ML4Code: Learning to Represent, Optimize, and Understand Code

S. VenkataKeerthy

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Doctor of Philosophy

Department of Computer Science and Engineering

May 22, 2026

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

(Signature)

S. VenkataKeerthy

(Name)

CS17M20P100001

(Roll No.)

Approval Sheet

This thesis entitled **ML4Code: Learning to Represent, Optimize, and Understand Code** by S. VenkataKeerthy is approved for the degree of Doctor of Philosophy from IIT Hyderabad.

(Prof. Subhajit Roy) Examiner
Dept. of Computer Science and Engineering
Indian Institute of Technology Kanpur

(Prof. Rupesh Nasre) Examiner
Dept. of Computer Science and Engineering
Indian Institute of Technology Madras

(Prof. Maunendra Sankar Desarkar) Internal Examiner
Dept. of Computer Science and Engineering
Indian Institute of Technology Hyderabad

(Dr. Ramakrishna Upadrasta) Adviser
Dept. of Computer Science and Engineering
Indian Institute of Technology Hyderabad

(Prof. Sumohana Channappayya) Chairman
Dept. of Electrical Engineering
Indian Institute of Technology Hyderabad

அகர முதல எழுத்தெல்லாம் ஆதி
பகவன் முதற்றே உலகு.

— திருவள்ளுவர்

Acknowledgements

"Be grateful that on this raging sea you have a mind to guide you," said Marcus Aurelius. I have been fortunate to have Prof. Ramakrishna Upadrasta as my advisor. He is one of the most kind and compassionate persons I have come across—a true mentor who has always *held the doors open*, both academically and personally (and quite literally!). From the very beginning of my master's degree, he has been a constant source of support, encouragement, and inspiration. He gave me the freedom to explore my interests, whether in research, teaching, or mentorship roles, while providing invaluable guidance and insights along the way.

His unwavering belief in my potential has been a driving force behind all my endeavors. Without his encouragement to continue my PhD at IITH, this journey would not have been possible. When I had the idea of applying machine learning to compilers, he patiently heard me out, was open to exploring it, and provided the necessary resources. He nurtured my *crazy* ideas—often wild and half-formed—into meaningful research, never rushing even when progress was slow. From him, I learned to *imagine* and *visualize* solutions to problems, and to think big. I will always cherish our dinner-time and car ride conversations that ranged from research to life philosophies to world affairs and spirituality. He has been more than an advisor; he has been someone I could turn to for counsel on everything, and I am deeply grateful for his mentorship, friendship, and the opportunities he has provided throughout this journey.

I had the privilege of having Professors Albert Cohen, R. Govindarajan, and Maunendra Desarkar on my doctoral committee. I am deeply grateful to them for agreeing to serve on my committee and for providing critical feedback at various stages that strengthened this work.

I wish to sincerely thank Prof. Albert Cohen for being gracious with his time and attention. Our regular discussions, his insights, and his valuable feedback have greatly shaped various works presented in this dissertation. I am also grateful for his encouragement and support in pursuing a student researchership at Google, and for his continued mentorship.

I am immensely thankful to Mircea Trofin, my host at Google. His guidance was crucial in upstreaming IR2Vec and MIR2Vec to LLVM, a milestone I am particularly proud of. He introduced me to the LLVM community and went above and beyond to support my stay, offering valuable career guidance along the way. His mentorship and friendship made my time at Google truly memorable.

I am also thankful to the wonderful team at Google: Sriraman Tallam, David Li, Kazu Hirata, Aiden Grossman, Chaitanya Mamata Ananda, and many others. In particular, Sri became a great mentor and friend, generous with his time and guidance. Working with such a talented team was an enriching experience, and our diverse lunch-table conversations were always a highlight. Thanks also to the LLVM community and maintainers for their constructive feedback during the upstreaming process.

I owe special thanks to Prof. Y. N. Srikant and Prof. Maunendra Desarkar for their roles in the genesis and development of IR2Vec. What began as a course project with Prof. Desarkar and an informal presentation to Prof. Srikant during his visit to IITH in 2018 grew into meaningful collaborations; their guidance and inputs during the formative stages were pivotal in shaping the work.

I am equally grateful to Prof. Fernando Magno Quintão Pereira for his support and collaboration on VexIR2Vec. He brought clarity and precision to our presentation, and our conversations were always insightful and enjoyable.

I also thank Prof. Subrahmanyam Kalyanasundaram for his valuable inputs on the theoretical formulations underlying VexIR2Vec.

I am thankful to Prof. Praveen Tammana for insightful research discussions at the intersection of Networks and Compilers. Thanks to Professors Saurabh Joshi, Jyothi Vedurada, Ashish Misra, and Rajesh Kedia for their guidance on career and research directions. I also thank Prof. Antony Franklin, Head of the Department, for his support at different stages of my journey at IITH.

I am grateful to Dr. Dibyendu Das, Dr. Venugopal Raghavan and the AMD team for practical insights on compiler optimizations and machine learning for compilers. Their feedback helped bridge the gap between academic research and industrial practice. I also thank Sanjay Ladwa and team at Qualcomm for their time and thoughtful feedback. This research was partially supported by the Google PhD Fellowship (2021) and the Qualcomm Innovation Fellowships (2024 and 2025).

I thank Prof. B. S. Murty, Director of IITH, for approving my conversion from the master's program to the PhD. I am also grateful to Prof. M. V. Panduranga Rao, then Head of the Department, for facilitating this transition. Their support, along with that of Prof. Bheemarjuna Reddy and other faculty members, was instrumental in my decision to continue my research journey at IITH. I am also thankful to the staff members of the department and institute for their help during my stay at IITH.

I am thankful to Prof. B. Karthikeyan, my advisor at SASTRA University, who introduced me to research and guided me in my initial research journey. He has been someone I could always count on for advice and guidance. He believed in me more than *anyone else* and encouraged me to aspire for greater heights.

I have been fortunate to have learned from Late Prof. N. Sairam during my undergraduate days. A gem of a person and a brilliant teacher, albeit strict, his passion for teaching and research was infectious.

My journey at IITH would not have been the same without the remarkable members of the Compilers lab. My seniors, Dangeti Tharun Kumar and Anirudh Sundar, were great mentors during my early days. I also learned much from Dr. Utpal Bora, Dr. Shalini Jain, Gayatri P. K., Abhishek Patwardhan, and Santanu Das, whose guidance helped me find my footing in the lab.

Many of the projects presented in this dissertation grew out of collaborations with an exceptional group of people. Rohit Aggarwal, Siddharth Jain, and Shalini have been my closest collaborators turned friends; working with them, learning from them, and growing together has been a pleasure. Thanks also to Yashas Andaluri, Soumya Banerjee, Umesh Kalvakuntla, Rajiv Chitale, Shiv Kumar, Animesh Singh, Pranav Gorantla, P. S. Raghul, Anilava Kundu, Sayan Dey, and Aayush Shrivastava for their contributions.

I made lasting friendships during my time at IITH. Nilesh Shah has been a constant companion for technical and non-technical discussions alike. Early morning lab and gym sessions, evening walks, conversations over tea, and travels with Nilesh, Siddharth, and Dr. Kaushal Kumar Maurya defined the later years of my PhD. Kaushal's dedication to research has been an inspiration. Without these three, my journey would not have been the same. Akash Banerjee, Rohit, M. S. Nishanth, and R. Sharath were great companions during the early days at IITH, and I cherish the time spent together. Thanks to Dr. Piyushi Manupriya, Dr. Thamilselvam, Dr. Yenda Ramesh, Pirapuraj Ponnampalam, Maharaj Brahma, Harish S. A., Ranjitha K., Eti Chaudhary, Dr. Sreekanth Madisetty, Dr. Maruthi Inukonda, and many others for their discussions and friendship.

I also had the opportunity to work with various members of the lab, including Sri Harika, Kuldeep Gautam, Gagandeep Goyal, Siddhartha Neyagapula, Raj Ambekar, Sukrut Rao, Ganesh Vernekar, Pankaj Kukreja, Sagar Jain, Shraiysh Vaishay, Happy Mahto, and others. The interns at the lab, including Kaivalya Aole, Divesh Mahajan, Nishant Sachdeva, Manas Dalvi, and many others, have been a pleasure to work with, and their contributions were vital to the development of the VexIR2Vec webapp and

other projects. Thanks to all of them for their help and support.

I am deeply thankful to my family for their love, support, and sacrifices. Without them, nothing would have been possible. No words can adequately express my gratitude, yet I must try.

My parents—my mother Usha SoundaraRajan and my father P. SoundaraRajan, have been the beacon of my strength and support. They were always there for me, believed in me, and encouraged me to pursue my dreams even when it meant sacrifices on their part. They have been the invisible foundation of my life and research journey, upon which I have built all my aspirations.

Thanks also to my grandparents, A. Parthasarathy and A. D. Anausya, C. Jagannatha Babu and C. J. Vanaja, and A. D. Rangalakshmi (*Ranga avva*), and my uncle P. Thiruvengadam (*Chinnanna*) for their unshakable love. I have reaped their love, help, and support in all my endeavors. Their values of sincerity, integrity, and compassion have guided me through life and research.

Finally, I am grateful for the journey itself, the years of imagining this work, and the joy of seeing it come together.

நெரித்த திரைக்கடலில் நின்முகங் கண்டேன்;
நீல விசும்பினிடை நின்முகங் கண்டேன்;
திரித்த நுரையினிடை நின்முகங் கண்டேன்;
சின்னக் குமிழிகளில் நின்முகங் கண்டேன்;
பிரித்துப் பிரித்துநிதம் மேகம் அளந்தே,
பெற்றதுன் முகமன்றிப் பிறிதொன் றில்லை;
சிரித்த ஒலியினிலுள் கைவி லக்கியே,
திருமித் தழுவியதில் நின்முகங் கண்டேன்.

—மகாகவி சுப்ரமணிய பாரதியார்
கண்ணன் பாட்டு

Dedication

Sarvam Sree Krishnarpanam

Abstract

Alan Turing envisioned machines that learn from experience rather than following rigidly programmed rules. Today, while Machine Learning (ML) has transformed domains like computer vision and natural language processing, its application to code remains an emerging frontier. Unlike natural language, programs have precise semantics, and even minor changes can fundamentally alter behavior. This distinction necessitates representations that capture not just statistical patterns but the semantic essence of code. This dissertation advances *ML4Code*, the application of machine learning to code, by developing semantics-aware, scalable program embeddings and demonstrating their effectiveness in compiler optimizations and program understanding. As the title suggests, we tackle these three aspects in three parts: *representing*, *optimizing*, and *understanding* code.

Part I addresses the fundamental challenge of representing programs for machine learning applications. We introduce a family of embedding frameworks, IR2Vec, MIR2Vec, and VexIR2Vec, that operate at the Intermediate Representation (IR) level, making them language-agnostic and application-independent. Unlike existing approaches that treat code as token sequences, our embeddings are driven by program analyses, encoding syntactic and semantic properties through entities such as opcodes, types, and operands using representation learning on knowledge graphs. Our bottom-up design aggregates instruction-level representations into basic block, function, and program embeddings, avoiding out-of-vocabulary issues. Crucially, this design enables embedding generation through simple vocabulary lookups rather than expensive model inference, making our approach 2–3× faster than contemporary tools while consuming significantly less memory. We develop multiple embedding variants that incorporate progressively richer program analysis: Symbolic embeddings capture inherent semantic properties, Flow-Aware embeddings encode data and control flow semantics, Profile-Aware embeddings incorporate dynamic execution behavior, and Path-Aware embeddings represent execution paths for fine-grained similarity. These embeddings operate at different abstraction levels: IR2Vec on LLVM IR for architecture-neutral analysis from source code, MIR2Vec on Machine IR for backend optimizations requiring target-specific details, and VexIR2Vec on VEX IR lifted from binaries for scenarios where source code is unavailable. Notably, IR2Vec and MIR2Vec have been upstreamed into the LLVM compiler infrastructure, marking the *first* integration of program embedding techniques into a mainstream production compiler.

Part II develops an end-to-end framework for ML-driven compiler optimizations, addressing challenges that span across the compilation pipeline. We tackle heterogeneous device mapping and thread coarsening for OpenCL kernels, decisions traditionally governed by handcrafted heuristics, using IR2Vec embeddings with supervised learning. This achieves prediction accuracies of up to 91% and geometric mean speedups of 1.1–1.6× over baseline heuristics. At the backend, we address register allocation, an NP-complete problem at the heart of code generation, through RL4ReAl, a multi-agent hierarchical reinforcement learning framework using MIR2Vec embeddings. This approach matches or outperforms LLVM's highly tuned Greedy allocator while demonstrating cross-architecture generalization through transfer learning from x86 to ARM. To enable practical deployment of these ML-driven optimizations, we develop ML-Compiler-Bridge, a modular and extensible infrastructure that bridges ML frameworks with compilers while keeping them framework-independent. Its two-component design—serializers and model runners—allows new communication and serialization mechanisms to be added by overriding minimal methods. We provide inter-process model runners (gRPC, pipes) for flexible training with Python-based frameworks, and in-process runners (ONNX, TensorFlow-AOT) for low-latency deployment, combined with diverse serialization options (Protobuf, JSON, bitstream). This design accommodates scenarios ranging from simple feed-forward queries to complex multi-round RL interactions and graph neural networks. With multi-language APIs spanning C++, C and Python, ML-Compiler-Bridge integrates seamlessly with compilers (LLVM, MLIR, Pluto) and ML frameworks (PyTorch, TensorFlow, RLlib)—requiring as few as *three* lines of code to connect a model with a compiler pass. We demonstrate its versatility across four ML-enabled optimizations spanning different representations and learning paradigms, achieving 7–22× faster compilation than Python wrapper approaches, while reducing training time by approximately 2×.

Part III extends our embeddings to program understanding tasks, particularly code similarity analysis. Using VexIR2Vec embeddings derived from VEX IR, an architecture-neutral representation lifted from binaries, we develop a binary similarity framework that handles cross-compiler, cross-optimization, cross-architecture, and obfuscation scenarios. Our approach outperforms state-of-the-art techniques by 18–60% in binary diffing and achieves a mean average precision of 0.76 in function retrieval, surpassing the nearest baseline by 46%, while being 3× faster. For source-level similarity, IR2Vec embeddings enable program classification that exceeds prior work by up to 9%, with richer semantic encodings yielding better results.

Collectively, this dissertation establishes semantics-aware program embeddings as a unifying foundation for ML4Code. By grounding our representations in program analyses rather than treating code as token sequences, we enable machine learning models to capture the deeper structure and meaning of programs. The resulting techniques are scalable, generalizable, and semantically sound, making them suitable for both research exploration and production deployment. As we move toward Turing's vision of machines that learn from experience, these methods chart a course toward intelligent systems that can autonomously represent, optimize, and understand code.

Contents

List of Tables

List of Figures

Algorithms and Code Listings

Algorithms

Code Listings

List of Abbreviations

Compilers and Infrastructure

AOT	Ahead-of-Time (compilation)
API	Application Programming Interface
AST	Abstract Syntax Tree
BB	Basic Block
CFG	Control Flow Graph
GCC	GNU Compiler Collection
ILP	Integer Linear Programming
IR	Intermediate Representation
JIT	Just-in-Time (compilation)
LLVM	Low Level Virtual Machine
MIR	Machine Intermediate Representation
MLIR	Multi-Level Intermediate Representation
PDG	Program Dependence Graph
SSA	Static Single Assignment
XFG	Contextual Flow Graph

Machine Learning

A2C	Advantage Actor-Critic (algorithm)
AI	Artificial Intelligence
APPO	Asynchronous Proximal Policy Optimization (algorithm)
CBOW	Continuous Bag of Words (model)
DL	Deep Learning
EAN	Entity-Attention Network
FCNN	Fully Connected Neural Network
GBDT	Gradient Boosted Decision Trees
GGNN	Gated Graph Neural Network

GNN	Graph Neural Network
GRU	Gated Recurrent Unit
KG	Knowledge Graph
LLM	Large Language Model
LSTM	Long Short-Term Memory
MARL	Multi-Agent Reinforcement Learning
MDP	Markov Decision Process
ML	Machine Learning
ML4Code	Machine Learning for Code
MLP	Multilayer Perceptron
NLP	Natural Language Processing
ONNX	Open Neural Network Exchange (format)
OOV	Out-of-Vocabulary (words)
PCA	Principal Component Analysis
PPO	Proximal Policy Optimization (algorithm)
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
t-SNE	t-distributed Stochastic Neighbor Embedding
TransE	Translational Embedding model

Hardware and Systems

CPU	Central Processing Unit
DPU	Data Processing Unit
DSP	Domain Specific Processor
GPU	Graphics Processing Unit
gRPC	gRPC Remote Procedure Calls
NPU	Neural Processing Unit
OpenCL	Open Computing Language
SIMD	Single Instruction Multiple Data
SmartNIC	Smart Network Interface Card
VLIW	Very Long Instruction Word

Optimizations and Techniques

BCF	Bogus Control Flow (obfuscation)
FLA	Control-Flow Flattening (obfuscation)
PBQP	Partitioned Boolean Quadratic Problem
PGO	Profile-Guided Optimization
RD	Reaching Definitions
UD	Use-Definition chains

Metrics and Evaluation

AP	Average Precision
CDF	Cumulative Distribution Function
F1	F1 Score (harmonic mean of precision and recall)
FN	False Negative
FP	False Positive
LOC	Lines of Code
MAP	Mean Average Precision
RSS	Resident Set Size
RTT	Round-Trip Time
TP	True Positive

Benchmarks and Datasets

CoFo	CodeForces (dataset)
NPB	NAS Parallel Benchmarks
POJ	Programming Online Judge (dataset)
SPEC	Standard Performance Evaluation Corporation (benchmark suite)
TSVC	Test Suite for Vectorizing Compilers

List of Symbols

Embeddings and Representations

$[\![\cdot]\!]$	Embedding of an entity; $[\![\mathbf{I}]\!]$, $[\![\mathbf{BB}]\!]$, $[\![\mathbf{F}]\!]$, $[\![\mathbf{P}]\!]$ denote embeddings of instruction, basic block, function, and program respectively
$\langle h, r, t \rangle$	Knowledge graph triple (head, relation, tail)
$E(h, r, t)$	Energy function for knowledge graph triple
$\mathcal{G}$	Knowledge graph of program entities
$\mathcal{V}$	Vocabulary of learned seed embeddings
$\mathcal{V}_{lookup}$	Vocabulary lookup function mapping entities to vectors
$G(V, E)$	Graph with vertices V and edges E (e.g., CFG, interference graph)
W_o, W_t, W_a	Weights for opcode, type, and arguments in instruction embedding
C_i	Set of instructions involved in circular dependencies with instruction I_i
$d_{\max}$	Maximum circular dependency degree over all circularly-dependent instructions
$\mathbb{I}$	Identity matrix
C_i	Set of instructions involved in circular dependencies with instruction I_i
$d_{\max}$	Maximum circular dependency degree over all circularly-dependent instructions
$\mathbb{I}$	Identity matrix

Machine Learning

$\mathcal{L}$	Loss function
$\mathcal{M}$	Machine learning model
α	Attention weight or learning rate
γ	Margin parameter (in TransE) or discount factor (in RL)
τ	Temperature parameter (e.g., in NT-Xent loss)
$\mathbf{u}$	Learned attention vector
$\mathbb{R}^n$	n-dimensional real vector space

$d(\cdot, \cdot)$	Distance function (e.g., Euclidean distance)
$\| \cdot \|$	Vector norm (e.g., L_2)

Reinforcement Learning

π_ω	Policy learned by agent ω
S, A, R	State space, action space, and reward
$S_\omega, A_\omega, R_\omega$	State, action, reward for agent ω
Ω	Set of agents (in multi-agent RL)
$\omega_\upsilon, \omega_\tau, \omega_\varphi, \omega_\xi$	RL4ReAl agents: node selector, task selector, splitter, coloring

Register Allocation

$\chi(v)$	Set of available registers for variable v
$\chi^{\mathcal{T}}, \chi^{\mathcal{C}}, \chi^{\mathcal{I}}$	Type, congruence, and interference constraints
$L(v)$	Live range of variable v
$\delta(v)$	Degree of node v in interference graph
$\mu(v)$	Spill of variable v to memory
$\varphi(v, k)$	Live range splitting at point k
$K(v)$	Set of use points for variable v
$v \blacktriangleright r$	Assignment of physical register r to virtual register v
R^t, R_a	Physical registers of type t; allocatable physical registers

VexIR2Vec and Binary Similarity

$\mathcal{P}$	Set of peepholes extracted from a function
π	A peephole (sequence of basic blocks in CFG)
k, c	Maximum peephole length and minimum coverage count
$\mathcal{M}_{diff}, \mathcal{M}_{search}$	Binary diffing and searching functions

List of Definitions

List of Takeaways

List of Artifacts

All software artifacts developed as part of this dissertation are publicly available as open-source repositories.

IR2Vec	LLVM IR based program embeddings. Upstreamed into LLVM. (Chapters 3, 6) https://llvm.org/doxygen/IR2Vec_8cpp.html https://llvm.org/docs/CommandGuide/llvm-ir2vec.html https://github.com/IITH-Compilers/IR2Vec
MIR2Vec	Machine IR based embeddings. Upstreamed into LLVM. (Chapter 7) https://llvm.org/doxygen/MIR2Vec_8cpp.html
VexIR2Vec	VEX IR based binary embeddings. (Chapters 4, 10) https://github.com/IITH-Compilers/VexIR2Vec https://compilers.cse.iith.ac.in/VexIR2Vec
ml-llvm-project	ML-driven compiler optimizations. (Chapters 7, 8) https://github.com/IITH-Compilers/ml-llvm-project
ML-Compiler-Bridge	Framework for ML-compiler interactions. (Chapter 8) https://github.com/IITH-Compilers/ML-Compiler-Bridge
Program Similarity	Program similarity using IR2Vec. (Chapter 11) https://github.com/IITH-Compilers/IR2Vec-Classification

Chapter 1

Introduction

"Can machines think?"

— Alan Turing

Alan Turing's seminal work on abstract computing machines, which we now call *Turing machines* [Turing, 1936] laid the foundation of the modern world of computer science. Often regarded as the father of computer science, Turing was also a pioneer of Artificial Intelligence (AI). He proposed that, rather than explicitly hardcoding rules to perform a task (i.e., programs), machines should *learn from experience*. This idea gave rise to his famous question: "*Can machines think?*"[Turing, 1950].

Since Turing's time, AI and computing have been intertwined with the shared goal of automating reasoning and problem-solving. While both fields have evolved together, they have diverged into two distinct areas. The availability of large-scale datasets, significant computational power, and advances in the underlying algorithms have led to the success of Machine Learning (ML) in various domains like computer vision, natural language processing, and speech recognition. Meanwhile, software systems have grown in size and complexity, necessitating better tools and techniques for optimization and understanding.

This shift has led to a resurgence in the convergence of AI and computing in recent years. In particular, there has been increasing interest in applying Machine Learning to code, leading to a new interdisciplinary area we refer to as Machine Learning for Code (ML4Code). ML4Code aims to leverage the power of ML techniques to address various challenges in compiler optimizations and program understanding.

This dissertation builds on this vision. We explore how programs can be effectively

represented for ML4Code applications like compiler optimizations and program understanding. We aim to develop scalable and effective ML models that can learn from the programs and generalize across different workloads and architectures.

1.1 Goals and Objectives

In this dissertation, we aim to address three broad Research Questions (RQs) that are central to ML4Code applications.

1.1.1 Program Representations

Programs exhibit several similarities with natural languages: both have well-defined grammar, structure and semantics. However, they are fundamentally different. Programs are highly sensitive to the syntax and semantics, and even a small change can lead to a completely different behavior. Hence we need ML models that go beyond capturing the *statistical correlations* between the tokens in the programs. This fundamental distinction motivates our first Research Question.

RQ1: Program Representations

Can we develop program representations that capture **semantic meaning** at **multiple abstraction levels**, enabling effective transfer across diverse ML4Code applications while remaining **computationally efficient**?

A well-designed program representation is critical for the success of ML applications in the domain of code. These representations must capture program behavior and characteristics, enabling downstream models to learn the underlying tasks effectively and generalize across different programs. Importantly, obtaining such representations should not be computationally expensive, and they must be scalable to handle large volumes of code. Effective program representations are essential for a variety of applications, such as *compiler optimizations* and *program understanding*.

1.1.2 ML-Driven Compiler Optimizations

Traditional heuristic-based optimizations in compilers, while carefully designed by domain experts, often fail to generalize across diverse workloads and architectures. This

can lead to *suboptimal* performance of programs. In contrast, machine learning techniques that leverage **program embeddings** offer a promising alternative. By learning in a data-driven manner, ML models can predict *near-optimal* solutions that improve performance across diverse settings.

Unlike typical ML tasks, compiler optimizations require *correctness guarantees*— an optimization that improves performance but alters the program's behavior is unacceptable. Furthermore, for ML-based optimizations to be adopted in real-world compilers, they must be *scalable and practically viable*.

These challenges give rise to our second Research Question.

RQ2: ML-Driven Compiler Optimizations

Can we build compilers that **learn continuously** from past executions and **adapt** to different workloads and architectures, while ensuring **semantic correctness** and **practical deployability** in production systems?

1.1.3 ML-Driven Program Understanding

On the other hand, understanding programs is a fundamental challenge in software engineering, with applications in security analysis and software maintenance. One important aspect of program understanding is the similarity analysis—determining which (portions of the) programs exhibit the same functionality (semantic similarity) despite the differences in the way in which they are expressed (syntactic differences).

Traditional rule-based techniques rely on features or heuristics are limited in their ability to capture the complex semantics of the programs. Machine learning approaches using *program embeddings* offer potential to capture deeper semantics of the programs at various levels of abstractions, like source code and binary code. This is particularly important in scenarios like malware analysis, where the programs are often obfuscated and compiled to different architectures.

These challenges lead to our third Research Question.

RQ3: ML-Driven Program Understanding

Can ML models understand program semantics deeply enough to identify **functionally similar code** across **different languages**, **architectures**, and **obfuscation techniques**?

1.2 Contributions

To address the Research Questions outlined in Section 1.1, we make the following key contributions in this dissertation.

1.2.1 Program Embeddings

We design program embeddings at the intermediate representation (IR) level, making them architecture and language independent. These embeddings are trained in an unsupervised manner, allowing them to be adapted for different applications such as compiler optimizations and program understanding. Our approach involves:

- We develop embeddings for different representations of a program:
  - **IR and MIR**: Derived from source code, useful when source code is available. While the IR itself is architecture-neutral, MIR is Machine-specific IR that captures some architecture details.
  - **VEX-IR**: Lifted from binary code, enabling analysis when source code is unavailable.
- We propose different variations of embeddings that capture different levels of program semantics:
  - **Symbolic**: Captures inherent semantic information of programs.
  - **Flow-Aware**: Captures data and control flow semantics.
  - **Profile-Aware**: Captures runtime behavior.
  - **Path-Aware**: Captures execution paths for better program analysis.

The embeddings are designed in a bottom-up manner, starting from instruction-level entities (opcodes, types, operands) and aggregating them into representations for

basic blocks, functions, and programs. This results in an embedding space where similar entities are closer together.

Furthermore, the embeddings are computationally efficient, enabling scalability for large codebases. We evaluate the effectiveness of these embeddings on various tasks such as compiler optimizations, program similarity analysis, and program comprehension. Our experiments demonstrate that these embeddings effectively capture program semantics and generalize across different workloads and architectures.

Integration with LLVM

To enable practical adoption and foster further research, IR2Vec and MIR2Vec have been upstreamed into the LLVM[1] compiler infrastructure [VenkataKeerthy, 2025a]. IR2Vec is available under `llvm/Analysis` and can be used directly within LLVM passes via public APIs. MIR2Vec is available under `llvm/CodeGen`, enabling MIR-level embedding generation within LLVM backends. A standalone tool, `llvm-ir2vec` [VenkataKeerthy, 2025b], is provided in `llvm/tools` for embedding generation and vocabulary training. This integration makes IR2Vec and MIR2Vec the first program embedding techniques to be part of a mainstream compiler infrastructure.

1.2.2 ML-Driven Compiler Optimizations

We develop machine-learning-driven compiler optimizations using program embeddings, ensuring both performance improvements and semantic correctness. We propose an end-to-end framework that integrates ML models with the LLVM compiler infrastructure for effective ML-driven optimizations.

Our contributions include:

- **Heterogeneous Device Mapping**: We use IR2Vec embeddings to determine the optimal device mapping for OpenCL kernels between CPUs and GPUs to improve performance.
- **Thread Coarsening**: We use IR2Vec embeddings to determine the optimal thread coarsening factor for OpenCL kernels to enhance the efficiency of parallel execution for performance improvements.
- **Register Allocation**:

[1]LLVM was originally an acronym for "Low Level Virtual Machine," but it is now the full name of the project.

  - We develop RL4ReAl, a novel RL-based register allocator using MIR2Vec embeddings.
  - A multi-agent RL approach is embedded within LLVM, ensuring the correctness of generated code.
  - Our method is architecture-independent and evaluated on Intel x86 and ARM AArch64, achieving results comparable to or better than production-grade LLVM register allocators.

- **ML-Compiler-Bridge: A Framework for ML-Compiler Integration**:
  - Enables seamless ML model integration with optimizing compilers.
  - Supports both research and production use cases.
  - Applied across multiple optimization problems, compilers, and RL training environments (gym infrastructures).

Using ML-Compiler-Bridge, we integrate ML-driven compiler optimizations such as RL4ReAl, POSET-RL [Jain et al., 2022a], and RL-LoopDistribution [Jain et al., 2022b] within LLVM, demonstrating their practical deployment in real-world compilers.

1.2.3 ML-Driven Program Understanding

We leverage machine learning approaches to develop models for effective program understanding, particularly in program similarity analysis across both binary and source codes.

Our contributions include:

- **Binary Similarity Analysis**:
  - We propose a novel binary similarity technique using VexIR2Vec's Path-Aware embeddings.
  - We design VexNet, a Siamese network that projects embeddings into a similarity space where semantically similar codes are closer together.
  - Our experiments on binary diffing and searching demonstrate that VexNet effectively generalizes across different architectures while accounting for variations induced by changes in compilers, optimization levels, and obfuscations.

- **Source Similarity Analysis**:
  - We conduct program classification experiments using IR2Vec's Symbolic, Flow-Aware, and Profile-Aware embeddings.
  - We demonstrate that richer semantic encodings yield better classification accuracy, with Profile-Aware embeddings achieving the highest performance when runtime profiles are available.
  - We apply these embeddings to a real-world application for mapping packet processing programs to hardware devices.

1.2.4 Publications and Artifacts

Some of the above contributions have resulted in publications at peer-reviewed journals, conferences, and workshops. All source codes, trained models, and experimental artifacts developed as part of this dissertation have been released as open-source software to facilitate reproducibility and enable future research.

Table 1.1 provides a consolidated list of publications included in this dissertation, along with the corresponding talks, chapters, and links to open-source artifacts.

Table 1.1: Publications Included in this Dissertation

Publication	Venue	Talks	Ch.	Artifacts
IR2Vec [Venkata-Keerthy et al., 2020]	TACO 2020 (Presented in HiPEAC 2021)	US LLVM Dev'25 RHPL@FSTTCS'20	3, 6	IR2Vec LLVM RFC LLVM Analysis llvm-ir2vec Tool
RL4ReAl [Venkata-Keerthy et al., 2023b]	CC 2023	LLVM Perf@CGO'23	7	LLVM MIR2Vec ml-llvm-project
ML-Compiler-Bridge [Venkata-Keerthy et al., 2024]	CC 2024	EuroLLVM'23 LLVM Perf@CGO'24	8	ML-Compiler-Bridge ml-llvm-project
VexIR2Vec [Venkata-Keerthy et al., 2025]	TOSEM 2025 (Presented in FSE 2025)	RHPL@FSTTCS'25	4, 10	VexIR2Vec WebApp
Packet Processing [VenkataKeerthy et al., 2023a]	APNet 2022	—	11	IR2Vec-Classification

Notably, IR2Vec and MIR2Vec have been upstreamed into the LLVM compiler infrastructure, making them the first program embedding techniques integrated into a mainstream production compiler. The unified `ml-llvm-project` repository[2] hosts implementations of ML-driven compiler optimizations including RL4ReAl, ML-Compiler-Bridge, POSET-RL, and Loop Distribution.

Other publications that are not included in this dissertation include:

- POSET-RL [Jain et al., 2022a], published in ISPASS 2022.
- Loop Distribution using Reinforcement Learning [Jain et al., 2022b], published in LLVM-HPC 2022.
- Generating Millions of SCoPs [VenkataKeerthy et al., 2023d], presentation in IMPACT 2023.

1.3 Overview of the Dissertation

This dissertation is organized into three parts, each focusing on different objectives laid out in Section 1.1. Figure 1.1 provides an overview of the dissertation organization.

In **Part I**, we discuss different program embedding approaches. Specifically, in Chapter 3 we present variations of IR2Vec and MIR2Vec embeddings, and in Chapter 4 we present VexIR2Vec embeddings.

In **Part II**, we discuss our works on ML-driven compiler optimizations. In Chapter 6, we present ML-driven optimizations for heterogeneous device mapping and thread coarsening. In Chapter 7, we present RL4ReAl, a RL-based register allocator. In Chapter 8, we present ML-Compiler-Bridge, a framework for integrating ML models with optimizing compilers.

In **Part III**, we discuss our works on ML-driven program understanding. In Chapter 10, we present binary similarity analysis using VexIR2Vec embeddings. In Chapter 11, we present source similarity analysis using IR2Vec embeddings.

Finally, we present the summary in Chapter 12, discuss our perspectives and lay out the gaps and future directions.

Reading Aids Throughout the dissertation, we use highlighted boxes to aid the reader: **Definitions** introduce key concepts and terminology; **Key Ideas** summarize the core insights of each contribution; **Rationales** explain design decisions and their

[2] https://github.com/IITH-Compilers/ml-llvm-project

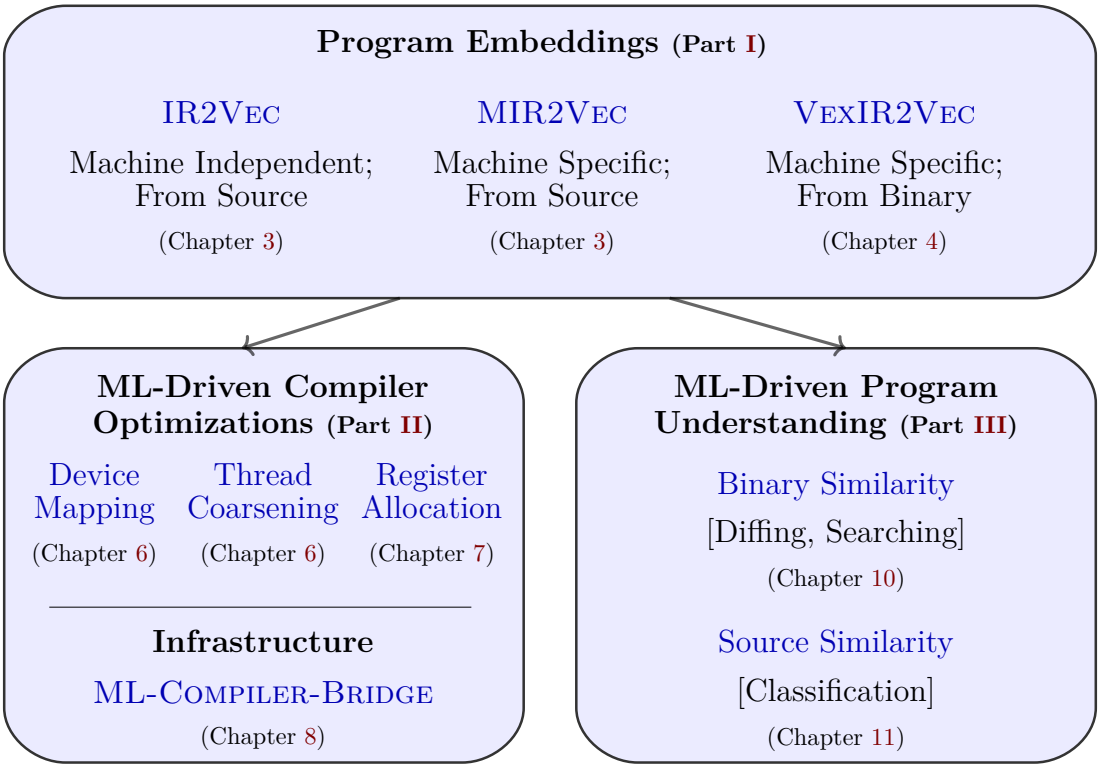


Figure 1.1: Overview and Organization of the Dissertation

justifications; **Visions** articulate broader goals and future directions; and **Takeaways** present the main findings from our experiments. Lists of Definitions and Takeaways are provided at the beginning of the dissertation for quick reference.

Part I

Program Embeddings

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

— Edsger Dijkstra

Chapter 2

Representing Programs as Vectors

2.1 Motivation and Background

With the growth of computing, comes the growth in computations. These computations are necessarily the implementation of well-defined algorithms [Cormen et al., 2009] as programs.

Thanks to the World Wide Web, these programs have become more accessible in various online platforms starting from programming tutorials to production quality code. Code snippets from open-source hosting sites (like GitHub, BitBucket), Community Question and Answer sites (like stack overflow) and downloadable binaries along with the relevant information gives rise to "Big Code" [Vechev and Yahav, 2016]. A good part of the Big code is largely attributed to several implementations of the same algorithm that differ in a multitude of ways like, their complexity and theoretical metrics of execution, and efficiency of implementation. While Big Data processing itself poses a lot of challenges, analyzing humongous volumes of code poses significant and much harder problems, because of the fact that most of the programming language questions are undecidable as stated by Rice's theorem [Rice, 1953].

Compiler Optimizations Compiler optimizations are essential to extract high performance on modern hardware architectures. Traditional compilers apply a series of optimizations to improve runtime efficiency, reduce memory usage, reduce power consumption and adapt code to specific hardware constraints.

However, choosing the best set of optimizations and parameters often involves solving NP-hard problems where exhaustive search is infeasible. Due to the complexity of the problem, most compilers rely on heuristic-based cost models to make optimization decisions. These heuristics, while generalizing well, do not always result in optimal performance.

Recent approaches leverage Machine Learning (ML) to improve the optimization decisions shifting the focus from fixed heuristic-based approach to adaptive, data-driven approach. For instance, ML-based techniques have been used to predict optimal loop unroll factors [Stephenson and Amarasinghe, 2005], make inlining decisions [Simon et al., 2013], determine thread coarsening factor [Magni et al., 2014], select optimal hardware device for code deployment in heterogeneous systems [Grewe et al., 2013], and drive vectorization decisions [Mendis et al., 2019b; Haj-Ali et al., 2020].

Program Comprehension Program comprehension variants involving program understanding is well understood to be undecidable [Rice, 1953]. So is the case in other software engineering and maintenance applications like code synthesis, bug detection. Conventional approaches primarily rely on static analysis techniques to approximate the program behavior [Hoare, 1969; Floyd, 1993; Cousot and Cousot, 1977; Cousot and Halbwachs, 1978]. However, these approaches are highly limited in terms of applications and can turn out to be unscalable.

Hence recent approaches leverage ML as an approximator to learn the properties of the programs. These learned properties can in turn be used to comprehend the semantic meaning of the programs. Some of such works include code description or comment generation [Iyer et al., 2016; Allamanis et al., 2016], method name predictions [Allamanis et al., 2015a; Alon et al., 2019b,a], code search and retrieval [Cambronero et al., 2019; Luan et al., 2019], algorithm recognition [Mou et al., 2016; Rosenblum et al., 2011], code synthesis [Rabinovich et al., 2017], and bug detection [Wang et al., 2016a; Gupta et al., 2017].

2.1.1 Learning Specialized Program Representations

The machine learning models typically require a fixed-size real-valued/numeric vector as input. To use programs as inputs to machine learning algorithms, it is essential to extract information from programs in a form that is *amenable to learning*.

Primarily, there are two ways of representing programs as inputs to such machine

learning algorithms – Feature-based representations and Distributed representations. Feature-based representation involves representing programs using hand-picked features—designed by domain experts [Fursin et al., 2011; Grewe et al., 2013; Magni et al., 2014]—specific for the particular downstream applications. Examples for the features could be the number of basic blocks, number of branches, number of loops, and even the derived/advanced features like arithmetic intensity.

On the other hand, distributed representations involve using a machine learning model to *automatically learn* to represent the input as a distributed vector [Bengio et al., 2013]. This learned representation—encoding of the program—is often called *program embedding*.

Definition 2.1: Program Embeddings

The *program embedding* is a distributed representation of the input program learned by using a machine learning model. It is a real-valued vector whose dimensions cannot be distinctly labelled, as opposed to that of feature-based representations.

Most of the existing works on using distributed learning methods to represent programs use some form of Natural Language Processing for modeling; all of them exploit the statistical properties of the code and adhere to the *Naturalness hypothesis* [Allamanis et al., 2018a] which states that -

Naturalness Hypothesis [Allamanis et al., 2018a]

Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build effective software engineering tools.

Such works that are based on the *naturalness hypothesis* primarily consider programs as sequence of tokens and use Word2Vec methods like skip-gram and CBOW [Mikolov et al., 2013a], or encoder-decoder models like Seq2Seq [Sutskever et al., 2014], or transformers [Vaswani et al., 2017] to encode programs as distributed vectors.

On the other hand, it is pertinent to understand that the programs are more than a sequence of tokens. They inherently have a structure that is defined by the data flow across the variables and control flow constructs like loops, branches, function calls, etc. Hence we believe that a carefully designed *embedding* which *encodes* these *semantic characteristics* of the program can be highly useful in making optimization decisions, in addition to being applied for software engineering. Hence, we *extend* the Naturalness hypothesis for programs to consider the program analyses information along with the

statistical properties for effective downstream applications in software engineering and compiler optimizations.

> **Definition 2.2: Extended Naturalness Hypothesis**
>
> *Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties* ***along with program analysis information****, can be exploited to build* ***effective*** *tools for software engineering* ***and compiler optimizations****.*

This *extended naturalness hypothesis* serves as the basis for the program embedding approaches described in this thesis. We believe that embedding meaningful semantic properties of the programs through program analysis information would lead to embeddings that are more effective for downstream applications. Moreover, such embeddings would be able to reduce the learning effort and complexity of the models of the downstream tasks thereby making the approach more practical and usable in real-world scenarios.

2.2 Levels of Abstraction of Programs

Programs can exist at various levels of abstraction, ranging from the source code, Abstract Syntax Tree (AST), and Intermediate Representation (IR) to Machine-specific Intermediate Representation (MIR), and finally, assembly or machine code. Each of these forms represents a distinct level of abstraction used by, or emitted by, different phases of the compiler, capturing various aspects of the program at each stage [Aho et al., 2006].

Figure 2.1 illustrates these distinct representations of programs, highlighting the abstraction levels they represent and the corresponding learning effort.

As shown in Figure 2.1, different levels of abstraction vary not only in the **learning effort** required by a model but also in the **amount of information captured** and the **language- or architecture-specificity** of the representation. At higher levels of abstraction, such as lexical tokens, the representation is language-specific and contains minimal structural information, which requires the model to learn syntactic and semantic regularities separately from scratch. This demands significant learning effort, as the model must process raw text with limited program-specific information.

Moving down to the **AST**, some of the hierarchical structure of the program is captured explicitly, providing a more informative representation that retains language-

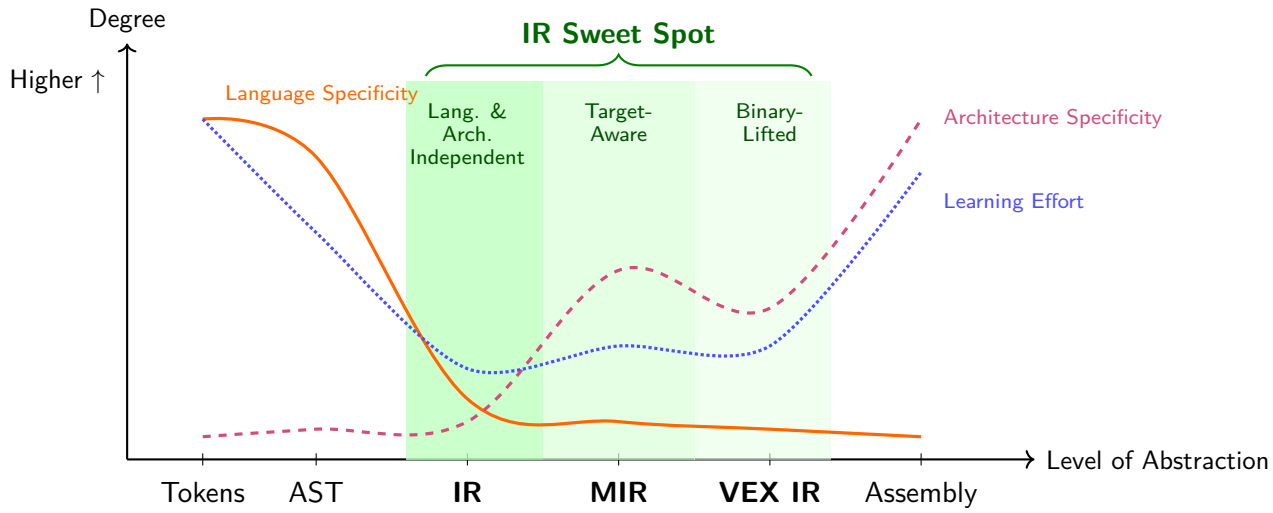


Figure 2.1: Different representations of programs across abstraction levels. Intermediate representations at various levels—from high-level IR to MIR and VEX IR—form a *sweet spot* that balances abstraction, semantic richness, and portability for learning program embeddings.

specific syntax but is still abstract in terms of program semantics. This reduces the learning effort slightly while still retaining some language dependence.

The **Intermediate Representation (IR)**, however, represents a *sweet-spot* in this trade-off. As a language- and machine-independent format designed for target-independent compiler optimizations, the IR is semantically rich, capturing essential program behaviors and structures without being tied to a specific language or hardware architecture. This makes it an ideal choice for learning program representations, as it provides a balanced level of abstraction that captures key information needed for effective training while remaining versatile across various programming languages.

Machine-specific Intermediate Representation (MIR) goes a step further by introducing some details specific to the target machine architecture. MIR provides a low-level view that can be highly valuable for analyses that require target-specific information like the opcodes, registers, and memory layout. This representation is useful for modeling code generation specific optimizations like instruction scheduling and register allocation that are specific to the target architecture. MIR supports fine-grained analysis of low-level operations while still offering a manageable abstraction compared to raw machine code.

At the lowest level, **assembly code** represents a human-readable, symbolic version of the fully compiled, executable form of the program. While it is highly architecture-specific and language-independent, assembly code contains detailed information about the program's execution at the hardware level. This makes it useful for tasks like reverse engineering, binary analysis, and security vulnerability detection.

However, even for such applications, as the assembly code is specific to the target architecture, it is less portable and harder to analyze across different platforms. Hence the binary analysis tools often lift the assembly code to a low-level intermediate representation like VEX-IR [Nethercote and Seward, 2007] to provide a more abstract view of the binary code. Such low-level IRs abstract out the target-specific details but remain close to the machine code.

Choosing the Appropriate Representation. Given the programs at different levels of abstraction, the choice of program abstraction should align with the goals of the analysis or application. Abstractions that are closer to source code, such as the lexical tokens or AST, are advantageous when syntactic and language-dependence are prioritized. While abstractions that are closer to the machine code, such as assembly code, offer fine-grained control and architecture-specific details, making them suitable for tasks that require a deep understanding of machine-level behavior.

However, as illustrated in Figure 2.1, intermediate representations at various levels occupy a *sweet spot* in the abstraction–specificity trade-off. While high-level IR (such as LLVM IR) is ideal for language-independent compiler optimizations, MIR becomes essential when target-specific details like register allocation and instruction scheduling are required. Similarly, when analyzing pre-compiled binaries where source code is unavailable, lifting assembly to a low-level IR like VEX IR enables portable analysis across architectures.

Crucially, all these intermediate representations share common characteristics that make them attractive for machine learning: they provide structured, semantically meaningful abstractions while remaining tractable for automated analysis. Whether one is optimizing compiler passes, scheduling instructions, or analyzing binaries, **some form of intermediate representation is almost always available and useful**. Hence, in this dissertation, we rely on intermediate representations at various levels as our primary abstraction for learning program embeddings—from LLVM IR for compiler optimization tasks to VEX IR for binary analysis applications.

2.3 Representation Learning

Representation Learning is a branch of machine learning that learns the representations of data by automatically extracting the useful features [Bengio et al., 2013]. Unsupervised models for learning distributed representations broadly fall under two major categories:

1. Context-window based embedding: Methods such as Word2Vec [Mikolov et al., 2013a], GloVe [Pennington et al., 2014] fall under this category. These include the skip-gram and CBOW models.

2. Knowledge graph embedding: Methods such as TransE [Bordes et al., 2013], TransR [Lin et al., 2015], TransD [Ji et al., 2015], TransH [Wang et al., 2014] fall under this category.

The context-window-based models operate on the basis of the surrounding context. The Knowledge graph embedding methods guide the learning of the entity representations based on the relationships that they participate in.

For program representations, we feel that it is important to consider semantic relationships rather than considering surrounding contexts; the latter could just mean the neighboring instructions. So, we prefer knowledge graph embedding models over context window embedding approaches. These knowledge graph embedding models use relationships to group similar datapoints together.

Knowledge Graph Representations

A knowledge graph [Paulheim, 2017] is a graph that captures the relationships between the entities that form the system under analysis. In our case, such entities are the constructs of the code: opcodes, types, values, etc. Typically, a knowledge graph models *entities* as vertices and the edges as the *relationship* between the entities.

Upon obtaining the knowledge graph, the representations of entities and relationships can be derived by using knowledge graph embedding approaches [Wang et al., 2017].

Formally, a knowledge graph $\mathcal{G}$ can be represented as a set of triplets $\langle h, r, t \rangle$, where h and t denote the *head* and *tail* entities connected by a *relation* r. The embeddings of h, r, and t in an n-d space are learned as *translations* from the head entity h to the tail entity t using the relationship r. Several knowledge graph embedding strategies have been proposed to represent the entities and relations in a continuous n-d space [Wang et al.,

2017]. These approaches learn a scoring function that returns a high value if $\langle h, r, t \rangle$ relation holds in $\mathcal{G}$, and a low value otherwise. This process results in learning the representations of the triplets ($[\![\mathbf{h}]\!] \in \mathbb{R}^n$, $[\![\mathbf{r}]\!] \in \mathbb{R}^n$ and $[\![\mathbf{t}]\!] \in \mathbb{R}^n$)[1] in an n-d space, where the third component of the triplet could be identified given the other two components using vector operations.

Using TransE

Of the many varieties available for KG embeddings, we use TransE [Bordes et al., 2013], a *translational representation learning* model, which embeds h, r and t on to the same high dimensional space. The model learns the relationship $[\![\mathbf{h}]\!] + [\![\mathbf{r}]\!] \approx [\![\mathbf{t}]\!]$, by minimizing the energy function, $E(h, r, t) = \|[\![\mathbf{h}]\!] + [\![\mathbf{r}]\!] - [\![\mathbf{t}]\!]\|_2^2$. The $\approx$ notation indicates that the embedding of h is closer to the embedding of t upon adding the embedding of the relation r. As relation r is used as a translation from the entity h to another entity t, this approach is called TransE.

Example 2.3.1:

Consider an LLVM IR instruction `store i32 %a, i32* %b`. One triplet extracted from this instruction is $\langle$`store`, `TypeOf`, `i32`$\rangle$, where $h =$ `store` (opcode), $r =$ `TypeOf` (relation), and $t =$ `i32` (type). TransE learns embeddings such that $[\![$`store`$]\!] + [\![$`TypeOf`$]\!] \approx [\![$`i32`$]\!]$. Intuitively, the relation `TypeOf` acts as a *translation* in the embedding space that moves the opcode embedding toward the type embedding. After training, opcodes that share the same type will have similar translations, causing semantically related entities to cluster together in the embedding space. This also enables learning implicit semantic analogies via vector arithmetic (see Section 3.8.2).

We train a model to minimize E using a margin-based ranking loss $\mathcal{L}$ given by the following equation.

$$\mathcal{L} = \sum_{\langle h,r,t \rangle \in \mathcal{G}} \sum_{\langle h',r,t' \rangle \notin \mathcal{G}} \max\left(0, \gamma + d([\![\mathbf{h}]\!] + [\![\mathbf{r}]\!], [\![\mathbf{t}]\!]) - d([\![\mathbf{h}']\!] + [\![\mathbf{r}]\!], [\![\mathbf{t}']\!])\right) \tag{2.1}$$

To minimize the loss function $\mathcal{L}$, we map code entities $\langle h, r, t \rangle$ to vectors that approximate the triangle equality; that is, $[\![\mathbf{h}]\!] + [\![\mathbf{r}]\!] \approx [\![\mathbf{t}]\!] \implies d([\![\mathbf{h}]\!] + [\![\mathbf{r}]\!], [\![\mathbf{t}]\!]) \approx 0$, where

[1] We use $[\![.]\!]$ to denote the embeddings in an n-dimensional space.

d is Eucledian distance. Likewise, if $\langle h, r, t\rangle \notin \mathcal{G}$, then it follows that $[\![\mathbf{h}]\!] + [\![\mathbf{r}]\!] \not\approx [\![\mathbf{t}]\!] \implies d([\![\mathbf{h}']\!] + [\![\mathbf{r}]\!], [\![\mathbf{t}]\!]') > 0$. Figure 2.2 provides an intuition on this objective function.

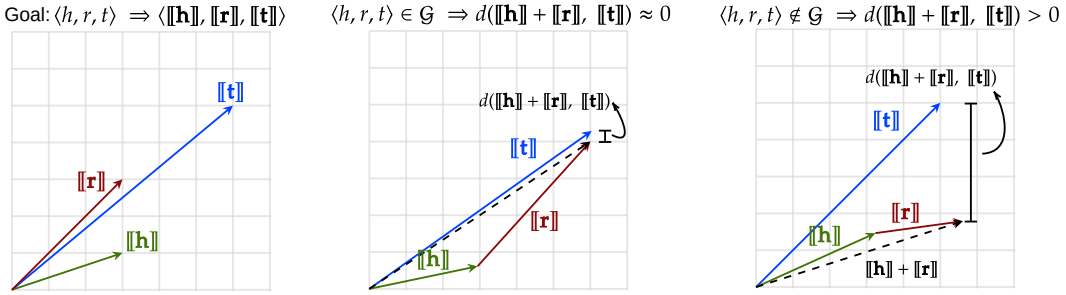


Figure 2.2: The construction of the vocabulary function $\mathcal{V}_{lookup}$ involves mapping entities $\langle h, r, t\rangle$ into triplets of vectors in a way that minimizes the distance between the vectors $[\![\mathbf{t}]\!]$ and $[\![\mathbf{h}]\!] + [\![\mathbf{r}]\!]$ whenever the triplet $\langle h, r, t\rangle$ belongs to the training set.

Additionally, $\mathcal{L}$ ensures that $d([\![\mathbf{h}]\!] + [\![\mathbf{r}]\!], [\![\mathbf{t}]\!])$ and $d([\![\mathbf{h}']\!] + [\![\mathbf{r}]\!], [\![\mathbf{t}']\!])$ are at least separated by a margin γ. At the end of this learning process, we obtain a vocabulary $\mathcal{V}_{lookup}$ containing the representation of each entity in an n-d Euclidean space $\mathbb{R}^n$. In the next section we shall explain how we derive the embedding of a program function out of the embedding of every entity that makes up this function.

Among the other Knowledge Graph Embedding models, TransE has relatively much smaller number of parameters to be learned, and has been used successfully in various algorithms for learning representations. Adopting this model ensures that the program representations are learned in a faster and effective manner that scales well on huge datasets [Ji et al., 2015; Han et al., 2018].

2.4 Our Schema of Program Embeddings

For modeling target-independent compiler optimizations, we use the LLVM IR for deriving the embeddings. LLVM IR [LLVM, 2018a] is a versatile, language-independent representation used by the LLVM compiler [Lattner and Adve, 2004]. LLVM IR captures the essential program semantics while providing access to a wide range of compiler analyses. For modeling target-specific optimizations, we use the Machine-specific Intermediate Representation (MIR) of LLVM that captures the target-specific details needed for low-level optimizations.

A key advantage of using LLVM IR and MIR is that they are **directly accessible**

within the compiler infrastructure. This enables seamless access to rich program analyses—such as control flow, data flow, type information, and alias analysis—without requiring separate extraction tools or preprocessing steps. Embeddings can be computed as part of the compilation pipeline itself, facilitating tight integration with compiler workflows and enabling practical deployment in production compilers.

Similarly, for modeling binary-level analysis, we use the VEX-IR , a low-level IR used by binary analysis tools like Valgrind [Nethercote and Seward, 2007]. VEX IR is readily available within the Valgrind framework, providing access to the binary analysis infrastructure and enabling the same tight integration benefits for binary analysis workflows.

Figure 2.3 illustrates our overall schema of program embeddings. Starting from a large **program corpus**, we first extract **intermediate representations** (LLVM IR, MIR, or VEX IR depending on the application). These IRs are then used for **pre-training** program embeddings, which involves two key steps: (1) learning a vocabulary of entity embeddings using TransE on the IR, and (2) refining these embeddings using program analysis information to produce Symbolic, Flow-Aware, Profile-Aware, or Path-Aware encodings. Finally, these pre-trained embeddings are **fine-tuned** for specific downstream applications such as compiler optimizations (Part II) and code similarity (Part III).

The key insight of our approach is the **decoupled** nature of embedding generation. The pre-training phase (vocabulary learning and refinement) is completely independent of the downstream application. This decoupling offers several advantages:

1. The vocabulary is learned **once** on a large corpus and can be reused across multiple downstream tasks without retraining.

2. Downstream models can be **smaller and faster** to train, as they build upon rich pre-trained representations.

3. The approach enables **transfer learning** across different compiler optimization and program analysis tasks.

Key Idea

Our decoupled approach results in a vocabulary that is *compact* yet *complete*—covering all IR constructs and avoiding Out-Of-Vocabulary issues. The vocabulary is learned through *unsupervised, task-independent pre-training* on a program corpus, requiring no labeled data. This lightweight pre-training, followed by lightweight fine-tuning

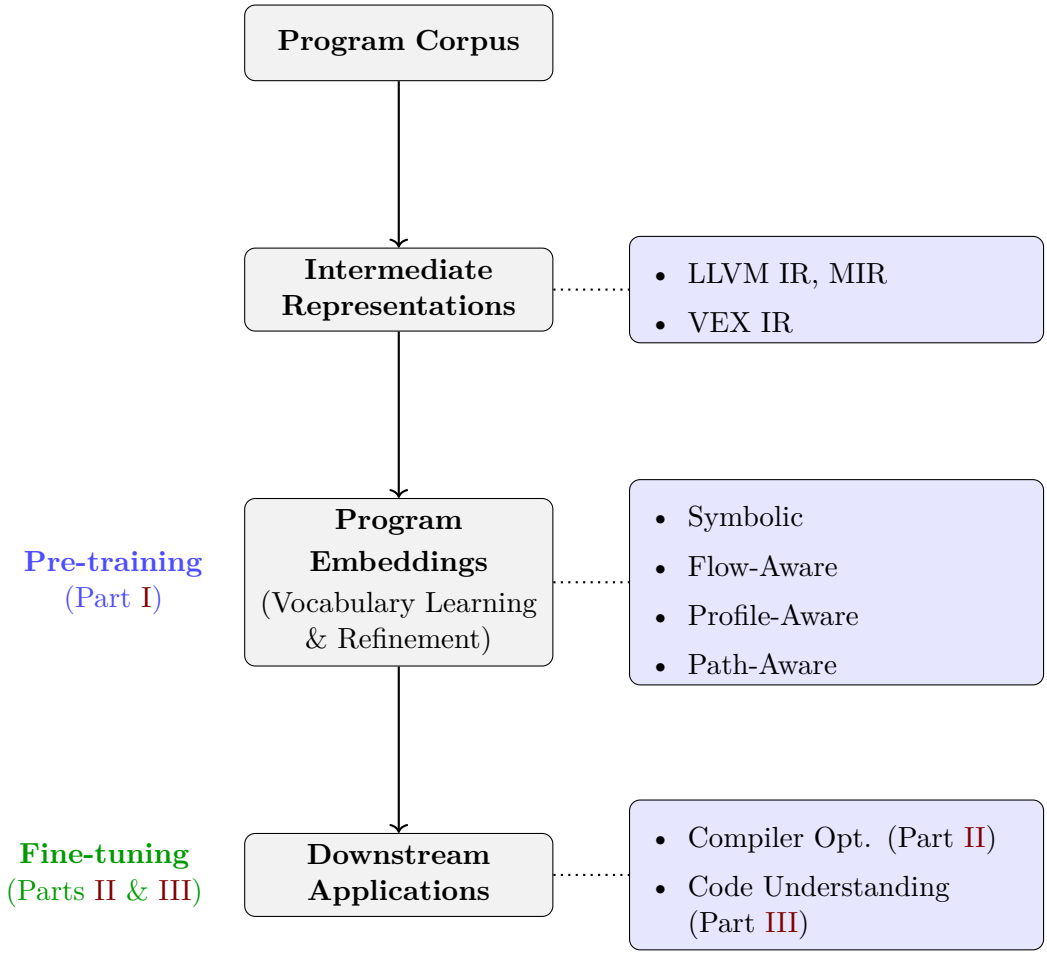


Figure 2.3: Schema of our program embedding approach. Starting from a program corpus, we extract intermediate representations and learn embeddings through unsupervised pre-training (Part I). These embeddings are then fine-tuned for downstream applications in Parts II and III. This decoupled design separates task-independent pre-training from task-specific fine-tuning.

> for downstream tasks, makes the approach *scalable* and *resource-efficient*—enabling practical deployment without the computational overhead of large end-to-end models.

Learning embeddings for these different program representations involves an unsupervised pre-training step to learn the vocabulary. This vocabulary is independent of the downstream application. The vocabulary is learned by training a representation learning model on a large corpus of programs in an unsupervised manner. We use TransE model [Bordes et al., 2013] for learning the embeddings. TransE is a translational knowledge graph embedding model that learns the embeddings of the entities. The learned

vocabulary consists of the embeddings of the entities in the programs.

The learned vocabulary captures the statistical properties along with the inherent semantic properties of the programs. Using the learned vocabulary, we derive the initial *symbolic embeddings* of the programs. Then, specific program analysis information like the control flow, data flow, and the program structure is used to refine these initial symbolic embeddings. This refined embedding typically encodes the semantic characteristics of the program thus adhering to the *Extended Naturalness Hypothesis for Programs*. The refined embeddings are then used for training the ML models for performing the downstream applications like compiler optimizations and program understanding in a supervised or semi-supervised manner.

2.5 Contributions

In this part of the dissertation, we address RQ1 posed in Chapter 1 on page 1 by presenting three different approaches to generate program embeddings for different levels of Intermediate Representation considering different program analysis information. Table 2.1 summarizes the embedding techniques, the intermediate representations, the program analysis information used, and the embedding variants.

Table 2.1: Program Embedding Techniques in this dissertation

Technique	**IR Level**	**Program Analysis / Transformations**	**Embedding Variants**
IR2Vec	LLVM IR	Liveness, Use-Def Chains, Reaching Definitions, Profile Information	Symbolic, Flow-Aware, Profile-Aware
MIR2Vec	Machine IR		Symbolic
VexIR2Vec	VEX IR	Random Walks on CFG, Copy/Constant Propagation, Redundancy Elimination	Path-Aware

IR2Vec**: LLVM IR based Program Embeddings** We present IR2Vec, a scalable program embedding generation technique that uses the LLVM IR as the input (Chapter 3). The embeddings come in three variants: *Symbolic* embeddings that capture the semantic relationships between IR entities, *Flow-Aware* embeddings that incorporate control and data flow information, and *Profile-Aware* embeddings that integrate execution frequency data.

MIR2Vec**: Machine IR based Program Embeddings** We present MIR2Vec, an extension of IR2Vec that generates *Symbolic* embeddings from the machine IR of LLVM (Section 3.4.3). MIR2Vec captures target-specific semantics while maintaining the abstraction benefits of intermediate representations.

VexIR2Vec**: VEX-IR based Program Embeddings** We present VexIR2Vec, an extension of IR2Vec that generates program embeddings from the VEX-IR (Chapter 4). VexIR2Vec uses random walk-based paths called *peepholes* to extract structural information from binary-lifted intermediate representations, and applies normalizing transformations such as copy propagation, constant propagation and folding, common subexpression elimination, and load-store elimination.

In each of these chapters, we describe the program embedding generation process involving the unsupervised vocabulary learning step followed by the refinement of the embeddings using different program analysis information available at the respective levels of Intermediate Representation. We evaluate the embeddings by posing analogy questions and visualizing the embeddings in a 2D space. We show the effectiveness of the embeddings in performing downstream tasks like compiler optimizations and program comprehension in Part II on page 90 and Part III on page 178 of the dissertation.

Outline of Part I

This part of the dissertation is organized as follows:

- Chapter 3 describes IR2Vec, including the vocabulary learning process using TransE, the three embedding variants (Symbolic, Flow-Aware, and Profile-Aware), and their evaluation through analogy tasks and visualization. This chapter also presents MIR2Vec as an extension for Machine IR.
- Chapter 4 describes VexIR2Vec, focusing on the VEX-IR representation, the peephole extraction process, and the path-aware embedding generation using random walks.

The experimental evaluation of these embeddings on downstream tasks—compiler optimizations and program similarity—is presented in Parts II and III respectively, following the decoupled approach illustrated in Figure 2.3.

Chapter 3

IR2Vec: LLVM IR based Program Embeddings

3.1 Introduction

As discussed in Chapter 2 on page 11, intermediate representations offer a compelling sweet spot for learning program embeddings—they are language-independent, semantics-preserving, and readily accessible within compiler infrastructures. Building on this foundation, we propose IR2Vec, an agglomerative approach for constructing a continuous, distributed vector to represent source code at different (and increasing) levels of IR hierarchy—Instruction, Function and Program. The vectors that are formed lower down the (program abstraction) hierarchy are used to build the vectors at higher levels.

We make use of the LLVM compiler infrastructure [Lattner and Adve, 2004] to process and analyze the code. The input program is converted to LLVM-Intermediate Representation (LLVM-IR), a language and machine-independent format. The initial vector representations of the (entities of the) IR, called seed embeddings, is learned by considering its statistical properties in a Representation Learning framework. Using these learned seed embeddings, hierarchical vectors for the new programs are formed. To represent Instruction vectors, we propose three flavors of encodings: *Symbolic*, *Flow-Aware*, and *Profile-Aware*. The *Symbolic* encodings are generated directly from the learned representations. Pertaining to the *Extended Naturalness hypothesis*, when the Symbolic encodings are augmented with the flow analyses information, they become *Flow-Aware*. The *Profile-Aware* encodings are generated by considering the profile information of the

program.

We show that the generic embeddings of IR2Vec provide superior results when compared to the previous works like DeepTune [Cummins et al., 2017a], Magni et al. [2014], and Grewe et al. [2013] that were designed to solve specific tasks. We also compare IR2Vec with NCC by Ben-Nun et al. [2018]; both have a similar motivation in generating generic embeddings using LLVM IR, though using different methodologies and techniques.

Key Idea

Symbolic embeddings capture the intrinsic semantics of LLVM IR entities—*what* an opcode or type represents—but are context-agnostic. Flow-Aware embeddings incorporate data flow information to encode *how* values propagate through computations. Profile-Aware embeddings further weigh these by execution frequency, capturing *which* contexts dominate at runtime.

Contributions: The following are our contributions:

- We propose a unique way to map LLVM-IR entities to real-valued distributed embeddings, called a seed embedding vocabulary.
- Using the above seed embedding vocabulary, we propose a concise and scalable encoding infrastructure to represent programs as vectors.
- We propose three embedding variants: *Symbolic*, *Flow-Aware*, and *Profile-Aware*, that are strongly based on classic program flow analysis theory and evaluate them through analogy tasks and visualization.
- Our novel methodology of encodings is *highly scalable* and performs better than the state-of-the-art techniques. We achieve an improved training time (up to 8640× reduction), our method is *non-data-hungry*, and it *does not encounter* Out-Of-Vocabulary (OOV) words.

Organization: This chapter is organized as follows: Section 3.2 provides the necessary background on LLVM IR and the motivation for IR-based embeddings. Sections 3.4–3.7 describe the methodology for constructing the seed embedding vocabulary and the *Symbolic*, *Flow-Aware*, and *Profile-Aware* encodings at instruction, function, and program levels. This section also presents MIR2Vec, an extension for Machine IR. Section 3.8 presents the experimental setup and evaluates the seed embeddings through

analogy tasks and visualization. Section 3.9 discusses the perspectives of IR2Vec, comparing the embedding variants and analyzing OOV issues. Section 3.10 discusses related works, and Section 3.11 concludes the chapter.

3.2 Background

3.2.1 LLVM and Program semantics

LLVM is a compiler infrastructure that translates source-code to machine code by performing various optimizations on its Intermediate Representation (LLVM IR) [Lattner and Adve, 2004]. LLVM IR is a typed, well-formed, low-level, *Universal IR* to represent any high-level language and translate it to a wide spectrum of targets [LLVM, 2018a]. Being a successful compiler, LLVM provides easy access to existing control and data flow analysis and lets new passes (whether analyses or transformations) to be added seamlessly.

The building blocks of LLVM IR include Instruction, Basic block, Function and Module (Figure 3.1). Every instruction contains opcode, type and operands, and each instruction is statically typed. A basic block is a maximal sequence of LLVM instructions without any jumps. A collection of basic blocks form a function, and a module is a collection of functions. This hierarchical nature of LLVM IR representation helps in obtaining embeddings at the corresponding levels of the program.

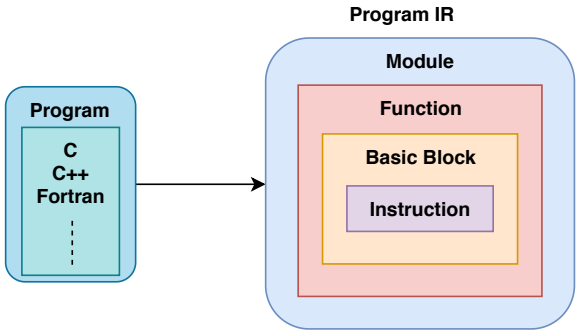


Figure 3.1: Building blocks of LLVM IR

Within a basic block, the flow of execution is sequential. Characterizing the flow of information which flows into (and out of) each basic block constitutes the data flow analysis. We study the impact of using such flow information as a part of the encodings.

The control flow of a program is primarily described by its branches. The probabilities of conditional branches can either be estimated statically or obtained dynamically. Static information is obtained by estimating the program profiles using heuristics and static analysis [Wu and Larus, 1994]. On the other hand, dynamic information is more accurate and involves dynamic profiling methods [Ball and Larus, 1992]. Such dynamic profiles typically involve executing the program on a representative input set and collecting the branch frequencies. From the profiling data, the branch probabilities can be estimated. This profile information is crucial in compiler optimizations like method inlining, loop unrolling, etc. for making better decisions.

Necessity for an LLVM-based Embedding. As discussed in Section 2.2, LLVM IR occupies a sweet spot in the abstraction hierarchy, making it ideal for learning program embeddings. Traditionally, machine-independent compiler optimizations were always considered to be part of the "middle-end" [Muchnick, 1997] of compilers. Using the LLVM infrastructure, carefully crafted heuristics have been implemented in various optimization passes to achieve better performance. As the power of using ML in compilers has been recognized [Cooper et al., 2002], many ML-driven approaches have been proposed [Joseph et al., 2007] as alternatives to these heuristics. However, they have primarily been feature-based, which means that integrating them into the middle-end is prone to challenges [Fursin et al., 2011], both at the modeling and engineering level.

Hence, there is a necessity for a *language- and architecture-agnostic embedding-based compiler framework* that bridges the gap between ML-based compiler optimizations and their adoptability in compiler infrastructures. IR2Vec addresses this gap by providing embeddings that integrate seamlessly with the LLVM compiler, offering a *generic compiler-based embedding*.

3.3 Overview of IR2Vec

Figure 3.2 illustrates the IR2Vec pipeline, from IR extraction to code vector generation. Instructions in IR can be represented as an Entity-Relationship Graph, with the instruction entities as nodes, and the relation between the entities as edges. A translational learning model that we discussed in Section 2.3 is used to learn these relations (Section 3.4.1). The output of this learning is a dictionary containing the embeddings of the entities and is called *Seed embedding vocabulary* ($\mathcal{V}_{lookup}$).

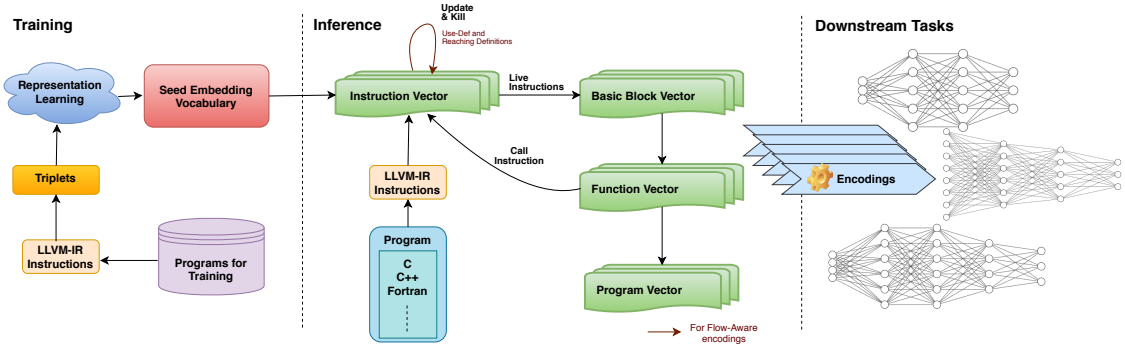


Figure 3.2: Overview of IR2Vec infrastructure

Definition 3.1: Seed Embedding Vocabulary

The *seed embedding vocabulary* ($\mathcal{V}_{lookup}$) is a dictionary mapping each IR entity (like opcodes, types, and generic arguments) to a fixed-dimensional real-valued vector, learned through representation learning on IR triplets. It serves as the foundation for constructing embeddings at all program abstraction levels.

The above dictionary is looked up to form the embeddings at various levels of the input program. At the coarsest level, instruction embeddings are obtained just by using the *Seed embedding vocabulary*. We call such encodings as *Symbolic* encodings (Section 3.4). We use the *Use-Def* and *Reaching definition* [Hecht, 1977; Muchnick, 1997] information to form the instruction vector for *Flow-Aware* encodings (Section 3.5). Then we use the *Profile information* of the program to form the *Profile-Aware* encodings (Section 3.6). The *Profile-Aware* encodings are obtained by propagating the *Flow-Aware* encodings with the profile information of the program.

The instructions which are *live* are used to form the *Basic block Vector*. This process of formation of a basic block vector is explained in Section 3.7. The vector to represent a function is obtained by using the basic block vectors of the function. The *Code vector* is obtained by propagating the vectors obtained at the function level with the `call graph` information.

3.4 Learning Instruction Embeddings

The instruction embeddings are first learned by considering the statistical properties of the IR. This embedding is learned using a translational learning model (Section 2.3). The

learned embeddings result in the *Seed embedding vocabulary*. The instruction embeddings are formed by using this vocabulary. In this section, we explain the process of forming the instruction vectors using the seed embedding vocabulary.

3.4.1 Seed Embedding Vocabulary for LLVM IR

Generic tuples

The opcode, type of operation (int, float, etc.) and arguments are extracted from the LLVM IR. This extracted IR is preprocessed in the following way: first, the identifier information is abstracted out with more generic information, as shown in Table 3.1. Next, the Type information is abstracted to represent a base type ignoring its width. For example, the type `i32` of LLVM IR is represented as `int`.

Table 3.1: Mapping identifiers to generic representation

Identifier	**Generic representation**
Variables	`VAR`
Pointers	`PTR`
Constants	`CONST`
Function names	`FUNCTION`
Address of a basic block	`LABEL`

Rationale

This abstraction intentionally discards value-specific information (e.g., which particular variable or constant is involved) in favor of the semantic and structural generalization. This yields a finite, OOV-free vocabulary that generalizes across programs.

Code triplets

From this preprocessed data, three major relations are formed: (1) `TypeOf`: Relation between the opcode and the type of the instruction, (2) `NextInst`: Relation between the opcode of the current instruction and opcode of the next instruction; (3) $\texttt{Arg}_i$: Relation between opcode and its i^{th} operand. This transformation from actual IR to relation ($\langle h, r, t \rangle$ triplets) is shown in Figure 3.3(a). These triplets form the input to the representation learning model.

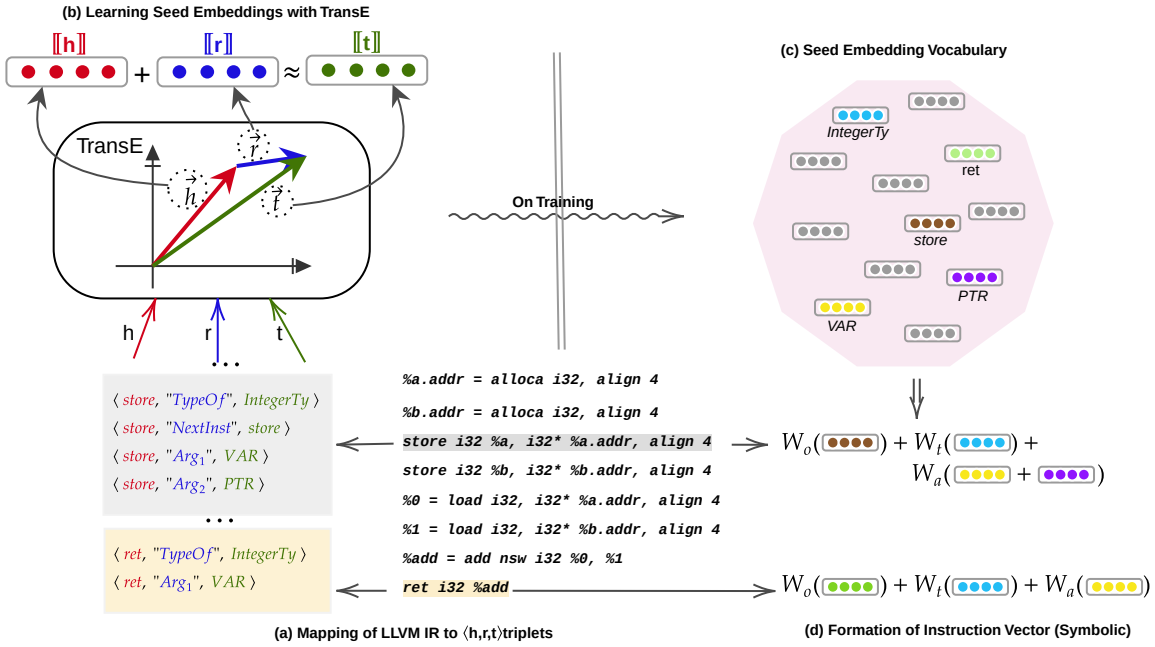


Figure 3.3: Overall schematic of IR2Vec: (a) [Data Collection] Mapping LLVM IR to $\langle h, r, t \rangle$ triplets. (b) [Vocabulary training] Learning representations using TransE. (c) [After Training] Obtained Seed Embedding Vocabulary. (d) [Inference] Formation of instruction vectors for new programs.

Example 3.4.1:

The LLVM IR shown in Figure 3.3 corresponds to a function that sums up the two integer arguments and returns the result. The first `store` instruction of LLVM IR in Figure 3.3(a) is of integer type with a variable as the first operand and a pointer as the second operand. It is transformed into the corresponding triplets involving the relations `TypeOf`, `NextInst`, $\texttt{Arg}_1$, $\texttt{Arg}_2$, as shown.

Learning Seed Embedding Vocabulary

As shown in Figure 3.3(b), the generated Code triplets $\langle h, r, t \rangle$ are used as input to the TransE learning model. On training, the model learns the representations of the entities forming the *Seed embedding vocabulary*, as shown in Figure 3.3(c). Since each entity (e.g., an opcode like `store`) participates in many triplets across the entire training corpus, its learned embedding aggregates information from all its contexts—enabling knowledge transfer across different parts of a program as well as across different programs.

3.4.2 Instruction Vector

Let the entities of instruction l, be represented as $O^{(l)}$, $T^{(l)}$, ${A_i}^{(l)}$—corresponding to Opcode, Type and ith Argument of the instruction and their corresponding vector representations from the learned *seed embedding vocabulary* be $[\![\mathbf{O}^{(l)}]\!]$, $[\![\mathbf{T}^{(l)}]\!]$, $[\![\mathbf{A_i}^{(l)}]\!]$. Then, an instruction of format

$$\left\langle O^{(l)}\ T^{(l)} {A_1}^{(l)} {A_2}^{(l)} \cdots {A_n}^{(l)} \right\rangle$$

is represented as a vector which is computed as:

$$W_o.[\![\mathbf{O}^{(l)}]\!] + W_t.[\![\mathbf{T}^{(l)}]\!] + W_a.\left([\![\mathbf{A_1}^{(l)}]\!] + [\![\mathbf{A_2}^{(l)}]\!] + \cdots + [\![\mathbf{A_n}^{(l)}]\!]\right) \tag{3.1}$$

Here, W_o, W_t and W_a are scalars ($\in (0, 1]$), the plus (+) denotes the element-wise vector addition operator, and the dot (.) denotes the scalar multiplication operator. Further, the W_o, W_t and W_a are chosen with a heuristic that gives more weightage to opcode than type, and more weightage to type than arguments:

$$W_o > W_t > W_a \tag{3.2}$$

This resultant vector that represents an instruction is the *Instruction vector* in **Symbolic encodings**.

Definition 3.2: Symbolic Encodings

A *symbolic encoding* represents instructions by combining vectors from the seed embedding vocabulary ($\mathcal{V}_{lookup}$) using weighted aggregation of opcode, type, and argument embeddings (Equation 3.1), without incorporating data or control flow information.

Example 3.4.2:

For the `store` instruction shown in Figure 3.3(d), the representations of opcode `store`, type *IntegerTy*, and arguments *VAR*, *PTR* are fetched from the seed embedding vocabulary, and the instruction is represented as $W_o.([\![\mathbf{store}]\!]) + W_t.([\![\mathbf{IntegerTy}]\!]) + W_a.([\![\mathbf{VAR}]\!] + [\![\mathbf{PTR}]\!])$. In the same figure, we also show a similar example of the `return` instruction.

3.4.3 MIR2Vec: Learning to Represent MIR Instructions

The above methodology can be easily extended to represent the instructions of the Machine IR (MIR) as well. This process is summarized in Figure 3.4.

MIR Entities Opcodes and MIR instruction arguments form the entities. MIR instructions are not typed, and hence the type information is not considered. Arguments primarily include physical and virtual registers, and immediate values. We abstract these arguments with generic identifiers as a preprocessing step.

Consequently, we create two different relations. (i) `NextInst`: Captures the relation between the current opcode and the next instruction opcode, (ii) $\texttt{Arg}_i$: Captures the relation between the opcode and the arguments of the instruction. Once the triplets are generated, we train the TransE model to obtain the embeddings for each of the entities.

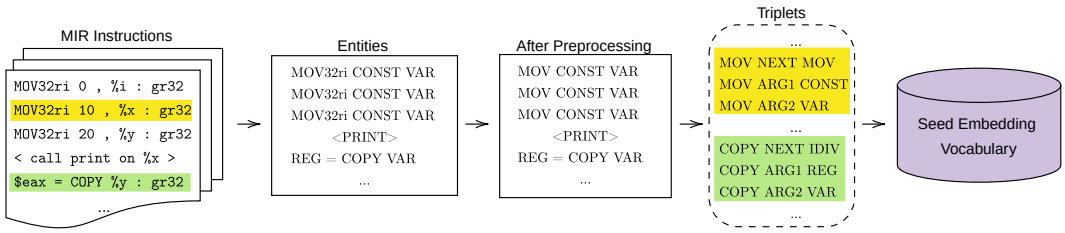


Figure 3.4: Illustration of learning MIR2Vec embeddings. The MIR Instructions are transformed into $\langle h, r, t \rangle$ triplets and the opcodes are grouped based on the operations. The resulting triplets are used as input to the TransE model to learn the embeddings. These learned embeddings form the seed embedding vocabulary of MIR2Vec.

Grouping of Opcodes MIR contains specialized opcodes, in terms of the operating width among other factors. MIR contains about 15.3K different possible opcodes in x86 and about 5.4K in AArch64. Obtaining a dataset to cover all such specialized operands would be highly infeasible, and in turn, would not generate good representations. Hence we mask out the opcodes based on their operating width, the source and destination locations (immediate, register, and memory) and group them together.

For example in x86, there are about 200 different `MOV` instructions operating on different bit width, sources, and destinations, like `MOV32r0`, `MOVZX64rr16`, `MOVAPDrr`, etc. All such opcodes are grouped together as a generic `MOV` token while forming the

triplets. The obtained triplets are fed to the TransE model to generate the embeddings for each entity-resulting in seed embedding vocabulary.

Representing Instructions For a given MIR instruction with opcode O and n arguments $A_1, A_2, \ldots, A_n$, its representation is computed as

$$W_o \cdot [\![\mathbf{O}]\!] + W_a \cdot ([\![\mathbf{A_1}]\!] + [\![\mathbf{A_2}]\!] + \cdots + [\![\mathbf{A_n}]\!]), W_o > W_a$$

where $W_o, W_a \in (0, 1]$ correspond to the weights of opcodes and arguments.

3.5 Embedding Data flow information

In this section, we explain the process of forming the instruction vectors using the flow information in the program. The flow information is used to form the *Flow-Aware* encodings. We primarily describe this process for LLVM IR, and the same can be extended to MIR as well.

An instruction in LLVM IR may define a variable or a pointer that could be used in another section of the program. The set of uses of a variable (or pointer) gives rise to the *use-def* (UD) information of that particular variable (or pointer) in LLVM IR [Hecht, 1977; Muchnick, 1997]. In imperative languages, a variable can be redefined; meaning, in the flow of the program, it has a different set of lifetimes. Such a redefinition is said to *kill* its earlier definition. During the flow of program execution, only a subset of *live* definitions would reach the use of the variable. Those definitions that *reach* the use of a variable are called its *Reaching Definitions*.

We model the instruction vector using such flow analyses information to form ***Flow-Aware*** encodings.

> **Definition 3.3: Flow-Aware Encodings**
>
> A *Flow-Aware encoding* extends the Symbolic encodings by specializing the abstracted generic representation of operands with the aggregated embeddings of its reaching definitions (Equation 3.3), thereby incorporating data flow information into the program representation.

Each A_j, which has been already defined, is represented using the embedding of its *reaching definitions*. The Instruction Vector for a reaching definition, if not calculated, is computed in a demand-driven manner.

Instruction Vector for Flow-Aware encodings

If RD_1, RD_2, ... , RD_n are the reaching definitions of $A_j{}^{(l)}$, and $[\![\mathbf{RD_i}]\!]$ be their corresponding representations, then, the encoding $[\![\mathbf{A_j}^{(l)}]\!]$ is calculated by aggregating over all the vectors of the reaching definitions as follows:

$$[\![\mathbf{A_j}^{(l)}]\!] = \sum_{i\,=\,1}^{n} [\![\mathbf{RD_i}]\!] \tag{3.3}$$

The Σ stands for the vector sum of the operands.

For the cases where the definition is not available (for example, function parameters), the generic entity representation of "VAR" or "PTR" from the learned *seed embedding vocabulary* is used.

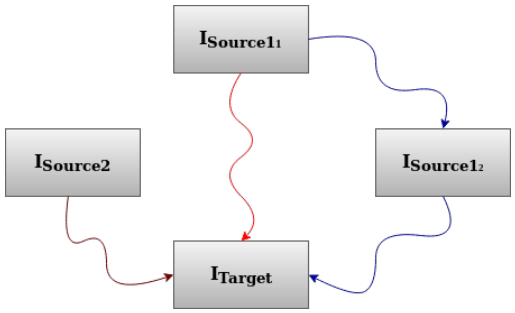


Figure 3.5: Illustration of propagating definitions to reach its use - Definition $I_{Source1_1}$ reaches I_{Target} either directly or via $I_{Source1_2}$. And, definition $I_{Source2}$ reaches I_{Target} directly. Hence, $[\![\mathbf{I_{Target}}]\!] = [\![\mathbf{I_{Source1_1}}]\!] + [\![\mathbf{I_{Source1_1}}]\!] + [\![\mathbf{I_{Source2}}]\!]$

Example 3.5.1: Flow-Aware Instruction Vector

An illustration is shown in Figure 3.5, where the instructions $I_{Source1_1}$ and $I_{Source2}$ reach I_{Target} as arguments. Here, the definition of $I_{Source1_1}$ could reach the argument A_{Target} of the instruction I_{Target} either directly or via $I_{Source1_2}$. And, definition $I_{Source2}$ reaches the argument A_{Target} of the instruction I_{Target} directly. Hence, $[\![\mathbf{A_{Target}}]\!]$ is computed as $[\![\mathbf{I_{Source1_1}}]\!] + [\![\mathbf{I_{Source2}}]\!]$.

An instruction is said to be *killed* when the return value of that instruction is redefined. As LLVM IR is in SSA form [Cytron et al., 1991; Lattner and Adve, 2004], each variable has a single definition, and the memory gets (re-)defined. Based on this, we categorize the instructions into two classes: ones which define memory, and the ones

that do not. The first class of instructions is *Write* instructions, and the second class of instructions is *Read* instructions.

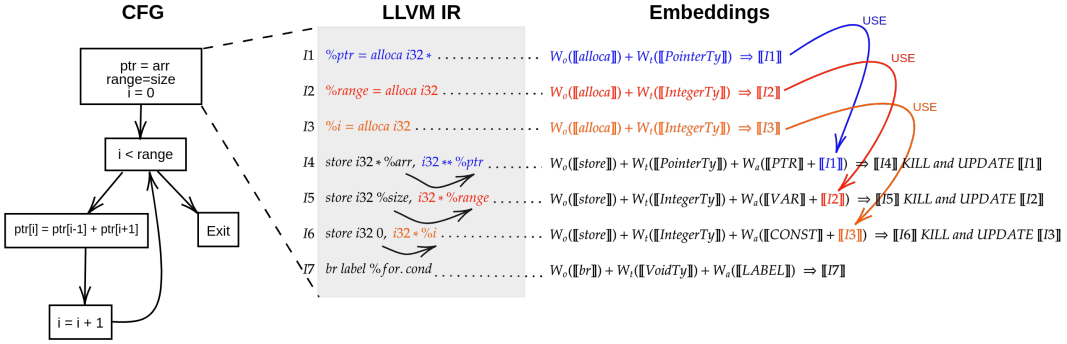


Figure 3.6: Illustration of generating intra basic block Instruction Vectors

For each instruction, the embeddings are formed as explained above. If these embeddings correspond to a write instruction, future uses of the redefined value will take the embedding of the current instruction, instead of the embedding corresponding to its earlier (killed) definition, until the current definition itself gets redefined. This process of *Kill and Update*, along with the *use* of *reaching definitions* for forming the instruction vectors within (and across) the basic block is illustrated in Figure 3.6 (and Figure 3.7) for the corresponding Control Flow Graph (CFG) respectively.

Example 3.5.2:

The CFG in Figure 3.6 corresponds to a function that takes `arr` and `size` as its two arguments. We expand the first basic block of this CFG and show its corresponding LLVM IR. The values of the two arguments are allocated memory in $I1$ and $I2$; this is followed by storing them to the local variables, `ptr` and `range` by the `store` instructions in $I4$ and $I5$. Similarly, memory for the loop induction variable `i` is allotted by $I3$ and is initialized to zero by $I6$.

Here, we show the process of propagating instruction vectors within the basic block.

Example 3.5.3:

The definition of `ptr` in $I1$ reaches the use of `ptr` in $I4$. So, the representation of $I1$ is used in its place as shown, instead of the generic representation of PTR. Also, the store instruction kills the definition of `ptr` in $I1$, as it updates the value of `ptr` with that of `arr`. Hence, in the further uses of `ptr`, the representation of $I4$ is used instead of $I1$. The same is the case for other store instructions in $I5$ and $I6$.

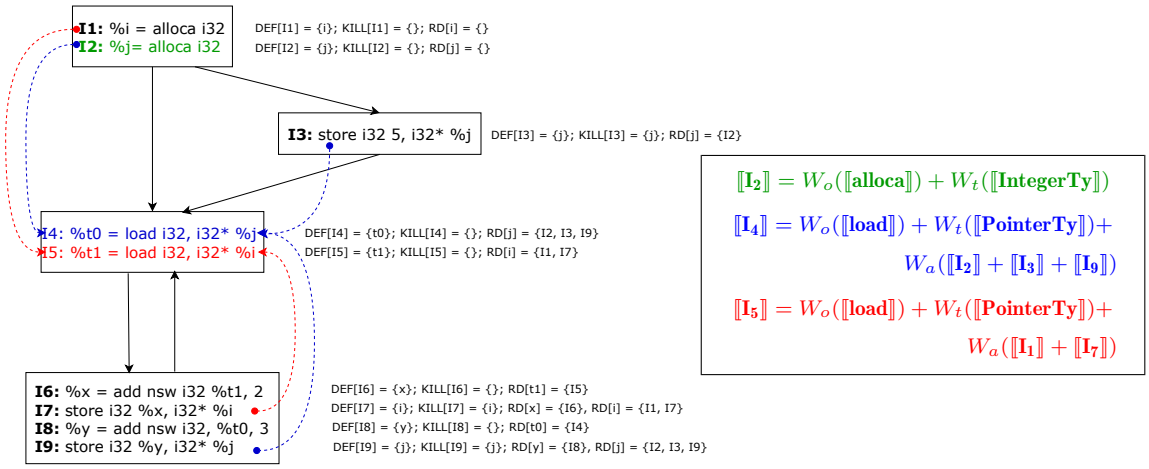


Figure 3.7: Illustration of generating inter basic block Instruction Vectors

Similarly, in Figure 3.7, we show how the propagation of instruction vectors happens across basic blocks. In this case, more than one definition can reach a use via multiple control paths. Hence, as explained in Equation 3.3, all possible definitions that can potentially reach a use are considered to represent the argument of that instruction.

Example 3.5.4:

In Figure 3.7(b), the definitions of `j` from $I2$, $I3$ and $I9$ can reach argument `j` of instruction $I4$ without being killed. Hence we conservatively consider all three definitions for representing the argument `j` in $I4$ as shown in Figure 3.7(c). Fig 3.7(b) also shows another such example for the instruction $I5$.

This resulting instruction vector which is formed by adding the exact flow information on the *Symbolic* encodings' instruction vector is used in obtaining ***Flow-Aware*** encodings.

Resolving Circular Dependencies

While forming the instruction vectors, circular dependencies between two write instructions may arise if both of them write to the same location and the (re-)definitions are reachable from each other.

Example 3.5.5:

To calculate $[\![\mathbf{I_4}]\!]$ (the encoding of I_4), in the CFG shown in Figure 3.8[a], it can be seen that the definitions of `i` from $I3$ and $I7$ can reach the argument `i` in $I4$. However, $[\![\mathbf{I_5}]\!]$ is needed for computing $[\![\mathbf{I_7}]\!]$ and is yet to be computed. But for computing $[\![\mathbf{I_5}]\!]$, $[\![\mathbf{I_7}]\!]$ is needed. Hence this scenario results in a circular dependency.

[a]Though this CFG is the classic irreducibility pattern [Hecht, 1977; Muchnick, 1997], it is easy to construct reducible CFGs which have circular dependencies.

This problem can be solved by posing the corresponding embedding equations as a set of simultaneous equations to a solver. For example, the embedding equations of I_5 and I_7 shown in Figure 3.8 would be:

$$\begin{aligned}
[\![\mathbf{I_4}]\!] =& W_o([\![\mathbf{load}]\!]) + W_t([\![\mathbf{IntegerTy}]\!]) + W_a([\![\mathbf{I_3}]\!] + [\![\mathbf{I_7}]\!]) \\
[\![\mathbf{I_5}]\!] =& W_o([\![\mathbf{store}]\!]) + W_t([\![\mathbf{IntegerTy}]\!]) + W_a([\![\mathbf{VAR}]\!]) + W_a([\![\mathbf{I_3}]\!] + [\![\mathbf{I_7}]\!]) \\
[\![\mathbf{I_7}]\!] =& W_o([\![\mathbf{store}]\!]) + W_t([\![\mathbf{IntegerTy}]\!]) + W_a([\![\mathbf{VAR}]\!]) + W_a([\![\mathbf{I_3}]\!] + [\![\mathbf{I_5}]\!])
\end{aligned} \tag{3.4}$$

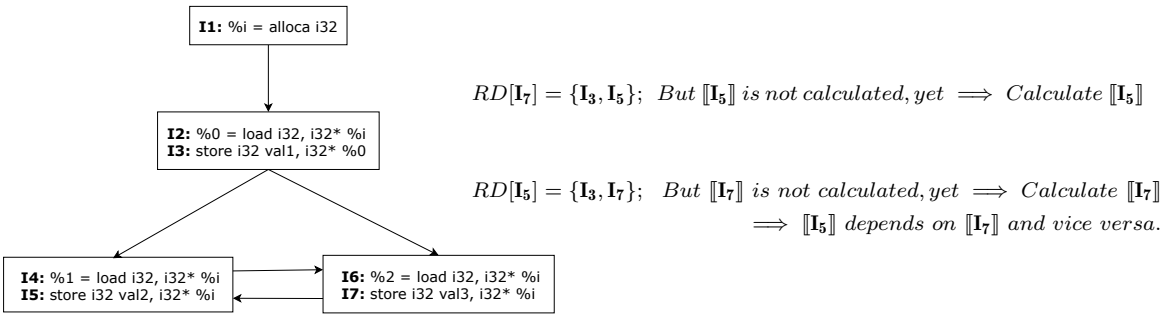


Figure 3.8: Control Flow Graph showing circular dependency

It can be seen that Equation 3.4 is a system of linear equations, where $[\![\mathbf{I_4}]\!]$, $[\![\mathbf{I_5}]\!]$ and $[\![\mathbf{I_7}]\!]$ are the unknowns that are to be calculated, while the rest of the values are known. Such embedding equations form a system of linear equations that can be posed to a solver to find the solution.

Just like any system of linear equations, there can be three cases of solutions.

1. **Unique solution:** In this case, the solution is obtained in a straightforward manner.

2. **Infinitely many solutions:** In this case, any one of the obtained solutions can

be considered as the result.

3. **No solution:** In this case, to obtain a solution, we can perturb the value of W_a as $W_a = W_a - \delta$ so that the modified system converges to a solution, with δ chosen randomly. If the system does not converge with the chosen value of δ, another δ could be picked, and the process can be iterated until the system converges.

We now show that, under the weight hierarchy constraint, the system always admits a unique solution—i.e., cases (2) and (3) do not arise in practice.

Matrix formulation. In general, each circularly-dependent instruction embedding $[\![\mathbf{I}_i]\!]$ satisfies an equation of the form

$$[\![\mathbf{I}_i]\!] = \underbrace{W_o \cdot [\![\mathbf{O}^{(i)}]\!] + W_t \cdot [\![\mathbf{T}^{(i)}]\!] + W_a \sum_{j \notin \text{cycle}} [\![\mathbf{A}_j^{(i)}]\!]}_{\mathbf{b}_i \text{ (known terms)}} + W_a \sum_{I_k \in \text{cycle}} [\![\mathbf{I}_k]\!]$$

Moving the unknown terms to the left-hand side and writing all such equations together in matrix form:

$$(\mathbb{I} - W_a \cdot \mathbf{A})\, \mathbf{x} = \mathbf{b} \tag{3.5}$$

where $\mathbb{I}$ is the identity matrix, $\mathbf{A}$ is a binary dependency matrix with $A_{ik} = 1$ when instruction I_i depends on I_k in the cycle (and 0 otherwise), $\mathbf{x}$ is the vector of unknown embeddings, and $\mathbf{b}$ is the vector of known terms. The system has a unique solution if and only if the coefficient matrix $(\mathbb{I} - W_a \cdot \mathbf{A})$ is non-singular.

Example 3.5.6: Matrix for the running example

For the circular dependency in Equation 3.4, the two unknowns are $[\![\mathbf{I_5}]\!]$ and $[\![\mathbf{I_7}]\!]$. Since I_5 depends on I_7 and vice versa, the dependency matrix is $\mathbf{A} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, and with $W_a = 0.2$ the coefficient matrix becomes $\mathbb{I} - 0.2 \cdot \mathbf{A} = \left[\begin{smallmatrix} 1 & -0.2 \\ -0.2 & 1 \end{smallmatrix}\right]$, which has determinant $1 - 0.04 = 0.96 \neq 0$—confirming a unique solution.

Definition 3.4: Circular Dependency Degree

For a system of circularly-dependent embedding equations involving n unknown instruction embeddings $[\![\mathbf{I}_1]\!], [\![\mathbf{I}_2]\!], \ldots, [\![\mathbf{I}_n]\!]$, let each depend on some subset of the other unknowns through its reaching definitions. The *circular dependency degree* of instruction I_i, denoted d_i, is the number of other unknown instruction embeddings that appear among its reaching definitions. The *maximum circular dependency degree* is

$d_{\max} = \max_{1 \le i \le n} d_i$.

Example 3.5.7: Circular Dependency Degree

In the circular dependency of Equation 3.4, instruction I_5 depends on I_7 (one circular dependency), and I_7 depends on I_5 (one circular dependency). Thus, $d_{\max} = 1$ for this example. If an instruction depended on two other instructions that were both part of the circular dependency, then $d_{\max} \geq 2$.

Why the matrix is always non-singular. Observing the structure of the coefficient matrix $(\mathbb{I} - W_a \cdot \mathbf{A})$, every diagonal entry equals 1 (from $\mathbb{I}$), and every other (off-diagonal) entry is either 0 or $-W_a$. Let S_j denote the set of instructions in the cycle that I_j depends upon, so the number of non-zero off-diagonal entries in row j is $|S_j|$, which is at most $d_{\max}$ (Definition 3.4). Hence, in each row j, the sum of absolute values of the off-diagonal entries is $W_a \cdot |S_j| \leq W_a \cdot d_{\max}$. Whenever $W_a \cdot d_{\max} < 1$, the diagonal entry $(= 1)$ strictly exceeds this sum, making the matrix *strictly diagonally dominant*. Such matrices are always non-singular (by the Lévy–Desplanques theorem), guaranteeing a unique solution. In the common case of pairwise circular dependencies, $d_{\max} = 1$ and the condition reduces to $W_a < 1$, which is always satisfied by the weight hierarchy.

Theorem 3.1 (Unique Solution for Circular Dependencies)**.** *Let $d_{\max}$ be the maximum circular dependency degree—the largest number of circularly-dependent reaching definitions that any single instruction in the cycle depends upon. If*

$$W_a \cdot d_{\max} < 1$$

then the system of linear equations arising from circular dependencies in Flow-Aware encodings (Equation 3.5) always has a unique solution. In particular, for pairwise circular dependencies ($d_{\max} = 1$), the weight hierarchy constraint $W_o > W_t > W_a$ (Equation 3.2) is sufficient.

Remark

The weight hierarchy heuristic (Equation 3.2) was originally designed for semantic reasons—prioritizing opcodes over types and types over arguments. For pairwise circular dependencies ($d_{\max} = 1$), it automatically satisfies the condition of Theorem 3.1. With standard weight choices (e.g., $W_a = 0.2$), even $d_{\max}$ up to 4 is easily handled.

In typical CFGs arising from structured programming constructs, $d_{\max}$ is small (often 1 or 2), so unique solutions are obtained for virtually all programs encountered. A detailed proof is given in Appendix A.3.

Table 3.2: Correspondence between classical dataflow analysis and Flow-Aware encodings

Aspect	**Classical DFA**	**Flow-Aware Encodings**
Domain	Abstract lattice $(L, \sqsubseteq)$	Vector space $\mathbb{R}^n$
Information	Lattice element $\ell \in L$	Embedding vector $[\![\mathbf{v}]\!] \in \mathbb{R}^n$
Confluence Operator	Meet $\sqcap$ / Join $\sqcup$	Summation $\sum$ (Equation 3.3)
Transfer Function	$f_{BB}(x) = \text{gen}_{BB} \cup (x \setminus \text{kill}_{BB})$	$W_o \cdot [\![\mathbf{O}]\!] + W_t \cdot [\![\mathbf{T}]\!] + W_a \cdot \sum [\![\mathbf{A}_i]\!]$
Convergence	Finite height + monotonicity	$W_a \cdot d_{\max} < 1$ (Theorem 3.1)

Takeaway 3.1: Correspondence of Flow-Aware Encodings with Dataflow Analysis

Flow-Aware encodings exhibit a correspondence with classical dataflow analysis, operating over a vector space rather than an abstract lattice.

- *Both frameworks propagate information along the CFG edges, transform it at nodes, and combine contributions where multiple definitions reach a use.*
- ***Confluence:*** *In classical reaching definitions, the confluence operator is set union; in Flow-Aware encodings, it is vector summation (Equation 3.3).*
- ***Transfer Function:*** *Classical DFA uses transfer functions with gen/kill sets; here, weighted aggregation (Equation 3.1) generates an instruction's embedding, while kill is handled separately via the Kill and Update process.*
- ***Cycles:*** *Cycles in the CFG induce recursive dataflow equations in classical DFA; cycles in the reaching-definitions graph induce circular embedding dependencies in Flow-Aware encodings.*
- ***Convergence:*** *Classical DFA guarantees convergence via finite lattice height and monotonicity; Flow-Aware encodings guarantee a unique solution when $W_a \cdot d_{\max} < 1$ (Theorem 3.1).*

Table 3.2 summarizes this parallel.

We note that this is a structural *correspondence—not a formal equivalence. Classical DFA operates over discrete lattices with exact set operations, while Flow-Aware*

encodings operate over continuous vector spaces with approximate aggregation. The correspondence is useful for understanding the design rationale, but the two frameworks differ in both their mathematical foundations and the nature of the information they propagate.

Handling Function Calls

Our encoding and propagation also take care of programs with function calls; the embeddings are obtained by using the call graph information. For every function call, the function vector for the callee function is calculated, and this value is used to represent the call instruction. A topological ordering of the strongly connected components (SCCs) in the call graph is used to resolve the order of processing the functions in the call graph.

For the functions that can be resolved only during link time, we just use the embeddings obtained for the `call` instruction. The final vector that is obtained encodes the function. The above procedure is applicable for recursive function calls as well.

3.6 Embedding Profile Information

Branches are the key in determining the program structure; which in turn, defines the *control flow* of the program. But, treating all the definitions which reach the current instruction by different paths equally may be misleading. So as to get a precise meaning of the program's control flow, each path through which the definitions could reach a use should be treated differently.

The key to doing the above is to take into account the probability of taking the path determined by each branch. This information can be obtained via static and dynamic analyses techniques which are readily available in compilers. Using `BranchProbabilityInfo` (BPI), an analysis pass in LLVM, it is possible to get an estimate of the probability with which the control flows from one basic block to another [LLVM, 2018b,c]. The probabilities of all the outgoing edges from a block naturally sum up to one. In program analysis, this information is highly valued and aids other optimization passes like determining *hotness* of a basic block, and similar instrumentation purposes. We use this `BPI` information to find the probability with which a *definition* could reach its *use*. In the context of IR2Vec, we use this probablity information to form the *Profile-Aware* instruction vectors.

Definition 3.5: Profile-Aware Encodings

A *Profile-Aware encoding* extends Flow-Aware encodings by weighting each reaching definition's contribution by the probability of the control-flow path through which it reaches the use point (Equation 3.6), based on the execution profile information.

Using BPI to construct Instruction Vector

If RD_1, RD_2, ... , RD_n are the reaching definitions of argument $A_j{}^{(l)}$, and $[\![\mathbf{RD_i}]\!]$ be their corresponding encodings, then Equation 3.3 is modified as follows:

$$[\![\mathbf{A_j}^{(l)}]\!] = \sum_{i\,=\,1}^{n} \sum_{k\,=\,1}^{\#paths} p_{ik}[\![\mathbf{RD_i}]\!] \tag{3.6}$$

Let p_{ik} be the probability of definition in instruction I_i reaching instruction I_l via path k. Then, $\sum p_{ik}$ is the sum of the probabilities of all such k paths reaching the instruction I_l from instruction I_i. The branch probability between two instructions I_x and I_y corresponds to the probability of reaching I_y from I_x.

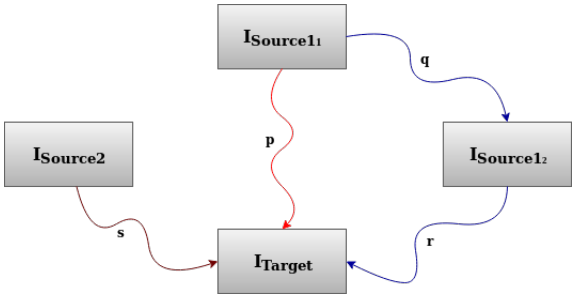


Figure 3.9: Illustration of propagating definitions to reach its use with branch probability information

Example 3.6.1: Profile-Aware Instruction Vector

An illustration is shown in Figure 3.9, where the instructions $I_{Source1_1}$ and $I_{Source2}$ reach I_{Target} as arguments. Here, the definition of $I_{Source1_1}$ could reach the argument A_{Target} of the instruction I_{Target} directly with a probability p or via $I_{Source1_2}$ with a probability $q{*}r$. Or, definition $I_{Source2}$ reaches the argument A_{Target} with a probability s. Hence, $[\![\mathbf{A_{Target}}]\!]$ is computed as $p[\![\mathbf{I_{Source1_1}}]\!] + q * r[\![\mathbf{I_{Source1_1}}]\!] + s[\![\mathbf{I_{Source2}}]\!]$.

Similar to the approach described for obtaining *Flow-Aware* encodings, the instruction vectors are formed by following the *Kill and Update* process within and across the basic blocks. The process of propagating instruction vectors within the basic block would remain the same as there is no branching within the basic block. The process of propagating instruction vectors across the basic blocks considers into account the branch probabilities between the basic blocks.

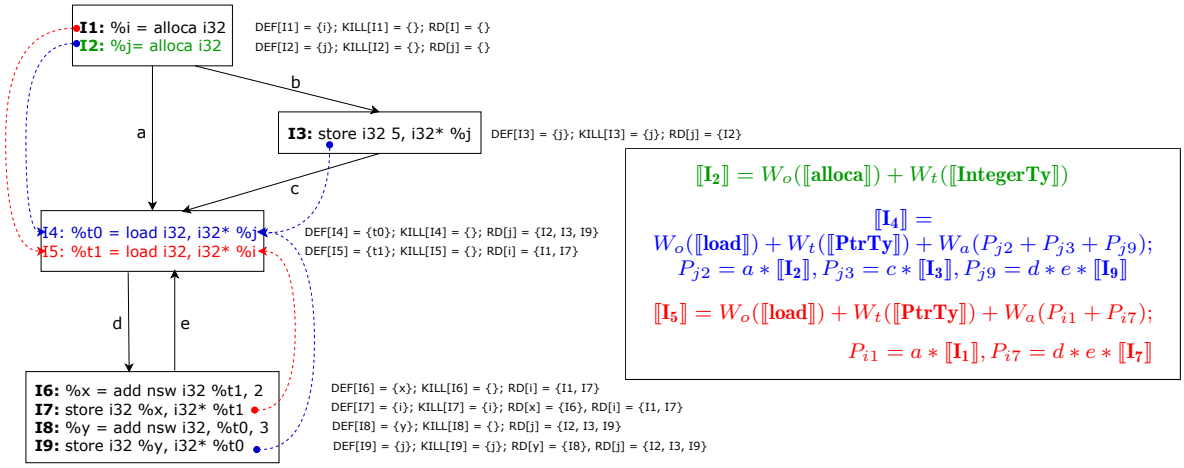


Figure 3.10: Illustration of generating inter basic block Instruction Vectors

Example 3.6.2:

Let us consider the example discussed in Figure 3.7 with the branch probabilities a, b, c, d and e between the basic blocks as shown in Figure 3.10. The definitions of `j` from $I2$, $I3$ and $I9$ can reach argument `j` of instruction $I4$ with probabilities a, c and $d * e$ respectively. Hence, the representation of argument j in $I4$ is computed as $a * [\![\mathbf{I2}]\!] + c * [\![\mathbf{I3}]\!] + d * e * [\![\mathbf{I9}]\!]$.

It can be easily observed that any circular dependencies that arise in this process can be resolved in a similar manner as described in Section 3.5. More broadly, the structural parallel with classical dataflow analysis (Table 3.2) extends to Profile-Aware encodings, with the confluence operator refined from unweighted summation to probability-weighted summation.

We summarize the process of obtaining the instruction vectors for the different flavors of the encodings in Algorithm 1.

3.7 Construction of Code vectors

Upon computing the instruction vector for the instructions of the basic block, we compute the cumulative Basic Block vector by using the embeddings of the instructions in the basic block.

In case of *Symbolic* encodings, the basic block vector is computed as the sum of the embeddings of all the instructions in the basic block.

$$[\![\mathbf{BB_i}]\!] = \sum_{k=1}^{m} [\![\mathbf{I_k}]\!] \tag{3.7}$$

However, in case of *Flow-Aware* and *Profile-Aware* encodings, the function vector is computed by considering the embeddings of those instructions that are not *killed*. For a basic block BB_i, the representation is computed as the sum of the representations of *live* instructions $LI_1, LI_2, ..., LI_m$ in BB_i.

$$[\![\mathbf{BB_i}]\!] = \sum_{k=1}^{m} [\![\mathbf{LI_k}]\!] \tag{3.8}$$

The vector to represent a function F with basic blocks $BB_1, BB_2, \ldots, BB_b$ is calculated as the sum of vectors of all its basic blocks as:

$$[\![\mathbf{F}]\!] = \sum_{i=1}^{b} [\![\mathbf{BB_i}]\!] \tag{3.9}$$

This process of obtaining the function vector is outlined in Algorithm 2.

The Algorithm 1 computes the vector representation for a particular instruction l. The representations of the opcode ($[\![\mathbf{O}]\!]$) and type ($[\![\mathbf{T}]\!]$) of the instruction are fetched from the seed embedding vocabulary. For computing the representation of $[\![\mathbf{A}]\!]$, we iterate over the list of arguments of the instruction.

In case of symbolic encodings, we directly use the generic representation of the argument from the seed embedding vocabulary irrespective of the argument's category.

In case of *Flow-Aware* and *Profile-Aware* encodings, we compute the representation of the argument based on the flow and profile information as follows: If the argument is a variable or pointer, we compute the representation of all its reaching definitions and sum them up to represent the argument, in case of *Flow-Aware* encodings. In case of *Profile-Aware* encodings, we multiply the probability of the definition reaching the use

with the representation of the definition. If the argument corresponds to a function, and the definition of that function is available, we find the representation of that function and use it as the representation of the argument. If the definition of the function is not available, then the generic representation of function in the seed embedding vocabulary is used to represent the argument. For the other cases, when the argument is a constant or an address of a basic block (its label), the corresponding generic representations are used. Finally, representations of all the arguments (of the instruction) are summed up to compute $[\![\mathbf{A}]\!]$. And, the instruction vector is computed as the weighted sum of $[\![\mathbf{O}]\!]$, $[\![\mathbf{T}]\!]$ and $[\![\mathbf{A}]\!]$.

Algorithm 1: Generate Instruction Vector

Inputs

- l: Instruction for which the embedding is to be computed
- $\mathcal{V}_{lookup}$: Seed Embedding Vocabulary
- *Encoding*: Type of encoding to be used - `Symbolic`, `Flow-Aware`, `Profile-Aware`
- W_o, W_t, W_a: Weights for Opcode, Type, and Arguments respectively

Output

- $[\![\mathbf{l}]\!]$: Embedding of the instruction l

$[\![\mathbf{O}]\!] \leftarrow \mathcal{V}_{lookup}(O^{(l)})$ `// Fetch value of Opcode of` l `from the vocabulary`
$[\![\mathbf{T}]\!] \leftarrow \mathcal{V}_{lookup}(T^{(l)})$ `// Fetch value of Type of` l `from the vocabulary`
$[\![\mathbf{A}]\!] \leftarrow \vec{0}$ `// Initializing n-d Argument vector to zeroes`
for *each argument* $A_i \in l$ **do**
  if $A_i \in$ *{VAR, PTR}* **then**
    if *Encoding is* `Symbolic` *or* A_i *is not a* `use` *of a definition* **then**
      $[\![\mathbf{A_i}]\!] \leftarrow \mathcal{V}_{lookup}(A_i)$
    else
      for *each reaching definition RD of* A_i **do**
        $[\![\mathbf{RD}]\!] \leftarrow$ getInstrVec(RD)
        if $[\![\mathbf{RD}]\!]$ *leads to cyclic dependency* **then**
          Resolve and obtain $[\![\mathbf{RD}]\!]$ as shown in 3.5
        if *Encoding is* `Profile-Aware` **then**
          $p =$ getProfileInfo(RD) `// Probability of` RD `reaching` A_i
        else
          $p = 1$ `// Flow-Aware encoding`
        $[\![\mathbf{A_i}]\!] \leftarrow [\![\mathbf{A_i}]\!] + p * [\![\mathbf{RD}]\!]$
  else if A_i *is FUNCTION* **then**
    if *Encoding is* `Symbolic` *or definition of* A_i *is not available* **then**
      $[\![\mathbf{A_i}]\!] \leftarrow \mathcal{V}_{lookup}(A_i)$
    else
      $[\![\mathbf{A_i}]\!] \leftarrow$ getFuncVec(A_i)
  else if A_i *is CONST* **then**
    $[\![\mathbf{A_i}]\!] \leftarrow \mathcal{V}_{lookup}(A_i)$
  else if A_i *is address of Basic block* **then**
    $[\![\mathbf{A_i}]\!] \leftarrow \mathcal{V}_{lookup}(A_i)$
  $[\![\mathbf{A}]\!] \leftarrow [\![\mathbf{A}]\!] + [\![\mathbf{A_i}]\!]$
return $[\![\mathbf{l}]\!] \leftarrow W_o * [\![\mathbf{O}]\!] + W_t * [\![\mathbf{T}]\!] + W_a * [\![\mathbf{A}]\!]$

Algorithm 2: Generate Function Vector

Inputs

- F: Function for which the embedding is to be computed
- *Encoding*: Type of encoding to be used - `Symbolic`, `Flow-Aware`, `Profile-Aware`

Output

- $[\![\mathbf{F}]\!]$: Embedding of the Function F

$[\![\mathbf{F}]\!] \leftarrow \vec{0}$ `// Initializing n-d Function vector to zeroes`
for *each basic block* $BB_i \in F$ **do**
  if *Encoding is* `Symbolic` **then**
    $worklist \leftarrow$ all instructions in BB_i
  else
    $worklist \leftarrow$ all *live* instructions in BB_i
  $[\![\mathbf{BB_i}]\!] \leftarrow \vec{0}$ `// Initializing n-d Basic block vector to zeroes`
  for *each instruction* $l \in worklist$ **do**
    Invoke Algorithm 1 to get instruction vector $[\![\mathbf{l}]\!]$
    $[\![\mathbf{BB_i}]\!] \leftarrow [\![\mathbf{BB_i}]\!] + [\![\mathbf{l}]\!]$
  $[\![\mathbf{F}]\!] \leftarrow [\![\mathbf{F}]\!] + [\![\mathbf{BB_i}]\!]$
return $[\![\mathbf{F}]\!]$

The Algorithm 2 computes the function vector. First, the representation of every basic block in the function is calculated. In case of Symbolic encodings, all the instructions in the basic block are considered. And, in other cases, only the live instructions are considered. The instruction vectors corresponding to these instructions are calculated by invoking Algorithm 1 and the basic block vector is obtained as the element-wise sum of these instruction vectors. Which, when summed up, forms the function vector.

If $[\![\mathbf{F_1}]\!], [\![\mathbf{F_2}]\!], \ldots, [\![\mathbf{F_f}]\!]$ are the embeddings of the functions $F_1, F_2, \ldots, F_f$ in a program, then the code vector representing the program P is calculated as the sum of the embeddings of all such functions as:

$$[\![\mathbf{P}]\!] = \sum_{i=1}^{f} [\![\mathbf{F_i}]\!] \tag{3.10}$$

3.8 Evaluation

In this section, we first describe the experimental setup and the training process of the embeddings. We then evaluate the effectiveness of the seed embeddings of IR2Vec.

The effectiveness of different flavors of embeddings is discussed in detail on various optimization and program identification tasks in Part II on page 90 and Part III on page 178.

3.8.1 Experimental Setup

We used the SPEC CPU 17 [Bucek et al., 2018] benchmarks and Boost library [Boost, 2018] as the datasets for learning the representations. The programs in these benchmarks are converted to LLVM IR by varying the compiler optimization levels (`-O[0-3], -Os, -Oz`) at random. The $\langle h, r, t \rangle$ triplets are created from the resultant IRs using the relations that were explained in Section 3.4.1. Embeddings of such triplets are learned using an open-source implementation of TransE [Han et al., 2018], which was explained in Section 2.3.

We wrote an LLVM analysis pass to extract the generic tuples from the IR so that they can be mapped to form triplets. There were $\approx 134M$ triplets in the dataset, out of which $\approx 8K$ relations were unique; from these, we obtain 64 different entities whose embeddings were learnt. The training was done with SGD optimizer for 1500 epochs to obtain embedding vectors of 300 dimensions.

These learned embeddings of 64 different entities form the *seed embeddings*. Additional related information is listed in the Table 3.4. We empirically set W_o, W_t and W_a to 1, 0.5, 0.2, respectively. The vectors at various levels were computed using another LLVM pass.

3.8.2 Evaluation of Seed Embeddings

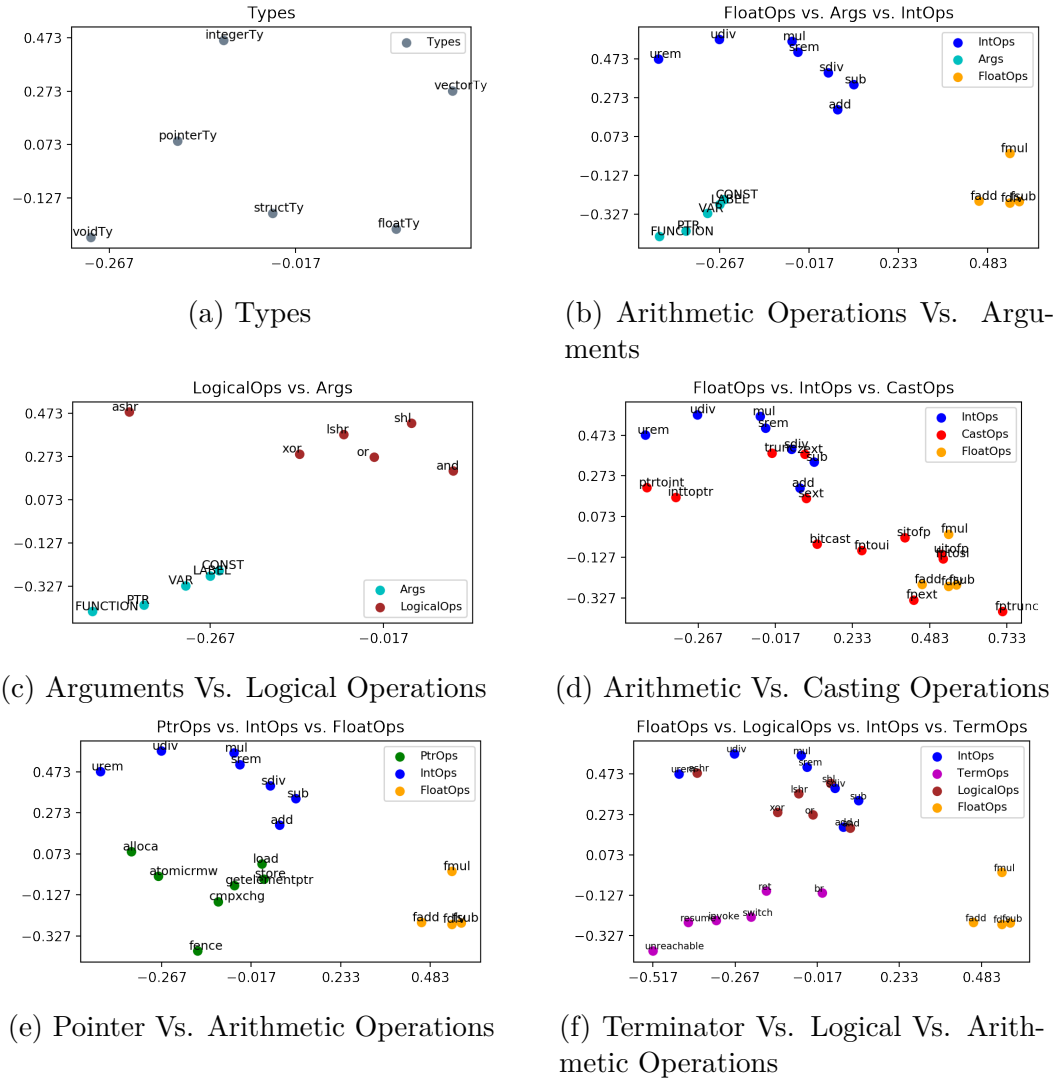
The effectiveness of the seed embeddings is analyzed by forming their clusters and by posing analogy questions to demonstrate if the relationship between the entities is captured effectively by the obtained seed embeddings.

Clusters

The entities corresponding to the obtained seed embeddings are categorized as groups based on the nature of the operations they perform — *Arithmetic operators* containing the integer and floating-point type arithmetic operations, *Pointer operators* which access memory, *Logical operators* which perform logical operations, *Terminator operators* which

form the last instruction of the basic block, *Casting operators* that perform typecasting, *Type* information and *Arguments*.

Clusters showing these groups are plotted using PCA (Principal Component Analysis), a dimensionality reduction mechanism to project the points from the higher to lower dimensional space [Wold et al., 1987]. Here, to visualize the 300-dimensional data in 2 dimensions, we use 2-PCA, and the resulting clusters are shown in Figure 3.11.



(a) Types

(b) Arithmetic Operations Vs. Arguments

(c) Arguments Vs. Logical Operations

(d) Arithmetic Vs. Casting Operations

(e) Pointer Vs. Arithmetic Operations

(f) Terminator Vs. Logical Vs. Arithmetic Operations

Figure 3.11: Comparison of embeddings of various seed entities

In Figure 3.11(a), we show the relation between various types. It can be observed that `vectorTy` being an aggregate type can accept any of the other first-class types and lies

approximately equidistant from `integerTy`, `pointerTy`, `structTy` and `floatTy`. And, `vectorTy` lies farthest from `voidTy` justifying that it is unlikely that `voidTy` elements will be aggregated together as vectors.

In Figure 3.11(b), we show that all integer based arithmetic operators are grouped together and are distinctly separated from floating-point based operators. It can be said that the analogies between the operators are captured. For example, the distance between (`add`, `fadd`) is similar to that of the distance between (`sub`, `fsub`). From Figure 3.11(b), 3.11(e) and 3.11(f), it can also be seen that the arithmetic operators are distinctly separated from the arguments, pointer operators and terminator operators. Similarly, from Figure 3.11(c), we can see that the logical operators are also distinctly separated from the arguments.

In Figure 3.11(d), we show the relationship between arithmetic and casting operators. It can be clearly seen that the integer based casting operators like `trunc, zext, sext`, etc. are grouped together with integer operators, and the floating-point based casting operators like `fptrunc, fpext, fptoui`, etc. are grouped together with floating-point operators. On observing Figure 3.11(d) and Figure 3.11(e), it can be seen that `ptrtoint` and `inttotpr` are closer to both integer operators and pointer operators. Figure 3.11(e) also demonstrates that the arithmetic operators are clearly distinct from pointer operators. Logical operators operate on integers, and hence they are grouped together with integer operators, as observed in Figure 3.11(f).

In summary, these clusters show that the obtained seed embeddings are indeed meaningful as they capture intrinsic syntactic and semantic relationships of LLVM entities.

> **Takeaway 3.2: Meaningful Semantic Clustering**
>
> *PCA projections reveal that semantically related LLVM entities—integer vs. floating-point operations, pointer vs. arithmetic operators, type hierarchies—form distinct clusters, validating that the learned embeddings encode meaningful program structure.*

Analogies

Queries of the form `a : b :: c : ?` are posed to the learned entity representation vectors. The missing value is computed by `b - a + c`, and the closest neighbor by Euclidean distance is considered to be the result.

Syntactic The syntactic information includes comparing the instructions based on the following:

Table 3.3: Sample Instruction Analogies

Syntactic	Semantic
zext : integerTy :: fpext : floatTy	sdiv : lshr :: mul : shl
trunc : fptrunc :: icmp : fcmp	fptoui : uitofp :: fptosi : sitofp
load : pointer :: store : constant/variable	sdiv : srem :: udiv : urem
fdiv : sdiv :: fmul : mul	shl : lshr :: or : and
call : ret :: switch : label	shl : ashr :: mul : div

- Datatypes (For example, integer type zero extension- `zext` vs. floating point type- `fpext`)
- Data structures (For example, instruction to shuffle element of vector type - `shufflevector` vs. instruction to get the address of an element - `getelementptr`) and
- Sequence of instructions which occur together (For example, *if* ***`load`*** *is equivalent to* ***`store`****, then* ***`landingpad`*** *is equivalent to* ***`invoke`***).

Semantic The semantic information reveals several interesting results:

- *If signed integer division is equivalent to right-shift operation, then integer multiplication is equivalent to left-shift operation.*
- *If integer to pointer conversion is equivalent to a pointer to integer conversion operator, then truncate is equivalent to zero extension operator.*

Table 3.3 shows some of the other such obtained analogies which imbibe the syntax, as well as the semantic information. A more detailed table (Table A.2) can be found in Appendix A.2.

> **Takeaway 3.3: Semantic Analogies from Unsupervised Learning**
>
> *The learned vocabulary correctly resolves semantic analogies such as* ***`sdiv:lshr::mul:shl`****, demonstrating that unsupervised training on LLVM IR triplets captures transferable program semantics without task-specific supervision.*

3.9 IR2Vec: Perspectives

In this section, we discuss some perspectives on IR2Vec.

3.9.1 Symbolic vs. Flow-Aware

The seed embedding vocabulary captures intrinsic syntactic and semantic relations at the entity level of the LLVM IR (Section 3.8.2). Hence, we believe that the primary strength of our encodings comes from the *seed embedding vocabulary*. This directly leads to the *Symbolic* encodings that achieve better performance than the other earlier methods on average. When the *Symbolic* encodings are augmented with flow information of the program, it results in *Flow-Aware* encodings. These encodings result in much more informative representation and hence lead to better accuracy and speedup than all other methods. This is evident from the results shown in Section 3.8.

However, this improvement in accuracy comes with a minimal overhead. Generating the *Flow-Aware* encodings take more time than *Symbolic* encoding, as it accounts for the time taken to generate and propagate the program flow information and to resolve circular dependencies in this process. We did an analysis of time taken by both of these encodings on a sample set of straightforward programs involving the family of sorting, searching, dynamic and greedy programs obtained from an online collection of programs [GeeksforGeeks, 2003].

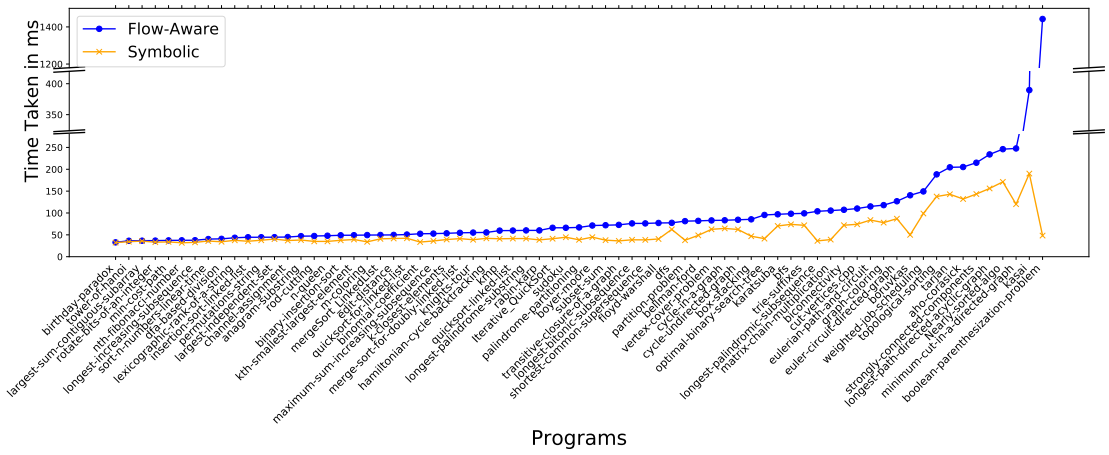


Figure 3.12: Comparison of time taken to generate the Symbolic and Flow-Aware encodings from *Seed Embedding Vocabulary*

The comparison of time taken to generate the *Symbolic* and *Flow-Aware* encodings from the *Seed Embedding Vocabulary* on the sample set is shown in Figure 3.12. It can be observed that, on an average, *Flow-Aware* encodings take 1.86 times more than that of *Symbolic* encodings. Their memory representations are of the same size, as both of

Table 3.4: Comparison Matrix: DeepTune vs. NCC vs. IR2Vec

Comparison metric	**DeepTune**	**NCC**	IR2Vec
Primary embedding	Token and Character level	Instruction level	Entity level
Files examined	Handpicked vocabulary	24,030	13,029
Vocabulary size	128 symbols	8,565 statement embeddings	64 entity embeddings
Entities examined	Application specific	≈640M XFG Statement Pairs	≈134M Triplets
Vocabulary training time	Task dependent	200 hrs on a P100	20 mins on a P100

them result in a floating-point vector in the same n-dimensions (300 dimensions for our setting).

3.9.2 Exposure to OOV words

For learning the representations of programs, the training phase of any method often involves learning a vocabulary that contains the embeddings corresponding to the input. When an unseen combination of underlying input constructs is encountered during inference, it would not be a part of the vocabulary and leads to *Out Of Vocabulary* (OOV) data points. In such cases, most of the models treat all the OOV words in a similar manner by assigning a common representation ambiguously, which may result in performance degradation.

Hence, to avoid OOV points, it is important to expose various and large (all possible) combinations of the underlying entities during the training phase. For example, for generating the embeddings at a statement-level of IR, all combinations of opcodes, types and arguments that can potentially form a statement should be exposed during training. Similarly, for generating token-based embeddings, all possible tokens that can possibly be encountered must have been exposed at the training time. But, both of these approaches would lead to a huge training space of $\mathcal{O}(|opcodes| \times |types| \times |arguments|)$ and $\mathcal{O}(|tokens|)$ (where $|tokens|$ can potentially be unbounded, with tokens being used more in the sense of a lexeme [Aho et al., 2006]) respectively. As it can be seen, covering such a huge intractable space is infeasible, and hence undesirable.

Consequently, the methods that learn statement level representations like NCC [Ben-Nun et al., 2018] face OOV issues. However, DeepTune [Cummins et al., 2017a] uses a Token and character-based hybrid approach to overcome this issue. DeepTune's method involves usage of the embeddings corresponding to the token if it is present in vocabulary. Otherwise, they break the token into a series of characters and use the corresponding embeddings of the character.

On the other hand, IR2Vec forms embeddings at the entity level of the IR, and hence, it is sufficient to expose a training space of only $\mathcal{O}(|opcodes|+|types|+|arguments|)$ to avoid *OOV* points. With this insight, it can be seen that IR2Vec can avoid the *OOV* issue even on exposure to a smaller number of programs at training time when compared to the other approaches. Consequently, this results in a smaller vocabulary and, hence, better performance than the other methods. A comparison of IR2Vec with DeepTune and NCC with respect to training and vocabulary is shown in Table 3.4.

A comparison of the number of *OOV* entities encountered by NCC and IR2Vec on the same set of programs used in Section 3.9.1 is shown in Figure 3.13. It can be seen that our method does not encounter any *OOVs* even when exposed to lesser training data, thereby achieving good scalability.

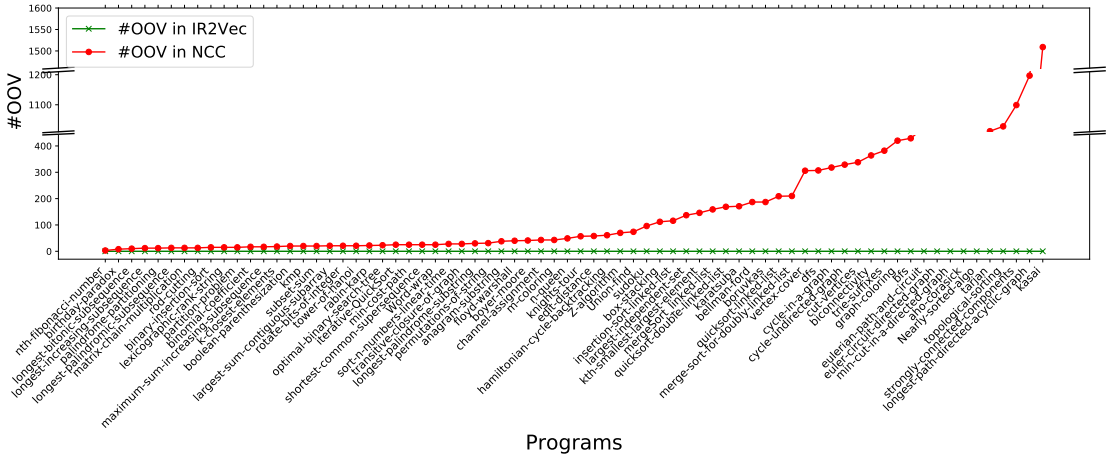


Figure 3.13: Comparison of the number of *OOV* entities encountered by NCC and IR2Vec

Takeaway 3.4: Zero OOV with Minimal Training

Entity-level vocabulary reduces training space from $O(|opcodes| \times |types| \times |args|)$ *to* $O(|opcodes| + |types| + |args|)$*, enabling* IR2Vec *to achieve zero OOV words even with 45% less training data than statement-level approaches.*

The modular design of IR2Vec—with vocabulary training decoupled from embedding generation also opens possibilities for future enhancements including the use of Large Language Models [Rozière et al., 2024; Wang et al., 2023b] for learning richer seed vocabularies. We discuss this and other future directions in Section 12.2 on page 253.

3.10 Related Works

Modeling code as a *distributed vector* involves representing the program as a vector, whose individual dimensions cannot be distinctly labeled. Such a vector is an approximation of the original program, whose semantic meaning is "distributed" across multiple components. In this section, we categorize some of the existing works that model codes, based on their representations, the applications that they handle, and the embedding techniques that they use. Then, we discuss the details of some specific recent works in this theme.

3.10.1 Representations, Applications and Embeddings

Representations Programs are represented using standard syntactic formats like lexical tokens [Allamanis et al., 2015a, 2016; Cummins et al., 2017a], Abstract Syntax Trees (ASTs) [Brauckmann et al., 2020; Alon et al., 2018; Raychev et al., 2015], and standard semantic formats like Program Dependence Graphs [Allamanis et al., 2018b], and abstracted traces [Henkel et al., 2018]. Then, a neural network model like RNN or its variants is trained on the representation to form distributed vectors.

We use LLVM IR [LLVM, 2018a] as the *base representation* for learning the embeddings in high dimensional space. To the best of our knowledge, we are the *first ones* to model the entities of the IR—Opcodes, Operands and Types—in the form of relationships and to use a translation based model [Bordes et al., 2013] to capture such multi-relational data in higher dimensions.

Applications In the earlier works, the training to generate embeddings was either application-specific or programming language-specific: Allamanis et al. [Allamanis et al., 2015a] propose a token-based neural probabilistic model for suggesting meaningful method names in Java; Cummins et al. [2017a] propose the DeepTune framework to create a distributed vector from the tokens obtained from code to solve the optimization problems like thread coarsening and device mapping in OpenCL; Alon et al. [Alon et al., 2019b] propose code2vec, a methodology to represent codes using information from the AST paths coupled with attention networks to determine the importance of a particular path to form the code vector for predicting the method names in Java; Mou et al. [2016] propose a tree-based CNN model to classify C++ programs; Gupta et al. [Gupta et al., 2017] propose a token-based multi-layer sequence to sequence model to fix common C program errors by students; Other applications like learning syntactic program fixes

from examples [Rolim et al., 2017], bug detection [Pradel and Sen, 2018; Wang et al., 2018] and program repair [Xiong et al., 2017] model the code as an embedding in a high dimensional space followed by using RNN like models to synthesize fixes. The survey by Allamanis et al. [2018a] covers more such application-specific approaches.

On the other hand, our approach is more generic, and both application and programming language independent. We show the effectiveness of our embeddings on two optimization tasks (device mapping and thread coarsening) in Section 3.8. We believe that the scope of our work can be extended beyond these applications, including program classification, code search, prediction of vectorization/unroll factors, etc.

Embedding techniques In encoding, entities are transformed into any numerical form amenable to learning, while in embedding, it is transformed to real-valued high dimensional vectors. Several attempts [Allamanis et al., 2015b; Ben-Nun et al., 2018; Vechev and Yahav, 2016] have been made to represent programs as distributed vectors in continuous space using word embedding techniques for diverse applications. Henkel et al. [2018] use the Word2Vec embedding model to generate the representation of a program from symbolic traces. They generate and expose embeddings for the program [Ben-Nun et al., 2018; Henkel et al., 2018], or the embeddings themselves become an implicit part of the training for the specific downstream task [Alon et al., 2019b; Pradel and Sen, 2018].

Our framework exposes a hierarchy of representations at the various levels of the program—Instruction, Function and Program level. Our approach is the first one to propose using seed embeddings. More significantly, we are the *first ones* to use program analysis (flow-analysis) driven approaches—*not* machine learning based approaches—to form the agglomerative vectors of programs beginning from the base seed encodings.

3.10.2 Similar works

The closest to our work is Neural Code Comprehensions (NCC) by Ben-Nun et al. [2018], who represent programs using LLVM IR. They use skip-gram model [Mikolov et al., 2013b] on conteXtual Flow Graph (XFG), which models the data/control flow of the program to represent IR. The skip-gram model is trained to generate embeddings for every IR instruction (inst2vec). So as to avoid Out Of Vocabulary (OOV) statements, they maintain a large vocabulary, one which uses large ($> 640M$) number of XFG statement pairs. A more thorough comparison of our work with NCC [Ben-Nun et al.,

2018] (along with DeepTune [Cummins et al., 2017a]) is given in Section 3.9.

Brauckmann et al. [2020] represent programs as Abstract Syntax Trees and annotate them with the control and data flow edges. A Graph Neural Network is used to learn the representation of the program in a supervised manner to solve the specific tasks of device mapping and thread coarsening. We achieve better performance on both the tasks, whereas they fail to show improvements in the prediction of the thread coarsening factor.

Another recent work is ProGraML [Cummins et al., 2021], which constructs a flow graph with data and control flow information from the Intermediate Representation of the program. They use inst2vec embeddings [Ben-Nun et al., 2018] to represent the nodes of the graph, and use a Gated Graph Neural Network to represent the program.

3.11 Summary

We proposed IR2Vec, a novel LLVM-IR based framework that can capture the implicit characteristics of the input programs in a task-independent manner. The seed embeddings were formed by modelling IR as relations, and the encoding was obtained by using a translational model. This encoding was combined with liveness, use-def, and reaching definition information, to form vectors at various levels of the program abstraction like instruction, function and module. Overall, this results in two encodings, which we term as *Symbolic* and *Flow-Aware*.

When compared to earlier approaches, our approach of representing programs is *non data-hungry*, takes *less training* time of up to 8640×, while maintaining a *small vocabulary* of *only 64 entities*. As we use entity level seed embeddings, we *do not* encounter any *OOV* issues. We demonstrate the effectiveness of the obtained encodings on two different tasks and obtain *superior* performance results while achieving *high* scalability when compared with various similar approaches.

The source code and other relevant material are available in https://compilers.cse.iith.ac.in/research/ir2vec.

Chapter 4

VexIR2Vec: VEX-IR based Binary Code Embeddings

4.1 Introduction

Building on the program embedding schema described in Chapter 2 on page 11, this chapter extends our approach to the domain of binary analysis. While IR2Vec (Chapter 3 on page 24) focuses on source-level LLVM IR, many practical applications require analyzing pre-compiled binaries where source code is unavailable.

Binary-level code analysis plays a crucial role in various domains ranging from reverse engineering, malware detection [Farhadi et al., 2014], and vulnerability analysis [Gao et al., 2018; Liu et al., 2020]. It is also crucial in other domains like binary translation, profile matching [Panchenko et al., 2019; Wang et al., 2000], code lifting [Martínez et al., 2023], and identifying IP and copyright infringements [Tang et al., 2018; Ming et al., 2016]. The applications of binary analysis in the domains of security and optimization tasks like profile matching can be modeled as similarity tasks. These tasks involve comparing two binaries or portion of binaries to determine their similarity.

Analysis of binary code is analogous to analyzing programs. However, analyzing the binary code for similarity is more challenging due to its low-level representation, which lacks explicit high-level abstractions like variables and functions. Additionally, two binaries compiled from the same source exhibit several differences in terms of the structure and organization depending on (1) the compiler used, (2) the compiler optimization level used, and (3) the target architecture. Moreover, there can be other deliberate code

manipulations that involve strategies for code obfuscation [Junod et al., 2015].

In recent times, ML-based approaches have become widely prevalent in addressing such challenges in binary similarity [Xu et al., 2017; Zuo et al., 2019; Ding et al., 2019; Duan et al., 2020; Massarelli et al., 2019; Pei et al., 2020; Yu et al., 2020] owing to their high success in recognizing patterns and generalizing them. Any such approach would involve disassembling the binary to assembly code [Ding et al., 2019; Massarelli et al., 2019; Pei et al., 2020], or to an Intermediate Representation [Chandramohan et al., 2016; David et al., 2017], or other Abstract Syntax Tree like representations [Yang et al., 2021] derived from the assembly.

Once the binary is lifted to the desired representation, it has to be represented in a numerical form as vectors so that it can be used as input to the underlying models. So, there exist feature-based and embeddings-based approaches. Feature-based approaches [Feng et al., 2016; Eschweiler et al., 2016] use predetermined features like the number of arithmetic instructions, loads/stores, basic blocks, etc., for representing binary snippets. While, embedding-based approaches [Ding et al., 2019; Duan et al., 2020; Massarelli et al., 2019; Wang et al., 2023a] rely on learning a distributed representation in a vector space for representing programs.

Limitations of previous work Much progress has been achieved in the domain of ML-based binary code similarity: recent works have demonstrated that ML-based approaches based on modern classification techniques are likely to outperform longstanding industrial anti-virus systems [Damásio et al., 2023; Salem et al., 2021; Sraw and Kumar, 2021; Li et al., 2024]. However, among the available binary similarity tools, we perceive a few limitations in terms of applicability, effectiveness, and usability:

1. **Architecture-Specific:** Many methods that rely on assembly code for binary similarity [Duan et al., 2020; Wang et al., 2023a] are tailored to specific architectures. As a result, these models struggle to generalize to binaries targeting new architectures, limiting their precision in cross-architecture analysis.

2. **Loss of Structural Context:** The existing approaches often fail to capture the shape and structural relationship within the Control-Flow Graph (CFG) of the binary due to reliance on instruction-level representations [Massarelli et al., 2019; Duan et al., 2020], or due to limitations of graph-based methods in handling large CFGs with thousands of basic blocks [Marcelli et al., 2022]. This loss of context impacts the accuracy and generalizability of the underlying approaches.

3. **Cluttered Intermediate Representations:** Intermediate Representations (IR), while designed to be architecture-neutral, often exhibit significant differences across the binaries compiled for different architectures and compilation settings. Disassemblers further exacerbate this problem with additional redundancies in instructions and memory/register accesses, thereby cluttering the IR and obscuring functional equivalence [Luo et al., 2023].

4. **Scalability:** Modern techniques, especially those leveraging large language models [Peng et al., 2021; Li et al., 2021; Wang et al., 2022a, 2023a; Zhu et al., 2023], face scalability challenges. These methods often require substantial computational resources during both training (e.g., weeks on GPU clusters) and inference, limiting their usability and making them impractical for time-sensitive or large-scale applications.

Proposed Approach To address these limitations, we present VexIR2Vec, an embedding approach that represents binary functions as distributed multi-dimensional vectors. VexIR2Vec incorporates five characteristics whose combined use is, to the best of our knowledge, novel:

1. **Architecture-Neutral:** The VexIR2Vec embedding of a binary is extracted from the VEX-IR intermediate representation, which is architecture neutral [Wang and Shoshitaishvili, 2017]. Thus, VexIR2Vec can compare binaries compiled to different targets without specialization (Section 4.2.1).

2. **Path-based Encodings:** VexIR2Vec appropriates the structural properties of functions by analyzing sequences of basic block paths, or *peepholes*, derived from CFGs via random walks. This approach emphasizes frequently connected and nested blocks (Section 4.3.1).

   Unlike traditional peephole optimizations that operate within a single basic block, our peepholes span multiple basic blocks, forming extended basic block sequences. However, the rewriting techniques we apply—such as strength reduction, canonicalization, and redundant load/store elimination—are inspired by classical peephole optimization principles.

3. **Normalizing Transformations:** VexIR2Vec applies *normalizing transforma-*

tions on peepholes by using rewriting rules inspired by compiler optimizations[1]. These transformations declutter the IR by removing irrelevant instructions and other syntactic details while preserving essential semantics[2]. We implement these normalizations on VEX-IR as a library called VexINE [3]. (Section 4.3.3).

4. **Scalable Embeddings:** The vocabulary of VexIR2Vec is learned from instruction opcodes and types, using representation learning techniques with simple feed-forward networks [Bengio et al., 2013; Wang et al., 2017] that avoids Out-Of-Vocabulary (OOV) issues common in other approaches [Marcelli et al., 2022; Zuo et al., 2019]. This learning methodology results in an infrastructure that is highly scalable (Section 4.4).

Key Idea

The intuition behind the design of VexIR2Vec is that decomposing two similar functions into a sufficiently large number of peepholes followed by normalization is likely to result in many of these peepholes having similar semantics with minimal obscurity. While normalization reduces disassembler-induced clutter, the underlying training process handles the critical task of identifying functional similarity, enabling the model to process cleaner inputs for more accurate analysis. The implementation of this intuition into an actual tool is able to address the limitations of previous works in terms of precision and scalability.

VexIR2Vec primarily reuses the principles of IR2Vec (Chapter 3) but extends them to the domain of binaries. While IR2Vec focuses on representing source code at different levels of the Intermediate Representation (IR) hierarchy, VexIR2Vec focuses on representing the IR of the binary functions as vectors. The encoding techniques used in VexIR2Vec are different from those used in IR2Vec. VexIR2Vec introduces a new encoding following the paths in the CFG. Path-aware encoding is more suitable for binary similarity for two reasons: (1) it is a cheaper way of capturing the structural relationships within the CFG given the large number of basic blocks in a binary function,

[1]The normalizations described in Section 4.3.3 include register optimizations, copy propagation, constant propagation, constant folding, redundancy elimination, and load-store elimination.

[2]These rules are applied to local sequences of instructions (the peepholes) without global code knowledge; hence, they are unsound. However, soundness is not important in this context for purpose of binary similarity.

[3]VexINE is intended to be pronounced similarly to 'vaccine', as VEX-seen. Just as a vaccine enhances the immunity, VexINE normalizes the IR, improving its quality for better embeddings.

and (2) it exposes a number of simple straightline paths that can be normalized effectively in a manner analogous to peephole optimizations, but to remove clutter and irrelevant details.

Contributions The following are our contributions:

- We introduce VexIR2Vec, a VEX-IR based, architecture-neutral embedding approach for binary functions.
- We propose a novel path-aware encoding that captures the structural relationships within the Control-Flow Graph (CFG) of binary functions.
- We introduce a new library called VexINE that applies normalizing transformations to the IR to remove clutter and irrelevant details.
- VexIR2Vec is scalable and avoids Out-Of-Vocabulary (OOV) issues, making it suitable for large-scale binary analysis tasks.

Organization The rest of this chapter is organized as follows: Section 4.2 provides the necessary background on VEX-IR and the motivation behind VexIR2Vec. Sections 4.3, 4.4 describe the approach used in VexIR2Vec. Section 4.5 presents the evaluation of the vocabulary of VexIR2Vec. In Section 4.6 we discuss the related works and present a comparison. Finally, Section 4.7 concludes the chapter.

4.2 Background

4.2.1 VEX-IR: The Intermediate Representation

The process of binary code analysis may optionally involve lifting the assembly code into some higher level program representation [Ding et al., 2019; Massarelli et al., 2019; Pei et al., 2020] such as Abstract Syntax Tree (AST) [Yang et al., 2021] or an Intermediate Representation (IR) [Chandramohan et al., 2016; David et al., 2017]. Once the binary is converted to the desired representation, different graph/hash matching techniques or ML approaches can be used to solve the underlying binary code analysis for the purpose of similarity.

Our approach uses VEX-IR, the IR used by Valgrind [Nethercote and Seward, 2007] and `angr` [Wang and Shoshitaishvili, 2017]. VEX-IR is derived from the assembly

Figure 4.1: VexIR2Vec uses `angr` for disassembling the binaries and obtaining the VEX-IR. Function embeddings are generated by training simple feed-forward neural networks.

code; however, it abstracts out many architecture-specific details, such as register names. Instead of using machine registers, instructions refer to variable names, which are in the static single assignment (SSA) form [Cytron et al., 1991; Rastello and Tichadou, 2022]. Thus, each variable name has only one definition site. VEX-IR presents another high-level characteristic: it is typed. Nevertheless, VEX-IR also preserves some machine-specific information, such as side-effects, pointer sizes, instruction flags, and calling conventions, for instance. As we explain in Section 4.2.3, we opted to work with VEX-IR because this format is architecture-neutral and open-source; hence, it is amenable to cross-architecture binary similarity analysis. Figure 4.1 shows how VEX-IR relates to the different phases of binary diffing described in the following sections.

Example 4.2.1:

Figures 4.2 (b) and (c) show two VEX-IR programs. These two programs implement the same sequence of three operations. They differ because they were produced out of different assembly codes. Each line of code represents an instruction in Figures 4.2 (a) and (d). Names such as `t26` represent variables. VEX-IR programs can use an unbounded surplus of these names. Indeed, the static single-assignment property ensures that each variable is defined with a new name. Memory addresses and registers, in contrast, do not follow this property. Thus, a register such as `r184` in Figure 4.2 (a) can be affected multiple times. The same is true for memory addresses such as `M1` or `M2`.

4.2.2 Challenges in Binary Similarity

The binary similarity problem is challenging for several reasons. The same source code might yield very different binaries due to the differences in compilers, optimization levels, target architectures, or obfuscations. Any one of these factors might change the instructions present in a program, as well as the control flow created by these instructions. This section discusses some of these challenges.

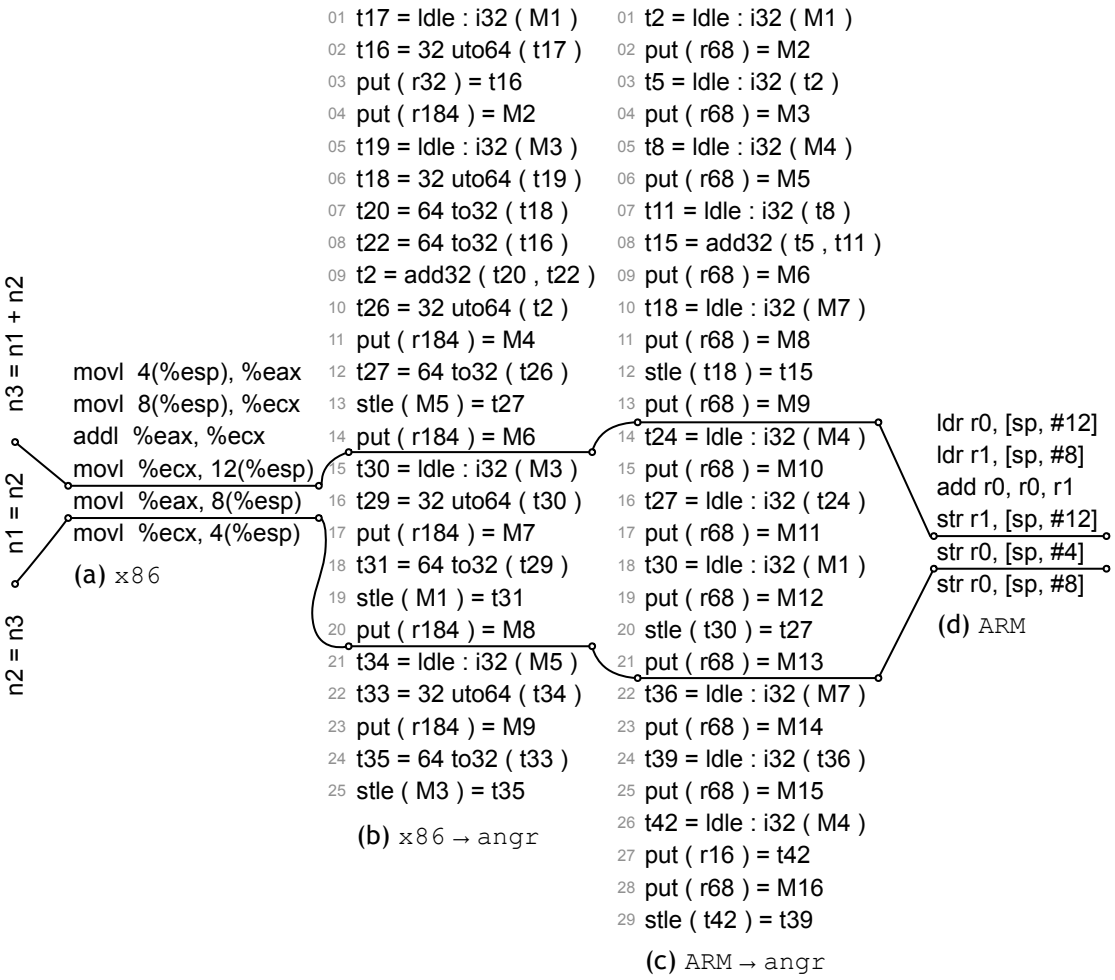


Figure 4.2: VEX-IR corresponding to the section of the program `n3 = n1 + n2; n1 = n2; n2 = n3;` to compute the Fibonacci series generated by `angr` from x86 and ARM binaries.

Compiler and Optimization Levels

It is well understood that different compilers generate different assemblies and binaries for the same source code. Such divergences are due to several factors, including the underlying optimizations and internal cost models used by these compilers at different stages of compilation. Moreover, even binaries produced by the same compiler out of the same source code can present substantial differences, depending on the optimization level used during code generation.

Structural Differences The control-flow graph of a program determines the possible paths through which execution can flow along that program. Many binary diffing tools rely on structural properties of control-flow graphs to compare programs [Karamitas and Kehagias, 2018; Bourquin et al., 2013; Xu et al., 2017; Li et al., 2021; Duan et al., 2020]. These tools first construct a CFG for each function in the binary, then compare them using graph isomorphism algorithms [Ullmann, 1976] or ML algorithms and models to identify similar or identical subgraphs. However, such algorithms leverage structural properties that can be significantly affected by how the binary code is produced. In particular, optimizations like loop unrolling, function inlining, and tail call elimination cause substantial changes to a program's CFG. Example 4.2.2 illustrates this issue by comparing control-flow graphs produced by GCC and Clang under different optimization levels.

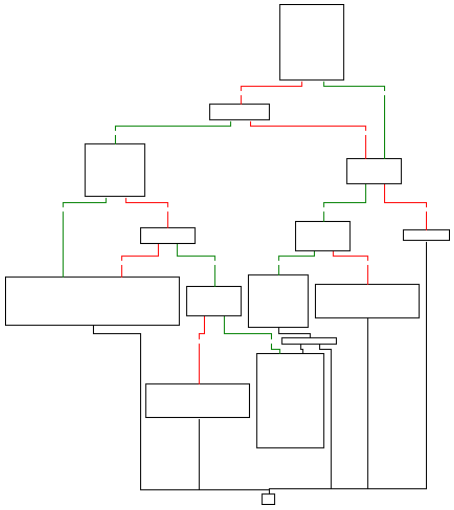
Example 4.2.2:

Figure 4.3 shows the CFGs corresponding to `adjust_relative_path` method of Binutils generated by GCC and Clang compilers at `-O0` and `-O3` optimization levels. The size of the blocks is proportional to the number of instructions they contain. As it can be observed, the topology of the CFGs, the number of basic blocks, and their size vary significantly across different optimizations and compilers.

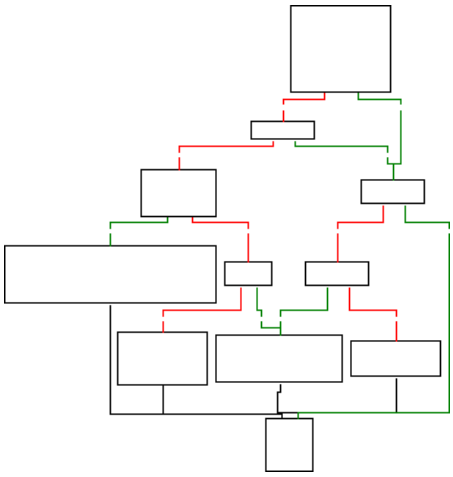
Instruction-Level Differences Many binary diffing tools use the hashing of code sequences to compare snippets of binary code [Jin et al., 2012; Pewny et al., 2015; David et al., 2017; Wang et al., 2000; Ayupov et al., 2024]. However, just like the structural properties previously mentioned, the instructions that make up the function also vary depending on how code is generated. Differences arise due to the heuristics used by the compilers to perform instruction selection and scheduling. In this regard, compilers reorder instructions differently across architectures to minimize pipeline stalls and improve execution efficiency. Furthermore, in architectures such as x86, the same operation can often be expressed in many ways. Such differences can be dramatic, as Example 4.2.3 demonstrates.

Example 4.2.3:

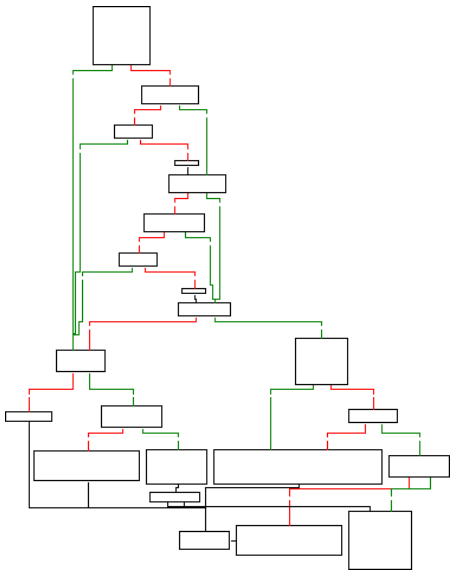
The heatmap in Figure 4.4 shows the frequency distribution of the x86 instructions in the binaries of FindUtils generated by Clang and GCC. The figure shows that these compilers use different instructions within and across optimization levels. For



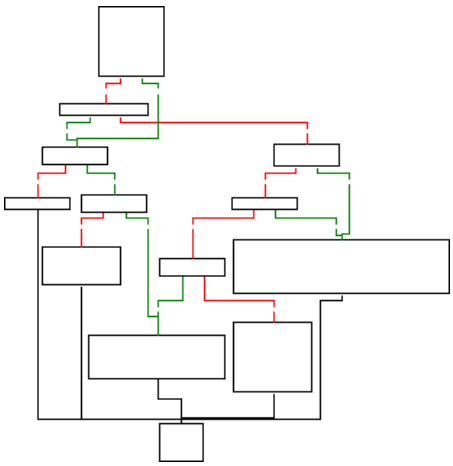
(a) x86 - GCC10 `-O0`



(b) x86 - GCC10 `-O3`



(c) x86 - Clang10 `-O0`



(d) x86 - Clang10 `-O3`

Figure 4.3: The CFG of `adjust_relative_path` method from `elfedit` of binutils project generated by GCC 10 and Clang 10 compilers under different optimizations.

instance, the `endbr64` instruction is emitted 4,682 times by GCC, whereas Clang generated this instruction only 71 times. Similarly, instructions like `inc` and `dec` are mostly emitted by both compilers when they optimize for code size at the `-Os` level.

Even among the vector instructions, Clang and GCC use different instructions as Figure 4.4 demonstrates.

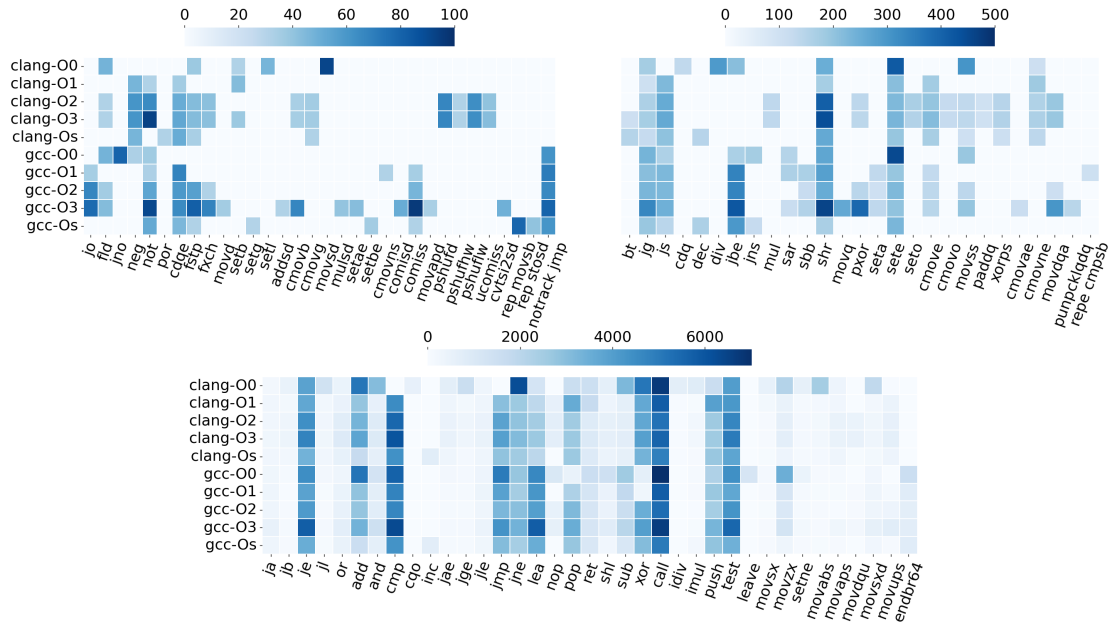


Figure 4.4: Heatmap showing the frequency distribution of the instructions across the binaries from FindUtils generated by Clang 10 and GCC 10 compilers with different optimization levels.

Target Architecture

The challenges in cross-architecture binary similarity stem from the differences in the syntax of assembly code and the structure of the induced CFG. Even when this target-specific assembly code is lifted to a common intermediate representation, the problem of matching them is still difficult because executables written in different assembly codes are unlikely to yield the same structure and instructions once converted to a common intermediate representation. Many factors might explain such differences:

1. Binaries compiled for different architectures may exhibit variations in memory layout, including the organization of data structures and the allocation of functions in memory.

2. Different architectures may use different byte orders (endianness) to store multi-byte data.

3. System calls and API functions can vary between architectures.

4. Binaries may dynamically link to different versions of system libraries.

5. Compilers for different architectures may optimize code differently.

Example 4.2.4:

Going back to Figure 4.2, on Page 64, we see the intermediate representation that `angr` generates for two binary programs that were compiled from the same source code: one of them runs in x86, the other in ARM. Noticeable differences between these intermediate representations include:

- **Redundant extensions/truncations:** Variable $t17$ in Figure 4.2(b) is extended from 32 to 64 bits in `L2`, followed by truncation back to 32 bits in `L8` before any use.

- **Redundant load-stores:** In `L13` of Figure 4.2(b), the variable `t27` is stored in the memory location $M5$, and in `L21` the same memory location is loaded to a new variable $t34$ and used before any write to the memory location. Such operations are typically introduced during the code generation phase of an SSA-based IR.

- **Special Register Updates:** The updates to special registers like instruction/stack pointer (`put` operations on $r184$ and $r68$ in Figure 4.2(b)/(c)) in a straight-line code.

- **Indirect Memory Accesses:** Indirect memory accesses arising out of architectural semantics like $t5$ in `L8` of Figure 4.2(c) vs. $t20$ in `L9` of Figure 4.2(b).

4.2.3 Normalized Instruction Traces to Minimize Code Differences

Our objective is to learn to represent the functions in the binary as embeddings, such that semantically similar ones are mapped onto vectors that are spatially close in a multi-dimensional Euclidean space. To achieve this goal, we aim to extract program embeddings from a representation that achieves the following properties:

Shape Sensitiveness: Instructions that run more often should be given higher importance. In other words, instructions that exist in confluence points (the post-dominators of branches) or within the cycles of the CFG should influence the embeddings more heavily.

Normalizable: The effect of redundant instructions as those seen in Example 4.2.4 should be discarded as much as possible. In this regard, the instructions should be *normalizable*, meaning that such redundancies should be pruned prior to the construction of the vector representation of programs.

Flow Insensitiveness: Recent findings have shown that embeddings that map binaries to vectors should be flow-insensitive to resist changes in the order of instructions [Damásio et al., 2023; Gorchakov et al., 2023]. This requirement is motivated by Example 4.2.2 which illustrates how the ordering of instructions and the control-flow graph of programs change once these programs are compiled to target different architectures.

To achieve these three goals, our approach builds from the *sequences of Basic blocks*, which we term as *peepholes*. Because this notion is fundamental to the presentation that follows, Definition 4.1 states it formally.

Definition 4.1: Peephole

Let $G = (V, E)$ be the control-flow graph of a program, where V is a set of *basic blocks*—instructions that always run in sequence. An edge $v_i \rightarrow v_j \in E$ denotes a branch from $v_i, v_j \in V$. A peephole is a list of k contiguous basic blocks that form a path in G.

4.3 Path-Aware Encoding: From Binaries to Normalized Peepholes

In the effort to map binaries to vectors using Path-Aware encoding, we first convert them to a set of peepholes, following Definition 4.1. In this section, we explain how we generate such peepholes. This process happens in multiple steps, starting with the VEX-IR representation of the binary and terminating with a collection of peepholes at the end of Phase I, as shown in Figure 10.1. Firstly, Section 4.3.1 shows how the peepholes

are generated from the VEX-IR representation of the functions. Secondly, Section 4.3.2 explains how peepholes are rewritten to abstract out the different syntax-oriented details that map to the same semantics in VEX-IR. Finally, Section 4.3.3 explains how this initial set of peepholes is further normalized via a number of transformations. These transformations, which are inspired by typical compiler optimizations, are applied by our VEX-IR Normalization Engine (VexINE). This final collection of peepholes is the input of Phase II (Figure 10.1).

4.3.1 Generating Peepholes

VEX-IR programs form *Control-Flow Graphs*: graph-like structures whose vertices denote basic blocks, and edges denote possible program flows. To effectively capture the structural properties of binary functions, we decompose these CFGs into straight-line sequences of basic blocks. Ideally, we would enumerate all possible execution paths through a function; however, this is computationally intractable for large CFGs with many branches. Instead, we approximate this enumeration using random walks, which sample representative paths while guaranteeing coverage of all basic blocks. With sufficient samples, the aggregate of these paths approximates the structure of the entire CFG. This approach forms the basis of our *Path-Aware* encoding.

> **Definition 4.2: Path-Aware Encodings**
>
> Path-Aware encoding captures structural relationships within the Control-Flow Graph (CFG) by aggregating embeddings along the program paths. Unlike Flow-Aware encodings (Definition 3.3) which propagate information through def-use chains, Path-Aware encoding samples paths via random walks on the CFG, producing peepholes (straight-line sequences of basic blocks) that approximate the control-flow structure. The embeddings of entities along these paths are combined to form the function representation.

The peepholes of Definition 4.1 are extracted from a control-flow graph via a random walk, which is implemented by Algorithm 3. Algorithm 3 translates a function F into a set of peepholes ($\mathcal{P}$), where each peephole $\pi \in \mathcal{P}$ is a sequence of a fixed number (k) of basic blocks. Algorithm 3 ensures that each basic block in the CFG is *covered* at least c times across $\mathcal{P}$. As Theorem 4.1 demonstrates, the expected running time of this algorithm is proportional to the product of this parameter, c, and the number of basic blocks in the program.

Algorithm 3: Generating peepholes via random walks on the control-flow graph

Inputs

- $G = (V, E)$: Control-flow graph
- k: Maximum length of a peephole
- c: Minimum number of visits per basic block

Output

- $\mathcal{P}$: Set of peepholes

Variables

- U: worklist of basic blocks from which the starting block is selected. Initially, $U = V$
- *count*: Number of visits per basic block. Initially, $count[v] = 0$ for every $v \in V$

while $|U| > 0$ **do**
  $v :=$ randomly sampled start block from U
  $\pi :=$ random path of length k, starting at v
  $\mathcal{P}$.append(π)
  for *each block* $b \in \pi$ **do**
    $count[b] := count[b] + 1$
  $U := \{\exists\ v \in V : count[v] < c\}$
return $\mathcal{P}$

Theorem 4.1. *If Algorithm 3 is invoked on a VEX IR function with $|V|$ basic blocks in the control-flow graph and parameters k and c, then it terminates in at most $c|V|$ steps.*

Proof. The worklist U, from which the start basic block is selected at each iteration, is updated to include only the basic blocks $v \in V$ that satisfy the condition $count[v] < c$. This ensures that at least one basic block in π that was not visited c times earlier is visited in each iteration. Hence, there is a monotonic decrease in the size of U, due to which the algorithm is guaranteed to terminate. This gives us a worst-case bound of $c|V|$ on the number of steps needed. The worst case occurs when the start basic block is the only one in π that is also in U (that has not been visited c times previously). □

Observation 4.2. *From our experiments, we note that the expected number of peepholes generated by using Algorithm 3 is a value close to $c|V|/2$.*

Rationale

In practice (from the experiments shown in Appendix D.2), we observe that as the value of k (maximum length of the peephole) increases, the number of peepholes needed tends to a value close to $c|V|/2$ (as opposed to the worst case $c|V|$). Intuitively, this means that in every step, roughly one more basic block from U is covered apart from the start basic block.

Below, we give an intuitive explanation of this:

- The average ratio of the number of edges and the number of blocks from the CFGs used in our dataset is around 1.3-1.4. This implies that the graph is sparse and has the structure of a directed tree with some additional edges.
- We also observe that the length of the longest path is about one-third of the number of basic blocks. If the randomly chosen start vertex of the peephole belongs to the first half of the longest path, then we are assured to hit at least 1/6 fraction of the basic blocks (assuming the peephole length k is long enough). The probability of hitting the first half of a particular longest path is again 1/6. Hence, with probability 1/6, we hit at least $|V|/6$ many basic blocks in addition to the starting block. This alone adds a contribution of $|V|/36$ (excluding the starting block) to the expected number of blocks to be covered.
- If the starting block is not a leaf in $G[U]$ (induced graph on the remaining basic blocks), we are assured that there will be at least one more block that will be covered.

In our experiments, we set $c = 2$, and hence, the expected number of peepholes generated is a value close to $|V|$.

The final VexIR2Vec embeddings are derived from these peepholes. By design, the instructions that occur in different peepholes will have higher contributions to the final VexIR2Vec vector—a direct consequence that such instructions occur more often in different *execution-contexts*. This observation motivates the design of Algorithm 3, which builds peepholes via random walks. Thus, the basic blocks that are more connected in the function's control-flow graph are likely to appear in more peepholes. Therefore, they tend to bear more weight on the final vector that represents a function. It can be seen that the number of peepholes produced by Algorithm 3 might vary for the same function across different runs due to the randomness induced by the algorithm. However, the expected number of peepholes can be derived from the parameters c and k, as Theorem 4.1 (and Observation 4.2) shows.

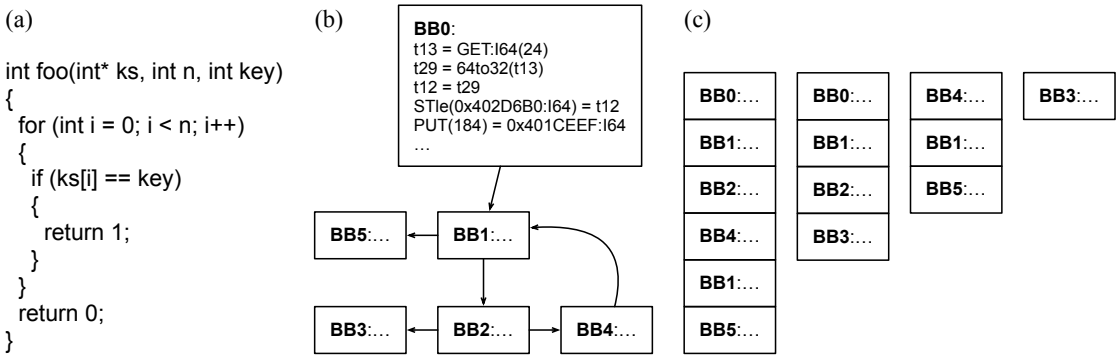


Figure 4.5: (a) Program written in C. (b) VEX-IR representation of the program, as obtained from an x86 binary. (c) Peepholes produced by Algorithm 3, using $k = 6$ and $c = 2$.

Example 4.3.1:

Figure 4.5 illustrates how peepholes are produced. This example assumes that Algorithm 3 is invoked on the function shown in Figure 4.5 (a), with $k = 6$ and $c = 2$. The peepholes are the sequences of basic blocks seen in Figure 4.5 (c). These peepholes are generated via a random walk on the graph that Figure 4.5 (b) shows. Notice that the same basic block might appear in different peepholes. Basic blocks can also appear multiple times in the same peephole if the control-flow graph contains loops. Thus, highly connected blocks, or blocks in loops, such as `BB1` in Figure 4.5 (b), are likely to contribute more instances to the peepholes.

4.3.2 Canonicalization of Peepholes

As seen in Example 4.2.4, VEX-IR is a rather prodigal format. It contains about 1095 opcodes and 18 types of operands [Shoshitaishvili et al., 2015]. Each opcode is specialized for the type and the bit-width of the operation that it implements. As an illustration, VEX-IR contains about 100 different ways to write an addition. This diversity is due to, among other things, types (integer, floating point, etc) and the size of the operands (8, 16, 32, and 64 bits). For example, the ARM instruction `adds R2, R2, #8` becomes a sequence of five operations: `t0 = GET:I32(16); t1 = 0x8:I32; t3 = Add32(t0,t1); PUT(16) = t3; PUT(68) = 0x59FC8:I32`, once converted into VEX-IR.

Here, vectors derived from VEX-IR instructions form the vocabulary and are used to compare binaries. The syntactic diversity of instructions that implement the same semantics complicates the generation of this vocabulary. Ideally, each different syntax should yield different semantics. Thus, to approximate this goal, we canonicalize the VEX-IR representation of a program. This preprocessing step follows a number of simple rewriting rules, which we enumerate as follows:

1. Opcodes representing the same semantics are replaced with a single opcode. For instance, we replace opcodes such as `Add8`, `Add16` and `Add32` with just `Add`.

2. The bitwidths and the endianness of the operands are masked out. For instance, we remove casts such as `32uto64` from the instructions.

3. Operations that use negative immediate values are converted to operations that use positive constants. For instance, `add(-1, t20)` is replaced with `sub(+1, t20)`.

4. Types are normalized into four primitive types, namely: Integer, Float, Double, and Vector.

5. Following the third and fourth rewriting rules above, all the constants, variables, and registers are abstracted using a corresponding generic representation.

6. Instructions involving indirect memory accesses are replaced with direct memory accesses.

The process of canonicalization of VEX-IR is analogous to the *normalization* step performed in Natural Language Processing [Jurafsky, 2000]. Canonicalization reduces

the number of unique entities to learn, in turn facilitating effective learning. Example 4.3.2 illustrates this preprocessing.

Example 4.3.2:

The canonicalized version corresponding to the example IR shown in Figure 4.2(b) is shown in Figure 4.6(a). Notice that the little-endian opcodes `ldle/stle` are generically represented as `load/store` while masking out the endianness and bitwidths.

The replacement of indirect memory accesses with direct accesses breaks architectural semantics. However, memory locations are masked out in the process of mapping peepholes to vectors. Thus, for our purposes, this kind of replacement is acceptable. Example 4.3.3 explains how this transformation is carried out.

Example 4.3.3:

The VEX-IR program generated from the ARM binary seen in Figure 4.2(c) shows several indirect accesses. For instance, the operands $t5$ and $t11$ of the `add` instruction in `L8`, are derived from $M1 \rightarrow t2 \rightarrow t5$ and $M4 \rightarrow t8 \rightarrow t11$ respectively. We replace such accesses by removing additional loads by introducing new memory addresses M_x and M_y as shown in Figure 4.8 (b).

4.3.3 Normalization of Peepholes by VexINE

Two IRs generated from binaries corresponding to the same source code might be vastly different due to compilation configurations, as explained in Section 7.2. Additionally, the process of disassembling and lifting from the binary to VEX-IR can also introduce additional redundant instructions that are not of importance for binary similarity. Thus, to get rid of such *uninteresting* instructions, we normalize the IR via a catalog of transformations that this section discusses.

Definition 4.3: VexINE: VEX-IR Normalization Engine

VexINE is a normalization engine that applies compiler-inspired transformations (copy propagation, constant folding, dead code elimination, load-store optimization) to VEX-IR peepholes, reducing syntactic variations while preserving semantic equivalence for binary similarity analysis.

These transformations follow conventional compiler optimizations [Muchnick, 1997]. We perform normalization directly on peepholes, not on functions. This approach sim-

plifies the implementation of the transformations because the peepholes are straight-line codes. The normalizations are implemented as part of our VEX-IR Normalization Engine (VexINE) module of VexIR2Vec. The VexINE module takes a peephole as input and produces a new *normalized* version of it. The implementation of VexINE relies on two principles:

1. Values used in a peephole without being defined within it are assumed to be parameters of the peephole and must be preserved.

2. Values defined within a peephole but not used within it are considered dead and can be eliminated.

```
1  t17 = load:i32(M1)
2  t16 = t17
3  put(r32) = t16
4  put(r184) = M2
5  t19 = load:i32(M3)
6  t18 = t19
7  t20 = t18
8  t22 = t16
9  t2 = add(t20,t22)
10 t26 = t2
11 put(r184) = M4
12 t27 = t26
13 store(M5) = t27
14 put(r184) = M6
15 t30 = load:i32(M3)
16 t29 = t30
17 put(r184) = M7
18 t31 = t29
19 store(M1) = t31
20 put(r184) = M8
21 t34 = load:i32(M5)
22 t33 = t34
23 put(r184) = M9
24 t35 = t33
25 store(M3) = t35
```

(a) Canonicalized IR

```
1  t17 = load:i32(M1)
2  t16 = t17
3  r32 = t16

5  t19 = load:i32(M3)
6  t18 = t19
7  t20 = t18
8  t22 = t16
9  t2 = add(t20,t22)
10 t26 = t2

12 t27 = t26
13 store(M5) = t27

15 t30 = load:i32(M3)
16 t29 = t30

18 t31 = t29
19 store(M1) = t31

21 t34 = load:i32(M5)
22 t33 = t34
23 r184 = M9
24 t35 = t33
25 store(M3) = t35
```

(b) After Register Optimizations

```
1  t17 = load:i32(M1)

3  r32 = t17

5  t19 = load:i32(M3)

9  t2 = add(t19,t17)

13 store(M5) = t2

15 t30 = load:i32(M3)

19 store(M1) = t30

21 t34 = load:i32(M5)

23 r184 = M9

25 store(M3) = t34
```

(c) After Copy and Const prop

Figure 4.6: Steps showing the normalizations performed by VexINE on the example shown in Figure 4.2(a) for binary similarity.

The effects produced by VexINE have two characteristics:

Local: Optimizations affect only one occurrence of an instruction, even if this instruction appears in multiple peepholes. In other words, the elimination of an instruc-

tion from a peephole bears no effect on the existence of this instruction within other peepholes.

Unsound: Due to the local nature of the optimizations, they are unsound. In other words, we remove instructions from a peephole without worrying if these instructions could be used outside the peephole.

In the rest of this section, we describe different normalizations that VEXINE performs on the canonicalized peepholes.

```
1  t17 = load:i32(M1)
3  r32 = t17
5  t19 = load:i32(M3)
9  t2 = add(t19,t17)
13 store(M5) = t2
15 t30 = load:i32(M3)
19 store(M1) = t30
21 t34 = load:i32(M5)
23 r184 = M9
25 store(M3) = t34
```

(c) After Copy and Const prop

```
1  t17 = load:i32(M1)
3  r32 = t17
5  t19 = load:i32(M3)
9  t2 = add(t19,t17)
13 store(M5) = t2
19 store(M1) = t19
21 t34 = load:i32(M5)
23 r184 = M9
25 store(M3) = t34
```

(d) After Common Subexp Elim

```
1  t17 = load:i32(M1)
3  r32 = t17
5  t19 = load:i32(M3)
9  t2 = add(t19,t17)
19 store(M1) = t19
23 r184 = M9
25 store(M3) = t2
```

(e) After Load-Store elim

Figure 4.7: (Contd. from Figure 4.6) Steps showing the normalizations performed by VEXINE on the example shown in Figure 4.2(a) for binary similarity.

Register Promotion: Memory addresses accessed via the `put` and `get` instructions are promoted to registers. Notice that this transformation breaks the static single-assignment property of VEX-IR. Consequently, a reaching-definition analysis becomes necessary to implement the transformations that follow. It can be seen that the register promotion exposes redundant, useless writes to the same register.

Redundant Write Elimination: Redundant writes to the same register are removed. This transformation, guided by a reaching-definitions analysis, is useful in reducing the number of updates to special registers like instruction and stack pointers.

Copy Propagation: Redundant copy instructions are eliminated after a reaching-definition analysis by replacing copies with their sources. As an example, Lines 6 and 7 in Figure 4.6 (b) are replaced with a single copy.

Constant Propagation and Folding: Variables initialized with constants are eliminated, and the constants are written directly at their use sites. Expressions that use only constants are replaced with the evaluated result.

Common Subexpression Elimination: Multiple evaluations of the same expression are replaced with the variable that holds the first computation. This transformation is guided by an available-expression analysis to ensure that the value computed by an expression does not change in between computations. Example 4.3.4 provides more details about this transformation.

Load-Store Elimination: Redundant store instructions in pairs of load and store operations are eliminated. A pair `t = load(M); store(M) = t` is deemed redundant if `t` is not updated in between the load and the store. This transformation is guided by a reaching-definition analysis, as the representation is no longer in the static single-assignment format.

Store-Store Elimination: Two consecutive stores to the same memory location without any reads in between cause the first store instruction to be removed.

Example 4.3.4:

Figures 4.6 and 4.7 show the successive effects of different normalizations. First, register promotion and redundant write elimination removes multiple instances of `put` instructions; hence, converting Figure 4.6 (a) into Figure 4.6 (b). Only the last effect of such instructions, in Line `L23`, is preserved. Secondly, a round of copy propagation maps Figure 4.6 (b) onto Figure 4.6 (c). Chained sequences of assignments are eliminated, and definitions are directly used where they are needed. As an example, variable `t17`, defined in Line `L1`, is directly used in Line `L9`. Finally, Figure 4.7 (d) illustrates a usage of common subexpression elimination. The redundant `load` on `M3` in `L15` is eliminated, and the use of `t30` in `L19` is replaced with `t19` defined in `L5`.

The combination of canonicalization rules and normalizations substantially simplifies the instructions in the peepholes that are used to produce vectors. The resultant codes, even when obtained from different instruction sets, tend to be more similar.

```
1  t17 = load:i32(M1)
2  r32 = t17
3  t19 = load:i32(M3)
4  t2 = add(t19,t17)
5  store(M1) = t19
6  r184 = M9
7  store(M3) = t2
```

(a) x86

```
1  t5 = load:i32(Mx)
2  t11 = load:i32(My)
3  t15 = add(t5,t11)
4  store(Mz) = t15
5  r68 = Mp
6  store(Mx) = t11
```

(b) ARM

Figure 4.8: Normalized VEX-IR generated by VEXINE for the example shown in Figure 4.2.

Example 4.3.5:

Figure 4.8 shows the x86 and the ARM versions of the peepholes earlier seen in Figure 4.2. As it can be seen, the syntactic gap between these two programs is substantially shorter than the syntactic difference between the two peepholes seen in Figure 4.2.

4.4 Path-Aware Encoding: From Peepholes to Embeddings

The peepholes obtained from the Path-Aware extraction (Section 4.3) are projected onto an n-dimensional Euclidean space as a vector of real numbers.

As described earlier (Section 2.4 on page 19), learning to represent functions as embeddings involves two steps: a one-time vocabulary learning step followed by the embedding generation step. The first involves learning an initial *task-independent* seed embedding vocabulary ($\mathcal{V}_{lookup}$) in an *unsupervised* manner. The second step involves mapping the normalized peepholes to vectors using the vocabulary $\mathcal{V}_{lookup}$.

4.4.1 Learning the Seed Embedding Vocabulary

Mapping VEX-IR as Entities and Relations

After normalization, the variables and constants are masked out with abstract placeholder tokens such as VAR and CONST. We decompose the normalized VEX-IR instructions into entities and relations. Such code entities, in turn, are features that characterize program instructions. Each entity is represented as a triplet formed by an *opcode*, a *kind*, and a *value*. In this chapter, we recognize three *kinds* of code entities, which we list as follows:

Arg: the entity describes the argument of an instruction; e.g., $\langle add, \mathbf{Arg}_1, VAR \rangle$ indicates that the first argument of an addition operation is a variable (instead of a constant or a memory address, for instance).

Type: the entity describes the type of instruction; e.g., $\langle add, \mathbf{Type}, INT \rangle$ indicates that an addition instruction operates on integer values.

Next: the entity describes the successor relation between instructions; e.g.: $\langle add, \mathbf{Next}, store \rangle$ indicates that a store instruction follows the add instruction.

These three categories capture the relationships between two entities at a time, resulting in $\langle h, r, t \rangle$ triplets. Such triplets are collected over a corpus of VEX-IR functions and are fed as an input to an ML model to learn the embeddings of the entities in the corpus. Such representation of entities results in a vocabulary.

> **Example 4.4.1:**
>
> Figure 4.9 shows the code entities derived from two instructions. As it can be observed, each instruction is decomposed into multiple $\langle h, r, t \rangle$ triplets using opcodes, kind (type, arg or next), and values. The decomposition captures the possible relationships defined by *kind*. For instance, entities of kind "type" capture the relation between an operation (ex.: a load or addition) and the type of the value that the operation manipulates (ex.: an integer).

Learning Seed Embedding Vocabulary

Once the entities and relations are identified, we learn the embeddings of these entities using the TransE model as described in Section 2.3 on page 17. The model learns the relationship $[\![\mathbf{h}]\!]+[\![\mathbf{r}]\!] \approx [\![\mathbf{t}]\!]$, by minimizing the energy function $E(h, r, t) = \|[\![\mathbf{h}]\!] + [\![\mathbf{r}]\!] - [\![\mathbf{t}]\!]\|_2^2$.

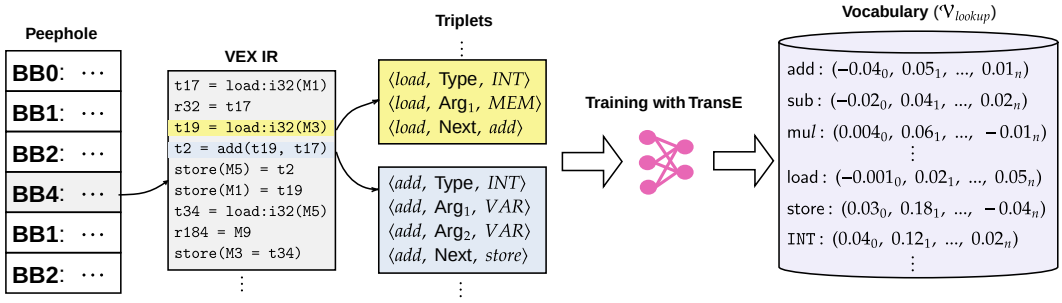


Figure 4.9: VEX-IR instructions from a corpus of binaries are decomposed into entities and relations to obtain triplets. These triplets are used to train the TransE model to obtain a vocabulary. The vocabulary contains the n dimensional representations of each of the entities of VEX-IR in the train set.

This training process results in a vocabulary $\mathcal{V}_{lookup}$ containing the representation of each entity of VEX-IR in an n-d Euclidean space $\mathbb{R}^n$.

4.4.2 Mapping Functions to Vectors

As described in Section 4.3, each program function is decomposed into a set of peepholes after Algorithm 3. Each peephole represents a sequence of basic blocks. Basic blocks are sequences of VEX-IR instructions. Each one of these instructions can be decomposed into the following entities: Opcode, Type, and Arguments. Thus, to construct the VEX-IR representation of a function, we combine the representation of these entities in the function's peepholes. As seen in Section 4.4, the representation of each entity is obtained using the $\mathcal{V}_{lookup}$ map.

If a VEX-IR instruction l is of format $[O^{(l)}T^{(l)}A_1^{(l)}A_2^{(l)}\cdots A_n^{(l)}]$, where $O^{(l)}$, $T^{(l)}$ and $A_i^{(l)}$ represent the opcode, type, and its i^{th} argument, then its representation is obtained as

$$[\![\mathbf{O}^{(l)}]\!] = \mathcal{V}_{lookup}(O^{(l)}), [\![\mathbf{T}^{(l)}]\!] = \mathcal{V}_{lookup}(T^{(l)}), [\![\mathbf{A_i}^{(l)}]\!] = \mathcal{V}_{lookup}(A_i^{(l)}).$$

The instruction is embedded as a combination of the embeddings of these entities that form it:

$$\left\langle [\![\mathbf{O}^{(l)}]\!], [\![\mathbf{T}^{(l)}]\!], [\![\mathbf{A}_1^{(l)}]\!] + [\![\mathbf{A}_2^{(l)}]\!] + \cdots + [\![\mathbf{A}_n^{(l)}]\!] \right\rangle \tag{4.1}$$

If the function F is decomposed into a set of normalized peepholes $\mathcal{P} = \{\pi_1, \pi_2, \ldots\}$, the representation of a peephole $\pi \in \mathcal{P}$ containing a set of instructions I^π is computed

as the pointwise addition of the representation of individual entities of its instructions in I^π.

$$[\![\pi]\!] = \left\langle \sum_{I \in I^\pi} [\![\mathbf{O}^I]\!], \sum_{I \in I^\pi} [\![\mathbf{T}^I]\!], \sum_{I \in I^\pi} [\![\mathbf{A}^I]\!] \right\rangle = \langle [\![\mathbf{O}^\pi]\!], [\![\mathbf{T}^\pi]\!], [\![\mathbf{A}^\pi]\!] \rangle \tag{4.2}$$

The representation $[\![\mathbf{F}^\nu]\!]$ of a function F is computed by summing the representations of entities across different peepholes:

$$[\![\mathbf{F}^\nu]\!] = \left\langle \sum_{\pi \in \mathcal{P}} [\![\mathbf{O}^\pi]\!], \sum_{\pi \in \mathcal{P}} [\![\mathbf{T}^\pi]\!], \sum_{\pi \in \mathcal{P}} [\![\mathbf{A}^\pi]\!] \right\rangle \tag{4.3}$$

Therefore, $[\![\mathbf{F}^\nu]\!]$, the embedding vector representing function F, is produced as a linear combination of the embeddings of the peepholes produced via Algorithm 3. This process of mapping functions to embeddings, parameterized by the $\mathcal{V}_{lookup}$ function, is summarized in Algorithm 4.

Algorithm 4: Mapping functions to vectors.

Inputs

- $\mathcal{P}$: set of normalized peepholes of the function F

Output

- $[\![\mathbf{F}^\nu]\!] = \langle [\![\mathbf{O}]\!], [\![\mathbf{T}]\!], [\![\mathbf{A}]\!] \rangle$: A tuple of n dimensional vectors corresponding to opcodes, types and arguments.

$[\![\mathbf{O}]\!], [\![\mathbf{T}]\!], [\![\mathbf{A}]\!] = \vec{0}$ // Initialize vectors to zero
for *peephole* $\pi \in \mathcal{P}$ **do**
  for *instruction* $l \in I_\pi$ **do**
    $[\![\mathbf{O}]\!] = [\![\mathbf{O}]\!] + \mathcal{V}_{lookup}(O^{(l)})$
    $[\![\mathbf{T}]\!] = [\![\mathbf{T}]\!] + \mathcal{V}_{lookup}(T^{(l)})$
    for *Arg* $A_i^{(l)} \in [A_1^{(l)} A_2^{(l)} \cdots A_n^{(l)}]$ **do**
      $[\![\mathbf{A}]\!] = [\![\mathbf{A}]\!] + \mathcal{V}_{lookup}(A_i^{(l)})$
return $\langle [\![\mathbf{O}]\!], [\![\mathbf{T}]\!], [\![\mathbf{A}]\!] \rangle$ as $[\![\mathbf{F}^\nu]\!]$

Function calls We also consider the function calls while obtaining the function vector ($[\![\mathbf{F}^\nu]\!]$). We compute the call graph of the binary. For each function call, the embedding of the callee is first computed and is used in the call site in place of the call instruction. This way of substituting the callee embedding in the call site has a similar effect to that

of the inlining optimization performed during the compilation process. This approach also provides additional information while handling wrapper functions. We follow the normal embedding process for representing call instructions if the definition of the callee is not available in the current binary.

4.5 Evaluation of Vocabulary

Our vocabulary is the set of 128-dimensional vectors that represent the entities that form the program's intermediate format: opcodes, types, and operands. As explained in Section 4.4.2, this vocabulary is encoded as a function $\mathcal{V}_{lookup}$, which maps entities to vectors. This section presents different experiments—essentially of a qualitative nature—that we have engineered to study the quality of $\mathcal{V}_{lookup}$ to encode meaningful semantic information.

Clustering

$\mathcal{V}_{lookup}$ encodes semantic information as points within a Euclidean Space. Thus, entities that are semantically related tend to be mapped to points that are close in this space, as illustrated in Figure 2.2 (Page 19). To provide some evidence that this semantic information is correctly encoded in $\mathcal{V}_{lookup}$, we use t-SNE [van der Maaten and Hinton, 2008] to project the 128-D entity vectors onto a bi-dimensional surface. Figure 4.10 shows the resulting image. For easier interpretation, we mark entity vectors into nine logical groups. These groups are either based on the type of the data they operate on (integers, floating-point numbers, vectors), or on the kind of operations (logic operations, loads, stores, comparisons, max/min). We also have a group for entities to denote user-defined symbols: variables, functions, constants, etc.

Figure 4.10 shows the distinct formation of smaller clusters in the case of integer types, float types, and store types. The cluster of logical types containing operations such as `xor` are closer to the integer types. This proximity comes from the fact that logical operations actuate on integer-type variables. Similar observations can also be made for some of the vector types lying in closer proximity to some of the Ext types, such as `truncv`.

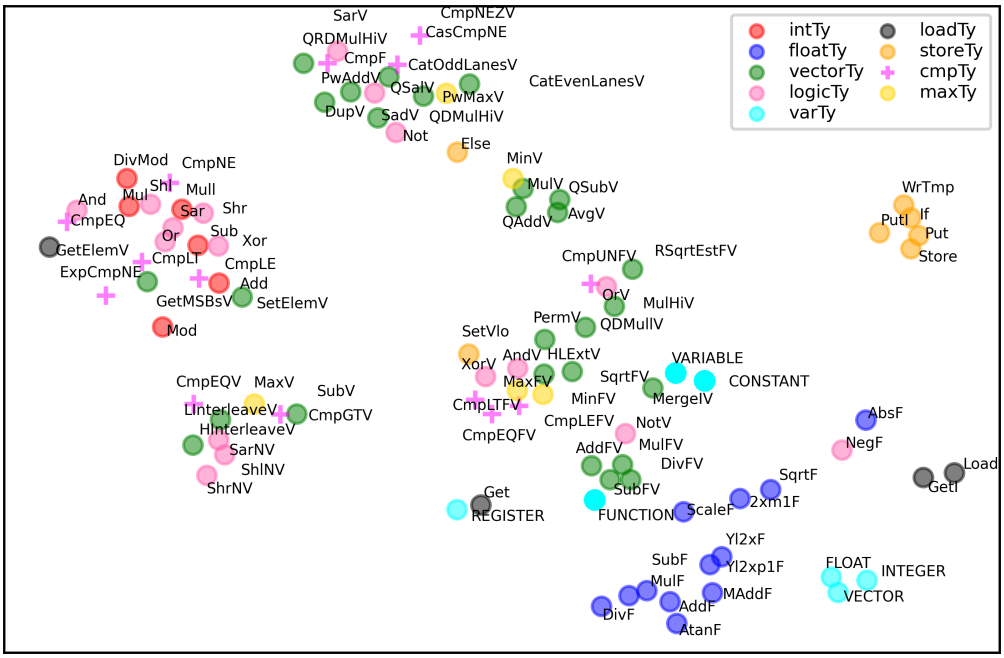


Figure 4.10: Vocabulary Clusters

> **Takeaway 4.1: Architecture-Neutral Semantic Clusters**
>
> *t-SNE projections of* VexIR2Vec *vocabulary reveal coherent clusters: integer types, float types, logical operations, and stores form distinct groups, confirming that VEX IR preserves semantic structure across architectures.*

Analogies

An *analogy* in a machine-learning embedding is a relationship between vectors that reflects a similar relationship between the entities they represent, often expressed in the form: "*a is to b as c is to d.*" VexIR2Vec lets us explore several kinds of analogies. For instance, an analogy such as "`get` is to `put` as `load` is to `store`" explores the similarity between different semantics of data movement instructions related to register and memory accesses. In order to probe VexIR2Vec's capacity to build meaningful analogies, we have designed 90 analogy queries to cover relations among operators, types, arguments, and their semantics. We represent an analogy query as $a : b :: c : ?$, meaning "*if a is to b, then c is to which entity?*" To answer the query, the missing value is computed as the entity the closest to $b - a + c$ by Euclidean distance.

Appendix B.1 gives the full list of 90 queries. Table 4.1 lists some of the analogies

Table 4.1: Sample Instruction Analogies

Syntactic	Semantic
getI : putI :: get : put	shl : mul :: shr : divmod
subf : addf :: subfv : addfv	mul : divmod :: and : or
add : addf :: sub : subf	get : put :: load : store
ext : integer :: extf : float	maxfv : minfv :: addfv : subfv
orv : vector :: or : integer	get : register :: geti : constant

that were correctly captured by $\mathcal{V}_{lookup}$. Notably, the vocabulary is able to capture intrinsic semantic information to capture that multiplication and division operations can be achieved using left and right shift operations, respectively. Other relations that characterize the type of operand for each operation are also captured. In this exercise, we observed the expected query result for 63 out of 90 queries, totaling an accuracy of 70%.

> **Takeaway 4.2: Cross-Architecture Semantic Preservation**
>
> *The* VexIR2Vec *vocabulary correctly answers 70% of semantic analogy queries (e.g.,* `shl:mul::shr:divmod`*), confirming that architecture-neutral VEX IR preserves the semantic relationships needed for cross-architecture binary analysis.*

Number of Dimensions

The queries explored in Section 4.5 gave us enough data to fine-tune hyperparameters (learning rate, batch size, and margin) besides the number of dimensions used in the representation of entities. As shown in Figure 4.11, we observe an increase in the accuracy of correctly answering analogies with the increase in the number of dimensions until 128. Further increase in the number of dimensions resulted in a fall in accuracy.

OOV study

The performance of supervised learning approaches typically depends on the quality and comprehensiveness of the training data. As discussed in Section 3.9.2 on page 53, when the training data does not cover the entire vocabulary of the language, unseen words can occur at test time, leading to Out-Of-Vocabulary (OOV) issues. These OOV

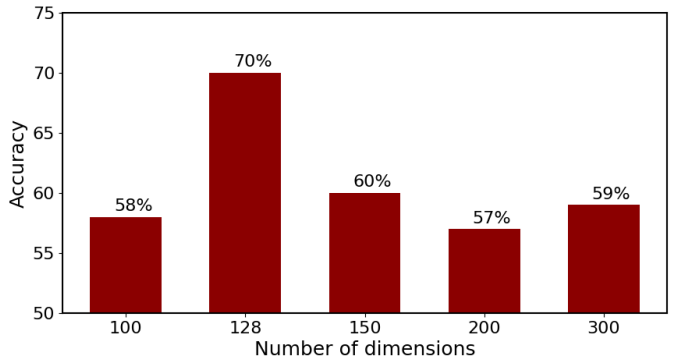


Figure 4.11: Accuracy vs Dimensions

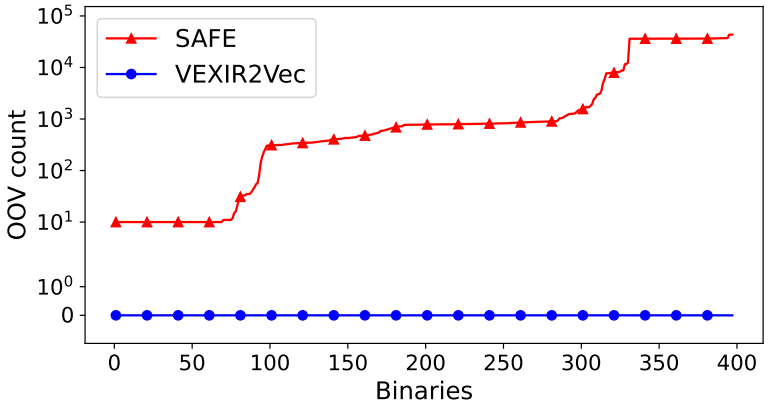


Figure 4.12: OOV Occurrences

words, which are not seen by the model during training, create challenges in accurately predicting and understanding the input binaries during inference.

The approaches that do not model input properly suffer from serious OOV issues. Similar to our earlier discussion on OOV in Section 3.9.2 on page 53, the approaches on binary code that learn the representations at the instruction or basic block level encounter a higher number of OOV as there could be a very high number of combinations of opcodes (O), types (T), and the number of arguments (A). Consequently, binary similarity approaches like SAFE [Zuo et al., 2019] and InnerEye [Zuo et al., 2019] encounter OOV issues. To avoid OOV issues, such approaches should use a training set comprising all possible combinations of opcodes, types, and arguments, leading to an enormous training space of $|O| \times |T| \times |A|$. Covering this huge space is highly infeasible.

In contrast, VexIR2Vec learns the representations at the entity level, reducing the

training space to $|O| + |T| + |A|$. These entities exist in limited numbers and can be quickly learned. By covering all possible opcodes, types, and arguments individually, it is easier to avoid OOV issues with a limited training set in VexIR2Vec.

Figure 4.12 shows the number of OOV occurrences encountered by SAFE and VexIR2Vec during inference on a collection of 400 randomly chosen binaries. It can be observed that VexIR2Vec did not face any OOV issues. SAFE, in contrast, encountered a large number of ($\approx 10^5$) OOV words. This study does not consider the implementations of OPC and BinFinder, because they are feature-based, not relying on a vocabulary of embeddings.

> **Takeaway 4.3: Eliminating OOV in Binary Analysis**
>
> VexIR2Vec*'s entity-level vocabulary achieves zero OOV words across 400 binaries, while instruction-level approaches like SAFE encounter* $\approx 10^5$ *OOV occurrences—demonstrating practical scalability for real-world binary analysis.*

4.6 Related Works

VexIR2Vec involves a two-step learning process—pre-training a vocabulary to derive the final representation of the function as a series of lookups, followed by fine-tuning with VexNet to learn similarity using global attention. This way of learning vocabulary representation draws its inspiration from IR2Vec (Chapter 3), which uses representation learning for obtaining program embeddings. However, there are several key differences between the two methods:

Embeddings: IR2Vec models function embeddings through a heuristic-based accumulation of the entities of the LLVM-IR instructions, whereas VexIR2Vec uses global attention to learn to combine the different entities of the VEX-IR instructions and metadata. Section 10.6.5 shows that using the attention mechanism enhances the effectiveness of our proposed approach.

Sampling: VexIR2Vec is extracted from our notion of peepholes (Definition 4.1). The same program instruction might appear in multiple peepholes or multiple times within the same peephole. As seen in Section 10.6.1, the length of peepholes and the minimum number of times each basic block contributes to them are configurable parameters of VexIR2Vec. In contrast, IR2Vec is extracted from LLVM IR, where each IR instruction is visited exactly once.

Normalization: The construction of VexIR2Vec involves normalizing the VEX-IR using unsound architecture-independent optimizations implemented as VexINE, as explained in Section 4.3.3. In contrast, IR2Vec does not use any such form of normalization.

Canonicalization and Normalization As explained in Section 4.3.2, to construct the VexIR2Vec embedding of a program, we simplify its intermediate representation, replacing concrete syntax (constants, bitwidths, types, etc) with abstract placeholders. Previous works on binary similarity have used similar forms of canonicalization. The most common techniques consist in masking out constants, pointers, and registers [Ding et al., 2019; Duan et al., 2020; Zuo et al., 2019; Massarelli et al., 2019; Farhadi et al., 2014; Wang et al., 2023a; Li et al., 2021].

Optimizations, similar in purpose as the normalization step of Section 4.3.3, have also been used in previous works. For instance, VulHawk [Luo et al., 2023] uses *def-use* information to prune off redundant and unused instructions in IDA-Pro Microcode [Hex-Rays, n.d.]. As another example, David et al. [2017] lift the binaries to LLVM IR [Lattner and Adve, 2004] and decompose the functions into *strands*, which are then subject to the off-the-shelf optimizations of LLVM. These optimized *strands* are then used for computing similarity through statistical methods. Their notion of strands corresponds to the *program slices* proposed by Weiser [Weiser, 1984]. In this regard, David et al.'s work differs from ours in two aspects: first, their optimizations are sound; second, they are not restricted to straight-line code sequences. However, the differences between our work and David et al. go much beyond optimizations: they extract embeddings from single program slices, whereas we extract them from peepholes, which might include multiple occurrences of the same instruction. In terms of software engineering, we notice that extracting executable program slices is much more complex than extracting peepholes. Due to this complexity and a lack of a public artifact, we could not compare David et al.'s work and ours.

4.7 Summary

We introduced VexIR2Vec, an architecture-neutral — VEX-IR-based — embedding framework designed to represent binary codes as inputs for ML models. For representing binary code, we proposed *path-aware* peephole based embeddings that capture the control-flow characteristics of the functions. First, we decompose the functions into

peepholes derived from their CFG and normalize them by using our VEX-IR Normalization Engine (VexINE). Then, we follow an unsupervised pre-training to learn the seed embedding vocabulary of VEX-IR so as to represent the peepholes. We demonstrated that the seed embeddings capture intrinsic syntactic and semantic information present in the binary code. We also showed that our representations avoid out-of-vocabulary issues that are common in other tools that operate on binary code.

The source code and other relevant material are available in https://compilers.cse.iith.ac.in/VexIR2Vec.

Part II

ML-Driven Compiler Optimizations

"Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs."

— John Backus

Chapter 5

Using Machine Learning for Compiler Optimizations

5.1 Evolution of Compilers and Optimizations

Compilers don't just translate code—they decide how to optimize. From choosing which functions to inline, to determining how to allocate registers, to selecting the order of optimization passes, compilers constantly navigate complex trade-offs that directly impact program performance. How these decisions are made has evolved significantly over the decades, progressing through distinct eras that reflect advances in both hardware and our understanding of program optimization.

Early Compilers: Automation of Translation The journey began in the 1950s with the development of the first practical compiler by Grace Hopper for the A-0 system [Hopper, 1952]. This marked a paradigm shift, enabling automated translation of symbolic mathematical programs into machine code and freeing programmers from the complex details of machine code. A significant advancement came with the seminal work of Böhm [1954] on compiler bootstrapping, demonstrating that a compiler could be written in the language it compiles—enabling successive generations of compilers to build upon their predecessors.

Heuristic-Driven Compilers: The Era of Optimization The development of Fortran by Backus et al. [1957] ushered in a new era where compilers became optimizers, not just translators. Fortran's optimizing compiler proved that abstraction

and performance could coexist, with techniques like loop unrolling, strength reduction, and register allocation ensuring that high-level code executed efficiently. Frances Allen's foundational work on control flow and data flow analysis [Allen, 1970, 1971] provided the theoretical underpinnings for increasingly sophisticated optimizations.

As hardware grew more complex—from general-purpose CPUs to domain-specific accelerators and heterogeneous systems [Aho et al., 2006]—compiler design evolved correspondingly. The introduction of Intermediate Representations (IR) decoupled front-ends from back-ends, enabling sophisticated analyses and portable optimizations. The Static Single Assignment (SSA) form [Cytron et al., 1991] simplified data flow analysis, enabling optimizations like constant propagation, copy propagation, and dead code elimination with greater precision and efficiency [Muchnick, 1997; Khedker et al., 2009].

LLVM [Lattner and Adve, 2004] emerged as one of the first open-source production compilers to adopt an SSA-based IR, providing a modular framework that achieved wide adoption in both academia and industry. Modern developments continue to extend these foundations with polyhedral techniques [Bondhugula et al., 2008; Grosser et al., 2012] and multi-level abstractions like MLIR [Lattner et al., 2021]. Despite these advances, one common aspect remains: compilers rely on *heuristics* which are expert-crafted rules that approximate the optimization decisions. These heuristics encode decades of domain knowledge, but they also impose fundamental limitations.

The Limits of Heuristics Ideally, compilers should compute optimal solutions for each optimization problem. However, this is infeasible: many compiler optimizations are computationally intractable. Several have been formally proven to be NP-hard, including register allocation via graph coloring [Chaitin et al., 1981], software pipelining [Garey and Johnson, 1979], instruction scheduling (which maps to job-shop scheduling [Hennessy and Gross, 1983]), optimal instruction selection [Aho et al., 1989], and even flow-insensitive alias analysis [Horwitz, 1997]. More fundamentally, precise static analyses like alias analysis have been proven undecidable [Landi, 1992; Ramalingam, 1994]. Beyond these formally characterized problems, many optimizations—such as phase ordering, method inlining, vectorization, and loop transformations—present exponentially large search spaces with interdependent decisions [Muchnick, 1997].

To overcome this intractability, compilers have broadly adopted two strategies. The first is *heuristic-based optimization*, where expert-crafted rules serve as approximations to escape the computational cost of obtaining optimal solutions [Hind, 2001; Poletto and Sarkar, 1999; Muchnick, 1997]. Heuristics enable practical deployment by provid-

ing reasonable performance without exhaustive search. But they apply the same fixed strategies regardless of the input program. This leads to suboptimal results for programs that do not match the assumptions encoded in the rules.

The second strategy is *iterative compilation* or *autotuning* [Ansel et al., 2014], which searches the optimization space for each program through repeated compilation and profiling. This approach can discover better solutions than heuristics by adapting to program-specific characteristics, but at significant computational cost, as exploring even a subset of the search space can be heavily time-consuming and resource-intensive.

Neither approach fully exploits the optimization opportunities across modern programs and architectures. While heuristics lack adaptability, autotuning lacks efficiency. Figure 5.1 illustrates this trade-off.

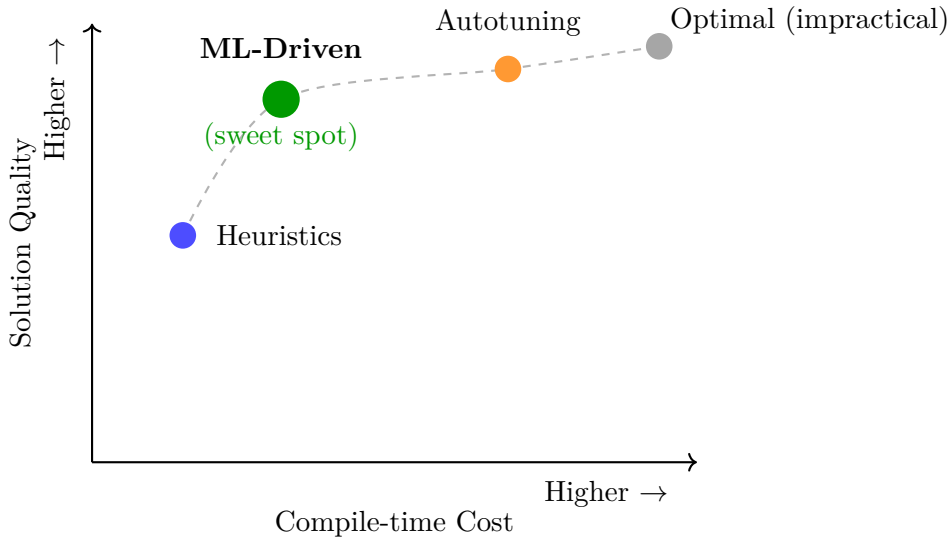


Figure 5.1: Trade-off between compile-time cost and solution quality. The dashed curve represents the Pareto frontier of feasible approaches. Optimal solutions are often impractical due to NP-hardness or complex search space. Heuristics are fast but produce lower-quality solutions; autotuning achieves high quality but at higher compile-time cost. ML-driven approaches occupy the sweet spot: learning patterns offline enables high-quality decisions with modest compile-time overhead.

Towards ML-Driven Compilers: Learning to Optimize These limitations motivate a fundamental shift in how we approach compiler optimization. Rather than encoding expert knowledge as static rules, can compilers *learn* to make better decisions?

This question echoes Turing's vision (introduced in Chapter 1 on page 1) that machines should learn from experience rather than follow hardcoded rules.

As articulated in RQ 2 (Section 1.1), we envision compilers that learn continuously from past executions and adapt to different workloads and future architectures:

> **Vision: Self-Learning Compilers**
>
> Compilers that **learn** to make optimization decisions, **adapt** to different workloads and architectures, and **evolve** their strategies based on observed performance.

Such self-learning compilers would not merely apply fixed optimization strategies but would generalize across programs and hardware platforms. This vision defines the current frontier in compiler evolution: *machine-learning-driven compiler optimizations*.[1] The remainder of this chapter explores what makes this vision promising (Section 5.2), the schema for realizing it along with the challenges involved (Section 5.3), and how this dissertation contributes to addressing them.

5.2 ML-Driven Compiler Optimizations

This vision of self-learning compilers builds on two decades of research applying ML to compiler optimization [Stephenson and Amarasinghe, 2005; Cavazos et al., 2007; Wang and OBoyle, 2018; Amarasinghe, 2020], contributing to the broader field of *Machine Learning for Code* (ML4Code) introduced in Chapter 1 on page 1. ML models can learn complex patterns from data, enabling them to make optimization decisions based on program characteristics and target architecture—automating what traditionally required extensive domain expertise.

As discussed in Part I on page 10, the *Extended Naturalness Hypothesis* suggests that programs exhibit statistical properties similar to natural language, and these properties—*along with program analysis information*—can be exploited for compiler optimizations. The program embeddings developed in Part I encode such semantic information and

[1] The recent emergence of Large Language Models (LLMs) for code raises the question of whether LLMs could directly make optimization decisions. While promising, LLMs face challenges for fine-grained compiler decisions: high inference latency (problematic for per-compilation decisions), difficulty in ensuring semantic correctness, and the need for effective program representations. The embedding techniques developed in this dissertation (Part I) can complement LLM-based approaches by providing structured semantic inputs. We discuss these directions further in Section 12.2 on page 253.

serve as the foundation for the ML-driven optimizations presented in this part of the dissertation.

This learning-based approach offers key advantages over heuristic-based methods, addressing the limitations illustrated in Figure 5.1:

Adaptability ML models can adapt to diverse programs and hardware architectures, making them more generalizable than fixed heuristics. This allows compilers to evolve faster alongside hardware advancements and emerging program domains without manual retuning.

Scalability Once trained, ML models infer optimization decisions quickly, significantly reducing the compile time overhead when compared to the autotuning approaches.

Optimality By learning from large amounts of training data, ML models can discover near-optimal solutions that capture complex relationships and exploit non-intuitive patterns that heuristics could miss.

However, ML-based approaches also introduce their own challenges: they require representative training data, face potential generalization issues across different programs and architectures, and involve overhead from model training and integration with compiler infrastructure.

5.3 Our Schema of ML-Driven Compiler Optimizations

Realizing the vision of self-learning compilers requires a systematic approach that spans data collection, program representation, model training, and deployment. Figure 5.2 illustrates this framework and how it maps to the structure of this dissertation. Each stage presents distinct challenges that we address through the contributions in this part.

Data Collection Training ML models for compiler optimizations requires datasets of programs with associated performance measurements or optimization labels. These may come from benchmark suites, synthetic generators [Cummins et al., 2017b], or fuzzers. Dataset curation remains an open challenge: existing benchmarks may not cover the diversity of real-world programs, while synthetic generation risks producing unrealistic patterns. We discuss this aspect further in Section 12.2 on page 253.

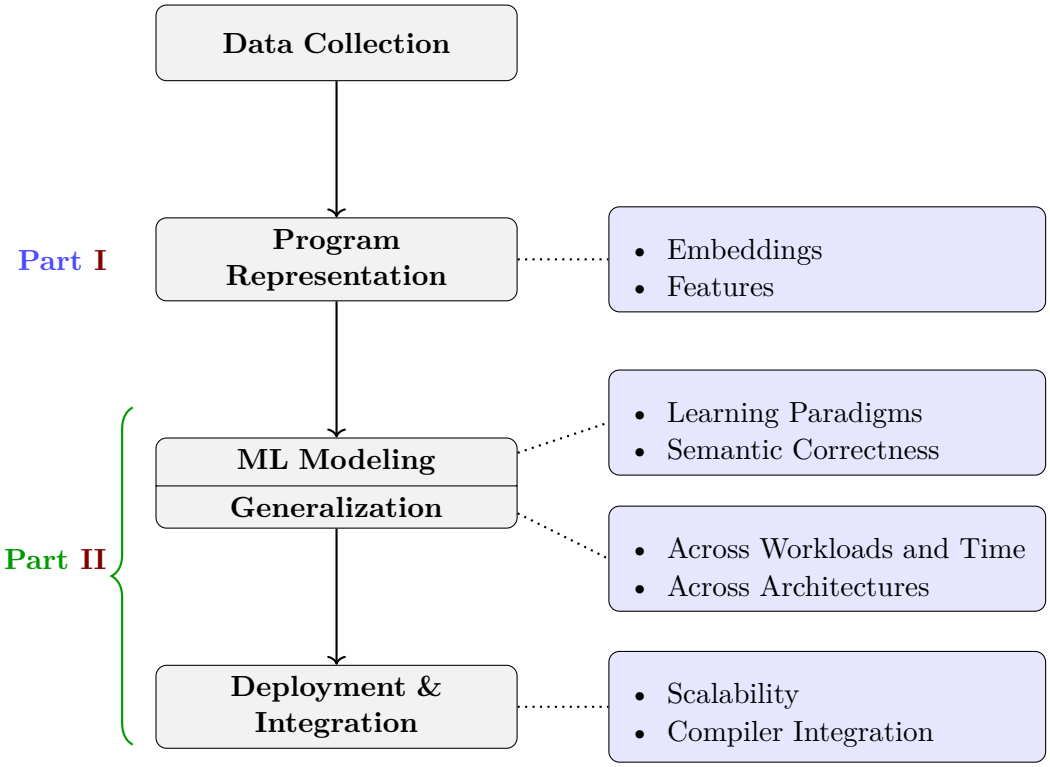


Figure 5.2: Schema for ML-driven compiler optimization, illustrating the stages from data collection through deployment. Program representation (Part I) provides the foundation, while learning, generalization, and deployment (Part II) complete the pipeline.

Program Representation Machine learning models operate on numerical vectors, yet programs are structured, symbolic artifacts. Early ML-driven approaches bridged this gap through hand-crafted features—counts of basic blocks, branches, and derived metrics like arithmetic intensity—designed by domain experts for specific tasks [Fursin et al., 2011; Grewe et al., 2013; Magni et al., 2014]. While effective, such features required significant engineering effort and often failed to transfer across problems.

Part I on page 10 develops an alternate approach through learned embeddings (IR2Vec, MIR2Vec, and VexIR2Vec) that encode semantic information directly from compiler intermediate representations. Wherever necessary these embeddings can be complemented with domain-specific features. Our register allocation work, for instance, combines MIR2Vec embeddings with interference graph structure and spill cost information.

Learning and Semantic Correctness The choice of learning paradigm depends on the structure of the optimization problem. Supervised learning suits problems where ground truth labels can be easily obtained. Reinforcement learning suits sequential decision problems or those with large exploration spaces where exhaustive labeling is infeasible. (Appendix C provides detailed background on these paradigms in the context of compiler optimizations.)

A critical concern is semantic correctness: unlike traditional compiler optimizations that are correct by construction, ML models learn statistical patterns that may not preserve program semantics. We address this through three strategies:

> **Definition 5.1: Inherently Safe Decisions**
>
> An optimization involves *inherently safe decisions* when all prediction choices yield semantically correct code. The ML model selects the *best-performing* option among alternatives that all preserve program behavior.

For example, in heterogeneous device mapping, any target device (CPU or GPU) yields correct execution; the model predicts which offers better performance. Such problems are naturally suited to supervised learning.

> **Definition 5.2: Compiler-Verified Decisions**
>
> An optimization involves *compiler-verified decisions* when some predictions may be invalid. The compiler validates each ML prediction before applying the transformation, falling back to default heuristics if the prediction is invalid.

For instance, not all vectorization factors are legal for a given loop. Correctness is guaranteed by the compiler's validation, while the model guides the search.

> **Definition 5.3: Constrained Learning**
>
> *Constrained learning* encodes semantic constraints directly into the learning process via *action masks* [Wang et al., 2016b]. The model predicts only among valid choices at each decision point, ensuring correctness while optimizing within the space of legal solutions.

This is essential for optimizations involving sequences of interdependent decisions or large combinatorial spaces where per-decision validation becomes impractical.

In this dissertation, we employ strategies (i) and (iii). Chapter 6 addresses device mapping and thread coarsening using supervised learning. Chapter 7 tackles register alloca-

tion, where constraints such as "no two interfering live ranges can share a register" are formalized as action masks within a reinforcement learning framework.

Generalization ML models must generalize beyond their training distribution to be useful in practice. This involves two dimensions:

(i) Across Architectures. Models trained on one hardware platform should transfer to new processors without complete retraining.

(ii) Across Workloads and Time. Models should handle diverse programs beyond the training set and improve over time. We adopt a policy improvement cycle [Trofin et al., 2021b] where the learned policy is fine-tuned on regression cases during deployment. While fully online continuous learning remains an open challenge, we address adaptability through policy improvement cycles and transfer learning, which enable models to refine their decisions on regression cases and retarget to new architectures.

Chapter 7 demonstrates both dimensions: transfer learning experiments show models trained on x86 adapting to AArch64 with minimal fine-tuning, while the policy improvement cycle enables continuous refinement on new and diverse workloads.

Deployment and Integration For ML-driven optimizations to be practical, they must integrate seamlessly into production compilers. This poses challenges in both directions: during training, ML models need compiler interaction to collect data and receive feedback; during inference, the compiler must invoke models with minimal overhead. Bridging Python-based ML frameworks with C++ compiler infrastructure is non-trivial. Naive approaches using process wrappers induce significant compile-time overhead [Jain et al., 2022b], while complex optimizations require richer communication than simple compiler flags [Trofin et al., 2021b].

Chapter 8 presents ML-Compiler-Bridge, a framework that provides a unified interface for efficient bidirectional communication between ML models and compiler infrastructure.

The following section details our specific contributions to each of these areas.

5.4 Contributions

In this part of the dissertation, we address RQ2 posed in Chapter 1 on page 1 by presenting three ML-driven compiler optimizations and a framework for ML-compiler integration towards realizing self-learning compilers. We use the program embeddings from Part I on page 10 to represent programs, ensure semantic correctness through action masks and inherently safe decisions, and demonstrate generalization via transfer learning and policy improvement cycles.

Table 5.1 maps our contributions to the schema stages described in Section 5.3, showing how each work addresses the key challenges in ML-driven compiler optimization.

Table 5.1: ML-Driven Compiler Optimizations mapped to our Schema

Optimization	**Representation**	**Semantic Correctness**	**Generalization**
Device Mapping & Thread Coarsening (Chapter 6)	IR2Vec embeddings	Inherently safe	Cross-platform evaluation
RL4ReAl: Register Allocation (Chapter 7)	MIR2Vec embeddings + graph structure	Action masks on legal registers	Transfer learning (x86 → AArch64); Policy improvement
Deployment & Integration: ML-Compiler-Bridge (Chapter 8) Inter-process (gRPC, Pipes) and in-process (ONNX, TF-AOT) model runners			

Heterogeneous Device Mapping and Thread Coarsening We present supervised learning based approaches for heterogeneous device mapping and thread coarsening prediction using IR2Vec embeddings (Chapter 6). For device mapping, we use IR2Vec embeddings to predict whether an OpenCL kernel should be mapped to a CPU or GPU for optimal performance. For thread coarsening, we predict the optimal coarsening factor for GPU kernels.

- Device mapping achieves 91.26% accuracy, improving over baselines by up to 12%.
- Thread coarsening achieves 1.05–1.10× geometric mean speedup, outperforming baselines while reducing training time by up to 8640×.

RL4ReAl**: Reinforcement Learning based Register Allocation** We present RL4ReAl, the *first end-to-end application* of reinforcement learning to solve the register allocation problem (Chapter 7). We formulate register allocation as a multi-agent

hierarchical RL problem, using MIR2Vec embeddings to represent interference graph nodes. Semantic correctness is ensured by constraining the action space to valid register assignments.

- Achieves performance on par with LLVM's greedy allocator, with 0.5–2% speedup on SPEC CPU benchmarks.
- Transfer learning enables 5× faster convergence when retargeting from x86 to AArch64, without loss in performance.
- Policy improvement cycle demonstrates continuous learning, recovering performance on regression cases.

ML-Compiler-Bridge: ML-Compiler Integration Framework We present ML-Compiler-Bridge, a library providing multi-language APIs for seamless interaction between ML models and compilers during training and deployment (Chapter 8). The library offers inter-process (gRPC, Pipes) and in-process (ONNX, TF-AOT) model runners.

- In-process runners reduce compile-time overhead by 7–22× compared to inter-process and Python wrapper approaches.
- Integrated with three compilers and four ML-enabled optimizations.
- Supports multithreaded compilation and distributed training with multiple workers.

ML-LLVM-Project Using ML-Compiler-Bridge, we integrate the optimizations presented in this dissertation (and our other ML-driven optimizations like loop distribution [Jain et al., 2022b] and phase ordering [Jain et al., 2022a]) with the LLVM compiler infrastructure. The ML-LLVM-Project[2] is an open-source fork of LLVM that provides an end-to-end infrastructure for developing ML-driven compiler optimizations. It integrates ML-Compiler-Bridge for model-compiler communication, IR2Vec and MIR2Vec for program embeddings, and four ML-driven optimizations in a unified manner. All components are configured, compiled, and linked during the regular LLVM build process.

[2] https://github.com/IITH-Compilers/ml-llvm-project

Outline of Part II

This part of the dissertation is organized as follows:

- Chapter 6 describes supervised learning approaches for heterogeneous device mapping and thread coarsening prediction using IR2Vec embeddings.
- Chapter 7 describes RL4ReAl, including the multi-agent hierarchical RL formulation, MIR2Vec embeddings for interference graphs, and transfer learning for retargetability.
- Chapter 8 describes ML-Compiler-Bridge, including the model runner implementations, serialization mechanisms, and the ML-LLVM-Project infrastructure.

Chapter 6

Heterogeneous Device Mapping and Thread Coarsening

6.1 Introduction

Heterogeneous computing systems that integrate CPUs, GPUs, and specialized accelerators have become central to modern computing, from data centers to mobile devices. Extracting optimal performance from these systems requires carefully mapping computations to appropriate devices and tuning execution parameters. Two fundamental optimization decisions are:

Definition 6.1: Device Mapping

Device mapping is the problem of determining the optimal execution target (e.g., CPU or GPU) for a computational kernel in a heterogeneous system, based on program characteristics and workload properties.

Definition 6.2: Thread Coarsening

Thread coarsening is an optimization that fuses multiple parallel threads into a single thread, reducing synchronization overhead and improving resource utilization. The *coarsening factor* specifies how many threads are combined.

Both problems have traditionally been addressed through handcrafted heuristics [Grewe et al., 2013; Magni et al., 2014] or complex deep learning models [Cummins et al., 2017a; Ben-Nun et al., 2018] that require extensive training data and computational resources.

In this chapter, we demonstrate that the IR2Vec embeddings developed in Chapter 3 on page 24 enable a simpler, more effective approach. By representing OpenCL kernels as fixed-size vectors that capture program semantics, we can use lightweight gradient boosting classifiers instead of complex sequential models.

As discussed in Section 5.3, these optimizations fall under the category of *inherently safe decisions*: any prediction (CPU or GPU for device mapping; any coarsening factor for thread coarsening) produces semantically correct code. The model's role is to select the *best-performing* option. This makes supervised learning a natural choice, where ground truth labels can be obtained based on runtime characteristics.

> **Key Idea**
>
> Program-level IR2Vec embeddings capture sufficient semantic information to enable simple classifiers to outperform complex deep learning models, while reducing training time by orders of magnitude.

Contributions

- We achieve state-of-the-art accuracy (91.26%) for heterogeneous device mapping, improving over baselines by up to 12%.
- We achieve geometric mean speedups of 1.05–1.23× for thread coarsening, including the first positive speedups on NVIDIA Fermi GPU.
- We reduce training time by 360×–8640× compared to other deep learning approaches.

Organization Section 6.2 presents the heterogeneous device mapping task, including the dataset, methodology, and experimental results. Section 6.3 presents the thread coarsening prediction task with similar structure. Section 6.4 discusses related work, and Section 6.5 summarizes the chapter.

6.2 Heterogeneous Device Mapping

Heterogeneous computing systems, which integrate different types of processing units like CPUs and GPUs, and specialized accelerators like DSPs (Domain Specific Processors) and NPUs (Neural Processing Units) have become central to modern computing. Such

systems play a crucial role in a wide range of applications, ranging from high-performance computing to embedded platforms like Nvidia Jetson, and Qualcomm Snapdragon.

These heterogeneous systems offer opportunities to exploit the strengths of different processing units to achieve high performance. For instance, CPUs excel in handling latency-sensitive tasks, while GPUs designed for parallelism are adept at handling throughput-oriented tasks. However, mapping the tasks to the appropriate processing unit is a challenging task, as it requires a deep understanding of the characteristics of the workload and the underlying hardware architecture. Incorrect mapping can lead to performance penalties, including increased execution time, higher energy consumption, and inefficient resource utilization.

Traditional methods, such as those proposed by Grewe et al. [2013], rely on manually crafted heuristics based on workload features like arithmetic operations, memory accesses, and derived metrics such as arithmetic intensity (computation-to-communication ratio). While these approaches provide a foundational framework, they struggle to adapt to the growing complexity of modern workloads and heterogeneous architectures. With the recent advancements in machine learning, program embeddings based approaches such as DeepTune [Cummins et al., 2017a] and inst2vec [Ben-Nun et al., 2018] demonstrating improved accuracy and speedup over heuristic methods in predicting optimal device mappings. We propose to use IR2Vec embeddings to predict the optimal device mapping for OpenCL kernels in heterogeneous systems.

6.2.1 Dataset

We use the dataset provided by Ben-Nun et al. [2018], which was originally collected by Grewe et al. [2013] for this optimization. It consists of 256 unique OpenCL kernels from seven different benchmark suites comprising AMD SDK [AMD, n.d.], NPB [Seo et al., 2011], NVIDIA SDK [NVIDIA, n.d.], Parboil [Stratton et al., 2012], Polybench [Pouchet et al., 2018], Rodinia [Che et al., 2009] and SHOC [Danalis et al., 2010]. Taking the kernels and varying their two auxiliary inputs (data size and workgroup size) for each kernel, a dataset is obtained. This leads to about 670 CPU (Intel Core i7-3820) or GPU labelled data points for each of the two GPU devices – AMD Tahiti 7970 and NVIDIA 970.

6.2.2 Approach

Initially the OpenCL kernels are compiled to LLVM IR using the Clang compiler. The LLVM IR is then processed by IR2Vec to generate embeddings for the kernels. The embeddings are then used to train a machine learning model in a supervised learning setting to predict the optimal device mapping for the kernels. We model the device mapping task as a binary classification problem, where the model predicts whether a kernel should be mapped to the CPU or GPU.

We use the gradient boosting classifier [Friedman, 2002] to predict the optimal device mapping. Gradient boosting is an ensemble learning technique that builds a strong model by combining multiple weak models. It is a popular choice for classification tasks due to its high accuracy and robustness to overfitting. We use simpler models like gradient boosting classifier, as IR2Vec embeddings are generated at the program/function level directly without forming sequential data that need sequential neural networks like RNNs or LSTMs.

To ensure robust evaluation and prevent overfitting, we use ten-fold cross-validation [Kohavi, 1995], as used in previous approaches [Cummins et al., 2017a; Ben-Nun et al., 2018]. In each fold, the dataset is divided into ten parts, and the model is trained on nine parts and evaluated on the remaining part. This process is repeated ten times, with each part serving as the test set once. The final accuracy is computed as the average accuracy across all the folds.

6.2.3 Baselines

We compare the performance of IR2Vec embeddings with that of the original heuristic based approach proposed by Grewe et al. [2013], and the recent deep learning based approaches of DeepTune [Cummins et al., 2017a] and NCC [Ben-Nun et al., 2018].

Static Mapping Static mapping is a baseline heuristic in which a single device (CPU or GPU) is chosen for all the kernels. The device is chosen based on the average of runtimes of all the kernels on both the devices. On AMD Tahiti 7970, the best performing device is CPU, while on NVIDIA GTX 970, the best performing device is GPU.

Grewe et al. [2013] Grewe et al. use a feature-based approach to predict the optimal device mapping for OpenCL kernels. They use a combination of 10 static and dynamic features extracted from the OpenCL kernels using a custom LLVM pass

and the runtime respectively. Using these features, they train a decision tree classifier to predict the optimal device mapping.

DeepTune [Cummins et al., 2017a] DeepTune is a deep-learning-based approach that learns to represent the input code by modeling the program as a sequence of tokens. Given the sequence of tokens, LSTM network is trained in a supervised manner to predict the optimal device mapping.

NCC [Ben-Nun et al., 2018] inst2vec is a program-embedding-based approach that learns to represent the instructions of LLVM IR using skip-gram model [Mikolov et al., 2013a]. The instruction level embeddings are combined using LSTMs to predict the optimal device mapping.

Along with inst2vec encodings, the NCC framework also proposes the inst2vec-imm encodings to handle immediate values. For this, they formulate four *immediate value handling methods*. As there is no single consistent winner among these value-handling methods, NCC considers the best result out of them[1].

As the dynamic features used by Grewe et al. are not available as part of the embeddings, DeepTune and inst2vec concatenate the embeddings of the kernels with the dynamic features as additional input dimensions. We follow the same approach to ensure a fair comparison.

6.2.4 Experiment

We evaluate the performance of IR2Vec embeddings for the heterogeneous device mapping task by comparing the prediction accuracy and speedup against the baselines.

Setup

The embeddings for each kernel are computed using IR2Vec infrastructure. Gradient boosting classifier with a learning rate of 0.5, which allows a maximum depth of up to 10 levels and 70 estimators with ten-fold cross-validation is used to train the model. And, the classifier is trained separately for each CPU-GPU pair. Similar to the earlier methods, we obtain the runtimes corresponding to the predicted device, to calculate the speedup against the static mapping heuristics.

[1] As the source code for the immediate value handling methods is not available, we directly use the results reported in their paper for comparison.

Accuracy

Accuracy is computed as the percentage of correct device mapping predictions for a kernel by the model over the total number of predictions during test time. In Figure 6.1, we show the accuracy of mapping using the embeddings generated by IR2Vec and other methods. *Flow-Aware* and *Symbolic* embeddings achieve average accuracies of 91.26% and 88.72%, respectively.

Table 6.1: Percentage improvement in accuracy obtained by *Flow-Aware* embeddings when compared to other approaches

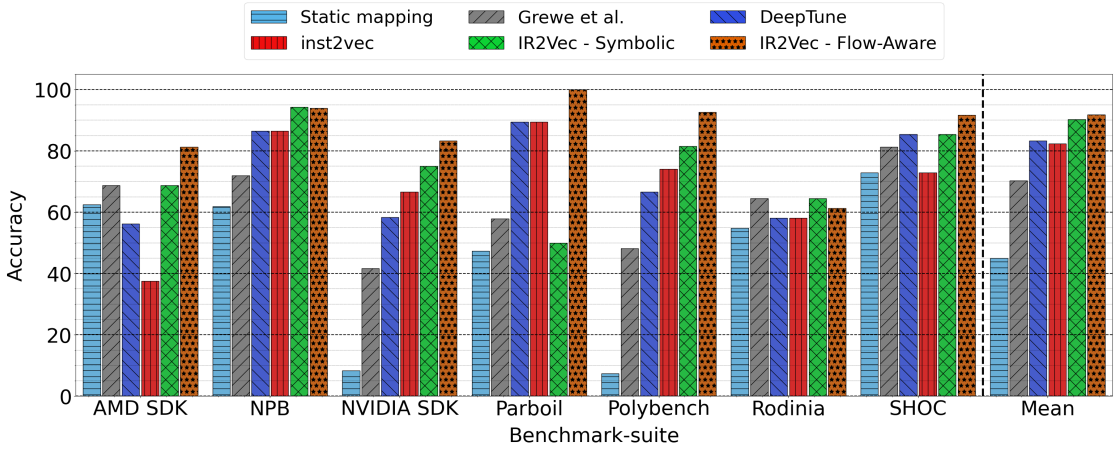
Architecture	**Grewe et al.**	**DeepTune**	**inst2vec**	**inst2vec-imm**	IR2Vec **Symbolic**
AMD Tahiti 7970	26.50%	10.93%	12.12%	5.38%	2.81%
NVIDIA GTX 970	22.96%	11.70%	9.69%	3.54%	2.92%

> **Takeaway 6.1: Lightweight Models Outperform Deep Learning**
>
> *Flow-Aware embeddings achieve 91.26% device mapping accuracy—over 10% better than prior neural network-based approaches—using a simple gradient boosting classifier, demonstrating that rich representations enable lightweight models to outperform complex baselines.*

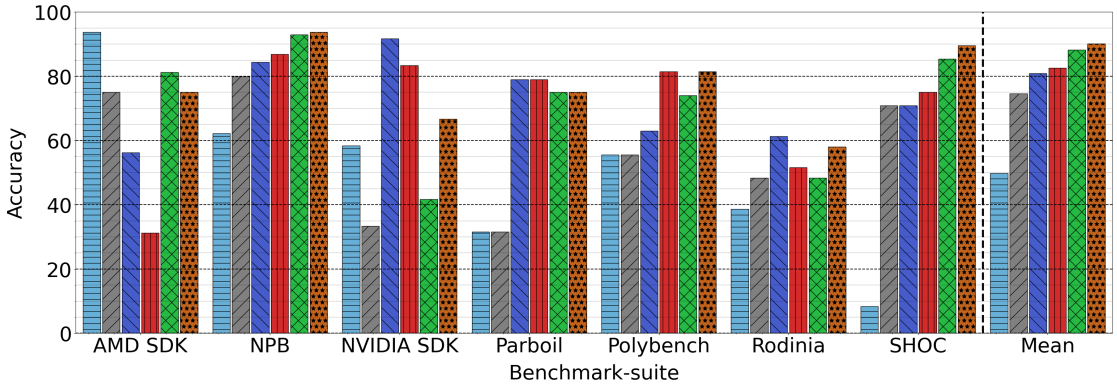
In Table 6.1, we show the percentage improvement of accuracy obtained by *Flow-Aware* embeddings over the other methods. It can be observed that the *Flow-Aware* embeddings achieve higher performance over the other methods [Grewe et al., 2013; Cummins et al., 2017a; Ben-Nun et al., 2018]. On AMD Tahiti 7970, our *Flow-Aware* embeddings achieve the highest accuracy in all 7 benchmark-suites. In addition, our *Symbolic* embeddings achieve second-highest in 4/7 benchmark-suites. Similarly, in NVIDIA GTX 970, *Flow-Aware* embeddings achieve the highest accuracy in 3/7 cases, and *Symbolic* embeddings achieve the highest or second-highest accuracy in 4/7 cases. In both the platforms, the IR2Vec embeddings achieve the highest average accuracy.

Speedups

In Figure 6.2, we show the normalized speedups achieved in comparison with static mapping as the baseline on both the platforms under consideration. The normalized speedup is calculated as the ratio of the speedup achieved on the predicted device to the

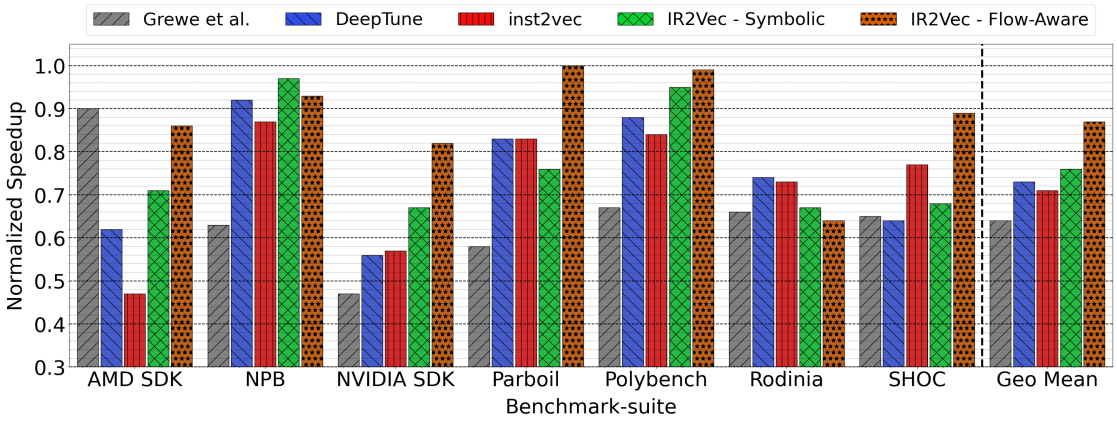


(a) AMD Tahiti 7970

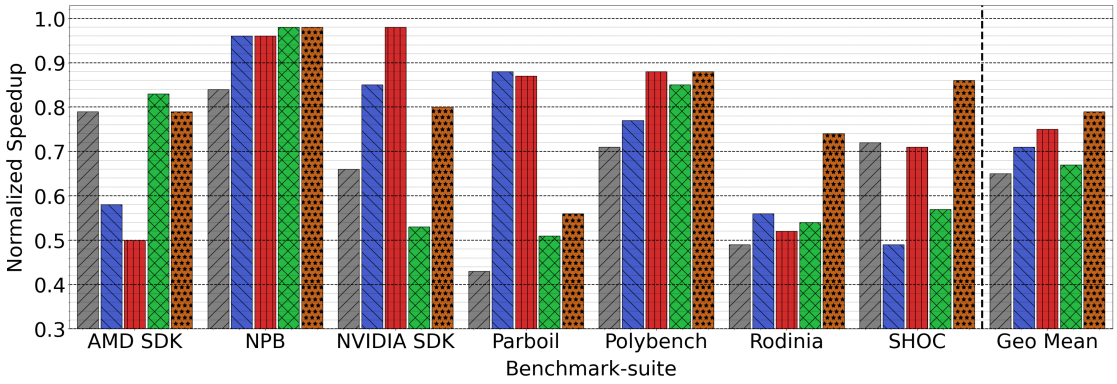


(b) NVIDIA GTX 970

Figure 6.1: Accuracy for heterogeneous device mapping task on various benchmark-suites



(a) AMD Tahiti 7970



(b) NVIDIA GTX 970

Figure 6.2: Speedups corresponding to the device mapping predictions by various methods. The speedups are normalized with respect to oracle, the maximum speedup that can be achieved.

maximum speedup achieved by any device for the given kernel and the GPU. Value of 1 indicates that the prediction achieves the best possible speedup.

With *Flow-Aware* embeddings, we achieve about 88.38% and 77.65% of the maximum speedup given by the oracle on AMD Tahiti and NVIDIA GTX respectively, when compared to the 70.76% on AMD Tahiti, and 70.66% on NVIDIA GTX by inst2vec and 72.90% on AMD Tahiti and 66.79% on NVIDIA GTX by Deeptune. (With *Symbolic* embeddings we achieve about 80.11% and 72.03% of the maximum speedup on AMD Tahiti and NVIDIA GTX.)

Flow-Aware embeddings achieve a geometric mean of 1.58× speedup on AMD Tahiti 7970 and a 1.26× speedup on NVIDIA GTX 970; in comparison, on the AMD platform, the speedups achieved by the state-of-the-art models of DeepTune and inst2vec is 1.46× and 1.4× respectively; while, on the NVIDIA platform, both DeepTune and inst2vec achieve a comparable 1.21× speedups. *Symbolic* encoding also performs better than the state-of-the-art by achieving a speedup of 1.56× on AMD Tahiti and 1.24× on NVIDIA GTX.

Table 6.2: Percentage improvement in speedups obtained by *Flow-Aware* embeddings over *Symbolic* embeddings and other baselines

Architecture	**Grewe et al.**	**DeepTune**	**inst2vec**[2]	IR2Vec **Symbolic**
AMD Tahiti 7970	29.65%	7.96%	12.95%	1.43%
NVIDIA GTX 970	13.77%	4.31%	3.93%	1.92%

In Table 6.2, we show the percentage improvement of speedups obtained by *Flow-Aware* embeddings in comparison with the earlier methods. It can be seen that the *Flow-Aware* embeddings perform better on both the platforms when compared to the other works.

On benchmarks like `ep` (*embarrassingly parallel*), where mapping them to GPUs gets the optimal performance, we map to GPU—in all the turns of 10-fold cross-validation—and hence achieve the highest possible speedup. Similarly, for `LU`, which gives an optimal performance when mapped to CPUs, we map the kernels that ought to be mapped to

[2]The inst2vec-imm results given in their paper are based on arithmetic mean, whereas our results are based on geometric mean. inst2vec-imm is reported to achieve a (arithmetic mean) speedup of 3.47× and 1.44×; whereas our *Flow-Aware* embeddings achieve a speedup of 3.51× and 1.47× on AMD Tahiti and on NVIDIA GTX respectively.

CPU correctly, with very high confidence (of about 95%) in various turns of 10-fold cross-validation. In comparison, DeepTune and inst2vec map LU to CPUs with a confidence of 77% and 85%, respectively.

Slowdown

Our predictions using *Flow-Aware* embeddings result in the *least* number of slowdowns both at the level of benchmarks and benchmark-suites. Our predictions result in a slowdown in 18/142 of benchmarks across two platforms; in comparison, inst2vec and DeepTune result in a slowdown in 33 and 32 benchmarks, respectively. Also, at an aggregate benchmark-suite level, the prediction using *Flow-Aware* embeddings leads to a slowdown in 3/14 cases across platforms, in comparison inst2vec and DeepTune result in slowdowns in three and six cases, respectively.

6.2.5 Training characteristics

Training for the device mapping task by IR2Vec takes $\approx$ 5 seconds on a P100 GPU, when compared to about 10 hours and 12 hours of training time taken by DeepTune and NCC respectively. This results in a *reduction of about* $\approx 7200\times$–$8640\times$ *in training time without a reduction in performance.*

The earlier works take much time for training because they involve training a large number of parameters: DeepTune uses $\approx 77K$ parameters, while NCC uses $\approx 69K$ parameters. In contrast, the IR2Vec predictions use Gradient Boosting, which is a collection of a small number of shallow decision trees. This reduction in time is primarily possible because the embeddings obtained by IR2Vec enable us to effectively use the Gradient boosting algorithm instead of the compute-intensive and data-hungry neural networks (LSTM in this case) that do not fit well in the cache hierarchy.

> **Takeaway 6.2: Orders-of-Magnitude Training Speedup**
>
> *By decoupling representation learning from the downstream task,* IR2Vec *enables the use of lightweight models (Gradient Boosting) instead of compute-intensive neural networks, achieving* $7200\times$–$8640\times$ *faster training while maintaining or improving accuracy.*

6.3 Prediction of Optimal Thread Coarsening Factor

Having demonstrated the effectiveness of IR2Vec embeddings for device mapping, we now present a related optimization: predicting the optimal thread coarsening factor for GPU kernels.

Thread coarsening is an important optimization technique in parallel computing, particularly for GPU devices. This optimization adjusts the amount of work done by a single thread by merging the work assigned to multiple threads [Volkov and Demmel, 2008]. This process reduces the overhead of thread management and synchronization, and reduces or eliminates the recomputation of the same data by multiple threads. The thread coarsening can also lead to decrease in the register pressure. However, these benefits come at the cost of increasing the computational workload per thread, which can potentially lead to a slowdown.

The primary objective of thread coarsening is to find a balance between the benefits of reduced overhead and the cost of increased workload. The thread coarsening factor, the number of threads that are fused together plays a significant role in determining this balance. Finding the optimal thread coarsening factor for a given kernel is non-trivial, as it depends on several factors based on the characteristics of the kernel and the target GPU architecture.

For example, memory-bound kernels that have excessive thread synchronization overhead can benefit from a higher coarsening factor, as it reduces the overall thread count while increasing the workload of individual threads. Similarly, compute-bound kernels with low memory access-to-computation ratios may see significant performance improvements with thread coarsening, as it enhances instruction-level parallelism (ILP) [Volkov and Demmel, 2008; Magni et al., 2013]. Modern GPUs have different architectures, that affects the outcome of the thread coarsening. For instance, the VLIW-based AMD Radeon GPU can exploit instruction level parallelism of the underlying kernel (eg., `nbody`) better than the SIMD-based AMD Tahiti GPU [Magni et al., 2014]. Hence, choosing a coarsening factor that gives the best speedup on one GPU may not necessarily give the best performance on another GPU or could even lead to a slowdown. Thus, predicting the optimal thread coarsening factor for a given kernel on a specific GPU architecture is a crucial, yet non-trivial task.

Traditional approaches, like those by Magni et al. [2014], relied on manual tuning or feature engineering to model the coarsening factor selection. With the recent ad-

vancements, deep learning models like LSTMs [Cummins et al., 2017a], RNNs [Ben-Nun et al., 2018], and GNNs [Brauckmann et al., 2020] have been proposed to predict the optimal thread coarsening factor. Despite the success of these models, they are computationally expensive and require a large amount of data for training. We propose to use IR2Vec embeddings to predict the optimal thread coarsening factor for a given kernel on a specific GPU architecture.

6.3.1 Dataset

We use the dataset curated by Ben-Nun et al. [2018] which was originally proposed by Magni et al. [2014] for predicting the optimal thread coarsening factor. The dataset consists of 17 OpenCL kernels from AMD SDK [AMD, n.d.], NVIDIA SDK [NVIDIA, n.d.], and Parboil [Stratton et al., 2012] benchmarks. The kernels are executed on 4 different GPUs – AMD Radeon 5900 (Cypress), AMD Tahiti 7970 (Tahiti), NVIDIA GTX 480 (Fermi), and NVIDIA Tesla K20c (Kepler). This leads to a total of 68 data points, with each kernel executed on each GPU for different thread coarsening factors among $\{1, 2, 4, 8, 16, 32\}$. Coarsening factor 1 denotes the baseline runtime without any thread coarsening. The runtime of each kernel is recorded for each thread coarsening factor on each GPU.

6.3.2 Approach

The OpenCL kernels are first converted to LLVM-IR using the Clang compiler. Then the LLVM-IR is processed to obtain the Symbolic and Flow-Aware embeddings of IR2Vec. The embeddings are then used to predict the optimal thread coarsening factor for the given kernel on a specific GPU architecture. Similar to the device mapping task, we train a gradient boosting classifier in a supervised learning setting with 10-fold cross-validation to predict the optimal thread coarsening factor.

The choice of gradient boosting classifier is motivated by the relatively smaller size of the dataset. Other deep learning models like LSTMs and RNNs require a large amount of data for training and is challenging to effectively train on small datasets without overfitting. On the other hand, gradient boosting classifiers are known to perform well on small datasets and are less prone to overfitting.

The classifier is trained on the Symbolic and Flow-Aware embeddings of the kernels separately for each GPU architecture to choose the best coarsening factor among $\{1, 2, 4, 8, 16, 32\}$.

6.3.3 Baselines

We compare the performance of Symbolic and Flow-Aware embeddings of IR2Vec with the feature-based approach by Magni et al. [2014], and the deep learning models of DeepTune [Cummins et al., 2017a] and NCC [Ben-Nun et al., 2018].

Magni et al. [2014]: This approach uses a feature-based model to predict the optimal thread coarsening factor. A feature vector consisting of 17 features are extracted from the kernel. These features include the number of basic blocks, instructions, divergent instructions, branches, etc. Magni et al.'s approach predicts the coarsening factor of OpenCL kernels using a Multilayer Perceptron (MLP) based cascading binary decision model. Starting with the source code, they iteratively compute feature vectors and apply a binary classification to determine whether to coarsen further or finalize the coarsening factor at each step, creating a multi-stage prediction process.

DeepTune [Cummins et al., 2017a]: DeepTune models the problem of predicting the optimal thread coarsening factor as a multi-class classification problem. Given the tokens of the input kernel, DeepTune trains an LSTM model to predict the optimal thread coarsening factor for a given kernel on a specific GPU architecture. The embeddings are learned as part of this training process. As the number of data points in the dataset is very low, the authors use transfer learning technique where the model trained on the device mapping dataset (Section 6.2.1) is fine-tuned on the thread coarsening dataset. This variant of DeepTune is referred to as DeepTune-TL.

NCC [Ben-Nun et al., 2018]: As described in Section 6.2.3, NCC learns to represent the instruction of the LLVM IR corresponding to the input as embeddings called inst2vec. These instruction-level embeddings corresponding to the instructions of the kernel are modeled as a sequence and fed to an LSTM model to predict the optimal thread coarsening factor for a given kernel on a specific GPU architecture. As described in Section 6.2.3, NCC also proposes a variant of inst2vec called inst2vec-imm, which uses the immediate values in the program during training.

6.3.4 Experiment

We evaluate the performance of Symbolic and Flow-Aware embeddings of IR2Vec for predicting the optimal thread coarsening factor for a given kernel by comparing the

speedups obtained by the predicted coarsening factors with the speedups obtained by the coarsening factors predicted by the baselines.

Setup

For this experiment, we configure the gradient boosting model with a learning rate of 0.05 and 140 decision stumps, each with a maximum depth of 1. This choice of hyperparameters is deliberate, considering the trade-off between model complexity and the limited number of data points. The use of shallow decision trees (stumps) mitigates overfitting and ensures that the model generalizes well to unseen data. We use 10-fold cross-validation to evaluate the performance of the model.

Speedups

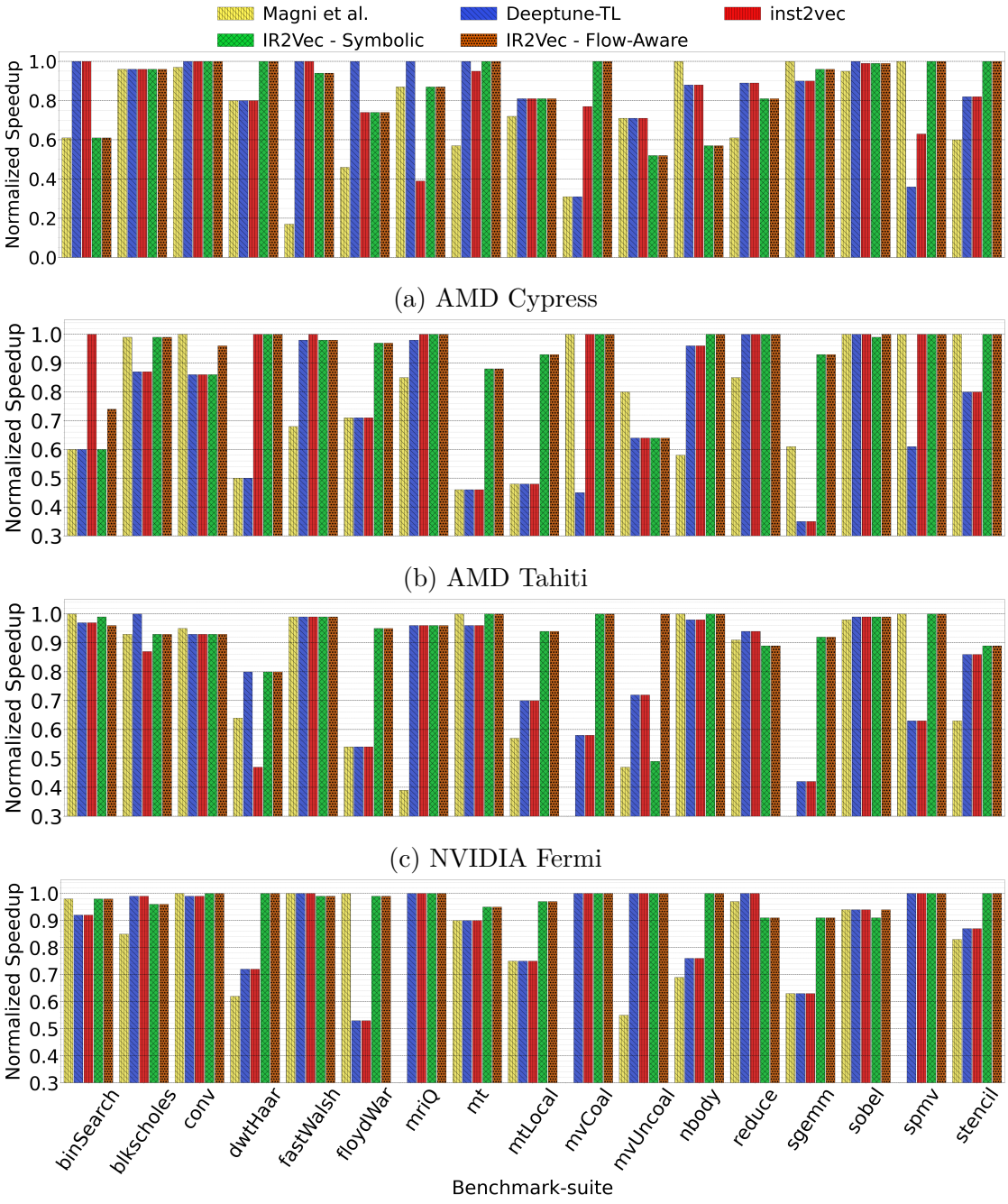
The speedup is calculated as the ratio of the runtime of the kernel with the predicted coarsening factor to the runtime of the kernel with coarsening factor 1.

Figure 6.3 showcases the normalized speedups achieved by our proposed IR2Vec embeddings, specifically the *Flow-Aware* and *Symbolic* embeddings, compared against baselines across the four GPU platforms. The normalized speedup is calculated as the ratio of the speedup achieved with the predicted coarsening factor to the maximum speedup achieved by any coarsening factor for that kernel on that GPU; Value of 1 indicates that the prediction achieves best possible speedup. The geometric mean of the speedups across all the kernels for each GPU platform is shown in Figure 6.4.

AMD Cypress On the AMD Radeon HD 5900 platform, both the *Flow-Aware* and *Symbolic* embeddings achieve a speedup of 1.2×, outperforming the speedup of 1.15× and 1.14× achieved by inst2vec and the DeepTune model with transfer learning (DeepTune-TL), respectively.

AMD Tahiti On the AMD Tahiti 7970, the *Flow-Aware* demonstrates a strong performance, achieving a speedup of 1.23×. The *Symbolic* encoding also performs strongly, with a speedup of 1.2×. Both significantly outperform DeepTune-TL and inst2vec, which achieve speedups of 0.9× and 1.04×, respectively.

NVIDIA Fermi We observe a significant improvement on the NVIDIA GTX 480 platform, where we are the *first* to report positive speedups. The *Flow-Aware* encoding



(a) AMD Cypress

(b) AMD Tahiti

(c) NVIDIA Fermi

(d) NVIDIA Kepler

Figure 6.3: Plot showing the normalized speedups achieved by predicted coarsening factors by various methods

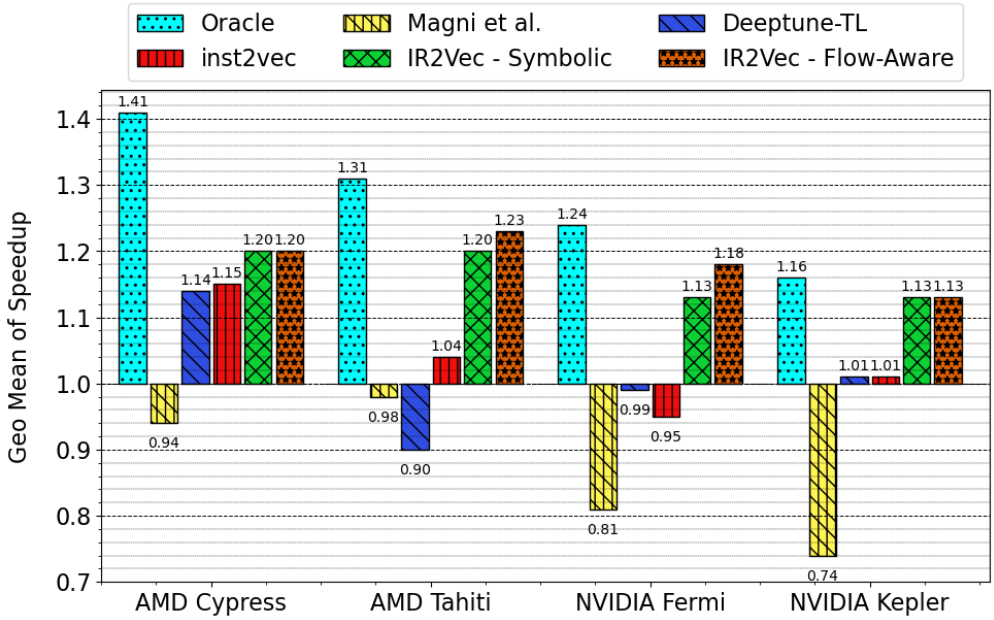


Figure 6.4: Plot showing the geometric mean of the speedups achieved by predicted coarsening factors by various methods in comparison to the maximum speedup achieved by the oracle with the best coarsening factor

achieves a speedup of 1.18×, and the *Symbolic* encoding closely follows with 1.13×, significantly outperforming inst2vec and DeepTune-TL, which achieve speedups of 0.95× and 0.99×, respectively.

NVIDIA Kepler On the NVIDIA Tesla K20c platform, both the *Flow-Aware* and *Symbolic* embeddings achieve a speedup of 1.13×, surpassing the 1.01× speedup achieved by DeepTune-TL and inst2vec.

Table 6.3: Percentage improvement in speedups obtained by *Flow-Aware* embeddings over *Symbolic* embeddings and other baselines

Architecture	**Magni et al.**	**DeepTune**	**DeepTune-TL**	**inst2vec**[3]	IR2Vec **Symbolic**
AMD Cypress	27.66%	5.26%	5.26%	4.35%	–
AMD Tahiti	25.41%	29.37%	36.56%	18.17%	2.08%
NVIDIA Fermi	45.31%	25.21%	18.89%	23.89%	3.98%
NVIDIA Kepler	52.84%	15.41%	11.98%	11.98%	0.18%

Table 6.3 provides a quantitative summary of the percentage improvements achieved by the *Flow-Aware* embeddings over *Symbolic* embeddings and over other prior approaches, reinforcing their consistent outperformance. Notably, the IR2Vec embeddings consistently outperform the baselines across all four GPU platforms.

These results demonstrate the robustness of IR2Vec embeddings in handling platform-specific challenges that previously hindered positive gains. The superior performance of *Flow-Aware* can be attributed to its ability to capture the flow semantics of the program, which is crucial for understanding and optimizing thread-level parallelism.

Slowdown

As observed by Magni et al. [2014], kernels such as `spmv` and `mvCoal` which exhibit irregular dependencies are less responsive to thread coarsening. As a result, no performance improvement can be achieved for these kernels. Our approach achieves baseline performance in such kernels without introducing any slowdown, whereas previous models show negative speedups. Specifically, DeepTune results in a slowdown of up to 0.36× on the AMD Cypress, and inst2vec causes a slowdown of up to 0.63× on both the AMD Cypress and NVIDIA Fermi platforms.

A similar trend is observed on the iterative version of the Jacobi stencil (`stencil`) kernel, where thread coarsening predictions by the baselines lead to a slowdown on all the platforms, except on NVIDIA Fermi. In contrast, IR2Vec embeddings achieve the baseline speedup without introducing any slowdown.

IR2Vec achieves the best performance on approximately 70% of the kernels across all platforms. Notably, the *Flow-Aware* embeddings rarely cause slowdowns—only in 8 out of 68 cases. Even in these cases, the performance remains within 10% of the baseline. In contrast, predictions made by inst2vec and DeepTune-TL result in slowdowns in 18 and 21 cases, respectively. We attribute this improved performance to the flow information embedded in the vectors, which allows for better handling of program properties in these kernels.

[3]As per the values reported by Ben-Nun et al. [2018], inst2vec-imm obtains an arithmetic mean of 1.28×, 1.18×, 1.11× and 1× speedup on AMD Cypress, AMD Tahiti, NVIDIA Fermi and NVIDIA Kepler; while IR2Vec *Flow-Aware* obtains an average speedup of 1.25×, 1.3×, 1.26× and 1.16× respectively.

Takeaway 6.3: Consistent Cross-GPU Speedups

IR2Vec *consistently achieves positive speedups across all four GPU architectures (1.13–1.23×), including the first reported gains on NVIDIA Fermi where prior methods produced slowdowns, demonstrating robust generalization across diverse hardware.*

6.3.5 Training characteristics

Training for the thread coarsening with IR2Vec embeddings takes ≈ 10 seconds on a P100 GPU, which is significantly faster than the training the models of DeepTune and NCC. DeepTune-TL takes ≈ 11 hours to train on the same dataset, while DeepTune and NCC take ≈ 1 hour to train the model with $77K$ and $69K$ parameters, respectively. This accounts for a reduction of training time by ≈ 360×–3960× while achieving better speedups.

Takeaway 6.4: Lightweight Training

Similar to device mapping, IR2Vec *achieves* 360×–3960× *faster training for thread coarsening compared to neural network baselines, reinforcing that decoupling representation learning from task-specific modeling enables efficient ML-driven compiler optimizations.*

6.4 Related Works

The problems of heterogeneous device mapping and thread coarsening have been widely studied in the literature, with various ML-based approaches. Due to their simple formulation using a supervised learning mechanism, these tasks have been among the most popular in ML-based compiler optimization.

The first ML-based approaches for heterogeneous device mapping and thread coarsening were introduced by Grewe et al. [2013] and Magni et al. [2014], respectively. Grewe et al. [2013] employed a decision tree model, while Magni et al. [2014] used a simple multi-layer perceptron. DeepTune [Cummins et al., 2017a] later introduced an end-to-end approach using LSTMs trained on a token-based representation.

Subsequent works further improved performance using program embeddings. NCC [Ben-Nun et al., 2018] introduced instruction-level *inst2vec* embeddings with RNNs, while ProgramL [Cummins et al., 2021] enhanced this by incorporating GNNs for a more

structured representation.

Recent evaluations [Dutta and Jannesari, 2024; Hakimi et al., 2021] demonstrate that our approach remains competitive, outperforming even contemporary works [Cummins et al., 2021; Jamsaz et al., 2023] while using a simple XGBoost based approach.

6.5 Summary

In this chapter, we studied the problem of optimal mapping of OpenCL kernels to heterogeneous devices and prediction of optimal thread coarsening factor. We proposed a novel approach by leveraging IR2Vec embeddings to represent the kernels and used gradient boosting classifiers to predict the optimal mapping and thread coarsening factor. We demonstrated that our approach outperforms the state-of-the-art methods both in terms of speed-up and scalability.

When compared to earlier approaches, our approach of representing programs is non-datahungry, takes less training time of up to 8640×. Our approach results in an improvement of average speedups by 8.5% for device mapping and 14.6% for thread coarsening while achieving least slowdowns.

The source code and other relevant material are available at https://github.com/IITH-Compilers/IR2Vec.

Chapter 7

RL4ReAl: Reinforcement Learning for Register Allocation

7.1 Introduction

Register allocation is one of the well-studied and important compiler optimization problems. It involves assigning a finite set of registers to an unbounded set of variables. Its decision problem is reducible to graph coloring, which is one of the classical NP-Complete problems [Garey and Johnson, 1979; Bouchez et al., 2006]. Register allocation as an optimization involves additional sub-tasks, more than graph coloring itself [Bouchez et al., 2006]. Several formulations have been proposed that return exact, or heuristic-based solutions.

Broadly, solutions are often formulated as constraint-based optimizations [Lozano et al., 2012; Kuchcinski, 2003], ILP [Appel and George, 2001; Barik et al., 2006; Chang et al., 1997; Nagarakatte and Govindarajan, 2007], PBQP [Kim et al., 2021], game-theoretic approaches [Quintão Pereira and Palsberg, 2008], and are fed to a variety of solvers. In general, these approaches are known to have scalability issues. On the other hand, heuristic-based approaches have been widely used owing to their scalability: reasonable solutions for practical benchmarks in near linear time. However, developing good heuristics is highly non-trivial and requires specialized domain expertise, on compiler construction as well as on hardware architecture. Various heuristics have been proposed over the past 40 years [Chaitin et al., 1981; Chow and Hennessy, 1990; Briggs et al., 1994], extending to recent times [Chen et al., 2018b]. They are often fine-tuned

for a particular architecture and yield non-optimal performance.

Recently, with the wide range of successes of Machine Learning (ML), ML-based approaches are being proposed to solve compiler optimization problems that have been known to be computationally expensive. These include classical optimizations like phase ordering [Fursin et al., 2011; Ashouri et al., 2017; Jain et al., 2022a], vectorization [Haj-Ali et al., 2020], function inlining [Simon et al., 2013], throughput prediction [Mendis et al., 2019a; Sykora et al., 2022]. However, the applicability and effectiveness of ML methods to compiler optimizations under hard semantic constraints remains poorly understood. Focusing on register allocation, we identified some of the main reasons.

- Register allocation is a complex problem, composed of multiple sub-tasks, including splitting, coalescing, spilling. These sub-tasks have to be considered in addition to modeling hardware complexities.
- ML-based allocation schemes should ensure correctness: no two variables in the same live range should be assigned to the same register, and the register types should be respected. Such semantic constraints should not suffer any approximation, unlike forgetful optimizations like function inlining.
- On a practical note, it is hard to integrate ML models and Reinforcement Learning (RL) algorithms in Python with compiler frameworks in C++ that are among the most complex pieces of software engineering.

Key Idea

Register allocation is formulated as a multi-agent hierarchical RL problem where semantic correctness is guaranteed by *action masks* that constrain agents to predict only among *valid* register assignments. Interference graph nodes are represented using MIR2Vec embeddings (Section 3.4), enabling learned policies that generalize across architectures via transfer learning and adapt to new workloads through a policy improvement cycle.

As discussed in Section 5.3, register allocation exemplifies optimizations involving sequential, interdependent decisions where compiler validation at each step is impractical. The allocator must make decisions about register assignments, live range splitting, and spilling, where the correctness of each choice depends on previous ones. We address this through *constrained learning*, encoding semantic constraints as action masks within a reinforcement learning framework.

Some initial attempts at addressing these challenges include Das et al. [Das et al., 2020] proposing a partially ML-based solution, Kim et al. [Kim et al., 2022b] leveraging RL to reduce the search space of PBQP, and the infrastructural Compiler-Gym [Cummins et al., 2022] approach to ease the RL-training process.

We propose a retargetable Reinforcement Learning (RL) approach to the REgister ALlocation (REAL) problem. We formulate a *multi-agent hierarchical reinforcement learning* optimization considering program-specific information: (1) to model the subtasks of register allocation like coloring, live range splitting and spilling, and (2) to encode the correctness constraints for preserving the semantics and hardware-compatible register assignments. The legality of the register allocations and assignments is preserved by imposing constraints on the action space, or outcome of each agent. As register allocation is a combinatorial problem, establishing the ground truth is hard, making RL a natural choice. It also facilitates the imposition of correctness constraints.

We leverage the LLVM infrastructure [Lattner and Adve, 2004] to build the first end-to-end RL framework addressing the above-mentioned challenges. The interference graph of a function is extracted from the Machine Intermediate Representation (MIR) of LLVM. Instructions within each node are represented as vectors using representation learning methods. For this purpose, we use MIR2Vec embeddings to represent MIR entities. These embeddings represent vertices of the interference graph that is traversed by RL agents. MIR2Vec embeddings are application-independent and may be used for other backend applications in the future. Finally, we propose LLVM-gRPC, a generic framework to facilitate communication between the RL model and the compiler during training and deployment. Our approach is architecture-independent; we show results on both Intel x86 and ARM AArch64. Further, we exploit transfer learning [Yosinski et al., 2014] to demonstrate the generalizability of trained models on new architectures.

Contributions

- We present RL4ReAl, the *first end-to-end application* of reinforcement learning to solve the register allocation problem, formulated as a multi-agent hierarchical RL problem.
- We ensure semantic correctness by encoding register allocation constraints as *action masks* that restrict agents to valid register assignments.
- We propose MIR2Vec embeddings to represent interference graph nodes, capturing MIR-level program semantics for RL agents.

```
// Source
i = 0
x = 10
y = 20
print x
z = y / x
i++
z = z + 10
i++
print y
print z
print i
```

```
MOV32ri 0, %i:gr32
MOV32ri 10, %x:gr32
MOV32ri 20, %y:gr32
<call print on %x>
eax = COPY %y:gr32
<clear edx>
IDIV32r %x:gr32, implicit-def ↩
    eax, implicit-def edx
%z:gr32 = COPY eax
%i:gr32 = ADD32ri %i:gr32, 1
...
<call print on %y, %z, %i>
```

Figure 7.1: (a) Example source code and (b) its Machine IR

- We demonstrate transfer learning enables 5× faster convergence when retargeting from x86 to AArch64, achieving better performance than training from scratch, along with a policy improvement cycle for continuous learning.
- We design LLVM-gRPC for compiler-model integration, which later evolves into the ML-Compiler-Bridge framework (Chapter 8).

Organization This chapter is organized as follows: Section 7.2 provides background on register allocation constraints and LLVM's register allocators, along with an overview of multi-agent hierarchical RL. Section 7.3 describes our RL formulation, including the environment, agent hierarchy, and reward structure. Section 7.4 presents the representation of interference graphs using MIR2Vec embeddings and GGNNs. The subsequent section describes LLVM-gRPC, our infrastructure for compiler-model integration, which forms the foundation for the ML-Compiler-Bridge framework presented in Chapter 8. Section 7.6 presents experimental results on x86 and AArch64, including transfer learning and policy improvement experiments. The chapter concludes with related work and a summary.

7.2 Background and Mathematical Model

We formulate the register allocation problem by defining different constraints. We also give an overview of LLVM register allocators and multi-agent RL.

7.2.1 Register Constraints

Optimizing compilers convert the source code into an Intermediate Representation (IR) where most target-independent optimizations take place. In the backend compiler, this IR is incrementally lowered to a machine-specific form. This representation in LLVM is called Machine IR (MIR). MIR at the stage of register allocation is very close to machine instructions, as instruction selection and other low-level optimizations have already been performed. After instruction selection, certain physical registers that are mandated by the architecture are immediately assigned. For instance, x86 processors mandate the output of 32-bit division to be stored only in `$eax` and `$edx` registers.

As shown by the example in Figure 7.1(a), `IDIV32` instruction divides the contents of `$eax` and `$edx` by `%x` and stores the result in `$eax`. Such mandatory assignments including calling conventions are made. Regalloc can now be reduced to assigning physical registers to the other *left-out* virtual registers ($\mathcal{V}$) while respecting the following constraints.

Type Constraint The register file ($\mathbf{R}$) of a machine consists of a collection of registers R^t belonging to different *types* (t): $\mathbf{R} = \bigcup_t R^t$. Assigning a physical register r to a virtual register v of type t, $v^t \blacktriangleright r$, should satisfy the register *type* constraint $\chi^{\mathcal{T}}(v^t) = \{v^t \blacktriangleright r : r \in R^t\}$.

> **Example 7.2.1:**
>
> In Figure 7.1(b), virtual register `%x` is of type `gr32` (32-bit general-purpose). It can only be assigned to physical registers of that type, such as `$eax`, `$ebx`, or `$ecx`, and not to other types.

Each virtual register is associated with a particular register type, and only registers belonging to that type can be assigned.

Congruence Constraint Real-world instruction set architectures like x86 and AArch64 have a hierarchy of register *classes*. For instance, 32 bit type of registers (like `$eax, $ebx`) are physically *part of* the 64 bit ones (like `$rax, $rbx`). We consider the registers that adhere to this *part of* relation as a congruence class $\mathcal{C}(r)$.

Example 7.2.2:

In x86, registers `$al`, `$ah`, `$ax`, `$eax`, and `$rax` are "chunks" of the same physical register, forming a congruence class with the hierarchy: `$al`, `$ah` $\sqsubseteq$ `$ax` $\sqsubseteq$ `$eax` $\sqsubseteq$ `$rax`. If a variable is assigned to `$eax`, no simultaneously live variable can use `$rax` or `$ax`.

The assignments for virtual register v should be among the set of registers that satisfy the following *congruence* constraint $\forall r' \in \mathcal{C}(r)$:

$$\chi^{\mathcal{C}}(v^t) = \{v_i^t \blacktriangleright r : \forall v_i, v_j \in \mathcal{V}, v_i \neq v_j, \nexists v_j \blacktriangleright r' \in L(v_i)\}$$

Here $L(v)$ corresponds to the live range of variable v, and is computed as $L(v) = [P_v^{def}, P_v^{end}]$; the definition of v occurs at program point P^{def}, and its last use is in P^{end}. Figure 7.1(a) gives an example. The live ranges of the corresponding variables are shown in Figure 7.2(a).

Interference Constraint Register allocation has been modeled as a graph coloring problem [Chaitin et al., 1981]. For each function in the program, it involves creating an *Interference graph* $G(V, E)$ defined as follows: the vertices of the graph are mapped to virtual registers (v) or physical registers (R_a), meaning $V \in (\mathcal{V} \cup R_a)$; the edges E are computed as $\{(v_i, v_j) : v_i, v_j \in V \wedge L(v_i) \cap L(v_j)\}$.

Definition 7.1: Interference Graph

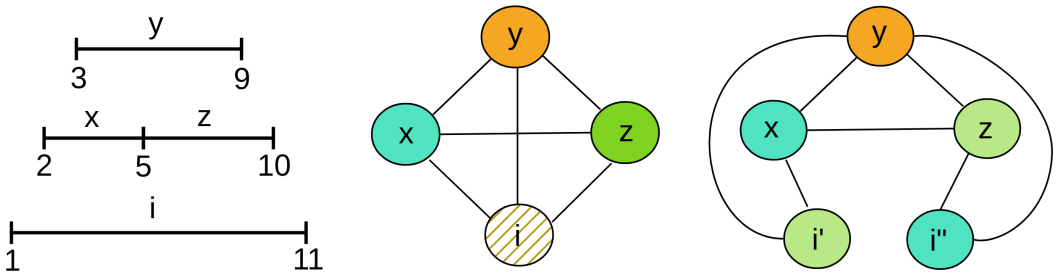
An *interference graph* $G(V, E)$ for a function represents the constraints on register assignment: vertices correspond to virtual and physical registers, and an edge (v_i, v_j) exists if and only if the live ranges of v_i and v_j overlap. Two adjacent vertices cannot be assigned the same physical register.

Example 7.2.3:

In Figure 7.2(b), variables `x` and `y` are connected by an edge because their live ranges overlap (both are live at program points 3–5). Therefore, `x` and `y` must be assigned to different physical registers.

The *interference* constraint says that no two adjacent nodes in G should be allocated the same color. The set of registers satisfying this constraint is given by:

$$\chi^{\mathcal{I}}(v^t) = R^t \setminus \{r : \forall u((u, v) \in E \wedge (u \blacktriangleright r))\}$$



(a) Live Intervals

(b) Without splitting i - one spill

(c) After splitting i - no spills

Figure 7.2: Register allocation with and without splitting

In summary, for a given virtual register v of type t, the set of available registers for allocation $\chi(v^t)$ is defined as the set of registers that satisfy the (i) type, (ii) congruence, and (iii) interference constraints:

$$\chi(v^t) = \chi^{\mathcal{T}}(v^t) \cap \chi^{\mathcal{C}}(v^t) \cap \chi^{\mathcal{I}}(v^t)$$

7.2.2 Live Range Splitting and Spilling

The above formulation of register allocation in terms of graph coloring is well known and natural, as a *decision problem*. Yet register allocation as an *optimization problem* is actually *much more* than graph coloring. For instance, when there are not enough physical registers available, deciding which variable (virtual register) has to be spilled to memory is important, as memory accesses take far more time than register accesses.

Definition 7.2: Spilling

Spilling a variable v, denoted $\mu(v)$, involves storing its value to memory and reloading it when needed. Each spill incurs a *spill cost* $M(v)$ based on access frequency. Register allocators minimize both the number of spills and total spill cost.

Example 7.2.4:

A loop induction variable accessed in every iteration would incur high spill cost if stored to memory. For a machine with only 3 registers, the example in Figure 7.1 is not *3-colorable*, requiring at least one variable to be spilled.

Hence, register allocators try to reduce the spill cost $M(v)$, in addition to minimizing the number of spills.

Definition 7.3: Live Range Splitting

Splitting a live range $L(v)$ at program point k creates two shorter live ranges: $L(v') = [P_v^{def}, P_v^k]$ and $L(v'') = [P_v^{k+1}, P_v^{end}]$. This can reduce interference and make previously uncolorable graphs colorable, avoiding spills.

Example 7.2.5:

In Figure 7.2(b), splitting variable i at program point 5 into (i', i'') reduces interference, making the graph *3-colorable* without requiring any spills.

Determining which variable to split, and at which point, is non-trivial.

7.2.3 Register Allocators in LLVM

The LLVM compiler currently has four register allocators: Fast, Basic, Greedy, and PBQP, ranked according to implementation complexity. They are implemented as passes and operate on one function at a time.

Fast is an improved version of the linear scan algorithm [Poletto and Sarkar, 1999] and operates at the basic block level. Basic is an improved variation of Fast, and operates at the function level [Xavier et al., 2012]. Greedy was developed [Jakob, 2011] to address the shortcomings of Basic; it iteratively combines four strategies: splitting, spilling, coalescing (merging of live ranges), eviction (de-allocating the already-allocated physical register). Each of these strategies is driven by greedy heuristics. Greedy is a complex, highly tuned, default register allocator at `-O3` optimization level. It iterates over the virtual registers in multiple rounds and obtains a legal physical allocation if possible. It includes *highly tuned heuristics* (i) for placing of proper spill code, (ii) to minimize the load-store instructions while favoring register moves by applying strategies like node-splitting/eviction/last-chance-recoloring, etc. PBQP is the only solver-based mechanism in LLVM, and models it as a Partitioned Boolean Quadratic Problem to obtain allocations [Hames and Scholz, 2006]. These allocators are a result of significant (*man-decades*) amount of engineering by expert compiler writers; and are continually improved to address the regressions on a case-by-case basis. Not all allocators implement all strategies; live range splitting only takes place in Greedy, whereas coalescing is present in Greedy and PBQP, and eviction is in iterative allocators like Greedy and not PBQP.

7.2.4 Multi-Agent Hierarchical RL

Reinforcement learning (RL) is a branch of machine learning that often tries to solve the problems where enumerating ground truth is either hard or infeasible. The learning happens with *experience* where the agent learns a policy to determine the best possible action based on its *observation* from the *environment*. Depending on the *goodness* of action, the environment gives positive or negative rewards so as to course-correct the learning. With the evolution of DL, methods like PPO [Schulman et al., 2017] that use gradient-based approaches to learn better policy have become prevalent.

Depending on the problem formulation, there can be single or multiple agents to solve the problem. When the problem is modeled with multiple agents, it is called *Multi-Agent Reinforcement Learning* (MARL). If all the agents work together towards a common goal, learning is said to be cooperative. If they compete against each other to achieve a goal, the learning is said to be competitive. In certain cases, there can be a mix of both of these.

> **Definition 7.4: Hierarchical Multi-Agent RL**
>
> In *hierarchical* multi-agent RL, agents are organized in levels where higher-level agents determine which lower-level agents act and when. In *cooperative* MARL, all agents work toward a common goal, sharing rewards based on the final outcome.

MARL is an active field of research that has accomplished huge success in gaming [Silver et al., 2017], robotics [Kalashnikov et al., 2018], navigation [Almasan et al., 2020], and autonomous driving problems [Kiran et al., 2021].

We formulate register allocation as a number of smaller subproblems using multiple agents to model node selection, task selection (among splitting, spilling, or coloring), splitting, and coloring. These agents work cooperatively to achieve a beneficial register allocation. Another categorization in the case of MARL problems is based on the schedule of the agents. If the agents act on the environment sequentially, it is the sequential variant of MARL. If the agents form a hierarchy, where the top-level agent determines how the agents at the lower level should act, the learning is said to be hierarchical. In RL4ReAl, task selection, splitting, and coloring are modeled in a hierarchical fashion.

7.3 Modeling RL4ReAl

We formulate register allocation as a Markov Decision Process (MDP) using hierarchical Reinforcement Learning (RL), modeling the sub-tasks of register allocation as lower-level tasks controlled by multiple agents. Figure 7.3 sketches the overall approach. It involves interactions between the LLVM compiler and the RL model for both training and inference.

7.3.1 Environment

We implemented a new `MLRegAlloc` pass in LLVM, to generate an interference graph (G), allocate, split and spill registers as predicted by the agents. This pass also generates a representation of G using MIR2Vec.

Definition 7.5: Action Masks

An *action mask* is a mechanism that restricts an RL agent's action space at each decision step to only *valid* choices. In register allocation, action masks encode the type, congruence, and interference constraints (Section 7.2.1), ensuring that agents can only select semantically valid registers. This guarantees correctness by construction, implementing the *constrained learning* strategy described in Section 5.3.

7.3.2 Agents

The task of allocating registers is split into multiple sub-tasks. Each of these tasks are modeled as agents $\{\omega_\upsilon, \omega_\tau, \omega_\varphi, \omega_\xi\} \in \Omega$, that learn their respective policies π_ω to optimally solve the low-level tasks. We formulate hierarchical agents for four sub-tasks as shown in Figure 7.3:

- Node selector (ω_υ): Top-level agent that learns to pick a node $v \in G$.
- Task selector (ω_τ): Mid-level agent that learns to select a task among $\{\chi, \varphi\}$ on v picked by ω_υ.
- Splitter (ω_φ): Low-level agent that learns to identify a split point k for v.
- Coloring Agent (ω_ξ): Low-level agent that learns to pick a valid color $\chi_i \in \chi$ or spill μ.

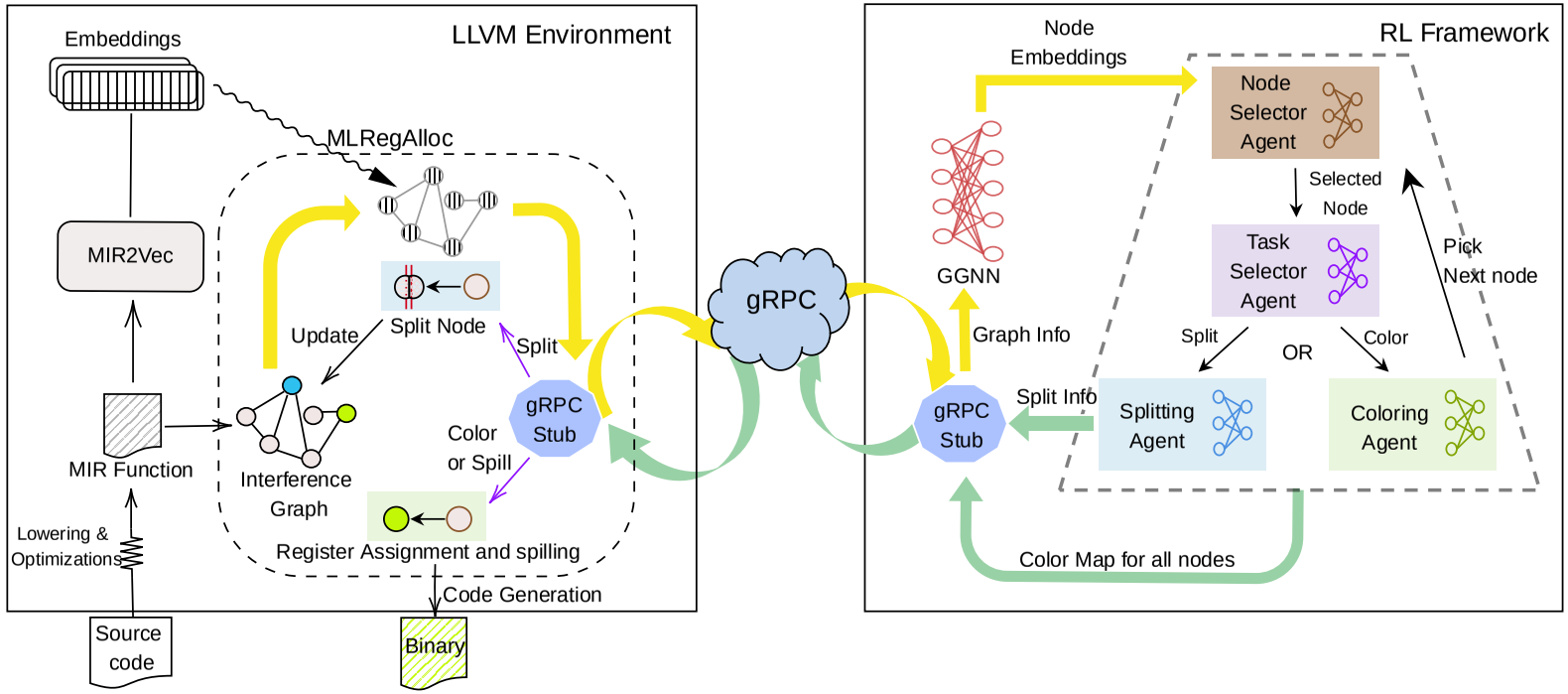


Figure 7.3: Overview of RL4REAL: The interference graph (G) generated from MIR is represented as MIR2VEC vectors and is passed as input to the RL Framework via LLVM-gRPC The agents in the RL framework perform splitting/coloring on G, and the register assignments (colors) are communicated back to LLVM via LLVM-gRPC. In case of splitting a vertex in G, the split points predicted by the agent are passed to the LLVM environment to perform splitting, and the updated graph is sent back to the model as response.

As it can be seen, each high-level agent invokes a low-level agent while following the timeline: $\omega_v \prec \omega_\tau \prec \{\omega_\varphi, \omega_\xi\}$. Each agent ω has its own state space S_ω, action space A_ω, and reward R_ω to learn a policy π_ω.

Coloring Agent (ω_ξ) In case of regular architectures like x86 and AArch64, all the registers of the same type/class have equal effect on the performance. However, the standard register allocators prefer some registers over others within the same class while considering several heuristics. One of the predominantly used heuristic is to give preference to the physical registers that are not (aliases of) callee saved registers. This is achieved by deriving an ordering of physical registers that can be allocated to a virtual register while satisfying the constraints described in Section 7.2.1. The goal of this ordering is to reduce the number of register moves. Hence, we design a simple model that can act as a coloring agent to learn the beneficial color assignments.

For a graph of V nodes and a set of registers $\chi(v)$ available at the instant, the state space of ω_ξ is given as a tuple $\langle [\![v]\!], |\chi(v)|, |V_{\text{nclr}}| \rangle$, where $V_{\text{nclr}} = V \setminus V_{\text{clrd}}$ are the nodes to be colored, v is the node that is picked by ω_v, and $[\![.]\!]$ denotes its embedding. Meaning, the coloring agent uses the following information to decide the register to be assigned: the embedding of the vertex v, the number of registers satisfying the constraints (see Section 7.2.1), and the number of uncolored nodes in G. If no registers are available, the coloring agent marks v for spilling. Hence the legal action space of ω_ξ is:

$$A(\omega_\xi) = \begin{cases} \chi(v), & |\chi(v)| > 0 \\ \mu(v), & \text{otherwise} \end{cases}$$

$\chi(v)$ gives the set of legal registers for v (Section 7.2.1). To improve performance, the agent should maximize the use of registers for vertices with higher spill weight. And, spill weight roughly corresponds to the importance of the node v. Hence the reward for the coloring agent is given as:

$$R(\omega_\xi) = \begin{cases} +M(v), & \text{if } \chi(v) \\ -M(v), & \text{if } \mu(v) \end{cases}$$

In our experiments, spill weight M is estimated using the *spill costs* computed by LLVM.

Splitter (ω_φ) Live range splitting $\varphi(v, k)$ corresponding to a variable v involves inserting a `move` instruction at the split point k and creating two new live ranges $L(v')$ and $L(v'')$ in place of a single original live range $L(v)$ as explained in Section 7.2.2. Selecting an appropriate split point plays an important role in making effective spilling and coloring decisions. For instance, the GREEDY in LLVM *greedily* selects the split points so as to carve out a region that can be allocated a register, while the other parts are spilled. In our model, the splitting agent predicts the split point in the live range among all the points of use.

Inserting the `move` instructions can be seen as a dataflow problem, that is analogous to the `phi` (`copy`) placement while creating (going out-of-) SSA form. We use dominance frontiers to place the `move` instructions appropriately to preserve the correctness. In Algorithm 5, we show how the move instructions are inserted. This algorithm directly builds from earlier works [Bouchez et al., 2006; Brisk et al., 2005; Hack et al., 2006] and dominance property [Cytron et al., 1991], so its soundness can easily be proved.

Algorithm 5: move-placement in live range splitting

Parameter: Virtual register v, Split point k
Rename $v \rightarrow v'$
At use point k do: $v'' \leftarrow \texttt{move}(v')$
Basic block $B \leftarrow block(v_k)$
for $i \in DominanceFrontier(B)$ **do**
    $v' \leftarrow \texttt{move}(v'')$, after last use$(v')$ in i
    Rename $v' \rightarrow v''$, $\forall use(v')$ between B and i

For predicting where to split the live range of a variable v, the node splitter considers the spill weights at each use of the variable $\mathcal{M}(v) = \{M(v_k) : \forall k \in K\}$, the distances between each successive use $D_v = \{D(v_i, v_{i+1}), \forall i, i+1 \in K\}$, and the embedding $[\![v]\!]$ of v. The use distance is the number of program points between two uses of v. Hence, the state space is given as a tuple $\langle [\![v]\!], \mathcal{M}(v), D_v \rangle$. For a given state, the agent learns an optimal program point $p \in K$ where v can be split. Hence the action space $A(\omega_\varphi) = K$.

The reward for the agent on splitting v into (v', v'') is based on the difference in spill weights of the variable before and after splitting. The agent gets a positive reward on reducing the sum of the spill weights, which indicate a reduction in complexity of coloring.

$$R(\omega_\varphi) = M(v) - \sum_{i \in \{v', v''\}} M(i)$$

Task Selector (ω_τ) For selecting a task (τ) among coloring and splitting, the agent ω_τ considers the parameters specific to each of the tasks: the representation of v, the number of available registers, the number of interferences, its life-time, spill weight. Hence the state space is formulated as the tuple: $\langle [\![v]\!], |\chi(v)|, \delta(v), |K(v)|, M(v)\rangle$. The action space of ω_τ is defined as:

$$A(\omega_\tau) = \begin{cases} \{\varphi, \chi\}, & |K(v)| \geq k \\ \chi, & otherwise \end{cases}$$

Here $|K(v)| \geq k$ indicates that v should have at least k uses to be considered for splitting. We define k as a hyper-parameter. We set $k = 2$ (1 definition and 1 use) in our experiments. We model the reward for this agent based on the outcome of the low-level tasks.

$$R(\omega_\tau) = \begin{cases} R(\omega_\xi), & \tau = \chi \\ 0, & \tau = \varphi \end{cases}$$

On choosing to color, the agent gets a reward based on coloring decision of ω_ξ. However, if the agent chooses to split, we defer from giving a reward as the goodness of the outcome is not known till a coloring decision is made.

Node Selector (ω_υ) It is well known that the order of picking a variable for allocation would highly affect the final outcome of register allocation. Usually, the iterative allocators use a priority queue to determine the order of allocation. The priority is derived from different heuristics including the spill cost of the variables, size of the live ranges, among others. We use a model (ω_υ) to figure out this order of allocation by predicting the variable/vertex to process (split/color) at every step of allocation after processing the previous vertex.

The state space of ω_υ comprises the embedding of each vertex in G obtained from a Gated Graph Neural Network (GGNN) as explained in Section 7.4. Along with these embeddings, the agent uses the spill weights of the nodes M to characterize the state. Hence, the state space is given as a tuple $\langle [\![G]\!], M(V)\rangle$ Its legal action space is $A(\omega_\upsilon) = V_{\text{nclr}}$. The learned policy is deemed *good* based on the final coloring decision of the node. Hence, the reward for this agent is also modeled based on the rewards of the coloring

agent (ω_ξ):

$$R(\omega_v) = \begin{cases} R(\omega_\xi), & \tau = \chi \\ 0, & \tau = \varphi \end{cases}$$

Global Rewards In addition to providing rewards to the agents at each step, we derive a global reward based on the throughput of the generated function estimated by LLVM-MCA [LLVM, n.d.]. The global reward is computed based on the difference between the throughputs of the code generated by RL4ReAl ($Th_{RL4ReAl}$) and Greedy (Th_{Greedy}) as:

$$R_G = \begin{cases} +10, & Th_{RL4ReAl} \geq Th_{Greedy} \\ -10, & Otherwise \end{cases}$$

This way of using throughput helps capture the overall impact of the allocation scheme in comparison with Greedy.

7.4 Representing Interference Graphs

Building on the program embedding foundations from Part I, we represent nodes of the interference graph using MIR2Vec embeddings. As described in Section 3.4 on page 28, MIR2Vec extends the IR2Vec methodology to LLVM's Machine IR, providing learned representations of MIR entities (opcodes, registers, operands) that capture semantic relationships at the backend level. These embeddings form the input to a Gated Graph Neural Network (GGNN) that learns to generate the representation of the state space for the RL agents.

Interference Graphs As mentioned earlier, the MIR at the stage of register allocation contains partial physical register assignments and virtual registers. The physical registers are assigned for the instruction operands that have restrictions on the particular register to be used. Virtual registers are used in all other places. Consequently, we need to take into account the edges corresponding to both virtual and physical registers. Virtual registers are marked with the register class so that assignments can only be one among the physical registers in that class.

For computing G, considering the interferences between the (physical $\longleftrightarrow$ virtual) and (virtual $\longleftrightarrow$ virtual) registers is sufficient as G is bidirectional and we do not need

to worry about the physical registers that are already assigned.

We use a collection of instructions in the live-range of a variable to represent a vertex of the interference graph. Each instruction is *represented* in $\mathbb{R}^n$ using MIR2Vec embeddings. Consequently, a vertex v is represented as a matrix of embeddings $[\![v]\!]$ in $\mathbb{R}^{m \times n}$, where m denotes the number of instructions in its live range.

Gated Graph Neural Networks (GGNNs) are widely used in programming language modeling [Cummins et al., 2021; Mendis et al., 2019b], especially when the inputs are modelled as graphs. GGNNs involve message passing between the nodes of the graph. Information propagates across multiple nodes to arrive at the representation for a given node. Also, they allow annotating the nodes and edges based on their types and properties, and consider these while learning a representation. We use GGNNs to process the embedded interference graph to get the final representation. This network transforms $\mathbb{R}^{m \times n} \rightarrow \mathbb{R}^d$. We set $d = n$ in our experiments and annotate the graph with the following node types, along with spill weights.

- **Not visited** — nodes that are not visited yet.
- **Spill** — nodes that are marked as spill.
- **Colored** — nodes that are assigned a register.

Such node representations are propagated through a GGNN by means of message passing. Messages received from adjacent nodes are aggregated and passed through a Gated Recurrent Unit [Cho et al., 2014] to yield a final representation.

7.4.1 Generalization: Transfer Learning and Policy Improvement

A key advantage of our RL formulation is that it enables *generalization* across architectures, workloads, and time—the third component of the ML-driven compiler optimization schema (Section 5.3). test

Retargetability via Transfer Learning The hierarchical agent structure is largely architecture-agnostic: the node selector, task selector, and splitter agents operate on interference graph structure and MIR2Vec embeddings, which encode program semantics independent of the target architecture. Only the coloring agent's output layer is architecture-specific, as it must match the target register file size. This design enables

transfer learning: a model trained on one architecture (e.g., x86) can be adapted to a new architecture (e.g., AArch64) by retraining only the coloring agent's final layer, while transferring the learned policies for node selection, task selection, and splitting. We validate this in Section 7.6.

Policy Improvement Cycle Traditional compiler heuristics are refined iteratively by experts who identify regression cases and tune the heuristics accordingly. Our RL framework enables a similar *policy improvement cycle*: when regression cases are identified during deployment, the learned policy can be fine-tuned on these specific cases to recover performance. This supports continuous learning without retraining from scratch, as the policy retains its general knowledge while adapting to new edge cases. We demonstrate this capability in Section 7.6.

7.5 Compiler Integration using gRPC

The integration of DL/RL models into optimizing compilers can be a hurdle, for both research and practical deployment. Previous works on ML driven vectorization [Haj-Ali et al., 2020] or phase ordering [Jain et al., 2022a] are adeptly controlled through optimization flags, carrying information from the predictions of the model to the compiler. Register allocation, however, poses an inherent difficulty as the ML-decisions and the optimization algorithm are deeply intertwined; the model predicts the final allocation based on the compiler generated code from its splitting decisions. One naïve approach is to rely on Python bindings for integration; however, this involves large overhead.

We propose LLVM-gRPC, a novel infrastructure for efficient communication between the Python model and C++ compiler to support *both training and inference*. LLVM-gRPC involves gRPC calls [gRPC, 2016] with the LLVM toolchain, leveraging its modular structure as an LLVM library. This gives the end-user the option of designing custom RPC calls that can operate on any of the module, basic-block, loop or function of the input program. LLVM-gRPC allows bi-directional communication between ML models and the compiler, during both training and inference. To our knowledge, this facility is not available in other frameworks.

We use protocol buffers [Protobuf] to describe the structure and serialize the data for communication. The proto-files are auto generated as headers that are linked with LLVM-gRPC. This library exposes the components to facilitate the communication with various libraries of LLVM by providing an option to either run a gRPC service, or to

connect to an existing one. RL4ReAl uses the above facility to create a custom gRPC service from within Clang. The backend then uses this gRPC service to communicate with the RL environment for either training, or inference. Our communications use RPC calls in synchronous mode that operates by inducing a blocking call, and waiting for a response for every message; the compilation is blocked till the allocation information is received from the model.

In inference, our `MLRegAlloc` pass (client) raises a request to the model (server) by sending the interference graph (G), along with the embeddings. When a decision to split a vertex is made, the model responds back with splitting request containing the split point, and the corresponding virtual register. The backend then materializes the splitting decision, and sends updates to the original G. This update contains the vertices and edges that are newly created as a result of splitting. Upon receiving this information, the model updates the interference graph using the received information continues the traversal. On processing all vertices of G, the coloring decisions are communicated to the compiler as a color map containing the coloring information corresponding to each virtual register. `MLRegAlloc` now assigns the physical registers as per the predictions, and generates the code. In case of the training flow, in addition to materializing the decisions made by the model, `MLRegAlloc` also responds with appropriate rewards to facilitate learning.

For example, a splitting decision by the model is communicated to the compiler, which then applies it and responds back with the update containing new interferences and live ranges. The model then updates the interference graph using the received information and continues the traversal. After processing all vertices of G, all the coloring decisions are communicated to the compiler as a color map. The backend compiler uses the allocation decisions made by the model to generate code.

We collect various metrics viz Average data sent/received per call, Max & Min data sent/received along with the total number of calls made between the compiler and RL4ReAl from our framework when inference is performed on SPEC 17. The *average data sent(received) per call* lies in the range of 50 KB–553 MB (20 Bytes–39 KB). The maximum(minimum) data sent from the compiler was 56 MB (60 Bytes). Likewise, the maximum(minimum) data received was 3.60 KB (10 Bytes).

On an average, about 100KB of data per gRPC call were transferred via LLVM-gRPC while compiling SPEC 2017 benchmarks, where the maximum (minimum) data transferred was 56 MB (10 Bytes). The larger values are for graphs of 500 vertices, along with embeddings.

7.6 Experimental Evaluation

We first discuss the experimental setup, followed by a characterization of the benchmarks. Then, we report the results on x86, followed by the results on AArch64 and an explanation of the results. Finally, we report improvements on the regression cases using policy improvements.

7.6.1 Setup

For training MIR2Vec representations, we randomly select 2K source files from SPEC CPU 2017 benchmarks and C++ Boost library. MIR triplets are generated by applying `-O3` optimization flag. The seed embedding vocabulary is obtained by training a TransE model [Bordes et al., 2013] on generated triplets (Section 2.3), by running an SGD optimizer over 1000 epochs to obtain an embedding vector of 100 dimensions. We obtain 1 Billion MIR triplets from which $\{675, 315\}$ entities and $\{25, 17\}$ relations are generated for $\{\text{x86}, \text{AArch64}\}$ respectively.

We target a complex x86 (Intel Xeon SkyLake W2133, 6 cores, 32GB RAM), and a simpler mobile AArch64 (ARM Cortex A72, 2 cores, 8GB RAM) processors. We consider allocations of general purpose, vector, floating point registers for both x86 and AArch64 (listed in Table 7.1); other registers like `eflags` are pre-assigned before any register allocation.

Table 7.1: Allocatable Registers in x86 and AArch64

Arch.	**Registers**
x86	[A-D]L, [A-D]X, [E,R][A-D]X, [SI,DI]L, [E,R][SI,DI], SI, DI, R[8-15][B,W,D], FP[0-7], [X,Y,Z]MM[0-15]
AArch64	[X,W][0-30], [B,H,S,D,Q][0-31]

Our framework is implemented as a pass `MLRegAlloc` in LLVM 10.0.1, using gRPC v1.34. We train the RL models using the PPO policy with the standard set of hyperparameters on the training set of functions selected from SPEC CPU 2017, until convergence of reward graph. There were about 11K and 30K functions (with 120–500 vertices) in SPEC CPU 2017 benchmark suite in x86 and AArch64 respectively, out of which we choose 5K functions at random (from SPEC 2017) for training. Training was done on a 32GB Tesla V100 GPU and sampling was done using 12 threads of a server with Intel Xeon Platinum 8168 processors.

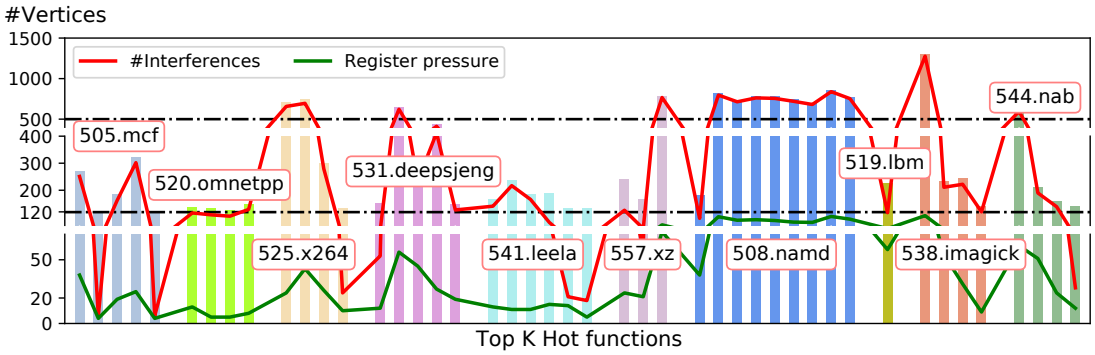


Figure 7.4: Correlation between #Vertices, #Interferences and Register Pressure

Model Architecture Each agent learns a policy depending on its model architecture. For GGNN, we use two fully connected (FC) layers to normalize the input, followed by a RNN layer for message passing. For the agents, we use simple neural networks with FC layers: node selector and splitter use four FC layers each, while task selector and coloring agents use three FC layers each with batch normalization. Everywhere, ReLU is used as the activation function.

Benchmarks In our experiments on x86, we consider C/C++ benchmarks with less than 1 MLOC (1 Million Lines of Code) from SPEC CPU 2006 and 2017. This constitutes (8 Int + 5 FP) 13 benchmarks in SPEC CPU 2006, and (8 Int + 5 FP) 13 benchmarks in SPEC CPU 2017 benchmark suites. We were able to successfully compile the benchmarks listed in Table 7.2. We observed 4 compilation errors, and 4 runtime errors due to the engineering and integration issues.

7.6.2 Characterization of Benchmarks

Let us now characterize the SPEC CPU 2017 benchmarks for choosing the graphs for experimentation. We study the 45 *hot* functions (profiled with the `perf` tool [Linux-Foundation, n.d.]) that take at least 5% of the total execution time of a benchmark. The number of vertices in the interference graphs of these hot functions along with the number of interferences and register pressure is shown in Figure 7.4. We compute register pressure as the maximum number of overlapping live ranges across all program points of a function.

It can be observed that out of 45 hot functions, 44 have more than 120 vertices in their interference graphs, while 31 (the majority) of them have between 120 and 500

vertices. It can also be seen that the graph size has a strong correlation with the number of interferences and the register pressure. Meaning, the vertices and edges show linear correlation (not quadratic), and the vertices and register-pressure also show near linear correlation (not quadratic) [Bouchez et al., 2006].

As the number of vertices and the register-pressure are positively correlated with each other, in general the graphs with higher number of vertices are harder to allocate. Hence, we consider the functions that have at least 120 vertices for allocation through RL4ReAl. We limit the maximum number of vertices to be 500, as most of the hot functions are in this range (120–500), and also for ease of training. Functions that are not in this range are processed using Greedy.

7.6.3 Runtimes on x86

We study the SPEC CPU 2017&2006 benchnmarks and compare the results with LLVM's allocators. Basic, Greedy and PBQP generally outperform the Fast allocator. However, there is no single allocator among these three that perform the best for *all* programs. Hence, we compare our results with these three allocators.

In Table 7.2, we show the runtimes obtained by Basic and the improvements obtained over it by other allocators for each benchmark. These are obtained by taking the median of three executions. Positive (negative) numbers indicate speedups (slow-downs) over Basic. For RL4ReAl, we train two models: one trained only using local rewards (L) and another with local along with the global reward (G).

On average, RL4ReAl-L (RL4ReAl-G) yields about 19s (17s) improvement over Basic; Greedy results in an improvement of about 22s. RL4ReAl-L (RL4ReAl-G) shows speedup over Basic in 14 (11) out of 18 benchmarks. The highest and second highest improvements in runtimes over Basic are highlighted in Table 7.2. In particular, RL4ReAl (L or G) results in highest or second highest improvements over Basic in 17 out of 18 benchmarks. As it can be seen, the runtimes obtained by our framework are very close to Greedy. In comparison, RL4ReAl-L (RL4ReAl-G) results in an improvement on 5 (6) benchmarks over Greedy. And those with slow-downs, runtimes of 12 (11) benchmarks are within 1% of Greedy, and only 1 show more than 4% slow-down.

To obtain confidence intervals, we ran benchmarks 8 times and observed that the noise was under 1% consistently across all benchmarks for a 95% confidence interval, except for `libquantum`, where the noise was 1.8% (1.2%) in RL4ReAl-L (RL4ReAl-

Table 7.2: Runtime improvement (s) on x86 over BASIC, highlighting the Highest and Second highest improvements.

Benchmarks	Runtime BASIC	Diff. from BASIC (BASIC- x)			
		PBQP	GREEDY	RL4REAL	
				L	**G**
401.bzip2	360.6	-7.3	7.5	-1.1	10.8
429.mcf	233.8	1.4	-2.9	2.7	-3.6
445.gobmk	322.3	-3.3	6.4	2.4	1.7
456.hmmer	284.3	1.8	6.1	5.0	-37.6
462.libquantum	256.4	-10.1	-1.1	-2.2	-6.7
471.omnetpp	305.7	0.7	0.4	1.2	1.2
433.milc	349.1	-16.6	0.1	-13.8	-7.0
470.lbm	184.0	-7.9	3.0	2.3	1.4
482.sphinx3	366.0	-37.5	1.6	-3.1	-2.7
505.mcf_r	344.9	4.5	-1.6	8.6	-4.7
520.omnetpp_r	475.7	6.4	6.4	2.4	2.8
531.deepsjeng_r	299.9	4.6	16.0	9.9	12.8
541.leela_r	439.5	1.6	7.1	0.4	1.9
557.xz_r	371.5	-0.6	11.9	12.1	-8.5
508.namd_r	236.5	2.5	23.5	9.1	23.8
519.lbm_r	261.8	1.4	57.7	50.9	58.1
538.imagick_r	479.3	-16.9	115.5	118.8	118.4
544.nab_r	417.5	5.8	132.1	131.3	134.4
	Average	-3.9	21.7	18.7	16.5

G). We also have empirical results on numerical kernels. On PolyBench [Pouchet et al., 2018] benchmark, our results show similar performance: RL4REAL obtains an average runtime of 43.5s (3.6s) in comparison 43.6s (3.6s) obtained by GREEDY on Extra-Large (Large) input size.

> **Takeaway 7.1: Competitive with Expert Heuristics**
>
> RL4REAL *achieves performance comparable to LLVM's production Greedy allocator—refined over decades of expert effort—while being learned automatically, validating that learned models can match and outperform hand-crafted heuristics for complex semantically constrained compiler optimizations.*

Table 7.3: % runtime diff. over Basic on hot functions

	SPEC CPU 2017			SPEC CPU 2006		
	Greedy	RL4ReAl		Greedy	RL4ReAl	
		L	**G**		**L**	**G**
Average	6.2	**7.3**	4.8	**-1.5**	-2.1	-1.6
# (val>0)	**23.0**	**23.0**	17.0	16.0	**17.0**	13.0
# (val<0)	**8.0**	**8.0**	14.0	19.0	**18.0**	22.0
Max	44.0	**44.4**	41.3	**12.7**	10.4	6.2
Min	-7.7	**-4.4**	-10.8	-51.4	-52.5	**-13.1**

Table 7.4: % speedup of Greedy and RL4ReAl over Basic on SPEC 2006 and 2017 (x86)

B/M	**Functions**	Greedy	RL4ReAl	**Diff.**
Top 5 functions with highest % speedup (over Greedy)				
401	BZ2_compressBlock	-51.3	-5.2	46.1
445	do_get_read_result	-12.0	-0.5	11.5
482	mgau_eval	-6.0	0.3	6.3
429	price_out_impl	-0.8	2.3	3.2
445	subvq_mgau_shortlist	-9.8	-6.9	2.9
538	GetVirtualPixelsFromNexus	8.3	28.8	20.4
538	SetPixelCacheNexusPixels	4.7	21.9	17.2
505	cost_compare	-7.7	8.1	15.8
557	lzma_mf_bt4_skip	-1.8	3.63	5.5
525	biari_decode_symbol	-2.7	2.7	5.4
Top 5 functions with highest % slow-down (over Greedy)				
456	P7Viterbi	2.2	-13.1	-15.3
482	vector_gautbl_eval_logs3	11.9	-2.5	-14.4
401	mainGtU	0.3	-9.6	-10.0
401	fallbackSort	12.6	6.2	-6.4
445	fastlib	4.8	-1.1	-5.9
557	lzma_mf_bt4_find	1.5	-10.7	-12.3
531	feval	26.4	17.7	-8.6
505	primal_bea_mpp	0.9	-7.6	-8.5
541	FastBoard::self_atari	3.7	-0.1	-5.8
541	qsearch	6.6	1.5	-5.0

Analysis of Hot Functions We did a study at function level, focusing on the hot functions. There were 35 and 31 allocated hot functions in SPEC 2006 and 2017. In Ta-

ble 7.3, we show the percentage difference in runtime improvements obtained by GREEDY and RL4REAL allocators in comparison to BASIC. It can be seen that RL4REAL results in improvements on largest number of functions, and minimum number of slow-downs over BASIC.

On average, in SPEC 2017 RL4REAL-L improves over BASIC by 7%, while GREEDY results in a similar improvement of about 6%. When it comes to SPEC 2006, to our surprise, GREEDY did not show improvement on average runtime among the hot functions. It is mainly due to a 51.3% slow-down observed on `BZ2_compressBlock` from `Bzip2` benchmark. In comparison, RL4REAL-G results in a lesser slow-down of about 5% on this function.

In `imagick` benchmark, RL4REAL (both L and G) obtains improvements over GREEDY, on all the three allocated hot functions: On `GetVirtualPixelsFromNexus` and `SetPixel- CacheNexusPixels` functions, RL4REAL improves by over 29% and 22%, where GREEDY results in an improvement of about 8% and 5%; on `MeanShiftImage` function, GREEDY and RL4REAL show a similar improvement of about 44%. We list the top 5 hot functions that show highest % speedup and % slow-down in comparison to GREEDY from both SPEC 2006 and 2017 in Table 7.4.

7.6.4 Retargeting to AArch64 via Transfer Learning

As discussed in Section 7.4.1, our RL formulation is largely architecture-agnostic, enabling transfer learning across targets. We validate this by evaluating RL4REAL on AArch64, comparing models trained from scratch against models transferred from x86.

Experimental Setup We cross-compile SPEC CPU 2006 and 2017 benchmarks from x86 targeting the AArch64 board. We skip benchmarks like `perlbench`, `h264ref`, `xalancbmk`, and `sphinx3` due to known cross-compilation issues or failures with standard LLVM allocators. Runtimes correspond to the median of three runs; for cases with significant variance ($> 20s$), we use median of five runs.

Transfer Learning Process We adapt the model trained on x86 (Section 7.6.1) to AArch64 by reusing all agent weights except the final layer of the coloring agent, which must match AArch64's register file. The transferred model converges in about one day, compared to five days for training from scratch, resulting in a 5× speedup. This efficiency gain validates the architecture-agnostic design: the learned policies for

Table 7.5: Improvement in runtimes (s) on AArch64 over Basic allocator, comparing RL4ReAl trained from scratch vs. transferred from x86. Highest and Second highest improvements are highlighted.

Benchmarks	**Runtime** Basic	**Diff. from** Basic (Basic- **x**)			
		PBQP	Greedy	RL4ReAl	RL4ReAl-**TL**
401.bzip2	1366.9	-41.1	15.6	12.8	7.4
429.mcf	1320.5	-12.7	-7.5	1.6	5.9
445.gobmk	992.8	15.6	26.1	14.5	24.9
462.libquantum	1627.6	-8.7	4.5	9.6	11.7
433.milc	1251.1	59.2	70.9	45.4	61.0
444.namd	855.3	2.7	21.8	18.8	23.8
470.lbm	1604.3	-6.4	-16.6	16.0	75.9
505.mcf_r	1535.1	25.9	1.9	-12.8	-22.7
508.namd_r	845	0.4	34.5	40.1	40
523.xalancbmk_r	979.1	8.1	-3.4	4.4	0.3
531.deepsjeng_r	777.2	10.0	30.5	4.5	4.9
541.leela_r	1067.9	-11.3	-0.1	-19.5	-3.3
557.xz_r	1163.2	3.7	22.2	21.3	20.6
519.lbm_r	1657	50.9	-1.6	39.8	13.6
538.imagick_r	1244.5	-3.9	75.8	65.6	67.7
544.nab_r	1170.7	-7.7	31.5	32.4	21.5
	Average	5.3	19.1	18.4	22.1

node selection, task selection, and splitting transfer effectively, with only the register assignment layer requiring retraining.

Results Table 7.5 compares the runtime improvements on AArch64. RL4ReAl trained from scratch achieves an average improvement of 18.4s over Basic, comparable to Greedy's 19.1s. The transfer-learned model (RL4ReAl-TL) achieves 22.1s average improvement—*better* than both the from-scratch model and Greedy. This demonstrates that transfer learning not only reduces training time but can also improve performance by leveraging knowledge from the larger x86 training corpus.

Transfer learning is effective because the fundamental task structure—interference graphs, coloring, and spilling decisions—remains similar across architectures. The learned policies encode general strategies that transfer well, with architecture-specific details captured through the MIR2Vec embeddings and the adjusted coloring agent.

Takeaway 7.2: Cross-Architecture Transfer Learning

Transfer learning across architectures (x86→AArch64) converges 5× faster and achieves better performance than training from scratch, validating the architecture-agnostic design of the RL formulation.

7.6.5 Policy Improvement on Regression Cases

As described in Section 7.4.1, our RL framework supports a policy improvement cycle for continuous learning. Here, we validate this by fine-tuning the learned policy on identified regression cases.

In traditional compilers, heuristic tuning for optimization is an iterative process: human experts identify regression cases, and heuristics are tuned to identify cases of regression. Similar to this spirit, Trofin et al. [Trofin et al., 2021b] (MLGO), propose a *policy improvement cycle*, where the learned RL policy is fine-tuned to check if it fares well on the regression cases.

In this section, we attempt to tune the learned policy to evaluate if the regression cases can be improved. For this purpose, we identified poorly performing benchmarks from each configuration: `milc` on RL4ReAl-L, `hmmer` and `xz` on RL4ReAl-G. The learned model is then retrained (fine-tuned) on the hot functions of these benchmarks. Upon training, we observe a positive improvement on all three regression cases: 12s on `milc`, 10s and 6s on `hmmer` and `xz` benchmarks.

The learned model is then retrained (fine-tuned) on the hot functions of these benchmarks. Upon training, we observe a positive improvement on all three regression cases: 12s on `milc`, 10s and 6s on `hmmer` and `xz` benchmarks.

This experiment makes a strong case for online or continuous learning [Alpaydin, 2010], where the learning continues during deployment for betterment of policy.

Takeaway 7.3: Continuous Policy Improvement

Fine-tuning the learned policy on regression cases recovers 6–12 seconds of runtime per benchmark, demonstrating that the generalization framework enables iterative refinement—mirroring how compiler engineers tune heuristics.

7.6.6 Discussion

We demonstrate performance results *on par with the best allocators* currently available in LLVM: RL4ReAl is *most frequently the best or second best allocator*, and there is *no single allocator* that performs best across all benchmarks (see Tables 7.2 and 7.5). It is well known that register allocation is one of the hard compiler optimization problems, and the baseline heuristics achieve excellent results that cannot be easily improved upon in terms of wall-clock time. For example, the studies of Pereira et al. [Quintão Pereira and Palsberg, 2008], Shin et al. [Shin and Sung, 2021], Kim et al. [Kim et al., 2022b], and the report on the PBQP solver [Community and conference, 2014] were *not able to* significantly outperform baseline heuristics across benchmarks, and they report performance numbers in the *same ballpark* as the ones that we obtain.

Here, we consider the major sub-tasks/strategies of register allocation: coloring, splitting and spilling, with a focus on building the first end-to-end RL model integrated with LLVM. As mentioned earlier, Greedy also admits register coalescing as one of its strategies. We consider coalescing along with other possible strategies like multi-allocation, register packing, spilling to vector registers, as possible incremental extensions for a future work. These additional strategies could result in further runtime improvements. It could however be noted that even without admitting these additional strategies—and just relying on the ones that are available in LLVM—RL4ReAl gives competitive numbers vs. the state-of-the-art register allocators in LLVM.

It can be noted that these results have been obtained fully automatically, against production-grade allocators tuned over many man-decades of experience and effort.

7.7 Related Work

Recently, several ML-supported compilers were proposed, leveraging representation learning techniques for compiler optimizations [Haj-Ali et al., 2020; Mendis et al., 2019a; Mammadli et al., 2020; Jain et al., 2022a]. These works use learned embeddings like inst2vec [Ben-Nun et al., 2018], IR2Vec (Chapter 3), Flow2Vec [Sui et al., 2020] for representing the input programs to the ML model. We model a complex register allocation problem using RL and propose MIR2Vec to represent programs in MIR form.

An initial attempt to solve register allocation using ML models by Das et al. [2020] uses an LSTM to come up with an initial coloring scheme; it undergoes a *correction phase* to rectify the inconsistency in coloring interferences. Their work focuses on the

graph coloring problem, and to our understanding, the solution was not integrated to obtain the final register assignments. Another recent work by Kim et al. [2022b] proposes an RL-based solution inspired from AlphaZero [Huang et al., 2019a] for solving PBQP constraints by reducing the search space; they use Monte Carlo Decision Trees to simulate solving the PBQP graphs. Their model focuses on an irregular and custom architecture for Automated Test Equipments. RL4ReAl is the first end-to-end application of RL for solving the generic register allocation problem; does not need a separate correction phase, and is integrated as a `MLRegAlloc` pass in LLVM, and reports results that are comparable to the register allocators in LLVM.

Compiler-Gym [Cummins et al., 2022] is a recent approach designed to leverage Python libraries for solving compiler optimization problems; it exposes RL environments and datasets for training. However, it currently does not support integrating these trained models in the compiler pipeline for both training or deployment. Another framework, MLGO [Trofin et al., 2021b] integrates trained ML/RL models within the LLVM compiler. For this purpose, the compiler loads a trained model and accesses it via C++ APIs of Tensorflow or ahead-of-time generated code (release mode). The framework is used in production, with improved decisions for inlining for size, and live-range eviction (in register allocation) when compared to the compiler's default heuristics. MLGO addresses a narrower space of the register allocation problem, but its deep integration of a precompiled ML model into LLVM hints at a path for further integration of deployment of our approach in a production compiler.

7.8 Summary

This chapter presented RL4ReAl, the first end-to-end application of reinforcement learning to the register allocation problem. We formulated register allocation as a multi-agent hierarchical RL problem, where cooperating agents model the sub-tasks of coloring, live range splitting, and spilling. Semantic correctness is guaranteed through action masks that encode register allocation constraints, ensuring agents can only predict valid register assignments, demonstrating the *constrained learning* strategy (Section 5.3).

We represented interference graph nodes using MIR2Vec embeddings, building on the representation learning foundations from Part I. Our experimental results demonstrate that RL4ReAl achieves performance on par with LLVM's highly-tuned greedy register allocator, with speedups on several benchmarks. These results were obtained fully automatically against production-grade allocators refined over many years of expert

engineering.

We demonstrated that transfer learning enables models trained on x86 to adapt to AArch64 with 5× faster convergence and better performance. A policy improvement cycle further enables continuous learning on regression cases. Transfer learning and policy improvement validate the generalization capabilities of our RL formulation across architectures, workloads, and time (Section 5.3).

The LLVM-gRPC infrastructure developed for this work provides bidirectional communication between ML models and the compiler during both training and inference. In the next chapter, we generalize this approach into ML-Compiler-Bridge, a comprehensive framework supporting multiple communication mechanisms for seamless ML-compiler integration.

Source code and artifacts for RL4ReAl are available at `https://compilers.cse.iith.ac.in/research/rl4real` [VenkataKeerthy et al., 2023c].

Chapter 8

ML-Compiler-Bridge: Putting It All Together

8.1 Introduction

With the success of Machine Learning (ML) models in various domains, there is a growing interest in applying ML to improve optimization heuristics in compilers [Cooper et al., 2002; Amarasinghe, 2020]. Several ML and Reinforcement Learning (RL) approaches have been proposed to improve optimizations like vectorization [Haj-Ali et al., 2020; Mendis et al., 2019b], loop unrolling, distribution [Stephenson and Amarasinghe, 2005; Jain et al., 2022b], function inlining [Simon et al., 2013; Trofin et al., 2021b], register allocation [Das et al., 2020; Trofin et al., 2021a; Kim et al., 2022b] (see also Chapter 7), prediction of phase sequences [Ashouri et al., 2017; Huang et al., 2019b; Jain et al., 2022a], among many others [Allamanis et al., 2018a; Wang and OBoyle, 2018]. More specifically, the widely used LLVM compiler [Lattner and Adve, 2004] has support for RL-based inlining decisions from version 11, and RL-based eviction decisions in its register allocator from version 14 [Trofin et al., 2021a].

Setting up an ML-based compiler optimization is a challenging task. In addition to model design, it involves specialized data collection, compiler engineering, packaging:

1. Preparing or generating the data sets for training the model [Cummins et al., 2017b; da Silva et al., 2021].

2. Engineering objective-specific features [Fursin et al., 2011], or extracting objective-

independent program embeddings [Alon et al., 2019b; Ben-Nun et al., 2018; Cummins et al., 2021] (see also Chapter 3), or a combination of both.

3. Setting up a training interface with the compiler, with examples ranging from communicating the output via compiler flags [Jain et al., 2022a], offline file logs [Haj-Ali et al., 2020], to generic gym APIs [Brockman et al., 2016] and recent compiler-specific gym APIs like CompilerGym [Cummins et al., 2022], PolyGym [Brauckmann et al., 2021] and Supersonic [Wang et al., 2022b].

4. Finally, building and deploying the compiler with the trained model for inference.

In most works, the process ends with step (3) and a simplified benchmark-oriented version of step (4) to evaluate the trained model. Indeed, while there exist a number of solutions for steps (1 & 2), proper methodology-based solutions for steps (3) & (4) that involve model-compiler interaction have not yet been adequately addressed.

The diversity of compiler optimizations and ML models is associated with an equally broad range of requirements for model-compiler interaction. In Table 8.1, we illustrate this on recent proposals. There exists multiple ML frameworks and even more types of ML models. A model's input may be a plain floating point vector, or tensors of different ranks and shapes. Outputs range from a unique Boolean decision to complex data structures. These need to be communicated with the compiler; it may be only once for simple scenarios, or many times and involving large amounts of data for more intricate ones. And this may involve extensive source code modifications for the sole purpose of implementing the compiler-model interface.

Some of these interactions have been explored in the literature and even landed in production; however, there *does not exist* a single generic method to address the vast diversity of scenarios that are imaginable and the trade-offs therein. Such a situation limits the scope, applicability and effectiveness of ML for compiler optimizations in the following ways:

- **Scalability:** Integrating a Python model with C++ code using wrappers induces significant [Jain et al., 2022b] compile time overhead: e.g. 6×–100×.

- **Integration:** Not all optimizations are simple enough that the outputs of the model can be communicated using flags (see Chapter 7 and [Jain et al., 2022b; Kim et al., 2022b; Trofin et al., 2021b]). As ML-based optimizations grow in popularity, flag-based approaches become unwieldy.

Table 8.1: Diverse ML and RL requirements across different works; unknown or unclear ones are left blank.

Work	Communication	Model Input	Model Output	Commn Frequency	#Agents	Model	Framework
SLP Vectorization [Mendis et al., 2019b]	–	LLVM IR	Instructions to pack	–	Single	GGNN	–
NeuroVectorizer [Haj-Ali et al., 2020]	Pragmas in source	Code2Vec vectors	Source with pragma	Once, at end	Single	FCNN	Keras, RLLib
Register Allocation [Das et al., 2020]	No integration	Interference graph	Coloured IG	None	NA	LSTM	TensorFlow
Register Allocation [Kim et al., 2022b]	–	PBQP Graph	Allocated PBQP graph	–	Single	GCN, ResNet	PyTorch
POSET-RL [Jain et al., 2022a]	Opt Flags	IR2Vec vectors	Pass sequence	Multiple/ episode	Single	FCNN	PyTorch
Loop Distribution [Jain et al., 2022b]	Python Wrappers	IR2Vec vectors	Distribution sequence	Once, at end	Two agents	GNN, FCNN	PyTorch
Inliner [Trofin et al., 2021b]	Precompiled TF	Features	Yes/No	Once, at end	Single	FCNN	TensorFlow
RegAlloc Eviction [Trofin et al., 2021a]	Precompiled TF	Features	LR index to evict	Once, at end	Single	FCNN	TensorFlow
RL4ReAl (Chapter 7)	gRPC	IG + node embeddings	Color map	Multiple/ episode	Four; hierarchical	GNN, FCNN	PyTorch, RLLib

- **Programmability:** ML models are typically written in Python across different frameworks like TensorFlow, JAX, PyTorch, etc. Expecting the model to be written *in C++ within* the compiler is *not ML developer-friendly.*

- **Portability:** Several proposals involve a *tight coupling* between the compiler and a specific ML framework; we however believe that a generic compiler infrastructure like LLVM *should remain ML-framework-independent.*

The existing gym libraries primarily aim at facilitating model training for research and reproducibility by providing a high-level integration. For example, the recent CompilerGym [Cummins et al., 2022] provides a high-level interface in the form of C++ wrapper methods *outside* the compiler to invoke *out-of-tree compiler APIs* to materialize the predicted actions. Such integration caters well to training certain interactions like Phase Ordering [Jain et al., 2022a]. However, other optimizations like register allocation (Chapter 7 and [Kim et al., 2022b; Das et al., 2020]), loop distribution [Jain et al., 2022b] and inlining [Trofin et al., 2021b] necessitate a *deeper* interfacing of the model within the compiler; with multiple rounds of interaction for both training and inference scenarios. Further, in these gym libraries, the inference flow is driven by Python: the compilation starts by invoking a Python process, breaking the isolation between the end user and the internal compiler algorithms; this limits deployment opportunities among other downsides. We discuss these issues in detail in Section 8.4.

Key Idea

ML-Compiler-Bridge provides a modular abstraction with *model runners* for communication and *serializers* for data encoding. Inter-process runners (gRPC, Pipes) support training with diverse ML frameworks, while in-process runners (ONNX, TensorFlow-AOT) enable low-overhead deployment. This separation of concerns allows researchers to develop models in Python and deploy them seamlessly within production compilers.

As discussed in Section 5.3, the *Deployment & Integration* stage of ML-driven compiler optimization presents significant challenges in bridging Python-based ML frameworks with C++ compiler infrastructure. ML-Compiler-Bridge directly addresses this stage by providing framework-independent abstractions that support both training (where rich interaction with ML models is essential) and deployment (where minimal compile-time overhead is critical).

To address these shortcomings, we propose ML-Compiler-Bridge, a library that allows ML model development within a traditional Python framework while providing

tightly coupled and efficient end-to-end integration with the compiler.

Definition 8.1: ML-Compiler Bridge

An *ML-Compiler Bridge* is an infrastructure abstraction that enables communication between ML models and compiler, providing mechanisms for data serialization, model invocation, and result deserialization across both training and deployment scenarios.

Our library bridges the compiler and ML model by providing a suite of communication approaches (model runners) and the related (de-)serialization mechanisms (SerDes) to cater to diverse scenarios. It also provides support for both inter- and in-process communication by exposing different model runners: gRPC and named-pipes for the former, and the TensorFlow and ONNX for the latter. Diverse SerDes options based on Protobuf , JSON , and native bitstreams improve efficiency and versatility. The appropriate model runner and SerDes can be chosen based on the usage scenario and requirements, and these may differ during training and inference. Our library provides C++ and Python APIs to expose model runners and SerDes for integration with compilers and ML frameworks respectively.

We show that the inter-process model runners effectively supports training. Once the model is trained, the in-process model runners provide interfacing of the model within the compiler in a transparent manner, with much lesser latency to aid in deployment. Besides, our both model runner and SerDes modules can be easily extended to support more forms of communication and serialization. Our library also provides C-APIs to aid in integration with C-based compiler infrastructures like Pluto, GCC, and SQLite.

We evaluate ML-Compiler-Bridge on four ML-driven optimizations in LLVM: RL-LoopDistribution, POSET-RL, RL4ReAl, and Inliner. We show that our library can be integrated with other compilers like Pluto [Bondhugula et al., 2008] and MLIR [Lattner et al., 2021] with minimal effort. We study the impact of communication and serialization options on compile time under different complex scenarios that the existing infrastructures could not handle. We conduct extensive evaluations to measure the overhead caused by each model runner and SerDes. We also study the impact of integrating ML-Compiler-Bridge with LLVM in terms of additional dependencies, compile-time, and binary size overhead. Here are our contributions:

- We propose ML-Compiler-Bridge, a library to enable the deeper integration of ML models and the compiler in a framework-independent manner.
- We provide a suite of *two* inter- and *two* in-process model runners, and *three*

(de-)serialization mechanisms (SerDes) to support different interaction scenarios.

- We provide multi-language user APIs: C++ and C APIs to interface model runners and serializers with compilers and Python APIs to interface inter-process model runners with ML frameworks.
- We show that our library is easy to integrate with *three* different compilers spanning different representations, and carry out extensive evaluations on *four* ML-driven optimizations on *two* versions of LLVM (V10, V17).
- We characterize the impact of each communication and serialization options on compilation and training times and other overheads.

Organization The remainder of this chapter is organized as follows: Section 7.2 provides background on ML-compiler interactions, covering the training and inference workflows. Section 8.3 presents the ML-Compiler-Bridge library, describing the model runner and serializer abstractions. Section 8.4 describes the ML-LLVM-Project, demonstrating integration with four ML-driven optimizations. Section 8.5 presents comprehensive evaluations of compilation and training time impact. Section 8.6 discusses lines of code, overhead, and characterization of model runners. Section 8.7 discusses related work, and Section 8.8 concludes the chapter.

8.2 Background

ML-driven Compiler Optimizations The process of supporting or fully implementing optimization decisions with one or more ML models involves the steps shown in Fig. 8.1. This process repeats until the end of the compilation process for each ML-based optimization. The above scheme is generic enough to capture any optimization involving single or multiple ML models with multiple two-way interactions. For the cases that would need multiple interactions, steps (1)–(7) are repeated until the final outcome.

More broadly, there are three actors involved in developing and using such an ML-driven compiler. (i) The Compiler expert who develops the compiler optimization, (ii) The ML expert who designs the ML model for the optimization problem, and (iii) The end-user who uses the compiler. Ideally, compiler experts should use the ML models with minimal understanding of the internals/process specific to ML modeling and the framework on which the model is built to arrive at the result. Similarly, ML experts

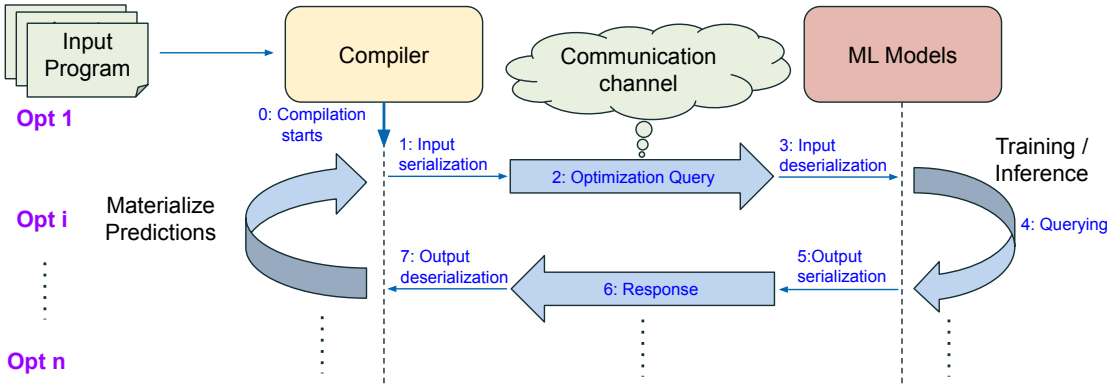


Figure 8.1: ML-driven compiler optimizations: (1) Inputs and other metadata required by the model are prepared in the appropriate format. (2) Serialized input is passed on to the model by a suitable communication channel. (3) Input is deserialized to appropriate format. (4) The model is queried to obtain optimization decisions as output. (5) Output is serialized, and (6) Sent back to the compiler optimization as a response. (7) The received response is deserialized, and optimization decisions are taken according to the output.

should instead design the models with minimal or no understanding of compiler internals, infrastructural details, and integration points, focusing on the optimization objectives and information flow. For the end-user, however, the presence of ML-compiler optimization should be *transparent*, and *indistinguishable* from the conventional (non-ML based) compilation process. To achieve this scheme of abstraction/segregation among all three actors, it is important to distinguish between the training and inference flows.

Training Typically, training the ML model becomes part of compiler development and build-up, and inference becomes part of the compiler deployment and execution. However, occasionally this boundary may shift towards the user, like domain-specific training or fine-tuning at deployment time. Since ML developers usually prefer developing models within a Python-based framework, the training process involving a C++ compiler infrastructure like LLVM requires a communication channel, typically inter-process, while catering to the needs of (de-)serializing data between the native types of C++ and Python. The distributed nature of training processes may also require extending communication beyond a single operating system node.

Inference When focusing on inference/deployment, compile-time and ease-of-use become crucial factors. The communication and serialization methods involved should take this into account, along with considering converting the Python model to a streamlined

C++ implementation. These factors are true even for the simplest forms of communication, like one-time evaluations of the ML model and communicating via flags. Making the flow transparent to the user also requires a deeper, end-to-end integration with the compiler.

There is no tool providing the necessary layers of abstraction between the three actors while supporting the required training and inference scenarios, not to mention ML-framework independence. *Designing such a library and evaluating its suitability for diverse use cases* is the challenge we tackle in this chapter.

8.3 ML-Compiler-Bridge

We propose an abstraction mechanism made of two main components: Serializer and Model Runner. The `SerDes` module (de-)serializes the data to/from the requested format, and the `MLModelRunner` module is responsible for communication with the model. The model runner obtains the serialized data, writes it to a communication channel, queries the model, and deserializes the output received from the model. ML-Compiler-Bridge exposes methods to be invoked by the user to interact with the model decoupled from serialization and communication. We provide three framework-independent model runners, gRPC, named-pipes, and ONNX, and one framework-specific TensorFlow model runner. These can be combined with three different serializations: Protobuf, JSON, and bitstream. The modular design enables new forms of communication and serialization to be added by overriding a minimal set of methods. Figure 8.2 shows the components and interactions of ML-Compiler-Bridge.

8.3.1 ML Model Runners

We provide two classes of model runners. The inter-process class provides the easiest mechanism to decouple Python models from a compiler running as a separate process. The in-process class assumes that the ML Model is readily available in a compiled form and can be accessed within the compiler through a specific API. Clearly, in-process communication is designed with inference and deployment in mind, while inter-process communication enjoys more diverse use cases. Model runners may support simple ML queries and feed-forward networks as well as more involved Reinforcement Learning (RL) algorithms or Graph Neural Networks (GNNs).

Internally, `MLModelRunner` is the abstract base class from which the other model

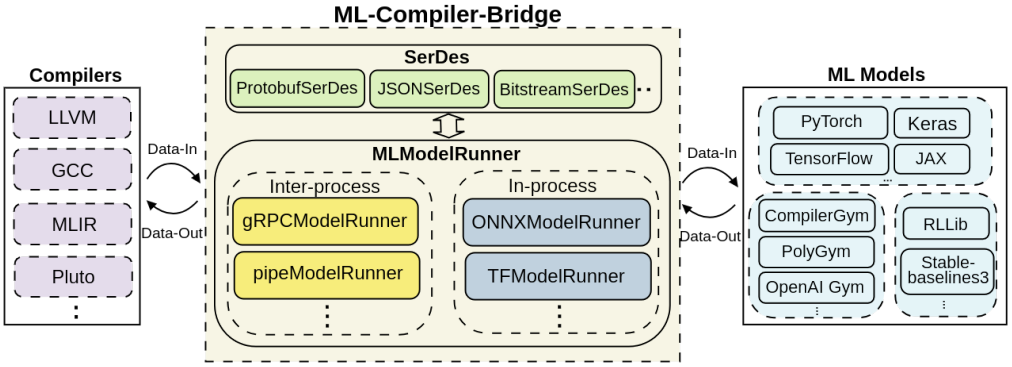


Figure 8.2: The compiler instantiates a model runner and sets the input features to be used by the model. `MLModelRunner` internally invokes `SerDes` to serialize the data in one of the supported formats and query the model. The returned decision is deserialized and provided to the optimization.

runners are derived (List. 8.1). It exposes two APIs: `populateFeatures()` populates the input features, and `evaluate()` queries the model. The latter returns the output of the model and is templated according to the expected output type. Internally, `evaluate()` invokes `evaluateUntyped()` that is to be overridden by the concrete model runner classes that derive from `MLModelRunner`. The `MLModelRunner` interfaces with the methods of `SerDes` using the `populateFeatures()` so as to serialize the inputs. The method `populateFeatures()` is implemented as a variadic function that uses variable-length key-value pairs as arguments. The key is a string identifier that describes the input, and the value is of template type.

Inter-process Model Runners

`gRPCModelRunner` uses gRPC, and may run the model and compiler on different machines. Whereas, the `pipeModelRunner` uses *named pipes* for single-machine scenarios. At training time, the compiler acts as a server and the Python-based ML model acts as a client. The sequence of steps is described as follows:

(1) Compilation starts and the compiler *listens* for queries at the `wait()` call inserted at the point of interest.

(2) The Python model starts training; this can be started concurrently with Step (1).

(3) When input from the compiler is required, the model sends requests to the compiler with appropriate queries and waits for the response.

```
class MLModelRunner {
public:
  // Populates inputs as key-value pairs
  template <typename T, typename... Types>
  void populateFeatures(std::pair<string, T> &var1, std::pair<↩
   string, Types> &...var2);
  // Exposed to the user; returns model's output
  template <typename T> T evaluate() {
    return *reinterpret_cast<T *>(evaluateUntyped());
  }
protected:
  // To be overridden by derived classes
  virtual void *evaluateUntyped() = 0;
};
```

Listing 8.1: Skeleton of `MLModelRunner` class

(4) The compiler gets out of the blocked state and processes the query to generate an appropriate response.

(5) The response is sent back to the client, and the model goes on to completing training on that input.

Inference follows the same steps, yet the compiler becomes the client and the model becomes the server so as to support a regular compilation process.

gRPC Model Runner gRPC [Wang et al., 1993] provides RPC methods specifying the type of input and output in Protobuf format [Protobuf]. During the build process of the library, the proto files are automatically translated to C++ and Python code by invoking the `protoc` compiler. The generated code defines the `Service` class that exposes the RPC methods to be overridden by the user in the optimization that makes use of `gRPCModelRunner`.

`gRPCModelRunner` takes in the server address and the port number at which the connection is to be established. In training mode, `gRPCModelRunner` starts the server and starts listening for an RPC call invoked by the model. The overridden RPC method is directly called by the Python model to generate new observations by applying the action predicted by the model. In inference mode, `gRPCModelRunner` starts the gRPC connection at the given address and port.

Pipe Model Runner As the name suggests, the pipe model runner relies on named pipes for inter-process communication (the `mkfifo` system call). Pipes provide a simple and effective means of communication that is local to the machine without any network or security constraints.

As pipes are unidirectional, the `pipeModelRunner` creates the read and write pipes for communication. The read pipe in the compiler obtains the data written by the model in Python, and the write pipe provides the data into the pipe that is read by the model on the other end. `read()` is a blocking call forcing the compiler to wait till data is written by the model. Once the data is written, the model gets to a blocking state by invoking `read()` on the second pipe waiting for the response from the compiler. The pipe model runner ensures proper opening, closing, and clean up. `pipeModelRunner` provides a simpler interface for establishing communication as the user directly invokes `evaluate()` after setting the inputs.

In-process Model Runners

In-process model runners are designed to provide an effective means of compiler deployment. It is important to optimize the inference time as it adds up to the overall compile time. One may obtain significantly lower compile times by removing inter-process communication overhead, and by turning the complications of a compiled model into an advantage, by reducing the query time compared to models running in Python. Serialization/deserialization overhead is also lowered.

ONNX Model Runner The Open Neural Network Exchange [, Linux Foundation] (ONNX) is an open format to represent machine learning models. Models built from various frameworks like TensorFlow, PyTorch, etc. can be represented in ONNX format in an interoperable manner. Additionally, it supports a wide variety of hardware architectures ranging from edge devices to general-purpose CPUs and GPUs. Once the model is trained in Python, it is converted into a common ONNX representation and is imported into the compiler via the ONNX runtime. `ONNXModelRunner` exposes the necessary wrapper APIs to read the ONNX model, query it with inputs and obtain outputs.

For RL, the agent is usually the learner trained to predict appropriate actions given the observations from the environment. Exporting a trained model to ONNX implies exporting only the agent. To facilitate RL-based interaction for a generic multi-agent scenario between the environment and the agents, `ONNXModelRunner` provides Environ-

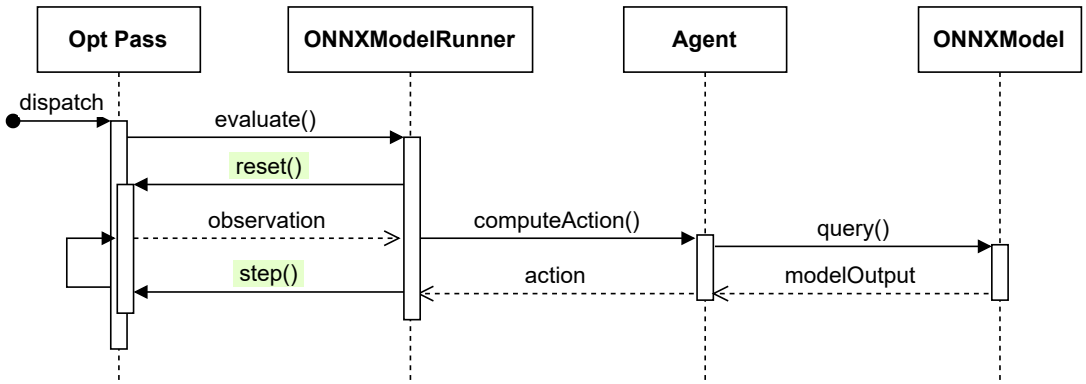


Figure 8.3: Sequence diagram indicating different events and the interaction between various classes for RL-based optimization by `ONNXModelRunner`. Only the methods that are highlighted are to be overridden by the user. Other methods are internal to the library.

ment and Agent classes separately and accesses the APIs internally. The sequence of events describing this interaction is shown in Figure 8.3.

`ONNXModelRunner` queries the model using the C++ APIs. A map containing the identifier of the agent (label) and the corresponding model path is passed while instantiating the `ONNXModelRunner`. In the case of multiple agents, the identifier of the next one to use is set by the `Environment` while returning the observation. `ONNXModelRunner` queries the corresponding agent with the observation to obtain the requested action. This process goes on until `Environment` invokes `setDone()`.

TensorFlow Model Runners This is a framework-specific model runner built on the TensorFlow ahead-of-time (AOT) saved model. There are two implementations: (i) *"Release Mode Model Runner"* used in production environments, (ii) *"Model Under Training Model Runner"* intended either for finetuning or when quickly evaluating candidate models and parameters.

The TensorFlow model runner uses the AOT *saved model compiler* which produces a header exposing the model as a C++ class, and a native object file with its implementation. The model runner reduces again to a simple adapter [Gamma et al., 1995] around that class. The compiler binary does not expose new runtime dependencies as it is statically linked, and this highly simplifies its deployment. Note that the model compiler can be configured to generate code by loading the weights from a file passed via the command line to the compiler.

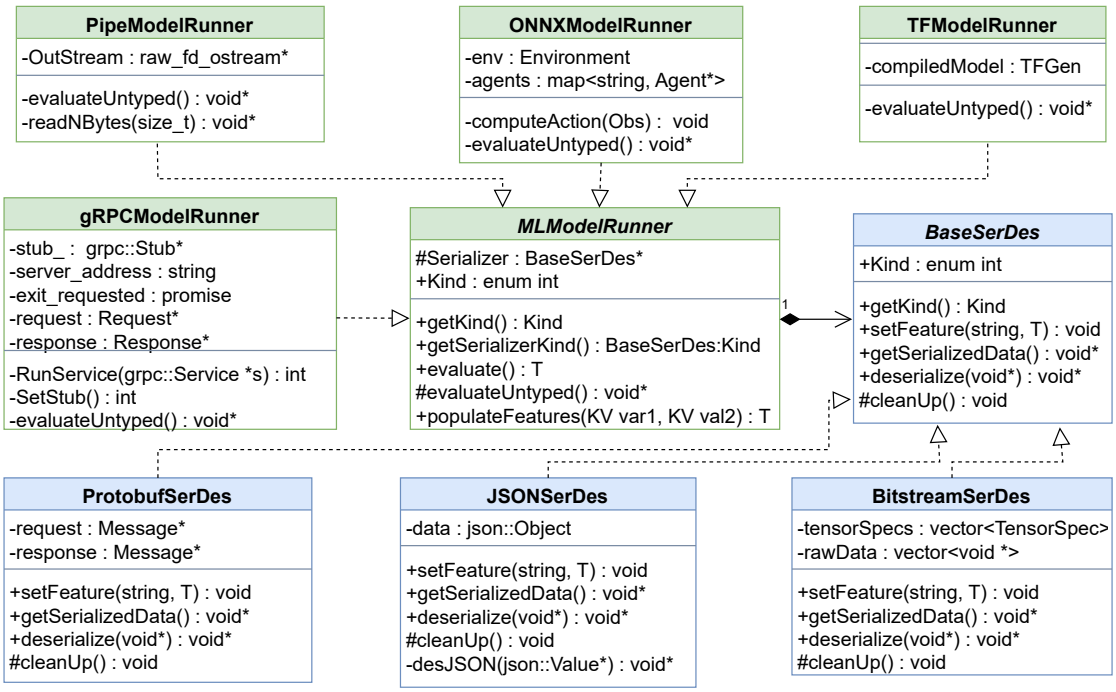


Figure 8.4: Class diagram of ML-Compiler-Bridge components: Model Runners and SerDes serializers/deserializers.

8.3.2 SerDes: Serializer and Deserializer Module

When data is transferred, specifically across two processes, it is important to convert data that is present in the native types (of C++ and Python) from one format to another. This is the purpose of (de-)serialization as implemented by the `SerDes` module.

The `MLModelRunner` interacts with `SerDes` to (de-)serialize C++ native data to model-specific types and back. The choice of (de-)serialization depends on the optimization and ML model. We currently provide three options: bitstream, JSON, and Protobuf. They vary in terms of usage scenario, usage effort, and (de)serialization time. SerDes effectively abstracts away the underlying mechanism while providing the flexibility of different serialization options.

Base SerDes Internally, each SerDes is derived from the `BaseSerDes` class that exposes the APIs to (de-)serialize inputs/outputs. `SerDes` uses key-value based serialization as described in Section 8.3.1. The `populateFeatures` method of `MLModelRunner` invokes the appropriate version of the overloaded `setFeature()` exposed by `BaseSerDes` to serialize inputs. These methods are overridden by the SerDes classes that derive

from `BaseSerDes` according to the underlying serializer. This class also exposes the `deserialize` method to deserialize the received data and is overridden by the derived classes to obtain the data in native types. Our library supports (de)serializing in basic (int, float, double, string, bool) and compound (vector, list) data types.

Protobuf SerDes Protobuf SerDes needs the user to provide the input and output data specifications in a proto file. These are compiled to generate the C++ and Python sources (Section 8.3.1). `ProtobufSerDes` serializes the input key-value pair by overriding the `setFeature` methods to set the appropriate fields of the *message* described in the proto file. Deserializing protobuf data to the native format only involves reading and returning the appropriate fields of the *message*. Except for providing the proto file, `ProtobufSerDes` is transparent to the user.

JSON SerDes `JSONSerDes` overrides the `setFeature` methods to populate the JSON buffer appropriately, given the key-value pairs. Similarly, the received data is deserialized by first converting it to a JSON object, then the JSON fields are casted to native types and returned. JSON SerDes is also transparent to the user.

Bitstream SerDes The bitstream starts with a JSON header which specifies the key (identifier), type and shape of the tensors, and the order in which they will be serialized. Tensor values themselves are dumped as raw bytes. The received bitstream is interpreted based on the type and shape specified in the header and converted to native types. Processing the header induces negligible overhead if communicated data does not involve complex data types. Internally, `BitstreamSerDes` overrides the `setFeature` methods similar to the other SerDes to expose the functionality.

Figure 8.4 shows the class diagrams [Gamma et al., 1995] of model runners and SerDes.

8.3.3 C-APIs

We provide C wrappers around the C++ implementation to integrate with C-based compilers. These wrappers are C++ files written in C-style. Each method internally queries the original C++ implementation and returns results in a way compatible with C calling conventions. This code is built as a separate library that may be linked with a C-based compiler.

```cpp
#include "MLCompilerBridge/MLModelRunner.h"
#include "MLCompilerBridge/yourMLModelRunner.h"

// Instantiate the required model runner with SerDes type
MLModelRunner *MLRunner = std::make_unique<yourModelRunner>(Arg,↩
    SerDes::Kind::yourSerDesType);
// Process Input Features
std::pair<std::string, InType> p = ... // Input
MLRunner->populateFeatures(p);
// Get ML Advice/Output
OutType advice = MLRunner->evaluate<OutType>();
// Use the obtained advice
...
```

Listing 8.2: C++ APIs of ML-Compiler-Bridge

```python
import CompilerInterface as CI

# Instantiate the required CompilerInterface with serdes type
interface = CI.YourCompilerInterface(Arg, yourSerdesType)
while True:
    # Send buffer data to compiler and wait for next request
    request = interface.evaluate()
    # Query model to get advice
    # Populates buffer with advice (serialized and stored in ↩
    serdes)
    interface.populate_buffer(advice)
    # Break on condition
```

Listing 8.3: Python APIs of ML-Compiler-Bridge

8.3.4 Extensions

Both MLModelRunners and `SerDes` can be easily extended to support new model runners and serializers. New runners may include TVM [Chen et al., 2018a], ahead-of-time compiled PyTorch models and FlatBuffers [Flatbuffers], and serialization also supports YAML formats. New model runners can be contributed by inheriting `MLModelRunner` and overriding the `evaluateUntyped` method according to the model runner. Similarly, a new (de)serializer can be added by inheriting `BaseSerDes` and overriding the `setFeature` and `deserialize` methods specific to the new serializer.

8.3.5 Error Checking and Recovery

The model runners and SerDes modules are designed to handle compiler/model crashes, communication failures, and infinite loops. The failures are handled appropriately by allowing graceful termination of the processes. In the case of gRPC, we have implemented an exponential backoff algorithm to attempt retries to overcome the failures due to the delays in communication resulting from any network-related issues and packet losses. The communication fails gracefully upon exhausting the number of retries. In all other cases, we use a timeout based mechanism for handling the failure. These mechanisms proved invaluable in practical experiments due to compiler bugs and network errors.

8.3.6 Compiler/ML Experts View

To use ML-Compiler-Bridge, developers need to invoke a minimal set of APIs by instantiating the necessary model runner with appropriate options specifying the SerDes type. List. 8.2 illustrates this on an example of invoking a user-defined model runner with a user-defined SerDes from the compiler. A similar API abstracting the communication and SerDes in Python is provided (List. 8.3) to query the ML model with inter-process model runners and respond back.

8.4 ML-LLVM-Project

We integrated ML-Compiler-Bridge with four ML-based compiler optimizations in LLVM: phase ordering [Jain et al., 2022a], loop distribution [Jain et al., 2022b], register allocation (Chapter 7) and method inliner [Trofin et al., 2021b]. The first three optimizations are built using RLLib [Liang et al., 2018] with PyTorch [Paszke et al., 2019] and LLVM V10, using program embeddings called IR2Vec (Chapter 3). The fourth optimization—inlining—uses TensorFlow [Abadi et al., 2015], is built within LLVM V17, and uses feature-based representations [Trofin et al., 2021b]. There are two ML-based register allocators (Chapter 7 and [Trofin et al., 2021a]) available for LLVM; we chose the former because it emphasizes finer-grained, high-bandwidth interactions with an ML model. All the components are configured, compiled and linked during the regular build process of LLVM. Integration challenges range from redesigning the entire framework of the original publication, to minor changes to the communication mechanisms.

8.4.1 Phase Ordering of Optimization Passes

POSET-RL predicts the ordering sequence of passes to jointly optimize code size along with execution time. An RL agent is trained with the DDQN algorithm [van Hasselt et al., 2016] to predict a subsequence as action, given program embeddings as input observation. There are about 15 predetermined subsequences provided by the authors. The predicted optimization subsequence is applied on the input program, and the embeddings corresponding to the transformed program are used as the new observation. This process goes on until reaching a threshold on the number of subsequences.

In the published version, the above process was not integrated within LLVM but driven from a Python model. An LLVM-opt process was spawned, passing the optimization sequence through a compiler flag for each prediction by the agent. In addition, embeddings involve spawning yet another process to invoke IR2Vec on the `.ll` IR file generated by the compiler. A similar strategy was in place for training.

We revisited the above using ML-Compiler-Bridge to operate directly within LLVM as a new transformation pass. Our new `PosetRL` implements a pass manager that applies the predicted optimization sequence, and also generates the next observation by invoking IR2Vec. The `MLModelRunner` communicates with the model and serializes the data to be transferred. The model communicates the predicted optimization subsequence as an integer ID (one among 15) to `PosetRL`, and the $\mathbb{R}^{300}$ module-level embedding vectors are sent to the model for the next prediction.

8.4.2 Loop Distribution for Vectorization and Locality

Jain et al. [2022b] improve loop distribution by modeling SIMD parallelization and locality optimization opportunities. It uses two RL agents with fully-connected networks to identify the vertex processing order and when to distribute. Along with these agents, a Gated Graph Neural Network [Li et al., 2016] processes the connected components of the dependence graph, where each node holds the embeddings for the corresponding instructions.

During training, a Python driver spawns a process to invoke the Loop Distribution pass. The RL model processes the input graph and predicts the sequence of instructions to be packed together as a loop. Upon applying the prediction, the rewards indicate the effectiveness of distribution. All these steps involve model-compiler interaction via file I/O. Inference itself is integrated with LLVM using Python wrappers.

Our approach eliminates the need for Python wrappers, file I/O and spawning new

processes. The model runners internally (de-)serialize data depending on the chosen `SerDes` and the `MLModelRunner`. For the runners that use serialization, the input graph is represented as key-value pairs, and a variable length matrix in $\mathbb{R}^{n\times 300}$ encodes the sequence of n 300-D instruction embeddings. The output takes the form a variable-length integer array with node identifiers that are to be distributed.

8.4.3 RL-Based Register Allocation

We also evaluate RL4ReAl, an RL-based register allocator implementing the splitting, coloring, and spilling sub-tasks as separate RL agents on LLVM's Machine IR. These RL agents pose a formidable engineering challenge in interfacing the model with the compiler during both training and inference. Unlike other optimizations that need one single communication at the end, RL4ReAl involves *multiple interleaved communications rounds* to obtain a new observation and let the relevant agent make the next prediction. Also them RL agents are arranged hierarchically: the outcome of one agent determines which agent would be invoked next. Unlike other use cases, this optimization involves transferring an interference graph where each variable is associated with a $\mathbb{R}^{n\times 100}$ matrix, and where each one of the n instructions in the live range of the variable is represented in 100-D, a variable-length integer array to specify interferences and use points, and a variable-length floating point array of spill weights. Other metadata like function name, file name, and status are also sent as string fields. The model returns key-value pairs mapping variables to split or color decisions. Both training and inference use gRPC and Protobuf serialization.

We investigate different communication and serialization improvements in the subsequent sections, with specialized scenarios for distributed training and deployment-friendly inference.

8.4.4 LLVM Inliner

The inliner pass traverses call sites in a bottom-up fashion, one connected component of functions at a time. For a given component a working queue is initialized with the set of all static call sites. As the algorithm marks some call sites for inlining, it appends the former callee's call sites to the work queue. The decision to inline or not is made in two steps. First, it determines legality and whether the user provided any guidance (always/never inline). Only if the operation is legal and non-mandatory, a heuristic determines its profitability.

The decision is driven by a simple RL-based model. It takes a number of scalar features characterizing both the caller/callee (instruction counts, basic block counts, maximum loop depth), the call site itself (the number of compile-time constant parameters), as well as module-wide features (the current number of functions and statically known call edges). For the published version [Trofin et al., 2021b], the cost metric was size, with no reliance on dynamic profile data. The implementation uses AOT compiled TensorFlow model for inference with C++ APIs. We modularized it to use any model runner.

8.5 Evaluation

We measure compilation time on an Intel Xeon SkyLake W2133 with 6 cores, 12 threads and 32GB RAM. Training time is measured on an Intel Xeon W1390P with 8 cores, 16 threads, 64GB RAM and an Nvidia 3060 GPU. We evaluate POSET-RL, RL-LoopDistribution and RL4ReAl with gRPC, Pipe and ONNX model runners and different SerDes options, and take the median of 3 runs. Most experiments use SPEC CPU 2006 and SPEC CPU 2017 benchmarks.

8.5.1 Impact on Deployment

Table 8.2 shows the POSET-RL compile time using different model runners. Among the in-process runners, we use ONNX for PyTorch models and RLLib. Overall, in-process runners achieve better compile times in all cases in comparison with any of the inter-process ones. Among the latter, gRPC has higher compile times (6.8–7.6%) compared to pipes, with JSON and bitstream SerDes. This is because of the overheads associated with establishing connections and invoking RPC methods. Pipes with Bitstream SerDes yield slightly higher performance than JSON SerDes due to the lower (de-)serialization overhead with bit streams. `ONNXModelRunner` yields a 7.2× speedup with POSET-RL compared to the original method in Section 8.4.1 that involved spawning new processes to invoke the compiler and other dependencies.

In-process model runners natively support multithreaded compilation, while inter-process model runners necessitate concurrently running multiple instances of the model resulting in a high memory and compute overhead. Table 8.3 shows compile times with in-process model runners on LLVM Inliner and RL4ReAl optimizations by varying the degree of parallelism. As LLVM Inliner and RL4ReAl respectively rely on TensorFlow

Table 8.2: Compile time (in seconds) for POSET-RL.

	Original	gRPC	Pipe + JSON	Pipe + Bitstream	ONNX
SPEC06	5,829	1,318	1,236	1,227	1,140
SPEC17	10,342	1,221	1,141	1,132	1,093

and PyTorch (and RLlib), we use TensorFlow and ONNX model runners accordingly. In comparison to the original gRPC-based inference flow of RL4ReAl, the ONNX runner reduces compile time by 22.4× and 19× using 8 threads and 1 thread respectively. Using RL4ReAl results in a higher compile time, as it involves a larger number of model-compiler interactions. This overhead is effectively reduced by using the model runners of ML-Compiler-Bridge.

Similar trends are observed for RL-driven loop distribution [Jain et al., 2022b] on TSVC [Boulton, n.d.] and the LLVM Test Suite [LLVM-Org, n.d.]. The ONNX model runner yields an improvement of 16× in comparison to the original Python wrapper.

Table 8.3: Multithreaded compile time with -O3 (in s) with in-process model runners. Compile time with gRPC is shown for RL4ReAl for comparison.

	gRPC	1 Thread	2 Threads	4 Threads	8 Threads
LLVM Inliner (TF Runner)	-	596	501	361	307
RL4ReAl (ONNX Runner)	5,572	291	257	248	248

Takeaway 8.1: Practical Deployment Overhead

In-process model runners (TF-AOT, ONNX) achieve 7–22× faster compilation than inter-process alternatives, enabling practical deployment of ML-driven optimizations without prohibitive overhead.

8.5.2 Impact on Training

In this section, we evaluate the effectiveness of ML-Compiler-Bridge during the training of POSET-RL and RL4ReAl. We use inter-process model runners for training.

Training Time

Figure 8.5(a) shows the cumulative training time and number of training iterations observed in POSET-RL. We obtain *large improvements* in the training time *across all the model runners*. We see similar trends with gRPC and Pipe, as explained in the previous experiment.

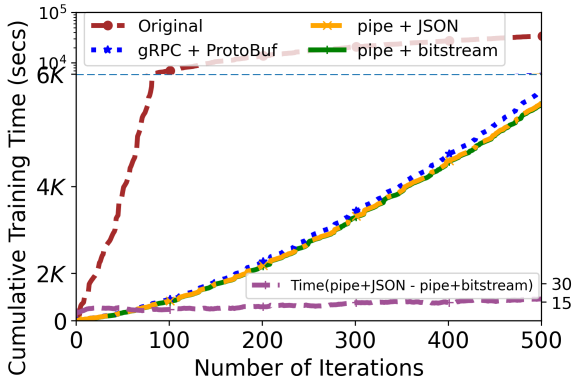
The original training process of POSET-RL involves spawning processes that takes $\approx$ 10Ks to complete 500 iterations. In comparison, the gRPC model takes about 5.7Ks, while the pipes with JSON and bitstream serialization options take about 5.5Ks each. Throughout the iterations, we observe an overhead of about 20s between JSON and bitstream serialization options. This minimal overhead is associated with the additional serialization effort involved while using JSON SerDes. However, using the inter-process model runners enables an *end-to-end integration of model and the compiler* while training yields a significant improvement.

Multi-Worker Support

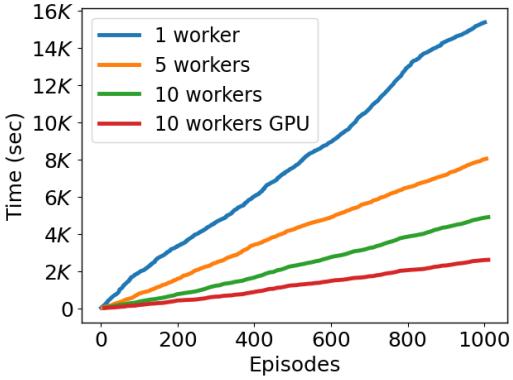
ML-Compiler-Bridge supports multi-worker training on both CPUs and GPUs. To support multiple workers while using gRPC, we expose a method taking an array of ports to establish connections with each worker. Similarly, multi-worker support with pipes is enabled by instantiating one pair of pipe per worker. We extended RL4ReAl to handle multi-worker scenarios; training times are shown in Figure 8.5(b) for CPU and GPU workers. Using 10 workers with a GPU trainer takes about 2 seconds per episode, while a CPU trainer with <10, 5, 1> workers takes <4s, 8s, 15s> respectively. We obtained similar trends among the workers even upon using pipes for communication.

Using Different RL Policies

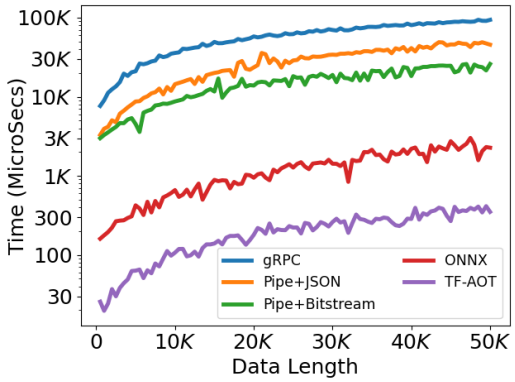
One may train and deploy models with different RL policies without impacting the compiler. For this experiment, we evaluate RL4ReAl with the different RL policies provided by RLlib. We perform hyperparameter tuning using Tune [Liaw et al., 2018]. We trained the models with PPO [Schulman et al., 2017], APPO [Schulman et al., 2017], and A2C [Mnih et al., 2016] policies until convergence. On the SPEC CPU 2017 benchmarks, this resulted in 2% improvement on average using the APPO policy. The PPO and A2C perform similarly to the original work.



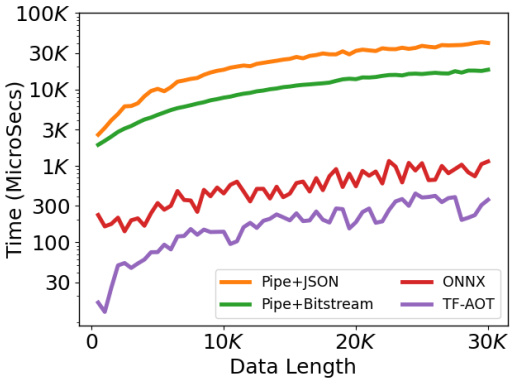
(a) Training times of POSET-RL with different Model Runners



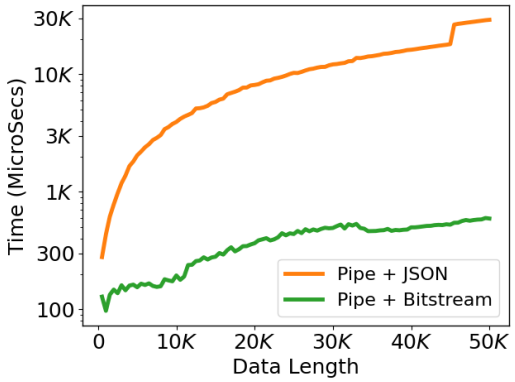
(b) Training times of RL4ReAl with CPU/GPU multi-workers



(c) Microbenchmarking of individual Model Runners



(d) MLIR performance



(e) Pluto performance

Figure 8.5: Performance characterization of model runners on different compilers and optimizations

Takeaway 8.2: Accelerated Training Infrastructure

ML-Compiler-Bridge enables: (i) ≈1.8× faster training through unified compiler-model communication, (ii) multi-worker scaling with 7.5× speedup using GPU workers, and (iii) flexible policy experimentation without modifying compiler code.

8.5.3 Round-Trip Time

Let us finally isolate the Round-Trip Time (RTT) of each model runner as a limit study of the achievable communication throughput. We consider random floating point vectors of increasing length ranging from 500 to 50K elements in steps of 500. The model itself is a single fully-connected layer that consumes the vector and returns a scalar float. Figure 8.5(c) shows the RTT of the whole process. The TF and ONNX runner achieves a very high throughput with a total RTT of 21 and 68ms respectively; while Pipes+JSON and Pipes+Bitstream yield 3154ms and 772ms respectively, and gRPC yields a larger RTT of 5948ms. These differences can be attributed to the serialization and communication overhead. The TF and ONNX runners benefit from in-process communication, proving to be suitable candidates for deployment. The higher throughput of TF is due to the AOT precompiled model. The Pipe runner proves to be a good candidate for training on local machines. And the gRPC runner provides support for training in a distributed environment. This makes *all the model runners important in their own way.*

8.5.4 Gym Integration

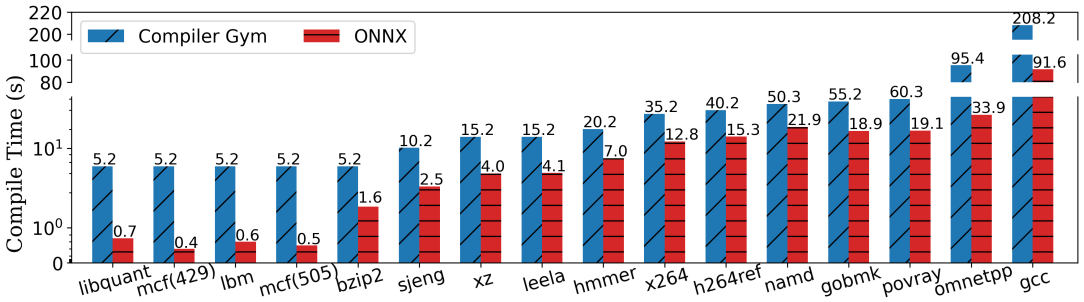


Figure 8.6: Compile times on using phase ordering for code size model with CompilerGym and ONNX model runner

We carried out additional experiments to evaluate the benefits of our library in the context of a state-of-the-art RL Gym. The two goals are to facilitate deployment and

to reduce compilation time by using in-process model runners. For this purpose, we trained the pass ordering for the code size of CompilerGym [Cummins et al., 2022] and exported the resulting model in the ONNX format. We then used our ONNX model runner within LLVM to materialize predictions and generate code. The inference times are shown in Figure 8.6, with speedups ranging from 2× to 13×. These are primarily due to gRPC overheads in CompilerGym, as shown in Figure 8.5(c).

8.5.5 Domain-Specific Compilers

Given LLVM's dominance in the general-purpose and backend compiler landscape, it forms the natural basis for most ML/RL tools (Table 8.1). However, some ML-based domain-specific optimizations target higher-level frameworks like MLIR [Lattner et al., 2021] and the polyhedral compilers Pluto [Bondhugula et al., 2008] and PoCC [Pouchet et al., 2010]. Let us illustrate the cases of MLIR and Pluto.

Integration with MLIR Given that end-to-end ML compilers based on MLIR are still undergoing rapid changes [Team, 2023a], we designed a simple experiment to demonstrate the integration of ML-Compiler-Bridge with MLIR. We wrote a custom pass in MLIR to communicate data with a dummy ML model to mimic a typical ML-compiler interaction. We use the same experimental setup as discussed in Section 8.5.3 and measure the round-trip time. The results are shown in Figure 8.5(c). This opens up ML-based optimizations in MLIR-native compilers such as IREE and OpenXLA [Team, 2023a], Triton [Team, 2023b], Polygeist [Moses et al., 2021], and many other frameworks.

Integration with Pluto We also experimented with the polyhedral source-to-source compiler Pluto. As Pluto is written in C, we use the C-APIs of ML-Compiler-Bridge for interfacing the models to illustrate the Pipe model runners and SerDes. We measured round-trip time using different SerDes and show the same in Figure 8.5(e). This integration opens new opportunities for ML-based polyhedral optimizations, including auto-scheduling and tile size selection.

> **Takeaway 8.3: Compiler-Agnostic Integration**
>
> *ML-Compiler-Bridge integrates with LLVM, MLIR, GCC, and Pluto with minimal code changes, validating a compiler- and ML-framework-agnostic design that decouples model development from compiler integration.*

Table 8.4: LOC to integrate model runners. gRPC shows LOC for API calls and RPC; Values in parenthesis indicate LOC in protobuf specification. Other serdes do not need any additional code.

Optimizations	Original	gRPC	Pipe	ONNX
POSET-RL	-	3+3 (4)	3	3
RL-LoopDistribution	65	3+3 (5)	4	3
RL4ReAl	75	10+3 (28)	4	3

8.6 Discussion

8.6.1 Lines of Code

In Table 8.4, we show the number of additional Lines of Code (LOC) to integrate ML-Compiler-Bridge with different compiler optimizations. We observe a significant reduction in LOC compared to the original published works.

We do not compare with the size of the published version of POSET-RL, as its model was not integrated with the compiler. With Loop distribution and RL4ReAl, the effort of writing Python wrappers and invoking protobuf and gRPC is completely removed. Among the available model runners and SerDes, only gRPC, ONNX and Protobuf involve (small) additional codes to handle RPC, environment, and Protobuf messages. It is pertinent to note that ML-Compiler-Bridge removes the tedious work of managing dependencies like gRPC and Python wrappers which was otherwise necessary.

8.6.2 Impact on Compile Time, Size and Memory

In Table 8.5, we show the compile time, binary size and average resident set size (RSS) used during compilation of Clang 10 with/without ML-Compiler-Bridge. The difference in binary size is $\approx$ 80KB, while the average RSS value differs by 400KB with the release build time increasing only by a few seconds. ML-Compiler-Bridge incurs only a negligible overhead in terms of binary size, compile time and memory upon statically linking with the production version of Clang.

Table 8.5: Comparisons of time taken to build clang and final binary size with/without ML-Compiler-Bridge

Characteristics	Native Clang	Clang with ML-Compiler-Bridge
Compilation Time	5m 7s	5m 15s
Binary Size	102.79 MB	102.87 MB
Average RSS	1.5538 GB	1.5542 GB

8.6.3 Characterization

As discussed earlier, different model runners exhibit different characteristics. During deployment, neither of the inter-process model runners offer multi-threaded compilation upon running a single model instance. It could be done by instantiating multiple model instances but this would consume unreasonable amounts of memory. The in-process model runners, however, do not face this problem. Though there is a separate serialization overhead involved with gRPC and pipe model runners, they are handled automatically *without the involvement* of the developer. Due to the nature of inter-process communication, there is a possibility of encountering communication errors arising from network and compiler crashes. We handle such cases as explained in Section 8.3.5. We summarize these characteristics in Table 8.6.

Table 8.6: Characteristics of different model runners

Characteristics	gRPC	Pipes	ONNX	TF
Multithreaded Compilation	✗	✗	✓	✓
Distributed Training	✓	✗	-	-
Need for separate model process	✓	✓	✗	✗
Autoserialization	✓	✓	-	-
Communication Fidelity	✗	✗	✓	✓
ML Framework agnostic	✓	✓	✓	✗
Additional code by compiler writer	Y	N	Y	N
Serialization Requirement	Y	Y	-	-
Time overhead	Y	Y	N	N

8.6.4 Limitations

As mentioned earlier, not all model runners are compatible with all ML models due to the nature of the underlying libraries. For instance, Tensorflow AOT compilation supports any Tensorflow or JAX model, but not PyTorch. Also, upon exporting the inliner model from TensorFlow to ONNX, we encountered an operator (`TFL-Bucketize`[1]) that is not supported by ONNX. To handle such cases, the ONNX runtime allows registering custom operators. Once exported, the models can be used seamlessly without restriction.

Similarly, protobuf does not natively support C runtime. Hence, our C APIs do not support using the gRPC model runner with protobuf serialization. The current TF AOT compilation generates C++ code thereby making it not usable directly with C. This issue can be mitigated by using TF C-APIs instead of using AOT models.

8.7 Related Work

RL environments for compilers come closest to our work, such as CompilerGym [Cummins et al., 2022], PolyGym [Brauckmann et al., 2021], Supersonic [Wang et al., 2022b]. These primarily aim at facilitating research and reproducibility, which are only two of the broader ambitions of our research (e.g., deployment, programmable compiler interface, finer-grained interaction). CompilerGym internally calls the compiler APIs from a C++ wrapper, and the communication between the Python model and the wrapper is established by predefined gRPC methods. This limits the functionality to only the APIs supported by the library and a particular compiler version with which the library is compatible. Supersonic [Wang et al., 2022b] also uses the CompilerGym way of interfacing via gRPC. And, to our understanding, PolyGym [Brauckmann et al., 2021] does not provide a programmable compiler interface.

The gym libraries and ML-Compiler-Bridge solve different problems; the former facilitates research and training, while our library aims to facilitate different interfaces for communication. We envision ML-Compiler-Bridge to supplement these gym environments by providing a variety of options for more diverse, finer-grained, and framework-independent interfacing of ML models with compilers facilitating the transition from research to production.

[1] https://www.tensorflow.org/mlir/tfl_ops#tflbucketize_tflbucketizeop

8.8 Summary

This chapter presented ML-Compiler-Bridge, a modular and extensible library to integrate ML models within compiler optimizations. The library provides inter-process model runners (gRPC, Pipes) for training scenarios where rich interaction with Python-based ML frameworks is essential, and in-process model runners (ONNX, TensorFlow-AOT) for deployment where compile-time overhead must be minimized. Different serialization options (Protobuf, JSON, bitstream) offer flexibility in trading off development effort against efficiency.

We demonstrated that ML-Compiler-Bridge enables seamless integration with minimal code changes—as few as 3 lines to connect a model and compiler pass—while also supporting deep interleaving of complex RL-based algorithms like RL4ReAl. The in-process model runners achieve 7–22× compile-time reduction compared to inter-process approaches, making ML-driven optimizations practical for production use.

The library exposes C++, C, and Python APIs for integration with diverse compilers (LLVM, MLIR, Pluto) and ML frameworks (TensorFlow, PyTorch, RLlib). We evaluated ML-Compiler-Bridge on four ML-enabled optimizations spanning different representations, learning paradigms, and interaction patterns, demonstrating its versatility across research and production environments.

Together with the ML-LLVM-Project, ML-Compiler-Bridge completes the infrastructure for the ML-driven compiler optimizations presented in this part of the dissertation: from program embeddings (Part I) through model training and deployment. Source code, documentation, and artifacts are available at `https://compilers.cse.iith.ac.in/research/mlcompilerbridge` [VenkataKeerthy and Jain, 2024].

Part III

ML-Driven Program Understanding

"Programming is the art of telling another human being what one wants the computer to do."

— Donald Knuth

Chapter 9

Using Machine Learning for Program Understanding

9.1 Program Understanding

We begin by defining program understanding, the overarching theme of this part of the dissertation.

> **Definition 9.1: Program Understanding**
>
> Program understanding encompasses techniques for *automatically analyzing* programs to extract meaningful *semantic knowledge*, enabling tasks such as similarity detection, comprehension, and generation.

The extracted knowledge and insights can be used for various purposes, such as identifying the program's functionality, recognizing the program's behavior, etc. Program understanding has been a cornerstone of software engineering, security analysis, and programming language research, and forms an important application area in *Machine Learning for Code* (ML4Code) introduced in Chapter 1 on page 1.

As software systems grow in complexity, with codebases spanning multiple architectures and programming languages, automated techniques for program understanding have become more essential than ever before. This challenge is further amplified in scenarios such as reverse engineering, malware analysis, and software maintenance, where programs may be deliberately obfuscated or rewritten to obscure their functionality.

9.1.1 Evolution of Program Understanding

Program understanding has remained a challenging and interesting problem in the field of software engineering. Given the undecidable nature of the problem [Rice, 1953], various approximations and heuristics have been developed to understand programs.

The early techniques relied on formal methods to systematically reason about the programs. Program analysis techniques such as control-flow and data-flow analyses [Allen, 1971] provided a structured way of understanding programs, helping compilers and debugging tools to detect unreachable code, variable misuse and other inefficiencies. Formalization of techniques like abstract interpretation [Cousot and Cousot, 1977; Cousot and Halbwachs, 1978], Symbolic execution [King, 1976] provided a powerful means of reasoning about the behavior of all possible program paths leading to more precise analysis. This led to the development of formal program verification [Floyd, 1993] and model checking [Clarke, 1997]. To complement the formal methods, rule-based and heuristic approaches were developed to understand programs [Giannesini et al., 1986]. These methods encode the knowledge in the form of rules or heuristics.

Though these methods were powerful, they were limited in their ability to capture the broader context and intent of the programs. The formal-method-based techniques also face challenges in scaling to large codes.

The advent of large-scale code repositories and advances in neural networks opened new possibilities for program understanding. Statistical models initially captured shallow patterns in code, while deep learning enabled learning of more complex program semantics. These data-driven approaches leverage the *Extended Naturalness Hypothesis* (Chapter 2), which suggests that combining program analysis information with the statistical properties of code leads to more effective program understanding. More recently, large language models have demonstrated remarkable code understanding capabilities, though their application to similarity tasks differs fundamentally from embedding-based approaches (as discussed in Section 12.2 on page 253). These ML-based methods enable different aspects of program understanding that are more complex in nature.

9.1.2 Aspects of Program Understanding

There are different aspects or flavors to program understanding, ranging from program comprehension/summarization, generation, and similarity as summarized in Figure 9.1.

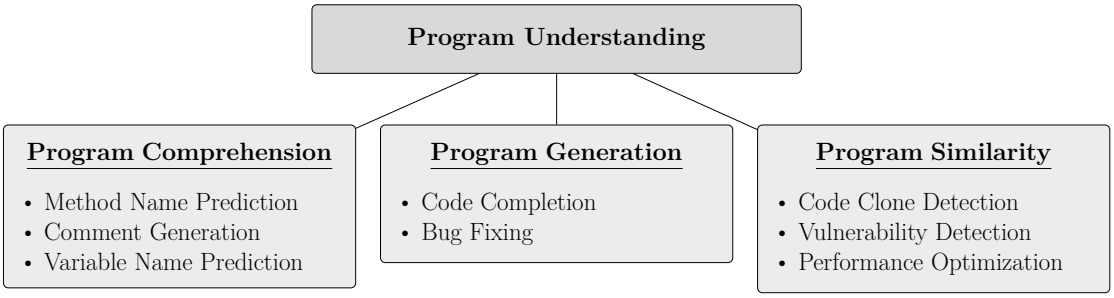


Figure 9.1: Different aspects of Program Understanding

Program Comprehension Comprehending programs involves generating a natural language description of the program's functionality. And, there exist several degrees of program comprehension; it could be as simple as predicting a short, meaningful method name [Alon et al., 2019a,b; Allamanis et al., 2015a], to generating code comments [Iyer et al., 2016; Hu et al., 2018].

Program Generation As the name implies, program generation involves automatically writing code snippets or programs that satisfy a given specification. Program comprehension and generation are two complementary tasks, where the former involves generating natural language descriptions given a code snippet, while the latter involves generating code snippets given a natural language description. Even here, there are different levels of program generation, such as code completion [Svyatkovskiy et al., 2020; Raychev et al., 2015], bug fixing [Gupta et al., 2017], etc. With the advent of large language models, program generation has seen significant improvements where specialized code models [Feng et al., 2020; Guo et al., 2021; Rozière et al., 2024; Wang et al., 2023b; Guo et al., 2022] have been developed to generate code snippets or programs.[1]

Program Similarity Among the aspects of program understanding, this dissertation focuses on program similarity.

> **Definition 9.2: Program Similarity**
>
> Program similarity is the task of identifying whether given programs exhibit similar functionality. Unlike program comprehension and generation, program similarity can be modeled as a classification or clustering task, where the goal is either to classify if

[1]The rapid progress in large language models for code has opened new frontiers in program generation and understanding. We discuss the relationship between embedding-based approaches and LLMs, along with future research directions, in Section 12.2 on page 253.

two programs are similar or to cluster similar programs together.

Program similarity can be useful in various scenarios, such as identifying code clones [Rattan et al., 2013; Farhadi et al., 2014], detecting bugs [Wang et al., 2016a; Gupta et al., 2017], and vulnerabilities [Gao et al., 2018; Liu et al., 2020], hints for improving code performance by identifying similar code patterns, etc.

In this dissertation, we focus on the program similarity aspect of program understanding. Specifically, we explore how machine learning can be used to identify the similarity between programs at the binary and source code levels. For *source-level similarity*, we employ IR2Vec's Symbolic, Flow-Aware, and Profile-Aware embeddings, which capture progressively richer program semantics from static and dynamic analysis. For *binary-level similarity*, we use VexIR2Vec's Path-Aware embeddings, which are specifically designed to capture control-flow structure through random walks on the CFG.

9.2 ML-Driven Program Similarity

Using machine learning for program understanding has been widely studied and explored in recent years [Pinku et al., 2024; Martinez-Gil, 2024]. Various ML modeling techniques have been developed to understand programs, ranging from statistical models to deep learning models to large language models. Code clones are typically classified into four types based on the degree of similarity [Rattan et al., 2013]:

Type I (Exact Clones) are identical program fragments differing only in whitespace, layout, or comments. They exhibit *syntactic similarity* and are the easiest to detect.

Type II (Renamed Clones) are structurally identical fragments with different identifier names for variables, types, or functions. Like Type I, they exhibit *syntactic similarity*, sharing the same tokens and structure at the text level.

Type III (Near-Miss Clones) are fragments with modifications such as added, deleted, or reordered statements. They exhibit *structural similarity*, sharing similar control flow or data flow patterns (e.g., similar CFGs or ASTs), even when their textual representation differs.

Type IV (Semantic Clones) are functionally equivalent programs that differ in both structure and syntax. They exhibit *semantic similarity*, the most general form that subsumes the other types, and are the most challenging to detect.

While Type I and Type II clones are relatively easy to detect, detecting Type III and Type IV clones is challenging. Variations in the program structure, algorithmic changes, and obfuscation techniques [Junod et al., 2015] make it difficult to detect these clones. As discussed earlier, detecting if two programs solve the same problem or have similar functionality is undecidable [Rice, 1953]. Hence, the problem of program similarity is often approximated using various techniques and heuristics [Hunt and Szymanski, 1977; Wang et al., 2000].

With the advent of deep learning and representation learning techniques, program similarity has seen significant improvements. Several techniques have been proposed to model the programs as vectors and then use these vectors to identify the similarity between programs [Ben-Nun et al., 2018; Cummins et al., 2021; Alon et al., 2019b; Damásio et al., 2023; Ding et al., 2019; Wang et al., 2023a]. ML-based techniques have been shown to be effective in identifying similar programs while being robust to variations in the program structure and syntax.

9.2.1 Our Schema of ML-Driven Code Similarity

In this dissertation, we study the program similarity problem using machine learning techniques. We focus on similarity at two levels: binary similarity and source code similarity. For solving both these problems, we propose a schema as depicted in Figure 9.2.

Dataset We start with a dataset of programs that we want to compare. The dataset can be obtained from various sources such as libraries, open judge datasets, etc.

Program Representations The programs in the dataset are then represented using Intermediate Representations such as LLVM IR, VEX-IR, etc. These representations are used to model the programs as vectors.

Embeddings The Intermediate Representations are then converted into embeddings using IR2Vec and VexIR2Vec techniques. These embeddings learnt in an unsupervised manner are used to encode the semantic information of the programs.

Modeling and FineTuning for Similarity The embeddings are then used to model the similarity between programs. As these embeddings are not specifically trained

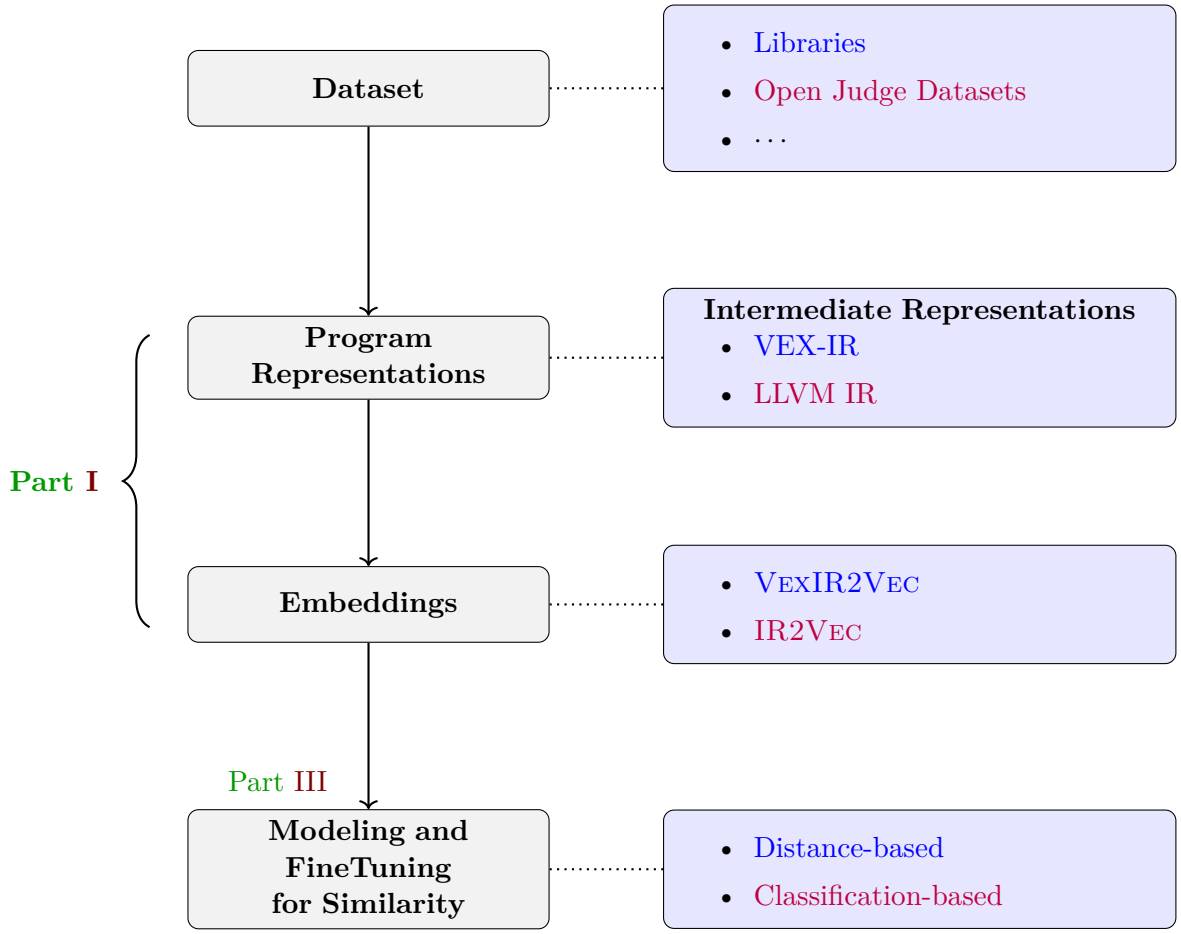


Figure 9.2: Our Schema of ML-Driven Binary and Source Code Similarity

for similarity, we fine-tune the embeddings using various techniques such as distance-based or classification-based methods to model the similarity between programs.

Though the schema is specific to the approaches we propose in this dissertation, it can be generalized to other approaches as well. The schema provides a structured way of understanding the problem of program similarity and the various steps involved in solving the problem using machine learning techniques.

Unlike the compiler optimization problems in Part II, where semantic correctness is ensured through constrained learning or compiler validation, program similarity is inherently a probabilistic classification task where all predictions are valid (the model outputs similarity scores or class probabilities). This makes program similarity naturally suited to supervised learning with distance-based or classification-based fine-tuning.

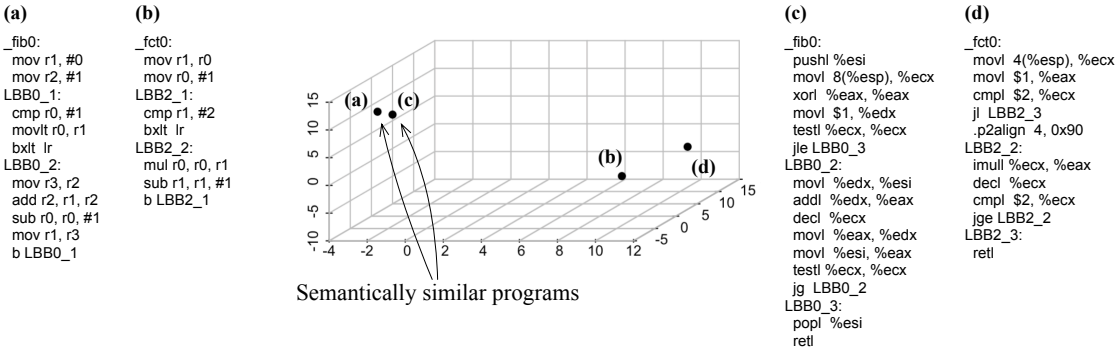


Figure 9.3: Example depiction of distance-based program similarity in a vector space. The programs are projected into a space where semantically similar programs (i.e., functionally equivalent) are closer to each other and dissimilar programs are farther apart.

9.2.2 Modeling Program Similarity

Program similarity can be modeled using various techniques ranging from distance-based methods to classification-based methods.

Distance-Based Methods Distance-based methods model the similarity between programs as a distance metric between the embeddings of the programs. The distance metric can be a simple Euclidean distance or cosine similarity between the embeddings depending on the approach used for training the embeddings for similarity.

Typically, during finetuning the embeddings for similarity, a distance metric is learned that minimizes the distance between similar programs and maximizes the distance between dissimilar programs. The distance metric can be learned using various metric learning approaches such as Siamese Networks [Koch et al., 2015], Triplet Networks [Schroff et al., 2015], etc.

The training results in a model that learns to project the input program embeddings into a space where similar programs are closer to each other and dissimilar programs are farther apart. Hence, at inference time, the distance metric is used to compare the similarity between programs. Figure 9.3 depicts an example of distance-based program similarity in a vector space.

Classification-Based Methods Classification-based methods model the similarity between programs as a classification task. Given a pair of programs, the model predicts if the programs are similar or dissimilar. The model is trained using a binary classification

loss function such as Binary Cross Entropy, Hinge Loss, etc. The training results in a model that learns to predict if the given pair of programs are similar or dissimilar.

Suitability As it can be seen, both distance-based and classification based methods are suitable in different scenarios. Distance-based methods are useful for searching similar programs in a large corpus of programs, while classification based methods are useful for identifying if two programs are similar or dissimilar. Depending on the approach used for training, distance-based methods can generalize to unseen classes of programs in a zero-shot or few-shot setting [Vinyals et al., 2016].

On the other hand, distance-based methods can be computationally expensive as they may involve computing the distance between all pairs of programs in the dataset. However, the problem of computing the distance between all pairs of programs can be mitigated by using Approximate Nearest Neighbors (ANN) methods such as those implemented in FAISS [Douze et al., 2025], HNSW [Malkov and Yashunin, 2016], or Annoy [Annoy, 2024]. Classification-based methods, on the other hand, are computationally efficient as they involve training a model that directly predicts the similarity between two given programs.

The choice of the method depends on the problem at hand. In this dissertation, we explore both distance-based and classification-based methods for modeling the similarity between programs.

9.3 Contributions

In this part of the dissertation, we explore how machine learning can address the challenges in program understanding, contributing to the broader goal of RQ3 posed in Chapter 1 on page 1: identifying functionally similar code across different languages, architectures, and obfuscation techniques. We focus on two key aspects: binary and source code similarity analysis. We use IR2Vec (Chapter 3) and VexIR2Vec (Chapter 4) techniques described in Part I on page 10 to develop architecture-neutral and language-agnostic similarity analysis techniques. By operating at the intermediate representation level, our approaches are inherently language-independent, enabling analysis across programs written in different source languages.

We show that the embedding-based techniques are effective in identifying similar programs while being robust to variations in the program structure and syntax. We evaluate the effectiveness of the embeddings under various adversarial settings such as cross-

optimization, cross-compilation, cross-architecture, and obfuscations. We also demonstrate the scalability of the embeddings and show that they are orders-of-magnitude faster than the state-of-the-art tools making them practical for real-world applications.

The contributions are summarized in Table 9.1.

Table 9.1: Summary of Contributions in Binary and Source Code Similarity

	Binary Similarity	**Source Code Similarity**
Embeddings	VexIR2Vec • Path-aware	IR2Vec • Symbolic • Flow-Aware • Profile-Aware
Method	Distance-based	Classification-based
Dataset	Real-world Linux libraries	Open Judge datasets
Applications	• Diffing • Searching	• Program classification • Device mapping

VexIR2Vec **for Binary Similarity** We use VexIR2Vec, a robust, architecture-neutral approach based on VEX-IR, to solve binary similarity tasks (Chapter 10). The Path-Aware encodings capture control-flow structure by sampling execution paths via random walks on the CFG, producing peepholes that approximate the function's behavior. We design VexNet, a feed-forward Siamese network that fine-tunes these embeddings for diffing and searching tasks. We evaluate VexIR2Vec against four baselines — BinDiff [Zynamics, 2024], DeepBinDiff [Duan et al., 2020], SAFE [Massarelli et al., 2019], and BinFinder [Qasem et al., 2023] — on a dataset comprising $2.7M$ functions and $15.5K$ binaries from 7 real-world Linux libraries compiled across 12 compilers targeting x86 and ARM architectures. The key insight is that combining representation learning, extraction of program structure via random walks, and normalization of code sequences is effective for binary similarity. VexIR2Vec introduces a decoupled methodology that separates vocabulary learning from training task-specific models. We implement VexIR2Vec as a parallel library consisting of the VEX-IR Normalization Engine (VexINE), an embedding extractor, and a tunable model (VexNet), available via command line and a web interface[2] (Figure 9.4).

[2] https://compilers.cse.iith.ac.in/VexIR2Vec

- Outperforms the nearest baseline in diffing by 40%, 18%, 21%, and 60% in cross-optimization, cross-compilation, cross-architecture, and obfuscation settings respectively.
- Achieves 0.76 mean average precision in searching, outperforming the nearest baseline by 46%.
- 3.1–3.5× faster than the closest baselines and orders-of-magnitude faster than other tools.

IR2Vec **for Source Code Similarity** We use IR2Vec embeddings (Symbolic, Flow-Aware, and Profile-Aware) for source code similarity tasks (Chapter 11). We study the effectiveness of different variations of IR2Vec embeddings for program classification using a feed-forward neural network. We evaluate IR2Vec on a dataset of $\approx 1M$ programs from open judge datasets (CodeJam, Code Forces) written in C and C++, with increasing levels of complexity. We also demonstrate the effectiveness of IR2Vec embeddings in detecting domain-specific codes such as Cyclic Redundancy Check (CRC) and cryptographic functions for hardware accelerator mapping.

- Symbolic, Flow-Aware, and Profile-Aware embeddings outperform histogram-based baselines by 4%, 8%, and 9% respectively.
- Demonstrated effectiveness in real-world device mapping applications.

Outline of Part III

This part of the dissertation is organized as follows:

- Chapter 10 describes the binary similarity problem and the approach we propose to solve the problem using VexIR2Vec.
- Chapter 11 describes the source code similarity problem and the approach we propose to solve the problem using IR2Vec.

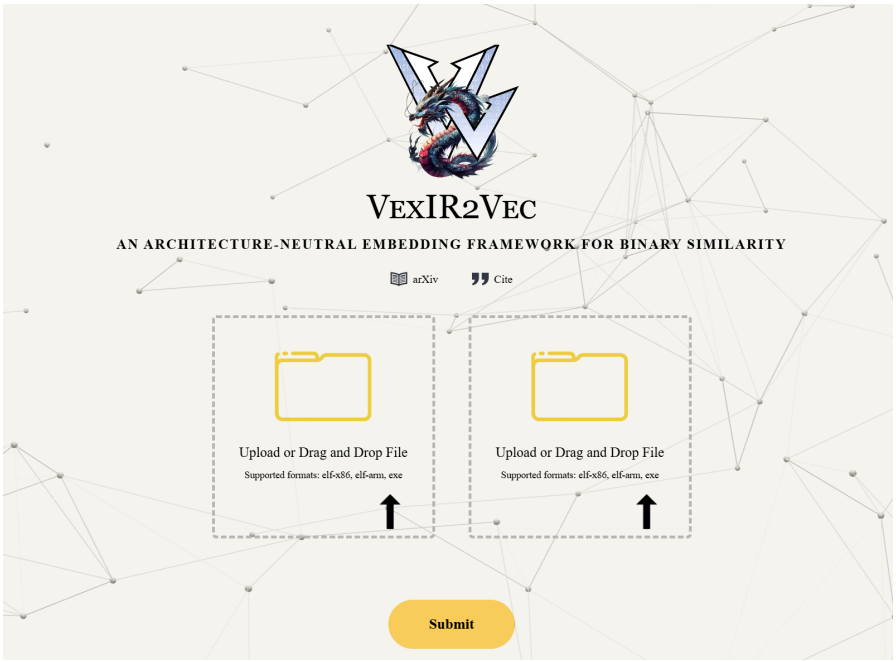


Figure 9.4: Web interface of VexIR2Vec. The user can upload two binaries and compare the similarity between the functions in the binaries.

Chapter 10

VexIR2Vec based Binary Similarity Framework

10.1 Introduction

Binary similarity is the task of determining whether two binary programs exhibit similar functionality, often originating from the same source code. Solutions to this problem enable applications in vulnerability analysis [Gao et al., 2018; Liu et al., 2020], malware detection [Farhadi et al., 2014], plagiarism identification [Ming et al., 2016], copyright authentication [Tang et al., 2018], profile matching [Panchenko et al., 2019; Wang et al., 2000], code lifting [Martínez et al., 2023; VenkataKeerthy et al., 2023a], and redundancy elimination [Xue et al., 2018].

This chapter focuses on two flavors of the binary similarity problem: diffing and searching. Diffing aims to identify similar or dissimilar regions (e.g. basic blocks, functions) between two binaries. Searching involves retrieving a binary function from a large pool of binary functions similar to the query code. These tasks become challenging in an *adversarial setting*, where binaries compiled from the same source code vary due to several factors: (i) Compiler choice (e.g., Clang, GCC, ICC); (ii) Compiler version (e.g., Clang 6.0.0 vs. Clang 17.0.0); (iii) Compiler optimizations (e.g., GCC `-O0` vs. GCC `-O3`); (iv) Target architecture (e.g., x86, ARM); and (v) Use of obfuscation (e.g., control-flow flattening or dead control flow). These variations in compilation configurations can significantly alter the binaries in terms of syntax, semantics, and structure, as we discussed in detail in Section 4.2.2 on page 63 of Chapter 4.

Binary similarity tools typically employ various representations to bridge the gap between raw binary and a more analyzable format. These representations include assembly languages [Ding et al., 2019; Massarelli et al., 2019; Pei et al., 2020], virtualized bytecodes [Chandramohan et al., 2016; David et al., 2017], or abstract syntax trees [Yang et al., 2021]. Once converted, several techniques are used to identify code similarities. These approaches are inherently heuristic because the general problem of determining program equivalence is undecidable [Rice, 1953]. Common heuristics leverage graph matching techniques [Zynamics, 2024], hashing of code sequences [Pewny et al., 2015; David et al., 2017; Wang et al., 2000; Ayupov et al., 2024], or Machine Learning (ML)—an approach that presently enjoy great popularity. ML models have emerged as the dominant approach due to their ability to learn complex relationships within code [Xu et al., 2017; Zuo et al., 2019; Ding et al., 2019; Duan et al., 2020; Massarelli et al., 2019; Pei et al., 2020; Yu et al., 2020].

As discussed earlier in Chapter 4, ML models for binary similarity require converting code into a numerical vector suitable for use as the model's input. These vectors can encode either handcrafted features or learned representations. Feature-based approaches [Feng et al., 2016; Eschweiler et al., 2016; Qasem et al., 2023; Damásio et al., 2023] use manually defined program characteristics, such as the number of opcodes, loops, and function calls, to represent the binary. In contrast, distributed representations are learned through representation learning techniques [Bengio et al., 2013]. This *learned representation* is a real-valued vector of a chosen dimensionality, conventionally referred to as an *embedding* [Ding et al., 2019; Duan et al., 2020; Massarelli et al., 2019; Wang et al., 2023a]. Embeddings capture complex relationships within the code that may not be easily captured by hand-crafted features.

In this chapter, we propose a novel binary similarity framework based on the VexIR2Vec embeddings introduced in Chapter 4. VexIR2Vec embeddings are learned from the VEX intermediate representation (VEX-IR) of the binaries. VexIR2Vec can be adapted to different binary similarity tasks, such as diffing or searching. To this end, we designed VexNet, a Siamese network [Koch et al., 2015] that *fine-tunes* the vocabulary to represent functions as points in an application-independent embedding space, where similar functions are closer to each other while dissimilar functions are far apart enabling broader applicability.

Key Idea

Binary similarity can be effectively solved by fine-tuning VexIR2Vec's *Path-Aware* embeddings through a lightweight Siamese network (VexNet) that learns to position semantically similar functions close together in the embedding space, while being robust to adversarial variations from compilers, optimizations, architectures, and obfuscations. The decoupled design, separating vocabulary learning from task-specific fine-tuning, enables both high precision and practical scalability.

Precision The evaluation in Section 10.5 shows that VexIR2Vec is more precise than BinDiff [Zynamics, 2024], DeepBinDiff [Duan et al., 2020], SAFE [Massarelli et al., 2019], and BinFinder [Qasem et al., 2023]. We also compare our approach with the representations created by using the histograms of opcodes, originally proposed by Damásio et al. [2023] for source-codes and with the GPT 4.0o large-language model (Section 10.5.1). Our evaluation uses a dataset made of $2.7M$ functions and $15.5K$ binaries built from 7 projects (Findutils [Findutils, 2024], Diffutils [Diffutils, 2024], Coreutils [Coreutils, 2024], cURL [Stenberg, 2024], Lua [lua, 2024], PuTTY [PuTTY, 2024], and Gzip [Gzip, 2024]) compiled with 12 different compilers targeting x86 and ARM. In the diffing scenario, VexIR2Vec's F1 Score outperforms the nearest baselines in cross-optimization, cross-compiler, cross-architecture, and obfuscation settings by 40%, 18%, 21%, and 60% respectively. In the searching scenario, VexIR2Vec outperforms the nearest baseline — BinFinder — by 46%, obtaining a mean average precision of 0.76 across different configurations.

Scalability Scalability is a critical factor that determines the practicality of binary similarity tools, as the number of binaries and functions in a codebase can be large. The experiments in Section 10.5 demonstrate the practicality of VexIR2Vec. The peephole extraction algorithm is linear on the number of basic blocks that constitute the CFG (Section 4.3.1) as opposed to the other approaches that use GNNs and matrix factorizations. As the normalizations are applied to the straight-line peepholes, the time taken is linear in the length of the peephole. As we use simple models, the training time of VexNet is about 5–8 seconds per epoch, resulting in an improvement of about 1080×–5000× in comparison to systems like SAFE [Massarelli et al., 2019] and BinFinder [Qasem et al., 2023]. Inference, i.e., the usage of the trained system, is equally fast. VexIR2Vec is $\approx 3.1\times$ faster than SAFE [Massarelli et al., 2019], $\approx 3.5\times$ faster than BinFinder [Qasem et al., 2023], and orders-of-magnitude faster than DeepBinDiff [Duan et al., 2020].

Availability The implementation of VexIR2Vec is publicly available as a Python library, which uses only open-source software. VexIR2Vec can either be used as a command-line tool, or via a web interface.

Contributions The following are our key contributions in applying VexIR2Vec's Path-Aware embeddings for binary similarity:

- We design VexNet, a lightweight Siamese network with attention that fine-tunes VexIR2Vec's Path-Aware embeddings for binary similarity tasks.
- We demonstrate state-of-the-art precision across four adversarial settings: cross-optimization (40% improvement), cross-compiler (18%), cross-architecture (21%), and obfuscation (60%).
- We achieve practical scalability with training times of 5–8 seconds per epoch and inference 3.1×–3.5× faster than competing approaches.
- We provide a publicly available implementation as a Python library and web interface.

Organization The rest of this chapter is organized as follows: Section 10.2 discusses our proposed approach, demonstrating how it addresses these challenges. Section 10.3 details our VexIR2Vec based ML model for similarity. Evaluation is presented in the subsequent sections. Section 10.4 describes the experimental setup, followed by results and ablation studies in Sections 10.5 and 10.6, respectively. Section 10.7 reviews related work, highlighting the novelty of our approach. Finally, Section 10.8 summarizes the paper's findings and potential future directions.

10.2 VexIR2Vec based Binary Similarity

Following the schema established in Section 9.2.1, we employ *distance-based* similarity through the Siamese network architecture, enabling both diffing and searching tasks.

Figure 10.1 outlines the approach that we propose to solve the binary similarity problem. Section 10.2.1 provides a brief overview of this approach, and Section 10.2.2 explains how it meets the list of requirements previously enumerated in Section 4.2.3.

10.2.1 From Programs to Vectors to Tasks

Figure 10.1 illustrates the overview of our pipeline designed to transform the binary representation of a program into a fixed-size vector. Our approach comprises three main phases:

I: Decomposing functions into a set of normalized peepholes (Section 4.3 on page 69).

II: Pre-training the vocabulary to embed these peepholes as vectors (Section 4.4 on page 79).

III: Fine-tuning the resulting vectors to address downstream tasks such as binary diffing and searching (Section 10.3).

Phase I: From Binaries to Normalized Peepholes In Phase I, we extract the VEX-IR from binaries generated with different compiler configurations. Functions are then decomposed into a set of peepholes derived from their corresponding Control-flow Graphs (CFG), as detailed in Section 4.3. The resulting intermediate representation (VEX-IR) undergoes normalization through VEX-IR Normalization Engine, VexINE. This engine applies transformations inspired by traditional compiler optimizations, effectively *normalizing* the VEX-IR representation of each peephole. Prior to invoking these passes, we canonicalize the input IR to abstract away details that are unnecessary to solve the binary similarity problem as discussed in Sections 4.3.2 and 4.3.3

Phase II: From Peepholes to Embeddings In Phase II, we learn—without supervision—the vocabulary of VEX-IR from a corpus of binaries as detailed in Section 4.4. This vocabulary facilitates the mapping of the *entities* in the intermediate representation (IR), such as opcodes, types, and arguments to n-dimensional vector representations. This step of learning a vocabulary occurs once in an *offline* manner. Subsequently, we derive n-dimensional representations for peepholes and functions using the learned vocabulary.

Phase III: Fine-Tuning Embeddings for Downstream Tasks In Phase III, we train VexNet, a simple feed-forward siamese network with global attention that learns to combine the embeddings generated at the entity level of the functions for addressing the binary similarity problem. The The objective of the model is to represent functions in a Euclidean space, grouping similar functions together while setting apart

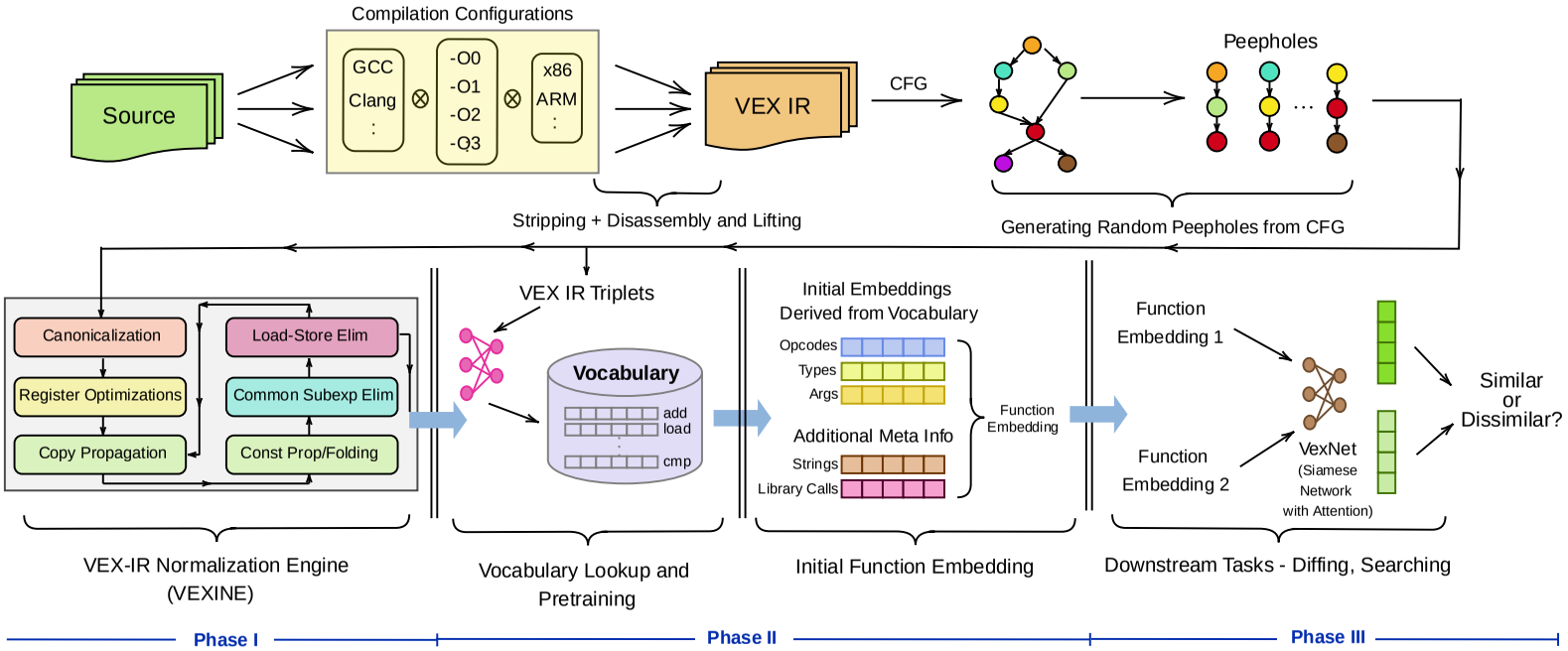


Figure 10.1: Overview of VexIR2Vec: The Control-Flow Graphs of the functions from VEX-IR are generated from different compilation configurations. Then, the peepholes are obtained and normalized by VexINE, the VEX-IR Normalization Engine by using different normalizing transformations. Function embedding is obtained from the embeddings derived from the Opcodes, Types, and Arguments, along with the strings and external library calls used in the function. This embedding is used as input to train VexNet, a simple feed-forward network in the Siamese setting designed to obtain the final embedding of the function.

dissimilar ones. Once trained, the representations generated by the model can be applied to various downstream tasks, such as binary diffing and searching. We describe the model architecture and the training process in detail in Section 10.3.

10.2.2 Meeting the List of Desirable Properties

The series of steps outlined in Figure 10.1 was conceived as a way to meet the three requirements previously stated in Section 4.2.3 on page 68; namely, shape-sensitiveness, normalization, and flow-insensitiveness.

Achieving Shape-Sensitiveness via Random Walks We generate peepholes as random walks on the CFG of the functions. Given three parameters: a function F, a length of the peephole k, and a coverage criterion c, peepholes of maximum length k are produced via random walks on the CFG of F, where each walk crosses k basic blocks. Production stops once each basic block is visited at least c times.

> **Rationale**
> The random walk tends to visit highly connected basic blocks more often. Thus, peepholes are more likely to contain blocks with high in and out degrees in the control-flow graph.

Achieving Normalization through Unsound Optimizations Before being represented as embeddings, peepholes are normalized using transformations inspired by conventional compiler optimizations. Currently, we apply the following transformations on each peephole: register optimization, copy propagation, constant propagation and folding, common expression elimination, and load-store elimination. Our normalizing optimizations are unsound [Morris, 1973]: they are applied to each peephole; hence, they can eliminate operations that are *dead* within a peephole, but that could be necessary outside it.

> **Rationale**
> The objective of VexIR2Vec is to solve binary similarity. These optimizing transformations act as a means of normalization. Thus, we want to be able to eliminate as much redundancy from the sequence of instructions that constitute a peephole as possible so as to preserve the essential relations between variables and operations. In this sense, soundness is not essential.

Achieving Flow-Insensitiveness through Commutative Accumulation Peepholes are transformed into vectors in two steps. First, each peephole is transformed into a single vector via a *mapping function* that converts instructions to vectors. Then, all these vectors are added together.

> **Rationale**
> Addition is commutative; hence, accumulation based on summation ensures that the ordering of instructions plays no hole in the construction of the final vector. However, instructions that appear in many peepholes (as a result of the random walk) will contribute more to the final vector.

With these desirable properties in place, we now describe how the resulting embeddings are fine-tuned for binary similarity tasks through our VexNet model.

10.3 Training Embeddings for Computing Similarity

The combined application of Algorithms 3 on page 71 and 4 on page 82 discussed in Sections 4.3 and 4.4 (Chapter 4) on a function yields its initial representation $[\![\mathbf{F}^{\nu}]\!]$. As seen in Algorithm 4, the initial function representation $[\![\mathbf{F}^{\nu}]\!]$ is derived from the vocabulary function $\mathcal{V}_{lookup}$. The construction of $\mathcal{V}_{lookup}$ depends only on the dataset corpus and is trained in an unsupervised manner; thus, it does not depend on any particular task to be solved by VexIR2Vec. Therefore, to be effectively used, the representation of $[\![\mathbf{F}^{\nu}]\!]$ must be fine-tuned to obtain the actual VexIR2Vec embeddings to solve particular tasks, such as binary diffing or searching. This section explains this process of fine-tuning.

10.3.1 VexNet—Siamese Network with EAN modules

This process of fine-tuning depends on a model that learns to combine the entities $\langle [\![\mathbf{O}]\!], [\![\mathbf{T}]\!], [\![\mathbf{A}]\!] \rangle$, along with metadata of the function like strings and external library calls. This resultant embedding is projected on an embedding space in such a way that semantically similar functions are spatially closer than dissimilar functions. We model this process of fine-tuning using the Siamese network [Koch et al., 2015] seen in Figure 10.2. This model is constructed using two identical Entity-Attention Network (EAN) modules. The rest of this section explains this model, which we call VexNet.

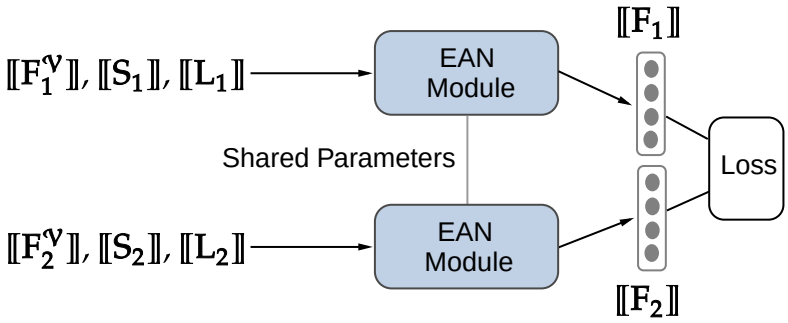


Figure 10.2: VexNet—Siamese network used for fine-tuning the embeddings. This network consists of an Entity-Attention (EAN) module (Figure 10.3) that learns to combine $\langle [\![\mathbf{O}]\!], [\![\mathbf{T}]\!], [\![\mathbf{A}]\!] \rangle$ and $[\![\mathbf{S}]\!], [\![\mathbf{L}]\!]$ to obtain a function embedding vector $[\![\mathbf{F}]\!]$ to learn similarity metric.

A Siamese network is a neural network configuration that contains two identical subnetworks that share the same learning parameters and weights [Koch et al., 2015]. We use the EAN module (Section 10.3.2) in the Siamese configuration to learn an embedding $[\![\mathbf{F}]\!]$ that differentiates similar and dissimilar binary functions. Thus, our training set consists of a collection of pairs of similar and dissimilar functions. In other words, pairs of binary codes obtained from the same source but compiled with different compilers, optimization levels, or architecture targets constitute similar pairs. Pairs that do not originate from the same source, in turn, form a set of dissimilar samples. Normalized temperature-scaled cross-entropy loss (NT-Xent) [Chen et al., 2020] is used to train the model. The model is trained end-to-end to fine-tune the initial embeddings $[\![\mathbf{F}^{\mathcal{V}}]\!]$ derived from the vocabulary to obtain the final VexIR2Vec embeddings $[\![\mathbf{F}]\!]$ of the function. This training process ensures that the embeddings of similar functions are positioned closer in the embedding space and the dissimilar functions are pushed farther apart.

Obtaining strings and library call representations $[\![\mathbf{S}]\!]$ and $[\![\mathbf{L}]\!]$

Strings and calls to the external library methods from *libc* and *libstdc++* seldom change with the variations in compilers, optimizations, and the target architecture. Hence, they can act as valuable additional information in the binary similarity analysis. Our approach leverages this information to enhance the effectiveness of the solution. To incorporate this data into our model, each string used in a function is represented using embeddings generated by a pre-trained model. We use a lightweight fastText model [Bojanowski et al., 2017] to obtain embeddings in $\mathbb{R}^n$. The embeddings for each string are summed

to produce a single representation for all strings used in the function, denoted as $[\![\mathbf{S}]\!]$. A similar process is applied to represent the library calls. The function names of these calls are treated as string tokens, and their representations are obtained using the fastText model. As with the strings, the representation of library calls, $[\![\mathbf{L}]\!]$, is obtained by summing the individual embeddings.

10.3.2 Entity-Attention Network

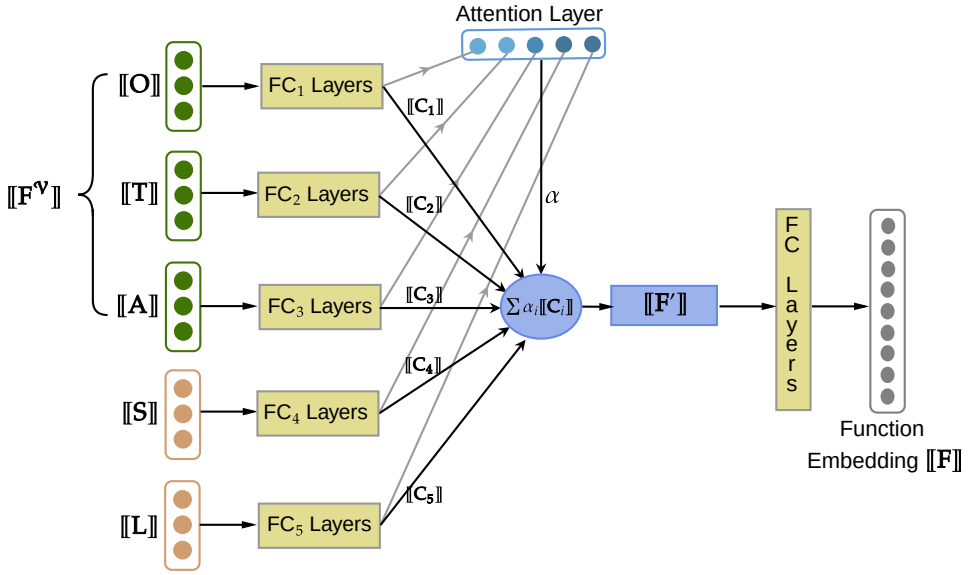


Figure 10.3: Entity-Attention Network (EAN) module used in the VexNet model. The EAN module learns to combine the embeddings of the entities and metadata to obtain the final function embedding.

The EAN module (Figure 10.3) is a simple feed-forward neural network with a global attention [Bahdanau et al., 2015] layer. This global attention layer combines the entities of the initial embeddings $[\![\mathbf{F}^v]\!] = \langle [\![\mathbf{O}]\!], [\![\mathbf{T}]\!], [\![\mathbf{A}]\!] \rangle$ in $\mathbb{R}^{3n}$ with the strings $[\![\mathbf{S}]\!]$ and library calls $[\![\mathbf{L}]\!]$ in $\mathbb{R}^{2n}$, projecting the resulting vector onto $\mathbb{R}^n$. It uses this projection to obtain the final function embedding $[\![\mathbf{F}]\!]$ in $\mathbb{R}^n$. Embeddings of the entities and metadata are passed through a set of independent, Fully Connected (FC) layers. These layers process the raw input embeddings derived from the vocabulary and metadata into context vectors $([\![\mathbf{C}_1]\!], [\![\mathbf{C}_2]\!], [\![\mathbf{C}_3]\!], [\![\mathbf{C}_4]\!], [\![\mathbf{C}_5]\!])$. The final context vector $[\![\mathbf{C}]\!]$ is produced as follows:

$$\begin{gathered}[\![\mathbf{C}_1]\!] = \mathrm{FC}_1([\![\mathbf{O}]\!]); \quad [\![\mathbf{C}_2]\!] = \mathrm{FC}_2([\![\mathbf{T}]\!]); \quad [\![\mathbf{C}_3]\!] = \mathrm{FC}_3([\![\mathbf{A}]\!]); \\ [\![\mathbf{C}_4]\!] = \mathrm{FC}_4([\![\mathbf{S}]\!]); \quad [\![\mathbf{C}_5]\!] = \mathrm{FC}_5([\![\mathbf{L}]\!]).\end{gathered} \tag{10.1}$$

$$[\![\mathbf{C}]\!] = [[\![\mathbf{C}_1]\!], [\![\mathbf{C}_2]\!], [\![\mathbf{C}_3]\!], [\![\mathbf{C}_4]\!], [\![\mathbf{C}_5]\!]] \tag{10.2}$$

The attention module learns to combine these context vectors into a single vector by learning the weights of each of the contexts. An attention vector $\mathbf{u} \in \mathbb{R}^n$ is learned during the training and is used to compute the attention weights. Attention weight (α_i) for each of the contexts ($[\![\mathbf{C}_i]\!], 1 \leq i \leq 5$ is computed as the dot-product of the context and attention vector, followed by a softmax function over all the attention weights.

$$\alpha_i = \frac{\exp([\![\mathbf{C}_i]\!]^T \cdot \mathbf{u})}{\sum_{j=1}^{5} \exp([\![\mathbf{C}_j]\!]^T \cdot \mathbf{u})} \tag{10.3}$$

Equation 10.3 ensures the attention weights

$$\alpha_i \in (0, 1); \qquad \sum_{i=1}^{5} \alpha_i = 1$$

Finally, the attention weights are used to aggregate the individual context vectors into a single vector in $\mathbb{R}^n$ according to Equation 10.4.

$$[\![\mathbf{F}']\!] = \sum_{i=1}^{5} \alpha_i [\![\mathbf{C}_i]\!] \tag{10.4}$$

This attention mechanism helps the network to automatically identify the relative importance of entities of the function. In this sense, notice that this differs from IR2Vec (Chapter 3), which relied on a fixed heuristic to weigh the relative importance of the different entities that make up a function. The aggregated function vector ($[\![\mathbf{F}']\!]$) is then passed through another set of fully connected layers to obtain the final function embedding $[\![\mathbf{F}]\!]$. This final embedding vector is a compact and more informative representation that captures the similarity metric of the binary function. We call this embedding the VexIR2Vec vector that represents the target function, as Definition 10.1 emphasizes.

Definition 10.1: Final VexIR2Vec Embeddings

If $[\![\mathbf{F}^{\nu}]\!]$ is the initial function representation derived from the vocabulary $\mathcal{V}$ (Algorithm 4) on a set of peepholes $\mathcal{P}$ (Algorithm 3), then the embedding obtained from the VexNet, $[\![\mathbf{F}]\!]$ is the final VexIR2Vec representation of the function F.

10.3.3 Binary Similarity Tasks

Upon training the VexNet model, the final function embeddings $[\![\mathbf{F}']\!]$ can solve binary similarity tasks like diffing and searching. Definition 10.2 formalizes these tasks.

Definition 10.2: Binary Similarity Tasks

A binary similarity task is a problem involving two collections of binary programs produced from the same or different source codes, albeit using different compilation settings. We recognize two types of binary similarity tasks:

Diffing: $\mathcal{M}_{diff}(A, B)$, where $A = \{a_1, a_2, \ldots, a_m\}$ and $B = \{b_1, b_2, \ldots, b_n\}$ are the two sets of VexIR2Vec vectors for the functions in binaries A and B, respectively. $\mathcal{M}_{diff}$ produces a list L_{diff} of $\min(m, n)$ pairs (a_i, b_j). Each $a_i, 1 \leq i \leq m$ and each $b_j, 1 \leq j \leq n$ appear only once in L_{diff}. The goal of $\mathcal{M}_{diff}$ is to match every function in A to its equivalent in B.

Searching: $\mathcal{M}_{search}(A, b)$, where $A = \{a_1, a_2, \ldots, a_m\}$ is a collection of m VexIR2Vec vectors obtained from different set of binaries, and b is a VexIR2Vec vector. $\mathcal{M}_{search}$ produces one vector $a_i, 1 \leq i \leq m$. The goal of $\mathcal{M}_{search}$ is to find, within A, a vector a_i representing a function that solves the same task as the function that b represents.

$\mathcal{M}_{diff}(A, B)$ assumes that each $a_i, 1 \leq i \leq m$, comes from a function extracted from a binary program P_A, being derived via the combination of Algorithms 3 and 4, following Definition 10.1. Similarly, each $b_i, 1 \leq i \leq n$, comes from a function extracted from a binary program P_B. We assume that P_A and P_B are binary programs derived from the same source code. Whereas, $\mathcal{M}_{search}(A, b)$ assumes that each $a_i, 1 \leq i \leq m$ comes from the functions extracted from a collection of different binary programs. And, the query function b is searched to retrieve similar functions among the pool of functions in A.

Example 10.3.1: Diffing

Let P_s be a source program, and let P_{ARM} and P_{x86} be binary versions of P_s, compiled for ARM and x86. If we let A be the set of all the VexIR2Vec vectors extracted from P_{ARM}, and B be the set of all the VexIR2Vec vectors extracted from P_{x86}, then $\mathcal{M}_{diff}(A, B)$ tries to match up the binary functions in P_{ARM} and P_{x86}.

Example 10.3.2: Searching

Let P_s be a source program, and let P_{O0}, P_{O1}, P_{O2} and P_{O3} be binary versions of P_s, compiled with different optimization levels. If we let A be the collection of VexIR2Vec vectors extracted from a corpus of binaries including all functions from P_{O1}, P_{O2} and P_{O3}, as well as other from unrelated programs P'. And, let b be VexIR2Vec representation of some function in P_{O0}, then $\mathcal{M}_{search}(A, b)$ searches for functions performing same task as b in A.

10.4 Experimental Setup

This section describes the experimental setup used in the evaluation discussed in Sections 10.5 and 10.6.

10.4.1 Dataset

The dataset used in our experiments comes from the following projects: Findutils [Findutils, 2024], Diffutils [Diffutils, 2024], Coreutils [Coreutils, 2024], cURL [Stenberg, 2024], Lua [lua, 2024], PuTTY [PuTTY, 2024], and Gzip [Gzip, 2024]. Binaries are generated by varying the compilation configuration, which includes architecture (x86 or ARM), compiler (Clang or GCC), compiler version (six versions per compiler), and optimization level (`-O0`, `-O1`, `-O2`, and `-O3`). Thus, in total, we use $2 \times 2 \times 6 \times 4 = 96$ compiler configurations. Experiments are performed on stripped binaries, which do not contain debug and symbol information. Stripping is done using the GNU `strip` utility. However, as we explain in Sections 10.5.1 and 10.5.2, we use unstripped binaries with debug information to aid in obtaining ground truth and identifying the same function across binaries. The size of individual unstripped binary files ranges from 9.2KB to 7.6MB.

We use 70% of the binaries from Findutils, Diffutils, and Coreutils for training and the remaining 30% for testing. All binaries from the other projects are used exclusively

for testing and studying the generalizability of our approach. The training set includes binaries compiled using 6 different versions of Clang (V6.0, 8.0.1, 9.0.1, 10.0.1, 11.1.0, and 12.0.1) and 6 different versions of GCC (V6.4.0, 7.5.0, 8.4.0, 9.4.0, 10.5.0, and 11.4.0). In total, the training set contains $10K$ binaries with $1.5M$ functions. The test set consists of about $5.5K$ binaries with $1.2M$ functions. These binaries are produced with the same compiler configurations, except that to reduce the time required for experiments, we use only three versions of each compiler: Clang V6.0, 8.0.1, and 12.0.1, and GCC V6.0, 8.0, and 10.0. These test-set functions are used in the experiments described in Section 10.5.1 and Section 10.5.2. Appendix D.1 summarizes the total number of projects, files, and functions, along with their sizes.

We use `angr` (V9.2.96) to disassemble the binaries and lift them to obtain the VEX-IR representation. Embeddings are constructed from the disassembled functions by creating peepholes following Algorithm 3, setting the maximum length of peepholes (k) to 72 and the minimum number of visits per basic block (c) to 2. The resulting peepholes are normalized as described in Section 4.3.3, and embeddings are obtained using the approaches described in Section 4.3 and Section 4.4.

10.4.2 Training

Training is the process of building a function that maps sets of peepholes $\mathcal{P}$ to vectors $[\![\mathbf{F}]\!]$, as explained in Sections 4.4 and 10.3.

Pre-Training to obtain $\mathcal{V}_{lookup}$

Section 4.4 defines the vocabulary function $\mathcal{V}_{lookup}$, which maps IR entities to 128-dimensional vectors. Each dimension is a 64-bit floating-point number. Here, $\mathcal{V}_{lookup}$ was learned from triplets extracted from two benchmark suites: SPEC CPU2006 and SPEC CPU2017 [Bucek et al., 2018]. These benchmarks were compiled using the Clang V6.0, V8.0.1, and V12.0.1, and GCC V6.0, V8.0, and V10.0 with different optimization levels (`O[0-3]`) targeting x86 and ARM.

We used `angr` to extract triplets and an open-source implementation of TransE [Han et al., 2018] to train the model. The dataset contained approximately $72M$ triplets, from which we identified 145 unique entities and 10 unique relations. The model converged after about 900 epochs using the Adam optimizer with a learning rate of 0.002. The margin γ was set to 3, and the batch size was 256. These hyperparameters were chosen based on tuning the validation loss.

Fine-Tuning—Training VexNet to obtain $[\![\mathbf{F}]\!]$

As explained in Section 10.4.1, to train the VexNet model (see Section 10.3), we use a dataset formed by Findutils, Diffutils, and Coreutils. The inputs for the VexNet model are the initial function embedding $[\![\mathbf{F}^{\nu}]\!] = \langle [\![\mathbf{O}]\!] \in \mathbb{R}^{1\times 128}, [\![\mathbf{T}]\!] \in \mathbb{R}^{1\times 128}, [\![\mathbf{A}]\!] \in \mathbb{R}^{1\times 128} \rangle$ obtained from the vocabulary $\mathcal{V}_{lookup}$, along with the embedding of strings ($[\![\mathbf{S}]\!] \in \mathbb{R}^{1\times 100}$) and external library calls ($[\![\mathbf{L}]\!] \in \mathbb{R}^{1\times 100}$) obtained using the fastText model.

A single fully connected layer with SiLU activation [Hendrycks and Gimpel, 2016] processes $[\![\mathbf{F}^{\nu}]\!], [\![\mathbf{S}]\!], [\![\mathbf{L}]\!]$ (Eq. 10.2). Before passing to the model, the inputs are L_2 normalized[1]. Internally, the fully connected layer converts the inputs into context vectors of 180 dimensions each. These context vectors are attended to by the attention layer to obtain $[\![\mathbf{F}']\!]$ in 180 dimensions (Eq. 10.4). $[\![\mathbf{F}']\!]$ is then processed by the final fully connected layer to obtain $[\![\mathbf{F}]\!]$ in 128 dimensions.

Deep metric learning models can suffer from *collapsing*, where the model learns to embed all or most of the inputs very close to each other at a negligible distance, rendering them indistinguishable [Tian et al., 2024; Xuan et al., 2020a; Schroff et al., 2015]. To overcome this phenomenon, we use batch easy hard mining [Xuan et al., 2020b] to select positive and negative samples for training and employ dropout (0.02) and batch normalization [Ioffe and Szegedy, 2015] as regularizers. We train the model with Adam optimizer with default parameters.

Other hyperparameters like batch size, learning rate, gamma (for Adam), and temperature (for NT-Xent loss) are tuned using RayTune [Liaw et al., 2018] to find the best-performing values. For hyperparameter tuning, we use the ASHA scheduler [Li et al., 2018] with Optuna search [Akiba et al., 2019] with default parameters to select the best model based on the Mean Average Precision (MAP) score on the validation set. We use an Ubuntu server with Intel Gold 6142 processors (32 threads) and a V100 GPU (32 GB) for hyperparameter tuning. Each trial is tuned until convergence, allowing a maximum of 200 epochs. This process results in the best model with a batch size of 4064, a gamma of 0.817, and a temperature of 0.05. Training a single trial is lightweight. For instance, training takes about 5–8 seconds per epoch on an Ubuntu workstation with Xeon(R) W-1390P processor with 16 threads, 64 GB RAM, and a 10 GB variant of an RTX 3080 GPU. In contrast, training SAFE and BinFinder on the same machine takes $\approx$ 1.5 hours and $\approx$ 7.5 hours per epoch, respectively.

[1]The L_2-norm is the vector norm, e.g., the square root of the sum of the squares of all the dimensions of the vector.

10.4.3 Baselines

We compare VexIR2Vec (V2V) with five different tools: BinDiff [Zynamics, 2024], DeepBinDiff [Duan et al., 2020], SAFE [Massarelli et al., 2019], Histograms of Opcodes [Damásio et al., 2023], and BinFinder [Qasem et al., 2023].

BinDiff BinDiff (BD) extracts the control flow graph (CFG) information of functions in a binary. It compares the number of blocks, edges, and calls between functions in the source and target binaries to obtain an initial match. BinDiff identifies similar function pairs and provides a similarity score and confidence value. It uses disassemblers like IDA Pro [Hex-Rays, n.d.] and Ghidra [Ghidra] to extract relevant information and convert it to the BinExport format. For our experiments, we use BinDiff (V7) to generate the BinExport files using Ghidra (V9.2.3) with the BinExport plugin (V12).

DeepBinDiff DeepBinDiff (DBD) uses an unsupervised learning technique to obtain embeddings from the control flow graph (CFG) of a function. A greedy matching algorithm is then used to provide block-level diffing between two binaries. We use the code provided by the authors of DeepBinDiff for our experiments. To perform function-level diffing, we follow the approach used by Codee [Yang et al., 2022], where two functions are considered similar if at least one pair of blocks between them is predicted to be similar.

SAFE SAFE uses Radare [Team, 2017] to disassemble binary files and trains a word2vec model for instruction-to-embedding mapping. This mapping is used to generate an embedding for each function. SAFE does not process functions if Radare returns instructions with invalid opcodes. We skip such instructions and allow SAFE to construct the function embedding from the remaining instruction sequence. We use the code and datasets provided by the authors of SAFE and the pre-trained model for the x86 experiments. However, the public distribution of SAFE does not include a pre-trained model for the ARM architecture. Therefore, we train the model using the cross-architecture dataset from SAFE to create a model for all our experiments involving ARM binaries.

Histograms of Opcodes In their recent study, Damásio et al. [2023] demonstrated that representing programs using histograms of opcodes is as effective as other source code embedding techniques. They showed that using opcode frequencies of LLVM IR [LLVM, 2018a] as input features can effectively identify similar codes, even when

compiler optimizations and obfuscation transformations are applied. We implemented this technique on binaries using `angr` and consider it as a baseline for comparison. We create a 144-dimensional feature vector, with each dimension representing the frequency of a specific opcode in VEX-IR of the function. We then train the VexNet model using these histogram representations on our training dataset. The model's hyperparameters were fine-tuned based on validation loss, similar to the process described in Section 10.4.2. The best model obtained through this process is then used for evaluation. In our experiments, we refer to this technique as OPC.

BinFinder BinFinder (BF) uses a similar setup as SAFE. We implemented the necessary feature extraction using `angr` as described in the paper to create a functional setup. We trained the BinFinder model on our training set using the provided training scripts and then evaluated the model on our test set for comparison.

10.5 Performance Evaluation

This section evaluates the performance of VexIR2Vec, in terms of accuracy and scalability, on the two binary similarity experiments formalized in Definition 10.2: function diffing and searching. This evaluation happens in an adversarial setting, involving binaries produced using different compilation configurations, as explained in Section 10.4.2. The goal of this evaluation is to provide answers to the following research questions:

- How well does our approach perform on the binary diffing task, withstanding the differences arising out of different architectures, compilers, optimizations, and obfuscations? (Section 10.5.1)
- How effective is our approach in searching and retrieving a similar function from a large pool of functions varying in compilation configurations? (Section 10.5.2)
- How scalable is our proposed tool in comparison with the other approaches? (Section 10.5.3)

10.5.1 Binary Diffing

We evaluate our approach on the binary diffing task at the function level, as described in Section 10.2 and compare it with different baselines.

Ground truth

The ground truth for the binary diffing experiment consists of function pairs from stripped source and target binaries, corresponding to the same source code function compiled using different configurations. To identify functions from the same source code, we first match functions from stripped binaries with their unstripped equivalents using their function addresses. Then, we obtain the debugging and symbol information from the unstripped functions. A pair of functions from the stripped source and target binaries with the same source filename and function name are considered identical and added to the ground truth.

Example 10.5.1:

To create a ground truth function pair between the `find` binaries compiled with x86-Clang12-O0 and x86-GCC10-O2, we first match the function addresses in the stripped and unstripped binaries to associate the source filename and function name from the unstripped function with the stripped function. This process is performed separately for each of the `find` binaries compiled with x86-Clang12-O0 and x86-GCC10-O2. Next, we pair functions from the stripped x86-Clang12-O0 binary with those from the x86-GCC10-O2 binary by checking for matching source filenames and function names. When a match is found, we identify two functions derived from the same source code and add this pair to the ground truth.

Metrics

We use precision, recall, and F1 score as the primary evaluation metrics for the binary diffing task.

- True Positive (TP): number of pairs of functions correctly predicted in the ground truth.
- False Positive (FP): number of pairs of functions predicted as similar but not present in the ground truth.
- False Negative (FN): number of pairs of functions in the ground truth that are not predicted to be similar.
- Precision ($\frac{TP}{TP+FP}$): the proportion of true positives among all positive results predicted by the model.

- Recall ($\frac{TP}{TP+FN}$): the proportion of true positives among all actual positive cases in the dataset.
- F1 score: the harmonic mean of precision and recall, computed as $\frac{2*Precision*Recall}{Precision+Recall}$.

DeepBinDiff and BinDiff directly generate matching function pairs for two binaries with a similarity score. For other approaches, we use the KDTree implementation from scikit-learn [Pedregosa et al., 2011] to compute the nearest neighbors for a source function among the set of target functions. KDTrees are binary search trees where data in each node is a K-Dimensional point. KDTrees provide an efficient means[2] of computation of nearest neighbors [Bentley, 1975]. We consider 10 neighbors in our experiments. A prediction is counted as a true positive (TP) if the target function is among the computed neighbors; otherwise, it is marked as a false positive (FP). Additionally, we plot the Cumulative Distribution Function (CDF) of F1-Score curves for each experiment.

Some functions identified in the ground truth are not present in the baselines. Such function pairs are considered false negatives according to our definitions. Experiments use a timeout of 7,200 seconds, beyond which we terminate the process for the pair of binaries under comparison. We skip evaluating DeepBinDiff on binaries from Lua, cURL, and PuTTY, and in experiments involving obfuscated binaries, as generating the embeddings and matching exceeded the 7,200-second timeout for almost 90% of the binary pairs. Additionally, we do not evaluate DeepBinDiff on cross-architecture experiments since the approach is architecture-specific and supports only x86.

Cross-Optimization Binary Diffing

We perform cross-optimization level binary diffing involving binaries compiled with Clang12 and GCC8 targeting x86 and ARM architectures. Table 10.1[3] shows the precision and recall values corresponding to the diffing between O0-O3 optimization level. Additionally, we studied the performance of diffing between O0-O2 and O1-O3 optimization levels (results are shown in Tables D.5 and D.4 of the Appendix). Our approach consistently achieves higher precision and recall values across all configurations compared to the baselines. It attains the best average F1-Score across all three experiments, at

[2]They have a linear-space complexity, an average logarithmic-time complexity, and a worst-case linear-time complexity.

[3]Henceforth, we add to the caption of each table the expected score of a random guesser (the null hypothesis). This score, being so low, demonstrates that the different binary similarity tools are able to learn non-trivial information.

Table 10.1: Precision and Recall Scores for Cross-Optimization Binary Diffing in O0 Vs. O3 setting. Missing pairs occur due to timeouts. The null hypothesis' expected score is 0.005.

	Precision						Recall					
	BD	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**	**BD**	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**
						ARM - Clang12						
Coreutils	0.08	0.17	0.11	0.36	0.35	**0.57**	0.05	0.09	0.09	0.47	0.46	**0.75**
Diffutils	0.37	0.24	0.17	0.44	0.49	**0.68**	0.27	0.19	0.13	0.57	0.65	**0.89**
Findutils	0.11	0.31	0.12	0.40	0.47	**0.68**	0.06	0.15	0.10	0.48	0.56	**0.82**
Gzip	0.07	0.15	0.16	0.32	0.41	**0.60**	0.05	0.16	0.11	0.42	0.54	**0.78**
Lua	0.20		0.04	0.13	0.19	**0.41**	0.19		0.03	0.21	0.31	**0.65**
PuTTY	0.07		0.04	0.06	0.21	**0.28**	0.04		0.04	0.08	0.29	**0.39**
cURL	0.14		0.12	0.50	0.58	**0.67**	0.12		0.08	0.57	0.67	**0.77**
						ARM - GCC8						
Coreutils	0.13	0.16	0.18	0.36	0.34	**0.51**	0.09	0.11	0.13	0.49	0.47	**0.70**
Diffutils	0.29	0.19	0.32	0.36	0.43	**0.60**	0.24	0.09	0.21	0.52	0.62	**0.86**
Findutils	0.13	0.18	0.22	0.41	0.42	**0.59**	0.09	0.12	0.18	0.52	0.55	**0.77**
Gzip	0.23	0.14	0.33	0.17	0.38	**0.50**	0.21	0.13	0.21	0.25	0.55	**0.73**
Lua	0.13		0.10	0.15	0.20	**0.32**	0.13		0.07	0.25	0.33	**0.52**
PuTTY	0.05		0.08	0.09	0.19	**0.28**	0.03		0.08	0.12	0.23	**0.39**
cURL	0.19		0.21	0.53	0.14	**0.57**	0.15		0.16	0.69	0.58	**0.76**
						x86 - Clang12						
Coreutils	0.12	0.12	0.24	0.33	0.34	**0.53**	0.09	0.36	0.20	0.44	0.44	**0.71**
Diffutils	0.48	0.17	0.31	0.45	0.43	**0.68**	0.44	0.55	0.22	0.60	0.57	**0.91**
Findutils	0.27	0.12	0.32	0.38	0.50	**0.66**	0.21	0.47	0.26	0.47	0.61	**0.82**
Gzip	0.22	0.11	0.35	0.38	0.42	**0.60**	0.18	0.66	0.25	0.49	0.54	**0.79**
Lua	0.26		0.17	0.11	0.21	**0.39**	0.24		0.12	0.17	0.32	**0.59**
PuTTY	0.11		0.12	0.06	0.20	**0.30**	0.10		0.12	0.08	0.27	**0.40**
cURL	0.25		0.35	0.42	0.51	**0.74**	0.20		0.25	0.48	0.59	**0.84**
						x86 - GCC8						
Coreutils	0.11	0.12	0.23	0.31	0.32	**0.50**	0.08	0.29	0.18	0.44	0.44	**0.69**
Diffutils	0.27	0.14	0.37	0.42	0.38	**0.60**	0.25	0.42	0.25	0.60	0.55	**0.86**
Findutils	0.10	0.10	0.26	0.34	0.40	**0.60**	0.08	0.31	0.21	0.44	0.52	**0.78**
Gzip	0.30	0.11	0.34	0.18	0.34	**0.52**	0.32	0.53	0.22	0.25	0.48	**0.75**
Lua	0.14		0.14	0.15	0.22	**0.31**	0.14		0.10	0.24	0.35	**0.50**
PuTTY	0.09		0.12	0.08	0.16	**0.29**	0.08		0.12	0.11	0.23	**0.41**
cURL	0.16		0.34	0.55	0.58	**0.79**	0.15		0.24	0.64	0.69	**0.93**

0.65, while the nearest baselines, BinFinder and Histograms of Opcodes (OPC), achieve F1-Scores of 0.47 and 0.41, respectively. On average, SAFE, DeepBinDiff, and BinDiff achieve F1 scores of 0.28, 0.27, and 0.26, respectively.

Comparison with GPT-4o Given the current popularity that large-language models enjoy today, a natural question is: what is the precision of VexIR2Vec when compared against them? To address this question, we compare VexIR2Vec with OpenAI's

ChatGPT (GPT-4o) on the task similarity problem. For this experiment, we selected function pairs from the stripped `gzip` binary compiled with clang12 at optimization levels O0 (binary A) and O2 (binary B). After disassembling the binaries with `angr`, we randomly chose 30 function pairs from the ground truth. These pairs were curated to fit within OpenAI's token input limits. The comparison involved writing the assembly code of functions from binary A to a file and providing the following prompt to GPT-4o:

> "*Consider yourself an expert in binary similarity. The attached file contains assembly code disassembled from a binary. I will provide additional assembly code functions from another binary. Both binaries were compiled from the same source. Your job is to identify the IDs corresponding to semantically equivalent functions.*"

Next, we fed the model the assembly code of functions from binary B, one at a time, and manually reviewed its responses to evaluate performance.

Performance Results GPT-4o demonstrated a strong understanding of the input prompts and generated plausible function IDs for all 30 inputs. However, its precision in correctly identifying semantically equivalent function pairs was limited. The model correctly identified 7 out of 30 pairs, yielding an accuracy of 23.33%. In the same zero-shot setting, VexIR2Vec achieved an accuracy of 62.12%, significantly outperforming GPT-4o despite having no prior exposure to the gzip binary during training. Due to token input limits and other restrictions on OpenAI's platform, we were unable to conduct a more exhaustive set of experiments. Nonetheless, this comparison underscores the effectiveness of VexIR2Vec relative to advanced LLMs[4].

Cross-Compiler Binary Diffing

To perform cross-compiler diffing, we consider binaries generated using Clang12 and GCC8 targeting x86 and ARM. Diffing is performed on binaries that target the same architecture but that were produced with different compilers or different versions of the same compiler. Table 10.2 shows the precision and recall for our approach and the baselines. Our approach achieves the highest precision, recall, and F1 scores, outperforming the baselines in all configurations.

[4]We also attempted a similar experiment with Google AI Studio's Gemini 1.5 Flash model. However, after processing a few inputs, the model consistently produced summaries of the input functions instead of the requested function IDs, making it unsuitable for meaningful comparisons.

Table 10.2: Cross-Compiler Binary Diffing - Clang12 Vs. GCC8. Null hypothesis' expected score: 0.006

	Precision						Recall					
	BD	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**	**BD**	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**
						ARM						
Coreutils	0.23	0.34	0.27	0.60	0.66	**0.78**	0.13	0.32	0.22	0.68	0.75	**0.88**
cURL	0.20		0.32	0.65	0.69	**0.79**	0.19		0.22	0.79	0.84	**0.96**
Diffutils	0.28	0.37	0.37	0.64	0.59	**0.78**	0.21	0.39	0.26	0.78	0.79	**0.94**
Findutils	0.41	0.53	0.32	0.60	0.69	**0.83**	0.29	0.36	0.27	0.67	0.78	**0.93**
Gzip	0.37	0.27	0.37	0.29	0.61	**0.71**	0.34	0.56	0.26	0.35	0.72	**0.84**
Lua	0.27		0.19	0.29	0.44	**0.52**	0.26		0.14	0.36	0.54	**0.64**
PuTTY	0.19		0.13	0.26	0.44	**0.49**	0.13		0.15	0.28	0.48	**0.55**
						x86						
Coreutils	0.25	0.16	0.32	0.58	0.63	**0.75**	0.22	0.53	0.27	0.66	0.71	**0.84**
cURL	0.30		0.48	0.76	0.82	**0.87**	0.22		0.35	0.81	0.88	**0.93**
Diffutils	0.42	0.18	0.44	0.65	0.62	**0.81**	0.43	0.32	0.31	0.80	0.76	**0.98**
Findutils	0.34	0.16	0.36	0.57	0.68	**0.82**	0.30	0.59	0.30	0.65	0.76	**0.92**
Gzip	0.34	0.08	0.43	0.32	0.62	**0.7**	0.33	0.72	0.32	0.37	0.70	**0.81**
Lua	0.34		0.29	0.35	0.42	**0.54**	0.33		0.23	0.43	0.51	**0.66**
PuTTY	0.22		0.21	0.38	0.43	**0.51**	0.20		0.24	0.41	0.47	**0.55**

We also carry out diffing by varying the compiler version. Table 10.3 shows the results of the comparisons between Clang6 and Clang12, Clang8 and Clang12, GCC6 and GCC10, GCC8 and GCC10. In our dataset, we observe that the cross-compiler version diffing is easier than other diffing experiments. Our approach results in the highest average F1 score of 0.93, while BinFinder, OPC, and SAFE result in 0.87, 0.8, and 0.53, respectively.

Cross-Architecture Binary Diffing

This experiment uses binaries generated by Clang (V12.0.1) and GCC (V10.0) at `-O0` and `-O3` optimization levels for x86 and ARM. We perform diffing between binaries generated by the same compiler and optimization level but for different architectures. Table 10.4 shows the results. VexIR2Vec achieves the highest precision and recall values in most configurations. Notice that in this experiment, VexIR2Vec is not consistently better than all the other baselines. For instance, in the clang12 `-O0` setup, VexIR2Vec trails BinDiff on the Lua binary, although it outperforms it in all the other six binaries on the same setting. On average, VexIR2Vec achieves the highest F1 score of 0.82, surpassing the closest baseline, BinFinder, which obtains an average F1 score of 0.67.

Table 10.3: Compiler Cross-Version Binary Diffing. Null hypothesis' expected score: 0.006

	Precision						Recall					
	BD	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**	**BD**	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**
	Clang6 Vs. Clang12											
ARM	0.72	0.59	0.51	0.80	0.93	**0.97**	0.57	0.86	0.43	0.80	0.93	**0.98**
x86	0.86	0.34	0.55	0.88	0.94	**0.98**	0.94	0.89	0.48	0.89	0.94	**0.99**
	Clang8 Vs. Clang12											
ARM	0.72	0.62	0.53	0.84	0.94	**0.97**	0.58	0.88	0.45	0.84	0.94	**0.98**
x86	0.88	0.36	0.56	0.89	0.95	**0.99**	0.96	0.92	0.50	0.89	0.95	**0.99**
	GCC6 Vs. GCC10											
ARM	0.58	0.52	0.49	0.72	0.75	**0.84**	0.49	0.82	0.40	0.79	0.83	**0.92**
x86	0.62	0.21	0.53	0.69	0.76	**0.83**	0.66	0.84	0.44	0.76	0.83	**0.91**
	GCC8 Vs. GCC10											
ARM	0.66	0.56	0.51	0.75	0.75	**0.85**	0.56	0.84	0.41	0.81	0.84	**0.92**
x86	0.71	0.21	0.55	0.74	0.77	**0.84**	0.81	0.86	0.46	0.82	0.85	**0.93**

Mixture-of-all Diffing

As an extreme case of adversarial diffing, we conducted experiments using a diverse set of binaries compiled with different compilers and varying optimization levels for both x86 and ARM. This setup poses a significant challenge to the binary diffing tools due to its mix of compilation configurations. Tables 10.5 and 10.6 present the compilation configurations, precision, and recall results for this experiment. Cross-architecture experiments are excluded for DeepBinDiff for the reasons mentioned in Section 10.5.1. VexIR2Vec outperforms all baselines across these configurations in terms of both precision and recall. On average, VexIR2Vec achieves approximately a 45% higher average F1-score compared to the best-performing baseline, BinFinder.

Figure 10.4 shows Cumulative Distributive Function plots for F1-scores. Ideal curves have a maximum increase in the fraction of data close to 1.0. Notice that VexIR2Vec achieves superior results than the baselines in all the experiments that Figure 10.4 summarizes.

Diffing on Obfuscated Binaries

Code obfuscation is a deliberate technique that alters code to make it difficult for humans to understand or interpret its underlying functionality. This section evaluates the ability of the different diffing tools to resist the effects of code obfuscation on the target binaries. To this end, we use O-LLVM to modify binaries before passing them to the diffing

Table 10.4: Cross-Architecture Binary Diffing. Null hypothesis' expected score: 0.005

	Precision					Recall				
	BD	**SAFE**	**OPC**	**BF**	**V2V**	**BD**	**SAFE**	**OPC**	**BF**	**V2V**
					Clang12 - O0					
Coreutils	0.53	0.10	0.41	0.64	**0.79**	0.43	0.09	0.45	0.67	**0.85**
cURL	0.83	0.10	0.41	0.83	**0.94**	0.72	0.08	0.41	0.83	**0.95**
Diffutils	0.71	0.14	0.41	0.70	**0.84**	0.60	0.12	0.44	0.74	**0.9**
Findutils	0.78	0.10	0.37	0.70	**0.85**	0.66	0.09	0.39	0.73	**0.9**
Gzip	0.81	0.08	0.50	0.71	**0.89**	0.78	0.07	0.53	0.76	**0.94**
Lua	**0.87**	0.02	0.12	0.61	0.77	**0.82**	0.02	0.12	0.61	0.77
PuTTY	0.56	0.03	0.10	0.50	**0.65**	0.55	0.03	0.12	0.55	**0.72**
					Clang12 - O3					
Coreutils	0.30	0.17	0.48	0.67	**0.84**	0.14	0.15	0.52	0.73	**0.92**
cURL	0.63	0.26	0.54	0.77	**0.82**	0.51	0.19	0.55	0.78	**0.83**
Diffutils	0.64	0.22	0.55	0.76	**0.88**	0.43	0.17	0.61	0.83	**0.96**
Findutils	0.45	0.23	0.52	0.69	**0.86**	0.21	0.20	0.56	0.74	**0.92**
Gzip	0.55	0.26	0.49	0.75	**0.86**	0.45	0.19	0.53	0.82	**0.94**
Lua	0.80	0.13	0.24	0.70	**0.82**	0.71	0.11	0.25	0.73	**0.85**
PuTTY	0.56	0.10	0.18	0.54	**0.67**	0.41	0.12	0.19	0.57	**0.71**
					GCC10 - O0					
Coreutils	0.78	0.29	0.43	0.71	**0.89**	0.65	0.26	0.46	0.76	**0.94**
cURL	**0.87**	0.36	0.10	0.18	0.21	0.78	0.28	0.47	0.82	**0.94**
Diffutils	0.88	0.39	0.41	0.79	**0.93**	0.76	0.33	0.43	0.82	**0.98**
Findutils	0.86	0.29	0.45	0.73	**0.95**	0.71	0.27	0.47	0.76	**0.98**
Gzip	0.84	0.31	0.49	0.84	**0.88**	0.82	0.25	0.52	0.89	**0.93**
Lua	**0.95**	0.17	0.34	0.59	0.88	**0.91**	0.16	0.34	0.59	0.88
PuTTY	0.59	0.12	0.10	0.47	**0.72**	0.57	0.13	0.12	0.52	**0.78**
					GCC10 - O3					
Coreutils	0.34	0.26	0.33	0.51	**0.61**	0.24	0.20	0.44	0.68	**0.81**
cURL	0.56	0.30	0.52	0.64	**0.75**	0.47	0.21	0.59	0.73	**0.86**
Diffutils	0.48	0.25	0.44	0.71	**0.79**	0.33	0.17	0.52	0.84	**0.93**
Findutils	0.45	0.26	0.34	0.65	**0.68**	0.31	0.22	0.42	0.80	**0.83**
Gzip	0.61	0.24	0.56	0.75	**0.87**	0.52	0.18	0.64	0.84	**0.97**
Lua	0.67	0.16	0.25	0.62	**0.81**	0.59	0.13	0.26	0.64	**0.83**
PuTTY	0.46	0.09	0.21	0.48	**0.67**	0.29	0.10	0.23	0.50	**0.7**

software. O-LLVM is a widely used obfuscation tool integrated with the Clang and LLVM toolchain (specifically, it runs on Clang version 4.0) [Junod et al., 2015]. O-LLVM offers three primary obfuscation schemes:

- **Bogus Control Flow (BCF)**: This scheme adds unnecessary branches guarded

Table 10.5: Cross-Compiler + Cross-Optimization + Cross-Architecture Binary Diffing. Null hypothesis' expected score: 0.005

	Precision					Recall				
	BD	**SAFE**	**OPC**	**BF**	**V2V**	**BD**	**SAFE**	**OPC**	**BF**	**V2V**
	x86-Clang12-O0 Vs. ARM-GCC10-O2									
Coreutils	0.10	0.08	0.21	0.28	**0.45**	0.05	0.05	0.34	0.43	**0.7**
cURL	0.10	0.09	0.25	0.42	**0.57**	0.06	0.06	0.33	0.56	**0.76**
Diffutils	0.19	0.09	0.25	0.31	**0.58**	0.11	0.06	0.37	0.51	**0.86**
Findutils	0.14	0.07	0.17	0.39	**0.56**	0.08	0.05	0.24	0.53	**0.76**
Gzip	0.13	0.08	0.12	0.39	**0.5**	0.10	0.05	0.17	0.54	**0.7**
Lua	0.19	0.04	0.06	0.20	**0.34**	0.16	0.03	0.08	0.30	**0.51**
PuTTY	0.06	0.03	0.04	0.19	**0.22**	0.04	0.03	0.05	0.24	**0.3**
	x86-GCC8-O1 Vs. ARM-Clang6-O3									
Coreutils	0.10	0.13	0.38	0.41	**0.64**	0.05	0.10	0.45	0.49	**0.76**
cURL	0.07	0.18	0.41	0.57	**0.74**	0.04	0.13	0.43	0.59	**0.77**
Diffutils	0.24	0.17	0.36	0.45	**0.72**	0.17	0.12	0.44	0.55	**0.89**
Findutils	0.19	0.14	0.39	0.47	**0.73**	0.09	0.12	0.45	0.54	**0.83**
Gzip	0.28	0.13	0.17	0.47	**0.59**	0.22	0.09	0.20	0.57	**0.72**
Lua	0.22	0.06	0.12	0.27	**0.46**	0.19	0.04	0.15	0.34	**0.57**
PuTTY	0.08	0.06	0.09	0.25	**0.29**	0.05	0.07	0.10	0.28	**0.32**

Table 10.6: Cross-Compiler + Cross-Optimization Binary Diffing. Null hypothesis' expected score: 0.006

	Precision						Recall					
	BD	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**	**BD**	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**
	x86-Clang12-O0 Vs. x86-GCC10-O2											
Coreutils	0.11	0.08	0.20	0.24	0.31	**0.48**	0.06	0.23	0.16	0.34	0.43	**0.67**
cURL	0.14		0.27	0.41	0.50	**0.73**	0.10		0.19	0.49	0.60	**0.86**
Diffutils	0.24	0.11	0.28	0.44	0.39	**0.63**	0.17	0.36	0.19	0.60	0.53	**0.86**
Findutils	0.27	0.08	0.21	0.26	0.38	**0.6**	0.16	0.26	0.17	0.34	0.49	**0.77**
Gzip	0.32	0.09	0.30	0.21	0.39	**0.55**	0.25	0.37	0.21	0.27	0.52	**0.71**
Lua	0.24		0.16	0.11	0.21	**0.36**	0.23		0.12	0.16	0.32	**0.53**
PuTTY	0.10		0.09	0.07	0.21	**0.24**	0.09		0.09	0.09	0.28	**0.33**
	x86-GCC8-O1 Vs. x86-Clang6-O3											
Coreutils	0.15	0.22	0.29	0.61	0.49	**0.72**	0.11	0.46	0.25	0.67	0.54	**0.79**
cURL	0.13		0.39	0.66	0.73	**0.87**	0.09		0.29	0.70	0.77	**0.91**
Diffutils	0.29	0.25	0.41	0.63	0.62	**0.81**	0.25	0.60	0.29	0.71	0.70	**0.92**
Findutils	0.20	0.24	0.32	0.60	0.51	**0.81**	0.15	0.57	0.27	0.66	0.56	**0.89**
Gzip	0.20	0.11	0.26	0.28	0.41	**0.54**	0.17	0.54	0.21	0.33	0.49	**0.64**
Lua	0.29		0.28	0.28	0.34	**0.51**	0.27		0.22	0.34	0.42	**0.62**
PuTTY	0.14		0.20	0.24	0.32	**0.37**	0.11		0.23	0.26	0.35	**0.4**

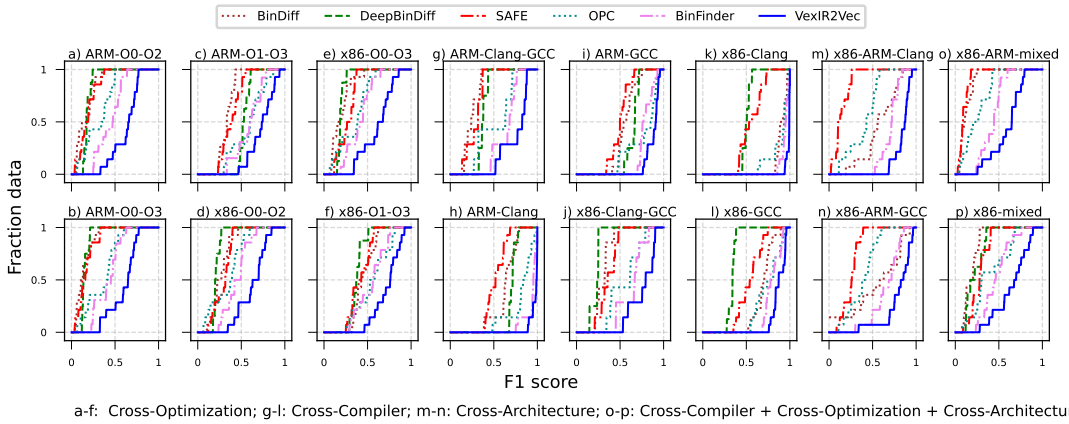


Figure 10.4: The cumulative distribution function (CDF) of F1-Scores in different adversarial settings.

by conditions (opaque predicates) that always evaluate to the same outcome. These branches cannot be eliminated by compiler optimizations, making the code's control flow harder to follow.

- **Control-Flow Flattening (FLA)**: This scheme flattens the Control Flow Graph by modifying the conditions within the graph and inserting additional code blocks that do not affect the program's functionality. In practice, the program becomes a large switch within a loop [László and Kiss, 2009].

- **Instruction Substitution (SUB)**: This scheme replaces a single instruction with a sequence of multiple instructions that achieve the same result. These replacements increase the complexity of the code without changing its behavior.

We generated obfuscated binaries using O-LLVM (Clang V4.0) on x86 at the `-O3` optimization level. In addition to applying the three individual obfuscation passes mentioned above, we also created a variant by applying all three passes together (denoted as "ALL"). These obfuscated binaries were then compared to their non-obfuscated counterparts generated using the GCC compiler (version 10.0) with the optimization level set to `-O0`. The experimental setup was identical to that described in Section 10.5.1. Table 10.7 shows the results of our approach and the baselines[5]. As shown in that table, VexIR2Vec achieves the highest precision and recall values in almost all cases.

[5]We do not consider the binaries from Lua in this experiment as the `angr` (V9.2.96) was not able to retrieve CFGs of the functions after obfuscation using Clang 4.

Table 10.7: Diffing between binaries generated by GCC-10 (with -O0) and the obfuscated version of binaries generated by Clang-4 (with -O3). Null hypothesis' expected score: 0.006

	Precision						Recall					
	BD	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**	**BD**	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**
						BCF						
Coreutils	0.07	0.15	0.24	0.22	0.31	**0.56**	0.04	0.16	0.35	0.28	0.39	**0.72**
cURL	0.05		0.34	0.29	0.68	**0.83**	0.02		0.56	0.31	0.74	**0.9**
Diffutils	0.23	0.14	0.27	0.20	0.39	**0.64**	0.13	0.32	0.49	0.27	0.52	**0.85**
Findutils	0.12	0.13	0.32	0.21	0.40	**0.68**	0.07	0.25	0.42	0.25	0.47	**0.81**
Gzip	0.24		0.32	0.17	**0.37**	0.27	0.16		**0.55**	0.21	0.46	0.34
PuTTY	0.04		0.23	0.06	0.17	**0.63**	0.02		0.21	0.08	0.22	**0.79**
						FLA						
Coreutils	0.09	0.14	0.20	0.15	0.35	**0.55**	0.06	0.07	0.30	0.20	0.46	**0.72**
cURL	0.16		0.22	0.22	0.70	**0.8**	0.08		0.39	0.25	0.80	**0.9**
Diffutils	0.22	0.14	0.20	0.10	0.40	**0.64**	0.16	0.23	0.36	0.14	0.54	**0.86**
Findutils	0.17	0.13	0.24	0.13	0.37	**0.63**	0.12	0.20	0.34	0.16	0.45	**0.77**
Gzip	0.25		0.22	0.20	**0.32**	0.28	0.18		0.41	0.27	**0.42**	0.37
PuTTY	0.08		0.17	0.06	0.22	**0.61**	0.07		0.16	0.08	0.30	**0.8**
						SUB						
Coreutils	0.11	0.15	0.25	0.34	0.31	**0.54**	0.06	0.08	0.38	0.45	0.42	**0.72**
cURL	0.07		0.35	0.56	0.66	**0.79**	0.05		0.63	0.65	0.76	**0.91**
Diffutils	0.28	0.16	0.27	0.44	0.39	**0.64**	0.18	0.31	0.51	0.60	0.54	**0.88**
Findutils	0.12	0.14	0.30	0.38	0.34	**0.65**	0.07	0.26	0.44	0.48	0.43	**0.82**
Gzip	0.25		0.32	0.32	**0.38**	0.25	0.17		**0.61**	0.43	0.51	0.34
PuTTY	0.07		0.30	0.08	0.18	**0.62**	0.05		0.29	0.11	0.24	**0.84**
						ALL						
Coreutils	0.06	0.14	0.16	0.08	0.29	**0.58**	0.04	0.03	0.24	0.11	0.36	**0.74**
cURL	0.00		0.16	0.15	0.48	**0.78**	0.00		0.26	0.16	0.52	**0.84**
Diffutils	0.29	0.11	0.15	0.11	0.35	**0.65**	0.20	0.24	0.26	0.14	0.45	**0.81**
Findutils	0.08	0.11	0.16	0.08	0.31	**0.61**	0.04	0.23	0.22	0.10	0.37	**0.74**
Gzip	0.10		0.17	0.10	**0.34**	0.28	0.07		0.28	0.12	**0.41**	0.35
PuTTY	0.04		0.11	0.03	0.14	**0.65**	0.02		0.10	0.04	0.17	**0.77**

VexIR2Vec obtains the highest average F1 score of 0.65, compared to BinFinder's 0.41. SAFE outperforms OPC, with an average F1 score of 0.29 versus 0.22. In general, it can be seen that the baselines do not perform well under obfuscation.

Takeaway 10.1: Robust Binary Diffing Across Configurations

VexIR2Vec *achieves the highest F1 scores across all adversarial diffing scenarios—cross-optimization, cross-compiler, cross-architecture, and obfuscated binaries—outperforming the nearest baselines by 22–58% and demonstrating that architecture-neutral lifting combined with semantic embeddings enables robust binary diffing.*

10.5.2 Searching and Retrieval

In this section, we evaluate our approach on the searching task described in Section 10.2.

Ground Truth

We create a pool of functions from the test set of x86 binaries generated with Clang (V8.0.1 and V12.0.1) and GCC (V8.0 and V10.0), at four optimization levels: `-O0`, `-O1`, `-O2` and `-O3`. In each search, functions from one compilation configuration are used as the query set. All the other functions, excluding the query set, form the set of search candidates. We obtain the function identifiers by following the same process described in Section 10.5.1 to form the ground truth. In any search set, there are 15 candidates that can match a query (2 compilers $\times$ 2 versions $\times$ 4 optimizations $-$ input query $=$ 15). On average, a query set has about $2K$ functions, and a search set has about $323K$ functions; thus, the chance of correctly retrieving a match for a query with a random guess is $\approx 15/323K$.

Metric

We use the Mean Average Precision (MAP) as the evaluation metric for the searching task. MAP is computed as $\frac{1}{Q}\sum_{q=1}^{Q} AP(q)$, where $AP(q)$ is the average precision for query q. Q is the total number of queries. $AP(q)$ is defined as $\frac{1}{N}\sum_{i=1}^{N} Prec(i) \times Rel(i)$. $Prec(i)$ is precision at i; $Rel(i)$ denotes the relevance at i. $Rel(i)$ is set to 1 when the retrieved function at position i matches the query function, and is set to 0 otherwise. N is the total number of retrieved functions that match the query function.

Mixture of all—searching

Each query function is searched against the candidate set of functions. Similar to Section 10.5.1, we obtain a list of 10 nearest functions to the query function among the set of candidates ranked by the Euclidean distance. A function from the list is considered to match the query function if both of them have the same source file and function name. We do not consider BinDiff and DeepBinDiff for the searching experiments as they are designed only for binary diffing. Figure 10.5 shows the MAP scores produced by SAFE, BinFinder, OPC, and VexIR2Vec for this task. VexIR2Vec consistently obtains higher MAP scores than the other baselines across all compiler configurations. The MAP score of VexIR2Vec is 0.76 on average, while SAFE's, BinFinder's, and

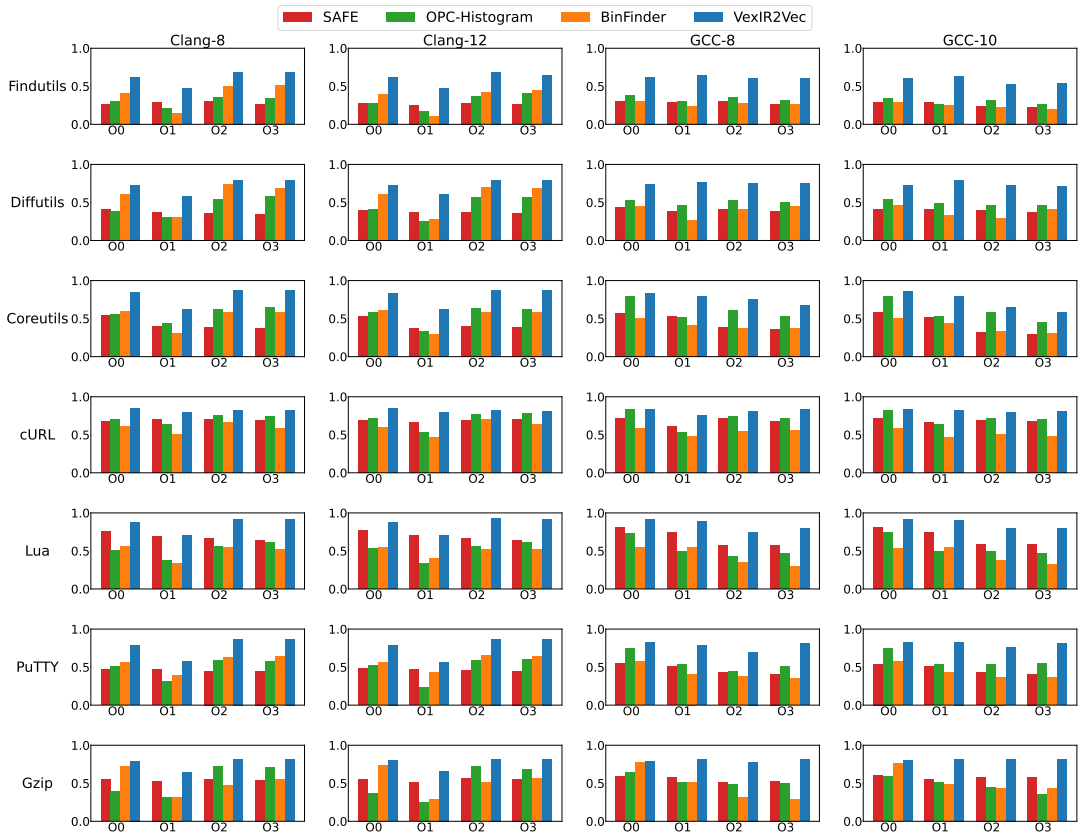


Figure 10.5: Mean Average Precision (MAP) scores for searching experiments. OPC's are 0.5, 0.47, and 0.52, respectively.

> **Takeaway 10.2: Superior Function Searching**
>
> VexIR2Vec *achieves 40–52% higher MAP scores than prior methods by lifting binaries to an architecture-neutral IR and applying the same semantic encoding principles developed for source code.*

10.5.3 Scalability

In this section, we evaluate the scalability of VexIR2Vec.

Time for Embedding Generation

The implementation of VexIR2Vec features two levels of parallelism:

Thread level: Each function of the binary is processed in parallel by different threads to obtain the function-level embedding.

Task level: Each binary is processed in parallel via a new process.

Figure 10.6 provides data that demonstrates that this parallelization yields benefits. The figure reports running times to generate embeddings for a randomly chosen set of 100 binaries from our dataset, whose sizes range between $15KB$ and $5MB$. Lines show the cumulative time observed for the different binaries. These binaries are sorted by size along the X-axis. The implementation of VexIR2Vec with one, two, four and eight threads takes 490, 335, 264, 221 seconds, respectively, to process the 100 binary files. Thus, the eight-threaded implementation of VexIR2Vec is 2.2× faster than its sequential version in this experiment.

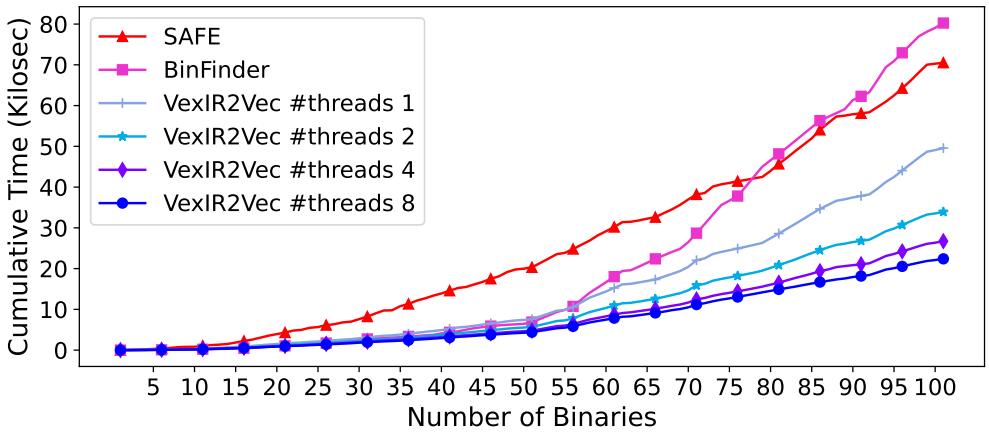


Figure 10.6: Time Taken to generate embeddings.

Figure 10.6 also reports running times to generate embeddings via SAFE and BinFinder. We omit BinDiff and OPC as they do not generate embeddings. We also omit DeepBinDiff because, in this experiment, this tool takes about $6,800$ seconds on average, timing out for binaries greater than $300KB$. SAFE and Binfinder take 700 and 795 seconds, respectively, to process the 100 binaries; hence, these tools tend to be much slower than VexIR2Vec. VexIR2Vec running with one thread achieves a speedup of 1.62× and 1.42× over BinFinder and SAFE, respectively. With eight threads, this speedup grows to 3.5× and to 3.11×, respectively. BinFinder and SAFE also seem to be impacted by worst asymptotic behavior: the time these tools need to produce embeddings grows orders of magnitude faster with the binary size compared to VexIR2Vec.

Time taken by VexINE

Figure 10.7 shows the time that VexINE takes to normalize the same 100 binaries used in Section 10.5.3. All the normalizations described in Section 4.3.3 can be implemented to run in linear time on the number of instructions that constitute a peephole. Linearity comes from the fact that peepholes are straight-line sequences of code; hence, each optimization visits each instruction only once. Figure 10.7 indicates that the time required to normalize peepholes grows more slowly than the overall time taken by VexIR2Vec to produce embeddings. The cumulative time taken by VexINE is about 25% of the time that VexIR2Vec takes to produce embeddings.

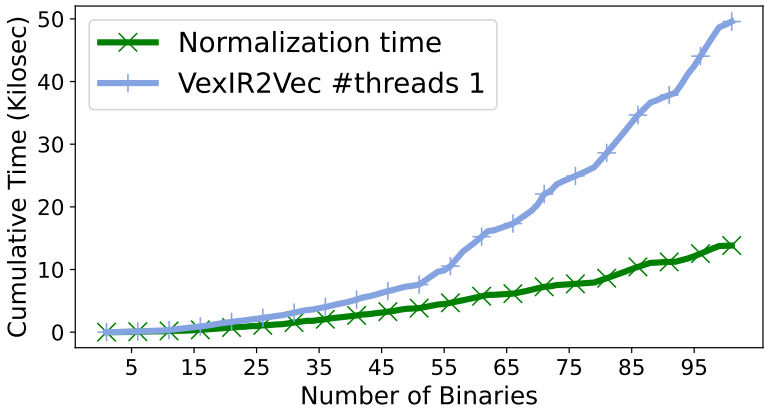


Figure 10.7: Time Taken to normalize embeddings.

> **Takeaway 10.3: Scalable Embeddings**
>
> VexIR2Vec *generates embeddings 3.1–3.5× faster than the nearest baselines with 8-thread parallelism and scales gracefully to larger binaries, demonstrating practical applicability to real-world binary analysis.*

10.6 Ablation Study

VexIR2Vec is customizable. For instance, its scalability and precision can be controlled by parameters of Algorithm 3, such as the length k of peepholes and the number c of times each basic block is visited to produce a peephole. Similarly, precision can be enhanced with the addition of phases such as the normalization proposed in Section 4.3.3,

which is entirely optional. In other words, VexIR2Vec can still be extracted from non-normalized peepholes. To evaluate the impact of different customizations on the precision and on efficiency of VexIR2Vec, this section provides answers to the following research questions:

- How the parameters of Algorithm 3 (k and c) impact the precision and the scalability of VexIR2Vec? (Section 10.6.1)
- What is the impact of the different types of normalization implemented by VexINE on the precision of VexIR2Vec? (Section 10.6.2)
- Can the proposed normalizations be adapted to work with other baseline classifiers, and in case such is possible, how would they change the precision of these tools? (Section 10.6.3)
- What is the contribution of strings and library calls to the precision of VexIR2Vec? (Section 10.6.4)
- What is the relative importance of the different program entities in the precision of VexIR2Vec? Or, in other words, how much attention does VexNet put on each kind of entity? (Section 10.6.5)

10.6.1 Impact of the Parameters of Random Walk

The random walk algorithm to generate peepholes described in Algorithm 3 is parameterized by the maximum length of the peephole k and the minimum number of visits per basic block c. The experiments discussed in Section 10.5 use $k = 72$ and $c = 2$. We have arrived at this configuration after testing different values. This section describes this search, using, to this end, the setting involving cross-architecture and mixture-of-all diffing experiments.

Figure 10.8 shows the variation in F1 score for different values of k. The F1 score increases with k from 1 to 72. This growth indicates the benefit of additional contextual information that comes with larger peepholes. However, past $k = 72$, we start observing a decrease in F1 scores. Indeed, $k = 144$ and $k = 32$ deliver almost identical results in the cross-architecture/mixture-of-all setting. This behavior suggests that context improves the precision of VexIR2Vec; however, only up to a certain limit.

Similar to the experiment on k, we vary the values of c, the number of times each basic block is visited by Algorithm 3, and record F1 scores. Figure 10.9(b) shows the results

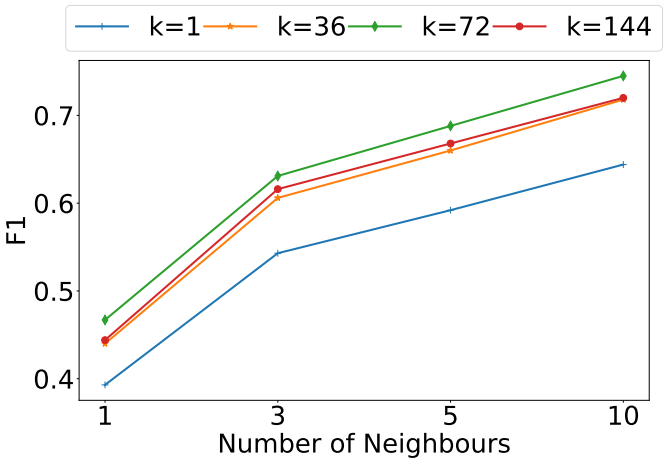


Figure 10.8: The impact of the parameter k in the random walk (Algorithm 3) on the F1 scores of VexIR2Vec.

of this experiment. F1 scores increase from $c = 1$ to $c = 2$. However, upon increasing c further to 3 and 5, we observe a reduction in F1 scores. The results obtained for $c = 3$ closely match those seen with $c = 1$. This behavior indicates a similar trend as that of k: it is better to visit a basic block multiple times than just once. However, there seems to exist a limit to how much information can be derived from the extra visits. In our case, this limit is two.

> **Takeaway 10.4: Optimal Context Window**
>
> *Peephole parameters exhibit diminishing returns: context improves precision up to a threshold (k=72, c=2), beyond which additional information introduces noise rather than signal.*

10.6.2 Effectiveness of Normalization

To analyze the impact of VexINE, we have created four sets of normalizations, each containing an increasing collection of the transformations introduced in Section 4.3.3:

- $N0$: No normalization.
- $N1$: Register optimizations, Redundant Write Elimination, and Copy propagation.
- $N2$: $N1$ along with Constant Propagation, Constant Folding, and Common Subexpression Elimination.

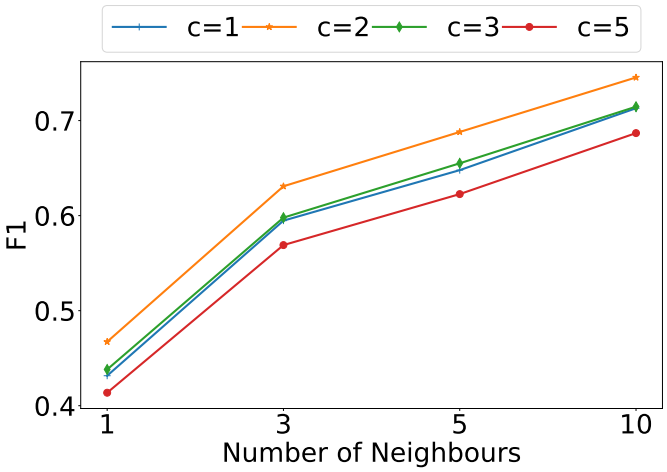


Figure 10.9: The impact of the parameter c in the random walk (Algorithm 3) on the F1 scores of VexIR2Vec.

$N3$: All the normalizing transformations described in Section 4.3.3.

Figure 10.10 shows the F1 score measured when VexIR2Vec is equipped with each of these sets of normalizing transformations. In these experiments, a new model is trained from scratch with the obtained embeddings generated by the normalized peepholes. We observe an increasing trend in performance with the increase in normalization level. Consequently, $N3$ yields the highest F1 score. $N0$, in turn, yields the lowest. Nevertheless, even in this poor configuration—without any support of normalization—VexIR2Vec is still able to produce higher F1 scores than the baseline approaches. This empirical observation supports the trend observed in Examples 4.3.4 and 4.3.5. These examples provide intuition on how successive normalizing transformations simplify peepholes, hence filtering out non-essential syntactic elements and revealing their essential semantic characteristics.

> **Takeaway 10.5: Progressive Normalization Benefits**
>
> *Normalization progressively improves precision, with each transformation level yielding higher F1 scores—yet the base embedding remains effective even without normalization.*

10.6.3 Impact of Normalization on Other Baselines

Although we have designed the normalizing transformations of Section 4.3.3 to work in tandem with VexIR2Vec, they can still be of service in other binary similarity tools.

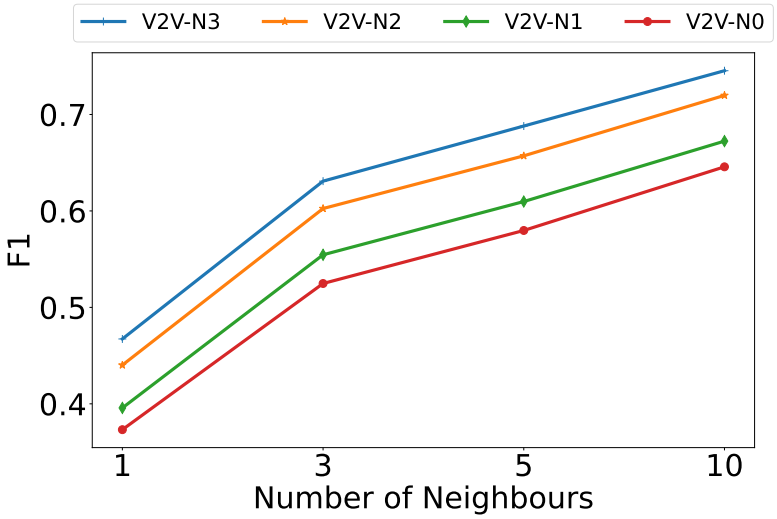


Figure 10.10: The impact of the normalizing transformations (Section 4.3.3) on the F1 scores of VexIR2Vec.

To understand this aspect, we consider two baselines: BinFinder and Histograms of Opcodes. These are the only baselines that we see how to augment with normalizations.

We have modified the implementation of these baselines, so that they would extract embeddings from normalized peepholes. Peepholes are extracted from the very Algorithm 3 that empowers VexIR2Vec. Decomposing functions into peepholes is fundamental to carry out this experiment, for the normalizing transformations were designed to work on straight-line sequences of instructions. Embeddings are extracted per peephole and then merged via vector addition, as done for VexIR2Vec.

Figure 10.11 shows the resulting F1 scores for these two modified baselines, using two versions of each one of them. The first version, with the suffix $N0$, is the original implementation without any normalization. The other version, with the suffix $N3$, is the modified version, which incorporates all the transformations described in Section 4.3.3. Figure 10.11 shows that normalization improves F1 scores in both the baselines. Notice, however, that even when equipped with the highest normalization level ($N3$, as explained in Section 10.6.2), the baselines still lag behind VexIR2Vec running on non-normalized peepholes. Nevertheless, this experiment implies that compiler-inspired normalizations, as a general pre-processing transformation, are general enough to be deployed onto different embedding functions.

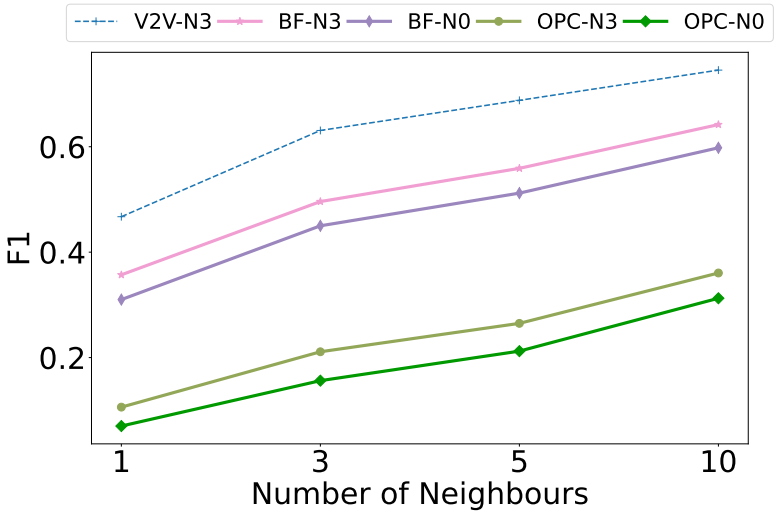


Figure 10.11: The impact of the normalizing transformations on the F1 scores of BinFinder & OPC.

> **Takeaway 10.6: Generalizable Normalization**
>
> *Compiler-inspired normalizations improve F1 scores across different embedding approaches, demonstrating that such normalizations are generalizable across different binary similarity methods.*

10.6.4 Contribution of Strings and Library Calls

As Section 10.3.1 explains, the final embedding $[\![\mathbf{F}]\!]$ that characterizes the VexIR2Vec representation of a function includes information extracted from strings and library calls. This information is encoded in the $[\![\mathbf{S}]\!]$ and $[\![\mathbf{L}]\!]$ vectors that Section 10.3.2 introduces.

We train the VexNet model only by considering $\langle [\![\mathbf{O}]\!], [\![\mathbf{T}]\!], [\![\mathbf{A}]\!] \rangle$; that is, without adding in the $[\![\mathbf{S}]\!]$ and $[\![\mathbf{L}]\!]$ vectors. Figure 10.12 shows the resulting F1 scores. On average, strings and library calls improve the F1 scores across different neighbors by about 0.1 (20%). Thus, strings and library calls provide VexIR2Vec with non-trivial information. They enhance the quality of the function embeddings and, consequently, the performance of VexNet. Nevertheless, even without strings and library calls, VexIR2Vec remains more precise than the competing baselines. For instance, while BinFinder and SAFE achieve average F1 scores of 0.58 and 0.32, respectively, VexIR2Vec, even without the extra data provided by strings and library calls, achieves an F1 score of 0.63.

Data extracted from strings and library calls is important because it tends to remain

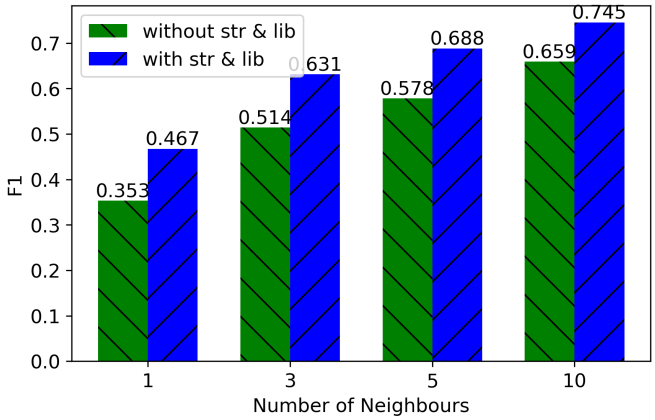


Figure 10.12: Impact of Strings and Library Calls on the F1 scores of VexIR2Vec.

unchanged, even in very aggressive adversarial settings. Thus, unless the compiler determines that the relevant portion of the code is *dead*, strings and library calls will persist in the different versions of the binary code. However, strings and library calls alone cannot be used as code embedding for two reasons. First, because many functions simply do not contain any data of this kind. Second, different functions might still contain exactly the same data, such as calls to I/O operations, for instance. Therefore, while strings and library calls cannot be the primary representation of a function, their inclusion alongside other features considerably improves the ability of VexIR2Vec to match binary code.

> **Takeaway 10.7: Complementary Auxiliary Features**
>
> *Strings and library calls provide stable, compilation-invariant signals that complement instruction-level embeddings, improving precision while remaining optional for core functionality.*

10.6.5 Attention Weights

As Section 10.3.2 explains, the final embedding that characterizes a function includes information taken from five vectors: operands ($[\![\mathbf{O}]\!]$), types ($[\![\mathbf{T}]\!]$), arguments ($[\![\mathbf{A}]\!]$), strings ($[\![\mathbf{S}]\!]$) and library calls ($[\![\mathbf{L}]\!]$). Section 10.6.4 provides some evidence that strings and library calls, although not essential, are important to ensure the high precision of VexIR2Vec. In this section, we analyze how the model perceives the other entities by examining the learned attention values.

Figure 10.13 examines the learned attention values of the different vectors that con-

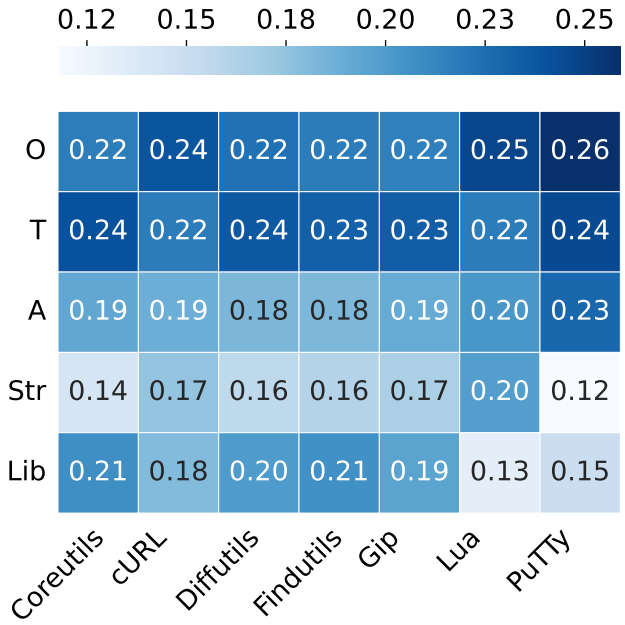


Figure 10.13: Heatmap of Attention Scores capturing the contribution of different entities (see Equation 4.1) to the F1 scores of VexIR2Vec.

stitute VexIR2Vec. The figure includes data taken from the entire test set described in Section 10.4. The heatmap reveals that VexIR2Vec assigns varying levels of importance to different features across various datasets (Coreutils, cURL, Diffutils, Findutils, Gzip, Lua, and PuTTY). Generally, Opcodes (O), Types (T), and Arguments (A) receive higher attention scores, highlighting their critical role in function representation. However, Strings (Str) and External Library calls (Lib) might, sometimes, receive higher attention than Arguments (A). This observation is especially true when they provide unique information. For instance, if a particular library call is only invoked in a certain function f, this feature will be essential to distinguish f from other routines. Notice that the model does not give any attention weights to strings and library calls if they are not present in the function.

We observe that unique or less commonly used strings tend to receive higher attention scores. For example, the string "all" which appears 64 times in our entire dataset, receives an attention score of 0.3. In contrast, commonly used strings like "%s", which appears $14K$ times, get a much lower attention score (0.07). Similarly, specialized or domain-specific library functions, such as `pow` and `sin`, receive more attention weights than commonly used functions like `malloc` and `free`. Hence, on average, projects like Coreutils and PuTTY, which have many common strings, receive a lower attention

weight for strings.

Figure 10.13 shows that VexIR2Vec learns to assign varying degrees of importance to different entities, aligning with their contextual contributions for binary similarity. This adaptive approach stands in contrast with works like IR2Vec [VenkataKeerthy et al., 2020], which assigns fixed weights to Opcodes, Types, and Arguments heuristically, with a predetermined importance order of $O > T > A$.

> **Takeaway 10.8: Learned Feature Importance**
>
> *The attention mechanism adaptively weighs program entities based on contextual importance, assigning higher weights to rare strings and specialized library calls while down-weighting common patterns.*

10.7 Related Work

Given the practical importance of binary similarity, extensive research has been dedicated to studying this problem[6]. This section covers different facets of this literature. Notice that much of this literature is defined by the fact that determining program equivalence is an undecidable problem [Rice, 1953]; hence, solutions to binary similarity are based on heuristics. Early implementations of such heuristics worked on source code [Hunt and Szymanski, 1977]. As a summary of this section, Figure 10.14 positions our work within this literature.

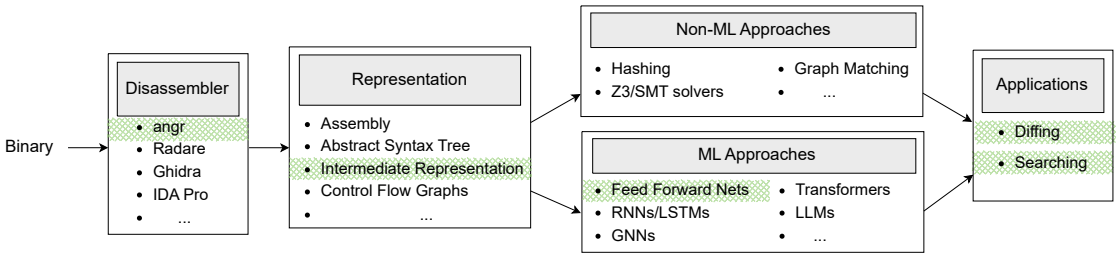


Figure 10.14: The different elements that constitute the Binary Similarity problem. The options used by VexIR2Vec under each facet are highlighted.

Binary code became a more intense focus of research in the nineties. Those initial efforts were mostly centered on sequence alignment algorithms [Coppieters, 1995; Baker

[6] As testimony to this variety, the "*Awesome Binary Similarity*" webpage catalogs about 214 publications as on June 2024 - https://github.com/SystemSecurityStorm/Awesome-Binary-Similarity

and Manber, 1998] and hash functions [Wang et al., 2000]. Canonicalization techniques, like register renaming [Debray et al., 2000], were proposed around that time. These techniques are still in use today [Collyer et al., 2023; He et al., 2024; Massarelli et al., 2019; Wang et al., 2022a; Li et al., 2021]. However, nowadays, research on binary similarity is directed towards machine-learning approaches [Ding et al., 2019; Massarelli et al., 2019; Duan et al., 2020; Li et al., 2021; Wang et al., 2023a; Yu et al., 2020].

In Table 10.8, we provide a detailed comparison of our approach with other related approaches on various design choices and applications, some of which we explain below. To provide the reader with some perspective on how different solutions to adversarial binary similarity compare, the column "Works Compared" in Table 10.9 shows which tools have been used as baselines during the evaluation of different approaches.

Disassemblers and Input Representations Binary similarity analysis begins by disassembling the input binary using disassemblers and binary analysis tools [Wang and Shoshitaishvili, 2017; Hex-Rays, n.d.; Ghidra; Team, 2017; Brumley et al., 2011]. These tools translate the binary code into human-readable formats like assembly code, Abstract Syntax Tree (AST), or Intermediate Representations (IR). Different approaches rely on different representations; however, a vast majority of binary similarity tools use assembly code [Qasem et al., 2023; Wang et al., 2023a; Zhu et al., 2023; Wang et al., 2022a; Ahn et al., 2022; Li et al., 2021; Yang et al., 2022; Duan et al., 2020; Pei et al., 2020; Yu et al., 2020; Massarelli et al., 2019; Ding et al., 2019; Zuo et al., 2019; Xu et al., 2017]. Works like Asteria [Yang et al., 2021] and Asteria-Pro [Yang et al., 2023] use AST, whereas David et al. [2017] and Peng et al. [2021] work on the LLVM IR. We use the VEX-IR representation generated by `angr` to model binary similarity. We chose VEX-IR because it is architecture-neutral and open-source. VexIR2Vec is heavily engineered to work on VEX-IR; however, its ideas could be employed on other representations. Section 10.6.3 supports this statement, demonstrating that the peepholes and normalizing transformations of Section 4.3.3 could be used to enhance different binary similarity tools.

Compilation Configurations Adversarial binary similarity explores the challenges of identifying differences in binaries resulting from various compilation configurations, including cross-compiler, cross-optimization, and cross-architecture settings. These variations pose significant challenges to the binary classifier. Consequently, they have received considerable research attention. However, not all existing implementations of

Table 10.8: Comparison of VexIR2Vec with earlier works. XO, XC, and XA indicate cross-optimization, cross-compiler, and cross-architecture support. D and S indicate support for Diffing and Searching tasks.

	Name	Disassembler	Input Representation	Comp Conf.			Approach		Tasks	
				XO	XC	XA	Models used	GPUs	D	S
⇒	VexIR2Vec	angr	VEX IR	Y	Y	Y	FCNN	1 RTX 3080	Y	Y
2023	BinFinder [Qasem et al., 2023]	angr	Assembly	Y	Y	Y	FCNN	8 TITAN	Y	Y
	Sem2Vec [Wang et al., 2023a]	angr	Assembly	Y	Y		RoBERTa, LSTM, GNN	1 Tesla V100 32G	Y	Y
	kTrans [Zhu et al., 2023]	IDA-Pro	Assembly	Y			BERT	4 V100		Y
	VulHawk [Luo et al., 2023]	IDA-Pro	MicroCode	Y	Y	Y	RoBERTa, GCN, ResNet	1 RTX 3090	Y	Y
	Asteria-pro [Yang et al., 2023]	IDA-Pro	AST		Y	Y	Tree LSTM			Y
2022	jTrans [Wang et al., 2022a]	IDA-Pro	Assembly	Y			BERT	8 A100	Y	
	BinShot [Ahn et al., 2022]	IDA-Pro	Assembly	Y	Y		BERT	2 RTX A6000	Y	
2021	Oscar [Peng et al., 2021]	RetDec	LLVM IR	Y			RoBERTa	8 Tesla V100	Y	
	PalmTree [Li et al., 2021]	BinaryNinja	Assembly	Y	Y		BERT	1 GTX 2080Ti		Y
	Asteria [Yang et al., 2021]	IDA-Pro	AST			Y	Tree LSTM		Y	
	Codee [Yang et al., 2022]	IDA-Pro, angr	Assembly	Y	Y	Y	Word2Vec	2 GTX RTX5000		Y
2020	DeepBinDiff [Duan et al., 2020]	angr	Assembly	Y			DeepWalk	Does not use	Y	
	Trex [Pei et al., 2020]		Assembly	Y	Y	Y	Hierarchical Transformers	8 RTX 2080-Ti		Y
	Order Matters [Yu et al., 2020]		Assembly			Y	BERT, ResNet, GRU			Y
2019	SAFE [Massarelli et al., 2019]	Radare2	Assembly	Y	Y	Y	RNNs	K80		Y
	Asm2vec [Ding et al., 2019]	IDA-Pro	Assembly	Y	Y		Word2Vec			Y
	Innereye [Zuo et al., 2019]	BAP	Assembly	Y		Y	FCNN, LSTM	Does not use		Y
2017	Gemini [Xu et al., 2017]	IDA-Pro	Assembly				structure2vec	1 GTX 1080	Y	
	David et al. [2017]	angr	LLVM IR	Y	Y	Y	Non-ML			Y
	BinDiff [Zynamics, 2024]	Ghidra	–	Y	Y	Y	Non-ML		Y	

binary similarity tools natively support all three scenarios.

Prior works by Pewny et al. [2015] and Redmond et al. [2018] initiated research on cross-architecture similarity. Their approaches, however, often suffer from limitations such as slow execution speeds [Eschweiler et al., 2016; Feng et al., 2016]. Existing works that rely on assembly codes for modeling binary similarity [Duan et al., 2020;

Wang et al., 2023a; Li et al., 2021; Ahn et al., 2022] are architecture-specific and cannot perform cross-architecture binary similarity analysis. This limitation is often overcome by the usage of feature-based representations [Qasem et al., 2023; Kim et al., 2023; Xu et al., 2017; Eschweiler et al., 2016; Chandramohan et al., 2016], with features extracted using binary analysis tools. Alternatively, unified representation spaces [Pei et al., 2020; Kim et al., 2022a; Zhang et al., 2020] are created by training models on assembly codes from multiple architectures; however, they do not generalize to unseen architectures.

VexIR2Vec overcomes these limitations by leveraging VEX-IR, the Intermediate Representation (IR) used by analysis tools like `angr` and Valgrind. Vexir offers a more architecture-neutral view of the program compared to assembly code. Consequently, the implementation of VexIR2Vec is effective in different adversarial settings, including cross-compiler, cross-optimization, and cross-architecture scenarios.

Approach Early methods for comparing binaries relied on statistical analysis and static code inspection. Tools like BinDiff [Zynamics, 2024] utilize graph isomorphism algorithms to identify code similarities. Other approaches employ specialized hashing techniques [Dullien, 2018; David et al., 2017] or solvers like Z3 [Pewny et al., 2015] and SMT [David et al., 2016] to perform more intricate comparisons.

More recent binary similarity techniques have seen the introduction of Machine Learning (ML) for generating n-dimensional vector representations of binaries. Asm2Vec [Ding et al., 2019] and SAFE [Massarelli et al., 2019] adapt the word2vec model [Mikolov et al., 2013a] for this purpose. However, the majority of current research leverages transformers [Vaswani et al., 2017] such as BERT [Devlin et al., 2019] and RoBERTa [Liu et al., 2019]. These methods often model Control Flow Graphs (CFGs) [Luo et al., 2023; Yu et al., 2020; Feng et al., 2016] using Graph Neural Networks (GNNs) [Wu et al., 2022]. Works like sem2vec [Wang et al., 2023a] even combine symbolic execution powered by Z3, GNNs, and transformers for a more comprehensive representation learning process.

ML-based approaches can represent binaries either as feature vectors or by learning distributed vector representations. Tools like BinFinder [Qasem et al., 2023] rely on predefined features like caller/callee information, status flags, and library calls. In contrast, Asm2Vec[Ding et al., 2019] and Codee [Yang et al., 2022] learn the input assembly representation by employing variations of the word2vec model [Mikolov et al., 2013a]. VexIR2Vec, in turn, leverages knowledge graph embeddings [Bordes et al., 2013] to model binary functions as distributed vectors.

Out-of-vocabulary Issue One of the main challenges in existing works is the out-of-vocabulary (OOV) problem. Approaches that learn instruction-level or basic block-level embeddings are highly prone to this issue [Massarelli et al., 2019; Zuo et al., 2019; Redmond et al., 2018]. In our experiments, we observed that SAFE encounters a very high number of OOV tokens, resulting in significant performance degradation. Works that use token-level embeddings generally face fewer OOV problems [Ding et al., 2019; Duan et al., 2020]. However, solely encoding tokens can overlook the semantics of the instruction.

Canonicalization and normalization techniques can help mitigate OOV issues to some extent. Nevertheless, OOV problems can still arise if numeric and string constants are not properly handled. Works that consider such constants as tokens within their vocabulary often suffer from OOV issues [Wang et al., 2023a; Li et al., 2021]. Improper canonicalization can lead to a very large vocabulary, increasing the likelihood of OOV problems [Zhu et al., 2023]. Additionally, works that model instructions and basic blocks as tokens and use LSTMs and transformers can suffer from OOV issues due to their fixed-length input requirements.

VexIR2Vec mitigates the OOV issue by learning entity-level representations of instructions with carefully designed canonicalization, resulting in a smaller vocabulary. During fine-tuning, instruction-level semantics are effectively captured, addressing the shortcomings of previous methods.

Resources Several existing binary similarity approaches face limitations when dealing with large binaries containing thousands of basic blocks [Haq and Caballero, 2021]. These limitations stem from the time and resources required for training or deployment. Techniques that leverage modern language models [Ahn et al., 2022; Wang et al., 2023a; Zhu et al., 2023; Luo et al., 2023; Li et al., 2021; Wang et al., 2022a; Peng et al., 2021; Yu et al., 2020; Pei et al., 2020] often require training millions of parameters. This requirement leads to lengthy training times and necessitates significant computational resources. For example, many of these approaches rely on large clusters equipped with multiple high-end GPUs, such as A100 and V100 [Qasem et al., 2023; Zhu et al., 2023; Wang et al., 2022a; Ahn et al., 2022; Peng et al., 2021; Pei et al., 2020]. Still, training can take weeks. These demands translate to substantial resource and time constraints during deployment, hindering practical usability.

In contrast, VexIR2Vec relies on lightweight feed-forward neural networks to model binary similarity. This low-computational requirement enables our model to train effec-

tively on a workstation equipped with a single RTX3080 GPU with 10 GB of memory. As seen in Section 10.5.3, each training epoch takes 5–8 seconds, hence being orders of magnitude faster than other tools.

Application Existing binary similarity approaches can be broadly categorized into two groups: those that learn a similarity score and those that learn a distance measure. The first category focuses on predicting whether a pair of binary snippets are similar or dissimilar [Duan et al., 2020; Zuo et al., 2019; Shalev and Partush, 2018]. The typical application of these methods is binary diffing, not searching. They become impractical for searching due to their reliance on pairwise comparisons, resulting in a quadratic time complexity.

In contrast, approaches that learn a distance measure [Massarelli et al., 2019; Qasem et al., 2023] are applicable to both searching and diffing tasks. VexIR2Vec falls into this category, leveraging the Siamese networks of Section 10.3.1 to learn a distance measure between programs. This design makes VexIR2Vec suitable for both diffing and searching.

The Quest for Reproducible Artifacts Tables 10.8 and 10.9 underscore the diversity of tools available for addressing binary similarity problems. As a consequence of this vast literature, much of the effort involved in our evaluation was spent trying to configure and run these tools. Some of these attempts met with success, as Section 10.5 shows, while others[7] did not. Artifacts such as VulHawk[8], CEBin[9], or Asteria-Pro[10] are not fully available. Other artifacts, such as BinFinder, are mostly available; however, to use it, we had to reimplement parts of the public implementation. The difficulty in

[7]We tried setting up Asteria [Yang et al., 2021], Codee [Yang et al., 2022], BinShot [Ahn et al., 2022], Sem2Vec [Wang et al., 2023a] but encountered several challenges in achieving a functional setup. Despite raising issues on the respective GitHub repositories and contacting authors via email, our queries either received no response or only partial assistance.

[8]The VulHawk repository (available at https://github.com/RazorMegrez/VulHawk) contains six unanswered issues, the oldest dating from June 7th, 2023. These issues are mostly requesting missing pieces of the artifact, such as code to generate datasets or to train the models. On November 12th, 2023, we wrote to the authors, requesting parts of the artifact to compare it with our techniques. To this date of this submission, we have not received an answer.

[9]The repository, available at https://github.com/Hustcw/CEBin contains an issue open on June 2024, where a user requests the model. The authors answer that they will "*release the model in the near future*".

[10]The Asteria-Pro repository (available at https://github.com/Asteria-BCSD/Asteria-Pro/issues/1) contains one unanswered open issue dating from February 8th, 2024, requesting the dataset.

obtaining reproducible artifacts in this domain has been noted by other authors [Haq and Caballero, 2021; Marcelli et al., 2022]. In majority of the cases [Yang et al., 2021; Ding et al., 2019; Xu et al., 2017; Luo et al., 2023], reproducibility is limited by the use of licensed/proprietary disassemblers such as IDA-Pro [Hex-Rays, n.d.] and Binary Ninja [Vector 35 Inc., 2024].

10.8 Summary

In this chapter, we discussed the problem of binary similarity and how Path-Aware embeddings of VexIR2Vec can effectively be used to solve it. VexIR2Vec is application independent, making it suitable to solve different binary similarity tasks such as diffing and searching. We designed VexNet, a simple feed-forward Siamese network, to learn the similarity metric between functions. Given the Path-Aware peephole-level embeddings, VexNet acts as a fine-tuning mechanism to capture similarity.

We demonstrated that VexIR2Vec is more scalable and highly accurate than previous works under various adversarial settings including cross-optimization, cross-compiler, and cross-architecture, and in the presence of deliberate obfuscating transformations. We also showed that the normalizing transformations applied by VexINE widely improve the performance of VexIR2Vec. We also experimentally showed that these transformations when applied to other representations like BinFinder and histograms of opcodes can improve their performance as well.

The source code, web application, and other relevant material are publicly available at https://compilers.cse.iith.ac.in/VexIR2Vec.

Table 10.9: Works Compared in VexIR2Vec and earlier works.

Name	Works Compared
VexIR2Vec	[Qasem et al., 2023; Damásio et al., 2023; Massarelli et al., 2019] [Duan et al., 2020; Zynamics, 2024]
BinFinder [Qasem et al., 2023]	[Massarelli et al., 2019; Shalev and Partush, 2018; Ding et al., 2019] [Li et al., 2019]
Sem2Vec [Wang et al., 2023a]	[Zynamics, 2024; Ding et al., 2019; Massarelli et al., 2019] [Yu et al., 2020; Li et al., 2021]
kTrans [Zhu et al., 2023]	[Xu et al., 2017; Wang et al., 2022a; Li et al., 2021]
VulHawk [Luo et al., 2023]	[Ding et al., 2019; Yang et al., 2021; Li et al., 2021] [Li et al., 2019; Massarelli et al., 2019; Pei et al., 2020]
Asteria-pro [Yang et al., 2023]	[Xu et al., 2017; Massarelli et al., 2019; Pei et al., 2020]
jTrans [Wang et al., 2022a]	[Feng et al., 2016; Xu et al., 2017; Massarelli et al., 2019] [Ding et al., 2019; Yu et al., 2020]
BinShot [Ahn et al., 2022]	[Xu et al., 2017; Ding et al., 2019; Li et al., 2021]
Oscar [Peng et al., 2021]	[Zynamics, 2024; Ding et al., 2019; Yu et al., 2020]
PalmTree [Li et al., 2021]	[Xu et al., 2017; Guo et al., 2019; Chua et al., 2017]
Asteria [Yang et al., 2021]	[Xu et al., 2017; Feng et al., 2016; Eschweiler et al., 2016]
Codee [Yang et al., 2022]	[Ding et al., 2019; Xu et al., 2017; Massarelli et al., 2019] [Duan et al., 2020; Yu et al., 2020]
DeepBinDiff [Duan et al., 2020]	[Ding et al., 2019; Zynamics, 2024]
Trex [Pei et al., 2020]	[Ding et al., 2019; Massarelli et al., 2019; Xu et al., 2017]
Order Matters [Yu et al., 2020]	[Xu et al., 2017]
SAFE [Massarelli et al., 2019]	[Xu et al., 2017]
Asm2vec [Ding et al., 2019]	[Chandramohan et al., 2016; Dullien, 2018]
Innereye [Zuo et al., 2019]	–
Gemini [Xu et al., 2017]	[Feng et al., 2016]
David et al. [2017]	–
BinDiff [Zynamics, 2024]	–

Chapter 11

Using IR2Vec for Code Similarity

11.1 Introduction

Program similarity refers to the problem of determining whether two programs or code fragments exhibit the same semantics. Similar to the binary similarity problem discussed in Chapter 10, program similarity is fundamentally undecidable [Rice, 1953]. Despite this theoretical limitation, numerous practical techniques have been developed to approximate program similarity, enabling applications across various domains:

- Security—Detecting plagiarism, malicious code, identifying intellectual property (IP) and license violations.
- Optimization—Replacing legacy code with efficient library implementations, mapping software to hardware accelerators.
- Software Maintenance—Supporting tasks such as code recommendation, search, retrieval, and detecting code duplication.

Traditional approaches to program similarity analysis primarily relied on syntactic representations, such as lexical tokens [Qiu et al., 2021] and abstract syntax trees (ASTs) [Mou et al., 2016; Ye et al., 2021]. While effective in some cases, these representations often fail to generalize across different implementations of the same *function* due to variations in coding style, compiler transformations, and optimization levels.

To overcome these limitations, recent research has explored semantic embeddings of programs that preserve functional behavior rather than mere syntax. In this chapter, we investigate the effectiveness of IR2Vec embeddings for program classification.

We define program classification as the task of assigning a program to a predefined category based on its semantics. We experiment with the different embedding techniques discussed in Part I to perform classification. Specifically, we employ Symbolic, Flow-Aware, and Profile-Aware variants of IR2Vec to assess the expressivity of these representations in classifying semantically similar programs. We conduct our experiments on multiple real-world competitive programming datasets, including POJ-104 [Mou et al., 2016], CodeJam [jam, 2003], and CodeForces (CoFo) [Gautam et al., 2025]. We compare the performance of IR2Vec embeddings against traditional feature-based techniques and other state-of-the-art embedding methods such as Milepost [Fursin et al., 2011] and Histograms of Opcodes [Damásio et al., 2023]. We show that the higher the program analysis information encoded in the input representation, the better the classification performance. Through these studies, we aim to provide insights into the strengths and limitations of various embedding techniques for program similarity analysis.

We also discuss an application of program similarity in the context of device mapping to hardware accelerators. We demonstrate how program similarity analysis can be used to identify similar code fragments for mapping to hardware accelerators. Specifically, we show the effectiveness of IR2Vec embeddings in identifying Network Functions (NFs) that can be mapped to SmartNICs and accelerators in DPUs (Data Processing Units) for efficient packet processing. We evaluate the performance of IR2Vec embeddings in identifying NF like CRC and cryptography functions that can be offloaded to hardware accelerators. We show that IR2Vec significantly outperforms the state-of-the-art approach in this domain [Qiu et al., 2021].

> **Key Idea**
>
> Program classification can be effectively solved by leveraging IR2Vec embeddings that encode progressively richer program semantics. By comparing Symbolic, Flow-Aware, and Profile-Aware variants, we demonstrate that higher levels of program analysis information, from static syntax to dynamic execution profiles, lead to better classification accuracy, especially as program complexity increases.

Contributions The following are our key contributions in applying IR2Vec for source code similarity:

- We systematically evaluate Symbolic, Flow-Aware, and Profile-Aware variants of IR2Vec embeddings for program classification, demonstrating that richer program analysis information leads to better classification accuracy.

- We achieve up to 92.71% accuracy on POJ, 86.52% on CodeJam, and 91.37% on CodeForces, outperforming state-of-the-art baselines by 4–9%.
- We demonstrate the practical applicability of IR2Vec embeddings for device mapping, significantly outperforming the state-of-the-art system on CRC and cryptography detection tasks.

Organization The rest of this chapter is organized as follows. Section 11.2 discusses the program classification task, the datasets used, and the evaluation methodology. Section 11.3 presents the application of program similarity in device mapping to hardware accelerators. Section 11.5 presents the summary of the chapter.

11.2 Program Classification

Following the schema established in Section 9.2.1, we employ *classification-based* similarity for source code analysis. Unlike the distance-based approach used for binary similarity in Chapter 10, we frame the similarity problem as supervised classification, where each program is assigned to a specification class based on its semantic functionality.

Given a finite set of program specifications and multiple implementations corresponding to each specification, the problem of program classification involves determining the specification that a given program implements. This can be formulated as a standard supervised classification problem, where each specification corresponds to a class, and each implementation serves as a data point belonging to that class.

Formally, program classification can be defined as follows:

Definition 11.1: Program Classification

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of n distinct program specifications (classes), and $P = \{p_1, p_2, \ldots, p_m\}$ be set of program implementations, where each $p_i \in P$ correctly implements a specification $s_j \in S$. Then the objective of program classification is to learn a function $f : P \to S$ that maps a given program p_i to s_j.

Example 11.2.1: Program Classification

Consider a competitive programming problem that asks to sort an array. Different submissions may use quicksort, mergesort, heapsort, or bubble sort with varying loop structures and variable naming conventions. Despite these algorithmic and syntactic

differences, all correct submissions belong to the same problem class. Program classification learns to recognize this semantic equivalence from the IR2Vec embeddings and assigns all variants to the same class.

We model this as an ML-based classification, where each program is represented using IR2Vec embeddings (Chapter 3). We train a simple feed-forward neural network to map the given program to its class. The experimentation setup and the approach is described in detail in Section 11.2.2

11.2.1 Dataset

To evaluate program classification, we conduct experiments on three different competitive programming datasets of increasing complexity.

POJ The POJ dataset, originally introduced by Mou et al. [2016], has been widely used in prior works on program classification [Ben-Nun et al., 2018; Cummins et al., 2021; Damásio et al., 2023; Ye et al., 2021]. It consists of 104 classes, each representing a unique problem specification. The dataset contains relatively simple programs that primarily involve basic algorithms and data structures, making it a well-suited benchmark for evaluating classification models.

CodeJam The CodeJam dataset is derived from Google's annual programming competition, Google Code Jam. We collected C and C++ submissions from 2008–2017, including only those programs that correctly passed all test cases. The dataset consists of more than 135 classes, covering a broader range of problem-solving techniques and algorithmic complexity than POJ.

CodeForces (CoFo) The CoFo dataset is based on submissions from CodeForces, a popular online competitive programming platform. The dataset includes solutions submitted to a variety of problem categories, capturing a wide range of algorithmic techniques and coding styles. Our CoFo dataset [Gautam et al., 2025] provides a more challenging classification task due to the diversity of problem-solving approaches.

We filter out each dataset to include only C and C++ programs and obtain LLVM IR using the Clang compiler without any optimizations. Then the dataset is split into training, validation, and test sets, with a 60%, 20%, and 20% datapoints for each class respectively. We only consider the classes that have at least 200 programs in the dataset. The statistics of the datasets considered for experiments are summarized in Table 11.1.

Table 11.1: Dataset Description

Dataset	POJ	CodeJam	CoFo
#classes	98	137	342
#programs	40,922	238,659	257,134
#programs in train dataset	24,508	143,138	154,144
#programs in test dataset	8,189	47,733	51,429
#programs in val dataset	8,225	47,788	51,561
Max LOC per class	5,677	56,820	60,702
Min LOC per class	22	13	27
Avg LOC per class	262	1,765	1,296

11.2.2 Approach

We use simple feed-forward neural networks to classify programs based on their functionality. Each program is represented using IR2Vec embeddings, and the network is trained to predict the class label of the program. We experiment with different variants of IR2Vec embeddings, including Symbolic, Flow-Aware, and Profile-Aware representations, to evaluate their effectiveness in capturing program semantics.

To obtain the Profile-Aware embeddings, we first instrument the programs to collect dynamic profile information, including the number of times each basic block is executed. We then use this profile information to generate the Profile-Aware embeddings, which encode the execution frequency of each basic block in the program. The Profile-Aware embeddings are expected to capture the runtime behavior of the program and provide additional context to the classification model. We use Clang's built-in profiling support (PGO) to collect the profile information.

The network architecture consists of an input layer, followed by one or more hidden layers with SiLU activation function, and an output layer with a oftmax activation function. The model is trained using the categorical cross entropy loss function and Adam optimizer. We use batch normalization and dropout to prevent overfitting and improve generalization. We hyperparameter tune the network architecture and training parameters, including the number of hidden layers, hidden units, learning rate, batch size, and dropout rate, using random search on the training set of the POJ dataset. The best configuration is then used for training and evaluation on the CodeJam and CoFo datasets.

For each embedding variant, we do a hyperparameter search to find the best ar-

chitecture and training parameters. We use the same architecture for all datasets and compare the performance of different embeddings on the test set. We evaluate the classification performance using accuracy as the primary metric and compare the results with feature-based techniques and other embedding-based approaches.

11.2.3 Baselines

We compare the performance of our approach with the following baselines:

Milepost [**Fursin et al., 2011**] Milepost is a machine learning based compiler optimization framework that uses a set of features extracted from the programs to predict the optimization parameters for a given program. We use the same features to train a classifier for program classification. There are 56 features in total, including number of instructions, number of branches, etc.

Histograms of Opcodes [**Damásio et al., 2023**] As described in Section 10.4.3, histograms of opcodes is a simple feature-based approach that captures the distribution of opcodes in a given program. The authors show the histograms of opcodes can perform on par or better than other embedding-based techniques on simple programs like POJ.

11.2.4 Experiments

We conduct experiments to evaluate the performance of different IR2Vec embeddings for program classification on the POJ, CodeJam, and CoFo datasets. We compare the accuracy of the embeddings on the test set and analyze the results to understand the strengths and limitations of each representation.

Experimental Setup

As described in Section 11.2.2, we use a feed-forward neural network to classify programs based on their functionality. We hyperparameter tune the network architecture on the training set of the POJ dataset and use the best configuration for training and evaluation on the CodeJam and CoFo datasets. We follow the same process of hyperparameter tuning followed by training for other baselines as well.

The best configuration obtained upon hyperparameter tuning for each embedding variants is shown in Table 11.2.

Table 11.2: Best Hyperparameter Configurations

Embedding Technique		Architecture (Layers, Units)	Dropout	Learning Rate
Opcode Histograms		4, [512, 128, 512, 256]	0.277	9.5×10^{-4}
Milepost		4, [256, 128, 512, 512]	0.263	1.2×10^{-4}
IR2Vec	Symbolic	3, [256, 512, 128]	0.263	1.2×10^{-4}
	Flow-Aware	5, [512, 128, 512, 128, 512]	0.288	1.4×10^{-4}
	Profile-Aware	5, [128, 256, 128, 128, 256]	0.144	2.9×10^{-4}

Out of all the datasets, only CoFo dataset has programs and the corresponding inputs to execute the programs. Hence, we could use these inputs to execute the programs and collect the profile information. This profile information is used to generate the Profile-Aware embeddings for the CoFo dataset as described by Algorithm 1 in Chapter 3.

Results

The obtained accuracies are shown in the Table 11.3. We observe that the IR2Vec embeddings outperform the baselines on all datasets, with the Flow-Aware and Profile-Aware embeddings achieving the highest accuracy.

We observe that the Profile-Aware embeddings, which capture the runtime behavior of the programs, show the best performance on the CoFo dataset, indicating the importance of dynamic profile information for program classification. The Flow-Aware embeddings exhibit the best performance on the POJ and CodeJam datasets due to their ability to capture the flow semantics of the programs. And, the Symbolic embeddings perform consistently well across all datasets, indicating their effectiveness in capturing the static structure of the programs.

The results demonstrate the effectiveness of IR2Vec embeddings in capturing program semantics and classifying semantically similar programs. The Flow-Aware and Profile-Aware embeddings, which incorporate additional context information, show improved performance over the Symbolic embeddings, highlighting the importance of capturing both static and dynamic aspects of the programs for program classification.

Another interesting observation is that as the complexity of the programs increases, semantic information becomes more critical for accurate classification. The CodeJam and CoFo datasets, which contain more complex programs than the POJ dataset, show a larger performance gap between the different embedding variants, with Flow-Aware and

Table 11.3: Program Classification - Accuracy

Embedding Technique	Accuracy
POJ	
Histogram	85.61%
Milepost	90.59%
IR2Vec - Symbolic	90.13%
IR2Vec - Flow-Aware	**92.71%**
CodeJam	
Histogram	77.52%
Milepost	79.44%
IR2Vec - Symbolic	81.84%
IR2Vec - Flow-Aware	**86.52%**
CoFo	
Histogram	82.56%
Milepost	85.50%
IR2Vec - Symbolic	86.17%
IR2Vec - Flow-Aware	90.46%
IR2Vec - Profile-Aware	**91.37%**

Profile-Aware embeddings significantly outperforming the Symbolic variant. Similarly, in the case of POJ dataset, the performance difference between IR2Vec embeddings and the baselines is less pronounced, indicating that simple feature-based techniques may be sufficient for classifying simpler programs.

It is also worth noting that with the increase in the amount of information captured, the accuracy of classification improves. The milepost features capture selected features of the programs, performs better than the histogram-based approach which captures only the opcode information. The symbolic variant of IR2Vec captures the inherent semantic properties along with the statistical correlation between the opcodes, types, and arguments, and performs better than the milepost features. The flow-aware variant captures the flow information in addition to the symbolic information, and the profile-aware variant captures the dynamic profile information in addition to the flow information. Both these variants outperform the symbolic variant, indicating that the additional context information improves the classification accuracy.

Takeaway 11.1: Semantic Depth Improves Accuracy

Classification accuracy improves progressively with semantic richness: Symbolic (86%) → Flow-Aware (90%) → Profile-Aware (91.37%), confirming that deeper program analysis translates directly to better ML performance on downstream tasks.

Impact of Optimizations We also evaluate the impact of compiler optimizations on the classification accuracy of the programs. We use the O3 optimization level to compile the programs and generate the IR2Vec embeddings. The results are shown in Table 11.4.

Table 11.4: Program Classification - Accuracy with O3 Optimization

Embedding Technique	Test Accuracy
POJ	
Histogram	89.71%
Milepost	89.64%
IR2Vec - Symbolic	90.29%
IR2Vec - Flow-Aware	**92.13%**
CodeJam	
Histogram	84.70%
Milepost	78.94%
IR2Vec - Symbolic	86.58%
IR2Vec - Flow-Aware	**86.63%**
CoFo	
Histogram	85.43%
Milepost	83.69%
IR2Vec - Symbolic	89.84%
IR2Vec - Flow-Aware	90.69%
IR2Vec - Profile-Aware	**92.83%**

We observe similar trends as with the O0 optimization level for the embeddings of IR2Vec and Milepost across all datasets. However, the performance of the histogram-based approach considerably improves with the O3 optimization level, indicating that the opcode distribution is more discriminative for optimized programs. Whereas, for other embedding-based approaches, the optimization has a marginal fine-tuning effect on the accuracy.

11.3 Application in Device Mapping

One of the important applications of ML-assisted program recognition is in the domain of device mapping. Here the main objective is to identify the code or code fragments that can be mapped to appropriate calls to highly optimal accelerators or libraries thereby offloading to appropriate devices to improve the performance [Woodruff et al., 2022; Martínez et al., 2023].

This application differs from the heterogeneous device mapping problem addressed in Chapter 6, where the goal was to predict whether an OpenCL kernel should execute on a CPU or GPU. Here, we focus on *code recognition*—identifying domain-specific code patterns (such as CRC or cryptographic functions) that can be replaced with optimized library calls or offloaded to specialized hardware accelerators like SmartNICs and DPUs (Data Processing Units).

In this section, we evaluate the efficiency of IR2Vec embeddings in detecting domain-specific codes such as Cyclic Redundancy Check (CRC) and various cryptographic functions. Identifying these codes is the first step toward replacing them with their corresponding library calls for optimized execution.

Baseline We consider Clara [Qiu et al., 2021] as the baseline for this experiment. Clara encodes the opcodes of the programs by using pattern matching algorithms and obtain a representation using one-hot encoding. This representation is then used as input to the ML models for classification. The primary motivation is to obtain insights that would enable automatic offloading of workloads on network accelerators like SmartNICs and DPUs [Netronome, 2017; Cavium, 2013; Nvidia, 2022]. Such offloading of Network Functions to appropriate accelerators and SmartNICs would improve the packet processing speeds thereby reducing the network latency [Kundel et al., 2021; Cui et al., 2021; Fang et al., 2021; Yan et al., 2019; Panda et al., 2021; Shah et al., 2020] .

Dataset We consider two different datasets for this problem. Both of these datasets are selected to suit the same theme of obtaining insights for offloading network functions to SmartNICs and DPUs.

CRC We consider the CRC dataset provided by Clara as one of the datasets for this application. The dataset consists of two classes – implementation of different variations of CRC code, and a collection of non-network functions (obtained from

POJ dataset [Mou et al., 2016]). The train set contains 22 CRC and $2.7K$ non-CRC programs; the test set contains 33 CRC and $1.9K$ non-CRC programs.

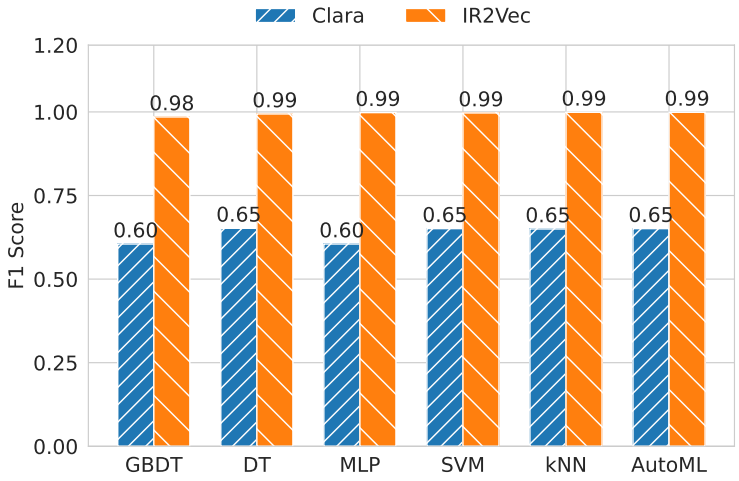
Cryptography Given the limited size of the CRC dataset, we manually curated a dataset containing the implementations of different cryptography algorithms from 6 different open-source libraries – OpenSSL [2024], Crypto++ [1995], Botan [2000], Nettle [Möller et al., 2001], WolfCrypt [2006], and Mbed-TLS [2009]. This dataset consists of encryption, decryption methods of 3 different cryptography algorithms – RSA, DES, and AES. This process resulted in about 121 different implementations with each of these three classes containing $(31, 48, 42)$ respectively.

We extend this dataset by applying compiler transformations to obtain different, semantically equivalent IRs for the same datapoint. We used 5 standard optimization sequences (`O[1-3]`, `Os`, `Oz`), and 300 random sequences of optimizations of varying length that resulted in a dataset containing $37K$ functions (121×306).

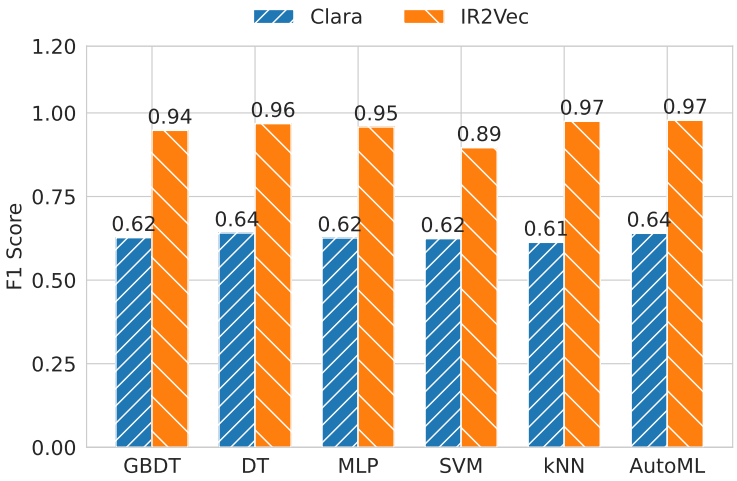
Experiment and Results We use Flow-Aware embeddings of IR2Vec for this experiment, as the inputs were not available to obtain the profiles for Profile-Aware embeddings. We use the same set of neural networks used by Clara for experiments – k-Nearest Neighbors (KNN), Support Vector Machine (SVM), Multilayer Perceptron (MLP), Decision Trees (DT), Gradient Boosting Decision Trees (GBDT), TPOT/AutoML [Olson et al., 2016] for training and evaluation. These models are implemented using scikit-learn, with standard hyperparameter settings. The models are trained independently, using the embeddings as inputs for binary classification (CRC dataset) and multi-class classification (cryptography dataset).

Given the dataset's skewed distribution, we use F1-score instead of accuracy for evaluation. IR2Vec embeddings consistently outperform Clara across all models on both datasets. The results, presented in Figure 11.1, demonstrate the effectiveness of IR2Vec embeddings in identifying domain-specific code. These findings highlight their potential for offloading network functions to specialized accelerators, improving efficiency and performance.

> **Takeaway 11.2: Domain-Specific Code Recognition**
>
> IR2Vec *embeddings enable accurate identification of domain-specific code patterns (CRC, cryptographic functions), demonstrating applicability to real-world offloading scenarios for SmartNICs and DPUs.*



(a) CRC Dataset



(b) Cryptography Dataset

Figure 11.1: F1 Scores obtained by Clara and IR2Vec

11.4 Related Works

Program similarity has been widely studied in the literature, with various approaches proposed to address the challenges of capturing semantic similarity. One of the early works in program similarity is by Mou et al. [2016], who introduced the Tree-Based

Convolutional Neural Network (TBCNN) for program classification using ASTs. Their approach captures the syntax and structure of programs by representing them as trees and applying convolutional operations to learn hierarchical features. Subsequently, Ben-Nun et al. [2018] used their instruction-level *inst2vec* embeddings to represent programs as sequences of instructions and used RNN models to perform program classification. They demonstrated that their approach outperformed traditional feature-based methods, highlighting the effectiveness of embedding techniques in capturing semantic information. Similarly, ProgramL [Cummins et al., 2021] uses GNNs with improved inst2vec embeddings and obtain improvements over the approach of Ben-Nun et al. [2018]. In our experiments, we observed that training this model takes significantly higher resources and time especially with the increase in the size of the dataset.

More recently, Damásio et al. [2023] showed that just considering histograms of opcodes as representation can be as powerful as other embedding-based techniques on a smaller dataset of 32 classes chosen from the POJ dataset. However, in our experiments we observe that as the complexity of dataset increases—in terms of number of classes and the length of programs—just considering the opcode information may not be sufficient. On the contrary, encoding more program semantics further enhances the performance.

Similar to our approach, Woodruff et al. [2022] uses ProgramL embeddings to identify the variants of FFT codes so that can be replaced with more efficient API calls. Once the variants are identified, they synthesize the programs carefully by using Input-Output matching to use the optimized APIs. Our device mapping work on NFs is similar in spirit to this work, but we focus on the identification of similar code fragments for offloading. Upon identifying the similar code fragments, we can use similar synthesis techniques to generate optimized code for the hardware accelerators.

11.5 Summary

This chapter explored the application of IR2Vec embeddings for program classification and device mapping tasks. We demonstrated that the semantic richness of IR-level embeddings enables effective classification of programs based on their functionality, achieving accuracies of up to 92.71% on POJ, 86.52% on CodeJam, and 91.37% on CodeForces (CoFo) datasets.

Our experiments revealed a clear hierarchy among embedding variants: Profile-Aware embeddings, which incorporate dynamic execution information, achieve the highest accuracy on datasets where runtime profiles are available. Flow-Aware embeddings excel

when only static analysis is feasible, while Symbolic embeddings provide a strong baseline across all settings. Notably, the performance gap between embedding variants widens as program complexity increases, highlighting the importance of encoding rich semantic information for challenging classification tasks.

Beyond program classification, we demonstrated the practical applicability of IR2Vec embeddings in device mapping, specifically for identifying network functions suitable for hardware acceleration. Our approach significantly outperforms the state-of-the-art system on both CRC and cryptography detection tasks, indicating the potential for embedding-based techniques in hardware-software co-design.

The techniques presented in this chapter, together with the binary similarity framework of Chapter 10, demonstrate the broad applicability of program embeddings for understanding code across different abstraction levels.

The source code for IR2Vec and the program classification experiments is available at https://github.com/IITH-Compilers/IR2Vec-Classification.

Chapter 12

Conclusion

In this chapter, we summarize the key insights, highlighting the contributions, perspectives, and future directions.

12.1 Summary

In this dissertation, we explore the intersection of machine learning and programs, focusing on ML4Code applications. The core idea of our work is that program embeddings—the machine-learned representation of programs—can encode rich semantic and structural properties of code. These embeddings can enhance compiler optimizations and various software engineering tasks by providing a learned understanding of programs at different abstraction levels.

Program Embeddings In addressing **RQ1** (Program Representations), Part I on page 10 of this dissertation focused on learning program embeddings using simple deep learning models. We introduce IR2Vec, MIR2Vec, and VexIR2Vec embedding techniques that leverage Intermediate Representations (IR) to capture both syntactic and semantic properties of code. Since IR abstracts away source language details, it serves as a robust means for learning embeddings that generalize across programming languages and applications. While IR2Vec derives embeddings from LLVM IR, which is independent of target architectures, MIR2Vec operates on Machine IR, making it more architecture-aware. VexIR2Vec, on the other hand, processes VEX IR extracted from binaries, enabling applications at the binary level.

To capture different properties of programs, we designed embeddings that encode

control flow, data flow, path structure, and runtime behavior. Symbolic embeddings capture intrinsic entity semantics, Flow-Aware embeddings encode data and control flow information, Path-Aware embeddings represent execution paths, and profile-aware embeddings integrate runtime behavior for precise flow information.

Generically across all program abstractions—LLVM IR, MIR, and binary—our embedding construction follows a bottom-up approach. It starts with learning a seed embedding vocabulary from instruction-level entities such as opcodes (O), types (T), and arguments (A). These embeddings are then aggregated at the basic block level and further at the function level (or any higher level), following the control flow structure. A naïve approach to representation learning would require encoding all possible opcode-type-argument combinations, resulting in an infeasibly large training space of $|O| \times |T| \times |A|$. Instead, our approach learns representations at the entity level, reducing the training space to just $|O| + |T| + |A|$. This avoids the OOV issues, as opcodes, types, and arguments exist in well-defined, relatively small sets—approximately 64 in LLVM IR, 300–600 in MIR, and 145 in VEX-IR. By ensuring coverage of all individual opcodes, types, and arguments, our method avoids the need for exhaustive training over combinatorial instruction variations.

This hierarchical design enhances scalability and generalizability. Once learned, the embeddings for programs can be constructed just by loading the vocabulary into memory while avoiding the model inference. This allows for efficient embedding construction for large programs, as the embeddings can be constructed in a single pass over the program without any additional dependencies. The learned embedding space preserves program semantics, naturally clustering similar entities and enabling complex analogical reasoning. Notably, IR2Vec and MIR2Vec have been upstreamed into the LLVM compiler infrastructure [VenkataKeerthy, 2025a], marking the first integration of program embedding techniques into a mainstream production compiler. A standalone tool, `llvm-ir2vec` [VenkataKeerthy, 2025b], is provided for embedding generation and vocabulary training.

ML-Driven Compiler Optimizations In addressing **RQ2** (Compiler Optimizations), Part II on page 90 of this dissertation explored how program embeddings can enhance compiler optimizations and introduced a framework for integrating machine learning models into compilers. By leveraging learned embeddings, we automate the optimization decisions and demonstrated their effectiveness in heterogeneous device mapping, thread coarsening, and register allocation.

For heterogeneous device mapping and thread coarsening, we use IR2Vec embeddings to predict optimal OpenCL kernel mappings and thread coarsening factors. A simple gradient boosting classifier trained on these embeddings significantly improves the performance, achieving speedups of 4–12% in device mapping and 4–24% in thread coarsening, while minimizing slowdowns compared to baseline approaches. Our embedding-based representations enabled the use of non-sequential models, leading to orders-of-magnitude faster training times.

For register allocation, we use MIR2Vec embeddings, which encode machine-specific LLVM-based representations, to optimize register assignments by using a multi-agent reinforcement learning approach. Our model learns optimal register allocations that generalize across different architectures, including Intel x86 and ARM AArch64. We further demonstrated the applicability of transfer learning to efficiently adapt to unseen architectures, often outperforming or matching the highly tuned Greedy allocator in LLVM, with up to 2% improvement in runtime on SPEC CPU benchmark suites.

To streamline the integration of ML models into compilers, we developed ML-Compiler-Bridge, an end-to-end infrastructure designed for modular and efficient interaction between machine learning models and compilation workflows. We demonstrate its support for a range of compiler infrastructures— including LLVM, MLIR, and Pluto—and ML frameworks like PyTorch, TensorFlow, and RLlib. By enabling seamless integration, ML-Compiler-Bridge significantly reduces the overhead of incorporating ML into compilation processes, leading to substantial compilation time improvements. We provide multi-language user APIs: C++ and C APIs for interfacing model runners and serializers with compilers, and Python APIs for interfacing inter-process model runners with ML frameworks. Its adaptability makes it applicable across a variety of optimization tasks across different target architectures, reinforcing the role of machine learning in modern compilers. We show that using appropriate interfacing would substantially reduce the training (by $\approx 2\times$) and compilation times (7–22$\times$).

ML-Driven Program Understanding In addressing **RQ3** (Program Understanding), Part III on page 178 of this dissertation explored how program embeddings can enhance program understanding tasks, particularly in binary similarity and program similarity.

In the binary similarity task, we show how VexIR2Vec embeddings can be used to learn the similarity between binaries. We use the embeddings to cluster binaries based on their similarity using a simple feed-forward Siamese network. We show that

the embeddings can capture the semantic information of the binaries and can be used to identify the similar binaries with high accuracy across various adversarial settings involving cross-compiler, cross-optimization, cross-architecture, and obfuscation transformations. We demonstrate the results on two different similarity tasks: diffing and searching. In diffing experiments, our approach outperforms the nearest baselines by 40%, 18%, 21%, and 60% in cross-optimization, cross-compilation, cross-architecture, and obfuscation settings, respectively. In the searching experiment, we achieved a mean average precision of 0.76, outperforming the nearest baseline by 46%. Our framework is highly scalable and is built as a lightweight, multi-threaded, parallel library using only open-source tools. VexIR2Vec is $\approx$ 3.1–3.5$\times$ faster than the closest baselines and orders-of-magnitude faster than other tools.

In the program similarity task, we show how IR2Vec embeddings can be used to learn the similarity between programs. We use the embeddings to train a simple feed-forward network-based classification model to classify programs based on their problem specifications. We show that the embeddings can capture the semantic information of the programs and can be used to classify the programs with high accuracy. Our approach outperforms the nearest baseline by 4%, 8%, and 9% upon using Symbolic, Flow-aware, and Profile-aware embeddings, respectively. These results highlight that incorporating richer program analysis information into the input representation improves classification performance, reinforcing the importance of embedding design in ML-driven program understanding.

12.2 Perspectives, Challenges and Future Directions

The findings of this dissertation open up several exciting directions for future research and potential ML4Code applications. However, as with any emerging technology, there are also notable limitations and challenges that must be addressed to further improve the utility and scalability of these approaches. We discuss some of these perspectives and future directions below.

12.2.1 Program Embeddings

Encoding more Program Information As demonstrated across different chapters, encoding richer program information in embeddings can lead to more effective representations. For instance, in device mapping and thread coarsening optimizations, the use of flow-aware embeddings yields higher speedups compared to symbolic embeddings. Similarly, in the VexIR2Vec and IR2Vec experiments on binary and program similarity tasks, embeddings that capture deeper semantic and structural aspects of programs consistently outperform those that encode less information.

There is significant potential in designing embeddings that capture additional program properties such as pointer aliasing, memory access patterns, and cache behavior. Encoding such details can result in more powerful and expressive representations that are applicable to a wide range of domain-specific optimizations and program understanding tasks. For instance, embeddings that incorporate precise loop information, such as program dependences, could open up new opportunities in polyhedral optimizations like loop tiling, loop fusion, etc. Similarly, embeddings that capture memory access patterns can be useful in modeling cache behavior and designing cache-aware optimizations.

An alternative direction, particularly when domain-specific features are already known, is to adopt a hybrid approach: concatenating handcrafted features with learned embeddings to form enriched representations. While embeddings capture generic, intrinsic properties of programs, domain-specific features can supplement them with targeted, task-relevant information. For example, in the register allocation task, concatenating embeddings with spill cost enables better modeling of inputs. This further reinforces the idea that augmenting embeddings with more relevant information—learned or explicit—can lead to better performance.

Modular Framework Design and Large Language Models A key architectural insight from our embedding approach is the decoupling of *vocabulary training* from *embedding generation* (Section 2.4 on page 19). The seed embedding vocabulary is trained offline (using TransE in our implementation), while embedding generation at compile time reduces to efficient dictionary lookups and vector operations. This modular design makes the vocabulary training component replaceable without affecting the embedding generation pipeline.

This opens an intriguing future direction: replacing TransE-derived vocabularies with LLM-derived embeddings. If large language models trained on code produce semantically richer entity representations that better capture relationships between IR

constructs, such LLM-enhanced vocabularies would still be used as lookup tables at compile time. This preserves the efficiency that makes our approach practical and scalable for compiler integration. The LLM inference cost is amortized across all programs during the one-time vocabulary training phase, not incurred during compilation.

This perspective suggests that our embedding techniques are not merely specific embedding algorithms but rather an *embedding framework* where the vocabulary learning method is a pluggable component. Exploring different vocabulary training strategies—including LLM-based approaches—while maintaining the efficient lookup-based embedding generation is a promising avenue for future research.

Scalability One of the key challenges in using embeddings for different applications is its scalability. In our work, we addressed this concern by adopting a bottom-up design approach and employing a seed embedding vocabulary as a lookup table, avoiding the overhead of running inference with machine learning models.

However, as we encode more *precise* program information in the embeddings, the challenge of scalability becomes more prominent. For instance, flow-aware embeddings take more time to generate compared to symbolic embeddings. Though some part of the solution to the scalability problem can be addressed by using efficient data structures and algorithms, there is still a need for more research in designing scalable embeddings that can capture more program information especially while dealing with large codebases.

Another dimension of scalability is to compute the embeddings with partial context. For instance, in the IR2Vec embeddings, we considered the entire function for obtaining flow information. However, in practice, it may not always be feasible to consider the entire function. In such cases, one can consider computing the embeddings with partial context, with limited instructions by abstracting the function to a smaller context. This approach has been explored in the ML-driven inliner available in LLVM [Trofin et al., 2021b; VenkataKeerthy, 2025a], where embeddings are computed only for the affected region rather than the entire function, significantly reducing overhead while retaining effectiveness. This can be particularly useful in scenarios where the functions are large or not fully available.

Extensions to Other Languages and Representations Our approach to designing embeddings primarily focused on intermediate representations derived from compiled programs, such as C, C++, and their corresponding binaries. This focus was driven by the widespread adoption of LLVM and Valgrind toolchains. While IR-based embed-

dings are a step toward generalization across different languages and representations, fully bridging this gap remains an open challenge. Nevertheless, we believe that the core ideas presented in this dissertation offer a foundation for extending these embeddings to other languages and representations.

For instance, these ideas can be extended to high-level languages like Java and Python, where the *lifecycle* of the program differs significantly from compiled languages. Similarly, VexIR2Vec can be extended to different binary representations and toolchains, broadening its applicability. These techniques can also be adapted to other intermediate representations like MLIR and other polyhedral representations.

However, such generalization is non-trivial. Embeddings designed for LLVM IR may not seamlessly translate to MLIR dialects due to structural differences and domain-specific abstractions. Designing embeddings for such emerging representations presents a rich space for future research, requiring a careful understanding of the unique constraints and semantics of each compiler infrastructure.

A compelling future direction is to move beyond representation-specific embeddings and aim for the design of *universal embeddings*—representations capable of capturing fundamental program properties across languages and IRs. Achieving this would require either constructing a shared latent space that inherently captures cross-language program semantics or designing a mapping function to translate embeddings between representations. Moving towards such a unified embedding paradigm could significantly improve portability and generalization, making embeddings more robust across compiler toolchains and programming environments.

12.2.2 ML-Driven Compiler Optimizations

Dataset One of the key challenges in using ML models for compiler optimizations is the availability of datasets. Even though programs are available prevalently in the public domain, not all of them are suitable for the optimization problem at hand. The programs need to be carefully curated to ensure that they are suitable for the optimization problem that is being addressed.

To cater to this, researchers often rely on generating synthetic programs or design domain-specific custom code generators to generate programs that are suitable for the optimization problem. Though some works on synthetic benchmark generation have been proposed [da Silva et al., 2021; Cummins et al., 2017b], there remains a significant need for more research in this direction to generate programs that suit a specific optimization

problem.

Our contributions in this direction include expanding datasets by applying targeted compiler transformations [VenkataKeerthy et al., 2023a], mutating certain properties of the existing programs to create new variants [Jain et al., 2022b] and generating domain-specific workloads through custom code generators [VenkataKeerthy et al., 2023d]. Collectively, these efforts represent a small step forward in addressing this challenge.

Bootstrapping from Heuristic-Based Optimizations Thanks to the decades of research in compiler optimizations, the optimizations are well understood and the heuristics undergo improvements over time to provide better performance on new architectures and other regression cases. A promising, but unexplored direction in this dissertation is leveraging these heuristics to bootstrap learning-based models.

This can be achieved by using approaches like behavioral cloning [Bain and Sammut, 1995], where the ML model initially mimics the decisions made by existing heuristics and subsequently improves upon them using additional learning and reward signals. Such a bootstrapping mechanism can be particularly helpful in constraining the vast exploration space of the models and providing better initializations for the models. Prior works, such as MLGO [Trofin et al., 2021b], demonstrate that this hybrid approach can lead to practical and effective solutions.

Performance As shown in our experiments, specifically in RL4ReAl, the ML models exhibit impressive performance, often matching or even surpassing highly-tuned heuristics in certain scenarios. However, these models do not consistently guarantee better performance across all cases. This is especially true when the heuristics are highly optimized for specific scenarios.

This highlights the opportunity, and the need for further improvements in model design and training methods. Incorporating recent advances in machine learning and reinforcement learning could significantly enhance model generalization and robustness. Equally important is acknowledging the value of decades of compiler research and integrating these insights into ML systems.

We strongly believe that the existing optimizations cannot be completely subsumed by the ML models, but rather the ML models can complement and aid the existing heuristic-based optimizations to achieve better results. This aspect of designing hybrid models that combine the best of both worlds is an interesting direction for future research.

Scalability Scalability remains a key challenge for deploying ML models within compiler toolchains, particularly in terms of inference latency and memory footprint. Since inference happens during compilation, any delay directly affects compile time, which is an essential factor for usability and adoption. Likewise, the size of the ML models determines their feasibility for integration in resource-constrained environments.

Our work on ML-Compiler-Bridge takes a step toward making ML-compiler integration more efficient, enabling on-the-fly inference while minimizing compile-time overhead. However, further research is needed to improve scalability, particularly when shipping compilers with embedded ML models. Techniques such as model quantization, pruning, and low-precision computation (like floating-point compression) are promising directions to reduce both inference time and memory usage [Bucilu et al., 2006; Choudhary et al., 2020]. Techniques such as model quantization, pruning, and low-precision computation (like floating-point compression) are promising directions to reduce both inference time and memory

Reward Signal and Performance Modeling In RL-based optimizations, the reward signal plays a crucial role in training the models. While designing rewards for code size optimizations is relatively straightforward, designing rewards for performance optimizations is challenging.

Rewards for performance optimizations can either be modeled statically or dynamically. In our work on register allocation, we used throughput estimates from LLVM-MCA as the reward signal, such static estimates are highly approximate and may not always reflect the actual performance of the program. On the other hand, dynamic rewards can be obtained by running the program on the target architecture and measuring the performance. However, this approach is highly expensive and may not always be feasible.

Hence, there is a need for more research in designing reward signals that can accurately model the performance of the programs. This can be achieved either by designing better, scalable static analysis techniques or by designing ML-based solutions that can accurately predict the performance of the programs. Though some efforts have been made in this direction, the existing solutions are limited in terms of scalability and accuracy [Hsu et al., 2023; Mendis et al., 2019a]. Embedding-based approaches for performance modeling is an interesting direction to explore in this context. Such reward models can be combined with self-play techniques [Zhang et al., 2024] to further improve the performance of the models.

Energy Optimizations The ML-driven optimizations discussed in this dissertation primarily focus on performance optimizations. Some of our other works [Jain et al., 2022a] have explored code size optimizations. However, energy optimizations are equally important in modern computing systems. Optimizations that minimize energy usage without compromising performance could be especially valuable for mobile and embedded systems. Incorporating energy consumption as a factor in the optimization process will require novel approaches to model energy use in addition to the existing performance metrics. This is an interesting direction for future research.

Autotuning As discussed in Chapter 5, autotuning is a popular technique to optimize the performance of programs. The existing autotuning techniques either rely on exhaustive search or use heuristics to guide the search. ML models can be used to guide the search in a more efficient manner. Recent works [Wu et al., 2023] have demonstrated the potential of ML-guided autotuning, showing significant improvements in convergence speed and result quality. An interesting direction for future work is to explore how embeddings can be used to further enhance autotuning.

Large Language Models The works presented in this dissertation primarily focus on using relatively simple neural networks to model the underlying tasks. This choice is mainly to balance the complexity of the models with the practicality of the solutions in real-world scenarios. However, the recent advancements in Large Language Models (LLMs) have demonstrated impressive capabilities in program understanding tasks [Feng et al., 2020; Guo et al., 2021; Rozière et al., 2024; Wang et al., 2023b; Guo et al., 2022].

An interesting direction for future research is to explore how precomputed embeddings can be used in conjunction with LLMs to improve the performance of the models. As discussed in the modular framework design above, LLMs could also enhance the seed vocabulary itself. Beyond this, embeddings can be used as the input representations to the LLMs, providing structured semantic information that complements token-based representations. Embeddings derived from deeper static or program analyses can serve as compact, structured inputs to LLMs, capturing intrinsic program properties that may not be directly evident from source tokens. LLMs, in turn, can provide complementary strengths by modeling rich contextual and sequence-based patterns. This synergy could lead to improved performance in various program understanding tasks.

Another direction would be to explore how LLMs can be used in compiler optimizations. Early works on using LLMs to model trivial optimizations [Cummins et al., 2025;

Taneja et al., 2025] have been proposed. A future research direction is to explore how LLMs can be used in more complex optimizations like register allocation, while ensuring correctness.

A major bottleneck in deploying LLMs within compiler toolchains is their scalability and computational cost. These models require substantial compute resources for training and inference, which poses challenges for integration into production compilers. Efficient model compression techniques, distillation methods, and hardware-aware optimizations will be critical to make LLM-based solutions practical.

12.2.3 ML-Driven Program Understanding

Program Comprehension and Generation The focus of this dissertation on program understanding has been largely on program similarity. However, program comprehension and generation are equally important aspects of ML-driven program understanding. Existing approaches in these domains predominantly rely on feature- or token-based representations. It would be worthwhile to explore how embeddings can be used in program comprehension and generation tasks.

For instance, embeddings can be used in comprehension tasks like comment generation, method name prediction, etc. by using the existing generative models that can generate natural language tokens. However, the embeddings should be supplemented with additional identifier information to make the generated comments or method names more meaningful and relevant.

Similarly, for the code generation tasks like language translation where the inputs and outputs are source codes, the embeddings can be used as the input representation, and the outputs can be generated using the existing sequence to sequence or other generative models. However, the challenge in this direction is to ensure that the generated code is syntactically and semantically correct. An interesting direction would be to explore how the models can be constrained to generate syntactically correct code by closely following the grammar rules of the target language [Svyatkovskiy et al., 2020].

Binary–Source Code Matching Another interesting direction would be to explore how embeddings can be used in matching the source code with the corresponding binaries or vice-versa. This task has applications in software security, IP protection, and reverse engineering.

This can be achieved by learning a common latent space that can capture the prop-

erties of both the source code and the binaries. The IR2Vec and VexIR2Vec embeddings can be used as the input representations for the source code and the binaries respectively. The embeddings can be learned in such a way that the similar programs in the source code and the binaries are positioned closely in the latent space. This can be achieved by using the existing Siamese network architectures that are designed for similarity tasks. Then the problem of matching the source and binary codes can be formulated as a similarity task by retrieving the closest neighbors in the latent space.

12.3 Closing Remarks

As Turing envisioned, intelligence in computing need not be manually programmed; it can emerge from learning. This dissertation takes a step towards that vision by demonstrating how *program embeddings* can be leveraged for ML-driven compiler optimizations and program understanding. By integrating learning-based techniques with software systems, we move closer to building *adaptive, self-improving, intelligent* compilers and program understanding tools.

The methods and techniques presented in this dissertation merely scratch the surface of the vast possibilities in adopting *ML for Code*. As software systems continue to grow in complexity and hardware architectures diversify, the need for intelligent, learning-based tools will only intensify. We believe these approaches will pave the way for developing *scalable, practical, and effective* solutions that can generalize across different architectures and workloads, ultimately transforming how we optimize and understand programs at scale.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org. 165

S. Ahn, S. Ahn, H. Koo, and Y. Paek. Practical binary code similarity detection with bert-based transferable similarity learning. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 361374, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450397599. DOI: 10.1145/3564625.3567975. URL https://doi.org/10.1145/3564625.3567975. 229, 230, 231, 232, 233, 235

A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491516, Oct. 1989. ISSN 0164-0925. DOI: 10.1145/69558.75700. URL https://doi.org/10.1145/69558.75700. 92

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811. 14, 53, 92

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page

26232631, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362016. DOI: 10.1145/3292500.3330701. URL https://doi.org/10.1145/3292500.3330701. 204

M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015a. Association for Computing Machinery. ISBN 978-1-4503-3675-8. DOI: 10.1145/2786805.2786849. 12, 55, 181

M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2123–2132, Lille, France, 07–09 Jul 2015b. PMLR. URL https://proceedings.mlr.press/v37/allamanis15.html. 56

M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In M.-F. Balcan and K. Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2091–2100. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/allamanis16.html. 12, 55

M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), July 2018a. ISSN 0360-0300. DOI: 10.1145/3212695. URL https://doi.org/10.1145/3212695. 13, 56, 150

M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018b. URL https://openreview.net/forum?id=BJOFETxR-. 55

F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 119, New York, NY, USA, 1970. Association for Computing Machinery. ISBN 9781450373869. DOI: 10.1145/800028.808479. URL https://doi.org/10.1145/800028.808479. 92

F. E. Allen. A basis for program optimization. In *IFIP Congress (1)*, pages 385–390, 1971. 92, 180

P. Almasan, J. Suárez-Varela, A. Badia-Sampera, K. Rusek, P. Barlet-Ros, and A. Cabellos-Aparicio. Deep reinforcement learning meets graph neural networks: exploring a routing optimization use case, 2020. 129, 317

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 404–419, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. DOI: 10.1145/3192366.3192412. URL http://doi.acm.org/10.1145/3192366.3192412. 55

U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019a. URL https://openreview.net/forum?id=H1gKYo09tX. 12, 181

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages (POPL)*, volume 3, pages 40:1–40:29, 2019b. DOI: 10.1145/3290353. 12, 55, 56, 151, 181, 183

E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010. ISBN 026201243X. 146

S. P. Amarasinghe. Compiler 2.0: Using machine learning to modernize compiler technology. In J. Xue and C. Jung, editors, *Proceedings of the 21st ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2020, London, UK, June 16, 2020*, pages 1–2. ACM, 2020. DOI: 10.1145/3372799.3397167. URL https://doi.org/10.1145/3372799.3397167. 94, 150, 318

AMD. AMD OpenCL accelerated parallel processing SDK . https://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/, n.d. [Online; accessed 18-May-2020]. 104, 113

Annoy. ANNOY library. https://github.com/spotify/annoy, 2024. Accessed: 2024-11-22. 186

J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: an extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and*

Compilation, PACT '14, pages 303–316, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328098. DOI: 10.1145/2628071.2628092. URL https://doi.org/10.1145/2628071.2628092. 93

A. W. Appel and L. George. Optimal spilling for cisc machines with few registers. *SIGPLAN Not.*, 36(5):243253, May 2001. ISSN 0362-1340. DOI: 10.1145/381694.378854. URL https://doi.org/10.1145/381694.378854. 121

A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Trans. Archit. Code Optim.*, 14(3), Sept. 2017. ISSN 1544-3566. DOI: 10.1145/3124452. URL https://doi.org/10.1145/3124452. 122, 150

A. Ayupov, M. Panchenko, and S. Pupyrev. Stale profile matching, 2024. 65, 191

J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The fortran automatic coding system. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), page 188198, New York, NY, USA, 1957. Association for Computing Machinery. ISBN 9781450378611. DOI: 10.1145/1455567.1455599. URL https://doi.org/10.1145/1455567.1455599. 91

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, May 7-9, 2015, Conference Track Proceedings*, San Diego, CA, USA,, may 2015. URL http://arxiv.org/abs/1409.0473. 199

M. Bain and C. Sammut. A framework for behavioural cloning. In *Machine Intelligence 15*, pages 103–129. Oxford University Press, 1995. URL https://www.cse.unsw.edu.au/~claude/papers/MI15.pdf. 257

B. S. Baker and U. Manber. Deducing similarities in java sources from bytecodes. In *1998 USENIX Annual Technical Conference (USENIX ATC 98)*, New Orleans, LA, June 1998. USENIX Association. URL https://www.usenix.org/conference/1998-usenix-annual-technical-conference/deducing-similarities-java-sources-bytecodes. 228

T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 59–70, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. DOI: 10.1145/143165.143180. URL http://doi.acm.org/10.1145/143165.143180. 27

R. Barik, C. Grothoff, R. Gupta, V. Pandit, and R. Udupa. Optimal bitwise register allocation using integer linear programming. In *Languages and Compilers for Parallel Computing*, LCPC, pages 267–282, 2006. DOI: 10.1007/978-3-540-72521-3_20. URL https://doi.org/10.1007/978-3-540-72521-3_20. 121

T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. Neural code comprehension: a learnable representation of code semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 35893601, Red Hook, NY, USA, 2018. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=3327144.3327276. 25, 53, 56, 57, 102, 104, 105, 106, 107, 113, 114, 118, 119, 147, 151, 183, 239, 248, 319

Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828, Aug. 2013. ISSN 0162-8828. DOI: 10.1109/TPAMI.2013.50. URL http://dx.doi.org/10.1109/TPAMI.2013.50. 13, 17, 61, 191

J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509517, sep 1975. ISSN 0001-0782. DOI: 10.1145/361002.361007. URL https://doi.org/10.1145/361002.361007. 208

C. Böhm. *On encoding logical-mathematical formulas using the machine itself during program conception. Translated 2016 by Peter Sestoft from Corrado Böhm: Calculatrices digitales. Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme. ETH Zürich (1951).* PhD thesis, PhD dissertation. French original published in Bologna, 1954. 91

P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017. DOI: 10.1162/tacl_a_00051. URL https://aclanthology.org/Q17-1010. 198

U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 101113, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. DOI: 10.1145/1375581.1375595. URL https://doi.org/10.1145/1375581.1375595. 92, 154, 173

Boost. Boost C++ Libraries. https://www.boost.org/, 2018. Accessed 2019-05-16. 48

A. Bordes, N. Usunier, A. Garcia-Durán, J. Weston, and O. Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 2787–2795, 2013. URL http://dl.acm.org/citation.cfm?id=2999792.2999923. 17, 18, 21, 55, 139, 231

Botan. Botan: Crypto and TLS for Modern C++. https://botan.randombit.net/, 2000. [Online; accessed 22-June-2022]. 246

F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 283–298, 2006. DOI: 10.1007/978-3-540-72521-3_21. 121, 133, 141

M. Boulton. TSVC_2. https://github.com/UoB-HPC/TSVC_2.git, n.d. Accessed 2015-09-16. 169

M. Bourquin, A. King, and E. Robbins. Binslayer: Accurate comparison of binary executables. In *PPREW*, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318570. DOI: 10.1145/2430553.2430557. URL https://doi.org/10.1145/2430553.2430557. 65

A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 201211, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371209. DOI:

10.1145/3377555.3377894. URL https://doi.org/10.1145/3377555.3377894. 55, 57, 113

A. Brauckmann, A. Goens, and J. Castrillon. Polygym: Polyhedral optimizations as an environment for reinforcement learning. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 17–29, 2021. DOI: 10.1109/PACT52795.2021.00009. 151, 176

P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428455, may 1994. ISSN 0164-0925. DOI: 10.1145/177492.177575. URL https://doi.org/10.1145/177492.177575. 121

P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, 2005. URL https://www.researchgate.net/publication/228923820. 133

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016. 151

D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: a binary analysis platform. In *Computer Aided Verification*, CAV'11, page 463469, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642221095. 229

J. Bucek, K.-D. Lange, and J. v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5629-9. DOI: 10.1145/3185768.3185771. URL http://doi.acm.org/10.1145/3185768.3185771. 48, 203

C. Bucilu, R. Caruana, and A. Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 535–541, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933395. DOI: 10.1145/1150402.1150464. 258

J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*,

ESEC/FSE 2019, page 964974, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. DOI: 10.1145/3338906.3340458. URL https://doi.org/10.1145/3338906.3340458. 12

J. Cavazos, G. Fursin, F. Agakov, E. Bonber, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197. IEEE, 2007. DOI: 10.1109/CGO.2007.32. 94, 318

Cavium. Cavium LiquidIOő Server Adapter Family. https://datasheet.octopart.com/CN6130-110SV-G-Cavium-Networks-datasheet-26366670.pdf, 2013. [Online; accessed 22-June-2022]. 245

G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981. DOI: 10.1016/0096-0551(81)90048-5. 92, 121, 126

M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 678689, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. DOI: 10.1145/2950290.2950350. URL https://doi.org/10.1145/2950290.2950350. 59, 62, 191, 231, 235

C. M. Chang, C. M. Chen, and C. T. King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers & Mathematics with Applications*, 34(9):1–14, 1997. DOI: 10.1016/S0898-1221(97)00184-3. 121

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. DOI: 10.1109/IISWC.2009.5306797. 104

T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018a. USENIX

Association. URL https://www.usenix.org/conference/osdi18/presentation/chen. 164

T. Chen, S. Kornblith, M. Norouzi, and G. Hinton. A simple framework for contrastive learning of visual representations. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1597–1607. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/chen20j.html. 198

W. Chen, G. Lueh, P. Ashar, K. Chen, and B. Cheng. Register allocation for intel processor graphics. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 352364, 2018b. ISBN 9781450356176. DOI: 10.1145/3168806. URL https://doi.org/10.1145/3168806. 121

K. Cho, M. B. van, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, EMNLP 2014, pages 1724–1734, Oct. 2014. DOI: 10.3115/v1/D14-1179. URL https://aclanthology.org/D14-1179. 136

T. Choudhary, V. Mishra, A. Goswami, and J. Sarangapani. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, 53:5113–5155, 2020. DOI: 10.1007/s10462-020-09816-7. 258

F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM TOPLAS*, 12(4):501536, oct 1990. ISSN 0164-0925. DOI: 10.1145/88616.88621. URL https://doi.org/10.1145/88616.88621. 121

Z. L. Chua, S. Shen, P. Saxena, and Z. Liang. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 99–116, Vancouver, BC, Aug. 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua. 235

E. M. Clarke. Model checking. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69659-9. DOI: 10.1007/BFb0058022. 180

J. Collyer, T. Watson, and I. Phillips. Faser: Binary code similarity search through the use of intermediate representations, 2023. 229

L. Community and L. D. conference. Pbqp register allocation. https://llvm.org/devmtg/2014-10/Slides/PBQP-update-and-in-the-wild.pdf, 2014. 147

K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002. DOI: 10.1023/A:1015729001611. URL https://doi.org/10.1023/A:1015729001611. 27, 150

K. Coppieters. A cross-platform binary diff. https://www.drdobbs.com/embedded-systems/a-cross-platform-binary-diff/184409550, 1995. [Online; accessed 12-Nov-2023]. 228

Coreutils. GNU Coreutils. https://www.gnu.org/software/coreutils/, 2024. [version 9.0; Online; accessed 08-May-2024]. 192, 202

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848. 11

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. DOI: 10.1145/512950.512973. URL https://doi.org/10.1145/512950.512973. 12, 180

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, page 8496, New York, NY, USA, 1978. Association for Computing Machinery. ISBN 9781450373487. DOI: 10.1145/512760.512770. URL https://doi.org/10.1145/512760.512770. 12, 180

Crypto++. Crypto++ő Library 8.6. https://www.cryptopp.com/, 1995. [Online; accessed 22-June-2022]. 246

T. Cui, W. Zhang, K. Zhang, and A. Krishnamurthy. *Offloading Load Balancers onto SmartNICs*, page 5662. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450386982. URL https://doi.org/10.1145/3476886.3477505. 245

C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017a. DOI: 10.1109/PACT.2017.24. 25, 53, 55, 57, 102, 104, 105, 106, 107, 113, 114, 119

C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99, 2017b. DOI: 10.1109/CGO.2017.7863731. 95, 150, 256

C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 2244–2253. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/cummins21a.html. 57, 119, 120, 136, 151, 183, 239, 248, 319

C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather. Compilergym: Robust, performant compiler optimization environments for AI research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 92–105, 2022. DOI: 10.1109/CGO53902.2022.9741258. 123, 148, 151, 153, 173, 176

C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather. Llm compiler: Foundation language models for compiler optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, CC '25, page 141153, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714078. DOI: 10.1145/3708493.3712691. URL https://doi.org/10.1145/3708493.3712691. 259

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991. DOI: 10.1145/115372.115320. 34, 63, 92, 133

A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. G. aes, and F. M. Q. ao Pereira. Anghabench: A suite with one million compilable c bench-

marks for code-size reduction. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 378390. IEEE Press, 2021. ISBN 9781728186139. DOI: 10.1109/CGO51591.2021.9370322. URL https://doi.org/10.1109/CGO51591.2021.9370322. 150, 256

T. Damásio, M. Canesche, V. Pacheco, M. Botacin, A. Faustino da Silva, and F. M. Quintão Pereira. A game-based framework to compare program classifiers and evaders. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, page 108121, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701016. DOI: 10.1145/3579990.3580012. URL https://doi.org/10.1145/3579990.3580012. 59, 69, 183, 191, 192, 205, 235, 237, 239, 241, 248

A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, page 6374, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589350. DOI: 10.1145/1735688.1735702. URL https://doi.org/10.1145/1735688.1735702. 104

D. Das, S. A. Ahmad, and V. Kumar. Deep learning-based approximate graph-coloring algorithm for register allocation. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, Workshop on the LLVM Compiler Infrastructure in HPC, pages 23–32, 2020. DOI: 10.1109/LLVMHPCHiPar51896.2020.00008. 123, 147, 150, 152, 153

Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. *SIGPLAN Not.*, 51(6):266280, jun 2016. ISSN 0362-1340. DOI: 10.1145/2980983.2908126. URL https://doi.org/10.1145/2980983.2908126. 231

Y. David, N. Partush, and E. Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 7994, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. DOI: 10.1145/3062341.3062387. URL https://doi.org/10.1145/3062341.3062387. 59, 62, 65, 88, 191, 229, 230, 231, 235

S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378415, mar 2000. ISSN 0164-0925. DOI: 10.1145/349214.349233. URL https://doi.org/10.1145/349214.349233. 229

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. DOI: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423. 231

Diffutils. GNU Diffutils. https://www.gnu.org/software/diffutils/, 2024. [version 3.8; Online; accessed 08-May-2024]. 192, 202

S. H. H. Ding, B. C. M. Fung, and P. Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, 2019. DOI: 10.1109/SP.2019.00003. 59, 62, 88, 183, 191, 229, 230, 231, 232, 234, 235

M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. The faiss library. *IEEE Transactions on Big Data*, pages 1–17, 2025. DOI: 10.1109/TBDATA.2025.3618474. 186

Y. Duan, X. Li, J. Wang, and H. Yin. DeepBinDiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium (NDSS)*, 2020. URL https://www.ndss-symposium.org/ndss-paper/deepbindiff-learning-program-wide-code-representations-for-binary-diffing/. 59, 65, 88, 187, 191, 192, 205, 229, 230, 232, 233, 235

T. Dullien. Funcsimsearch. https://github.com/googleprojectzero/functionsimsearch, 2018. [Online; accessed 13-July-2022]. 231, 235

A. Dutta and A. Jannesari. Mirencoder: Multi-modal ir-based pretrained embeddings for performance optimizations. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, PACT '24, page 156167, New York, NY, USA, 2024. Association for Computing Machinery. ISBN

9798400706318. DOI: 10.1145/3656019.3676895. URL https://doi.org/10.1145/3656019.3676895. 120

S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla. discovRE: Efficient cross-architecture identification of bugs in binary code. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. URL http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/discovre-efficient-cross-architecture-identification-bugs-binary-code.pdf. 59, 191, 230, 231, 235

S. Fang, Q. Liu, and W. Wu. Hypernat: Scaling up network address translation with smartnics for clouds. *2021 IEEE Global Communications Conference (GLOBECOM)*, Dec 2021. DOI: 10.1109/globecom46510.2021.9685551. URL http://dx.doi.org/10.1109/GLOBECOM46510.2021.9685551. 245

M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87, 2014. DOI: 10.1109/SERE.2014.21. 58, 88, 182, 190

Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 480491, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. DOI: 10.1145/2976749.2978370. URL https://doi.org/10.1145/2976749.2978370. 59, 191, 230, 231, 235

Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, and Y. Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, Nov. 2020. Association for Computational Linguistics. DOI: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139/. 181, 259

Findutils. GNU Findutils. https://www.gnu.org/software/findutils/, 2024. [version 4.9; Online; accessed 08-May-2024]. 192, 202

Flatbuffers. FlatBuffers. https://flatbuffers.dev/index.html, n.d. [Online; accessed 29-Aug-2022]. 164

R. W. Floyd. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*, pages 65–81. Springer, 1993. DOI: 10.1007/978-94-011-1793-7_4. 12, 180

J. H. Friedman. Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38(4):367378, Feb. 2002. ISSN 0167-9473. DOI: 10.1016/S0167-9473(01)00065-2. URL https://doi.org/10.1016/S0167-9473(01)00065-2. 105

G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. K. I. Williams, and M. O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *IJPP*, 39(3):296–327, Jun 2011. ISSN 1573-7640. DOI: 10.1007/s10766-010-0161-2. URL https://doi.org/10.1007/s10766-010-0161-2. 13, 27, 96, 122, 150, 237, 241, 318

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995. 161, 163

J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 896–899, 2018. DOI: 10.1145/3238147.3240480. 58, 182, 190

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0-7167-1044-7. 92, 121

K. Gautam, S. VenkataKeerthy, and R. Upadrasta. Cofo: Codeforces dataset for program classification, recognition and tagging, 2025. 237, 239

GeeksforGeeks. C/C++/Java programs. https://www.geeksforgeeks.org, 2003. Accessed 2019-08-15. 52

Ghidra. Ghidra: Software reverse engineering framework. https://ghidra-sre.org/, n.d. 205, 229

F. Giannesini, H. Kanoui, R. Pasero, and M. van Caneghem. *Prolog*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986. ISBN 0201129116. 180

A. V. Gorchakov, L. A. Demidova, and P. N. Sovietov. Analysis of program representations based on abstract syntax trees and higher-order markov chains for source code classification task. *Future Internet*, 15(9), 2023. ISSN 1999-5903. DOI: 10.3390/fi15090314. URL https://www.mdpi.com/1999-5903/15/9/314. 69

D. Grewe, Z. Wang, and M. F. P. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization*, CGO'13, pages 22:1–22:10, 2013. DOI: 10.1109/CGO.2013.6494993. URL https://doi.org/10.1109/CGO.2013.6494993. 12, 13, 25, 96, 102, 104, 105, 107, 119, 318

T. Grosser, A. Groesslinger, and C. Lengauer. Polly performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04): 1250010, 2012. DOI: 10.1142/S0129626412500107. URL https://doi.org/10.1142/S0129626412500107. 92

gRPC. gRPC Remote Procedure Calls. https://grpc.io, 2016. [Online; accessed 29-Aug-2022]. 137

D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang, and M. Zhou. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://openreview.net/forum?id=jLoC4ez43PZ. 181, 259

D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin. UniXcoder: Unified cross-modal pre-training for code representation. In S. Muresan, P. Nakov, and A. Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland, May 2022. Association for Computational Linguistics. DOI: 10.18653/v1/2022.acl-long.499. URL https://aclanthology.org/2022.acl-long.499/. 181, 259

W. Guo, D. Mu, X. Xing, M. Du, and D. Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1787–1804, Santa Clara, CA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/guo. 235

R. Gupta, S. Pal, A. Kanade, and S. Shevade. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 1345–1351. AAAI Press, 2017. DOI: 10.1609/aaai.v31i1.10742. 12, 55, 181, 182

Gzip. GNU Gzip. https://www.gnu.org/software/gzip/, 2024. [version 1.12; Online; accessed 08-May-2024]. 192, 202

S. Hack, D. Grund, and G. Goos. Register allocation for programs in ssa-form. In A. Mycroft and A. Zeller, editors, *Compiler Construction*, pages 247–262, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33051-6. DOI: 10.1007/11688839_20. 133

A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica. Neurovectorizer: end-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '20, page 242255, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370479. DOI: 10.1145/3368826.3377928. URL https://doi.org/10.1145/3368826.3377928. 12, 122, 137, 147, 150, 151, 152

Y. Hakimi, R. Baghdadi, and Y. Challal. Deep learning and classical machine learning for code mapping in heterogeneous platforms. In *2021 International Conference on Networking and Advanced Systems (ICNAS)*, pages 1–6, 2021. DOI: 10.1109/ICNAS53565.2021.9628950. 120

L. Hames and B. Scholz. Nearly optimal register allocation with pbqp. In *Modular Programming Languages (JMLC)*, volume 4228 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2006. DOI: 10.1007/11860990_21. 128

X. Han, S. Cao, X. Lv, Y. Lin, Z. Liu, M. Sun, and J. Li. OpenKE: An open toolkit for knowledge embedding. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 139–144, Brussels, Belgium, Nov. 2018. Association for Computational Linguistics. DOI: 10.18653/v1/D18-2024. URL https://aclanthology.org/D18-2024/. 19, 48, 203

I. U. Haq and J. Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54 (3), apr 2021. ISSN 0360-0300. DOI: 10.1145/3446371. URL https://doi.org/10.1145/3446371. 232, 234

H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *USENIX Security Symposium*. USENIX, 2024. URL https://www.usenix.org/conference/usenixsecurity24/presentation/he-haojie. 229

M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977. ISBN 0444002162. 28, 33, 37

D. Hendrycks and K. Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016. URL http://arxiv.org/abs/1606.08415. 204

J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 163174, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. DOI: 10.1145/3236024.3236085. URL https://doi.org/10.1145/3236024.3236085. 55, 56

J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422448, July 1983. ISSN 0164-0925. DOI: 10.1145/2166.357217. URL https://doi.org/10.1145/2166.357217. 92

Hex-Rays. Ida pro. https://hex-rays.com/ida-pro/, n.d. 88, 205, 229, 234

M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 5461, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134134. DOI: 10.1145/379605.379665. URL https://doi.org/10.1145/379605.379665. 92

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576580, Oct. 1969. ISSN 0001-0782. DOI: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259. 12

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9 (8):1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735. 319

G. M. Hopper. The education of a computer. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, ACM '52, page 243249, New York, NY, USA, 1952. Association for Computing Machinery. ISBN 9781450373623. DOI: 10.1145/609784.609818. URL https://doi.org/10.1145/609784.609818. 91

S. Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):16, Jan. 1997. ISSN 0164-0925. DOI: 10.1145/239912.239913. URL https://doi.org/10.1145/239912.239913. 92

M.-Y. Hsu, F. Hetzelt, D. Gens, M. Maitland, and M. Franz. A highly scalable, hybrid, cross-platform timing analysis framework providing accurate differential throughput estimation via instruction-level tracing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 821831, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. DOI: 10.1145/3611643.3616246. URL https://doi.org/10.1145/3611643.3616246. 258

X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 200210, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357142. DOI: 10.1145/3196321.3196334. URL https://doi.org/10.1145/3196321.3196334. 181

J. Huang, M. M. A. Patwary, and G. F. Diamos. Coloring big graphs with alphagozero. *CoRR*, abs/1902.10162, 2019a. URL http://arxiv.org/abs/1902.10162. 148

Q. Huang, A. Haj-Ali, W. S. Moses, J. Xiang, I. Stoica, K. Asanović, and J. Wawrzynek. AutoPhase: Compiler phase-ordering for HLS with deep reinforcement learning. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–308, 2019b. DOI: 10.1109/FCCM.2019.00049. 150

J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350353, may 1977. ISSN 0001-0782. DOI: 10.1145/359581.359603. URL https://doi.org/10.1145/359581.359603. 183, 228

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In F. Bach and D. Blei, editors, *Proceedings of*

the 32nd International Conference on Machine Learning, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR. URL https://proceedings.mlr.press/v37/ioffe15.html. 204

S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083. Association for Computational Linguistics, 2016. DOI: 10.18653/v1/P16-1195. 12, 181

S. Jain, Y. Andaluri, S. VenkataKeerthy, and R. Upadrasta. POSET-RL: phase ordering for optimizing size and execution time using reinforcement learning. In *International IEEE Symposium on Performance Analysis of Systems and Software, ISPASS 2022, Singapore, May 22-24, 2022*, pages 121–131. IEEE, 2022a. DOI: 10.1109/ISPASS55109.2022.00012. URL https://doi.org/10.1109/ISPASS55109.2022.00012. 6, 8, 100, 122, 137, 147, 150, 151, 152, 153, 165, 259

S. Jain, S. VenkataKeerthy, R. Aggarwal, T. K. Dangeti, D. Das, and R. Upadrasta. Reinforcement learning assisted loop distribution for locality and vectorization. In *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 1–12, 2022b. DOI: 10.1109/LLVM-HPC56686.2022.00006. 6, 8, 98, 100, 150, 151, 152, 153, 165, 166, 169, 257

S. O. Jakob. Greedy Register Allocation in LLVM 3.0. http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html, 2011. 128

C. jam. Google Code Jam. https://codingcompetitions.withgoogle.com/codejam/archive/, 2003. Accessed 2019-05-16. 237

A. T. Jamsaz, Q. I. Mahmud, L. Chen, N. K. Ahmed, and A. Jannesari. PERFOGRAPH: A numerical aware program graph representation for performance optimization and program analysis. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NeurIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/b41907dd4df5c60f86216b73fe0c7465-Abstract-Conference.html. 120

G. Ji, S. He, L. Xu, K. Liu, and J. Zhao. Knowledge graph embedding via dynamic mapping matrix. In *Proceedings of the 53rd Annual Meeting of the Association for*

Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 687–696, Beijing, China, July 2015. Association for Computational Linguistics. DOI: 10.3115/v1/P15-1067. URL https://www.aclweb.org/anthology/P15-1067. 17, 19

W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary function clustering using semantic hashes. In *ICMLA*, page 386391, USA, 2012. IEEE Computer Society. ISBN 9780769549132. DOI: 10.1109/ICMLA.2012.70. URL https://doi.org/10.1109/ICMLA.2012.70. 65

P. J. Joseph, M. T. Jacob, Y. N. Srikant, and K. Vaswani. Statistical and machine learning techniques in compiler design. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, 2007. DOI: 10.1201/9781315219967. 27

P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In B. Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015. DOI: 10.1109/SPRO.2015.10. 59, 183, 213

D. Jurafsky. *Speech & language processing*. Pearson Education, 2000. 74

D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. QT-Opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*, 2018. URL https://arxiv.org/abs/1806.10293. 129, 317

C. Karamitas and A. Kehagias. Efficient features for function matching between binary executables. In R. Oliveto, M. D. Penta, and D. C. Shepherd, editors, *SANER*, pages 335–345. IEEE Computer Society, 2018. DOI: 10.1109/SANER.2018.8330221. URL https://doi.org/10.1109/SANER.2018.8330221. 65

U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., USA, 1st edition, 2009. ISBN 0849328802. 92

D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Trans. Softw. Eng.*, 49(4):16611682, apr 2023. ISSN 0098-5589. DOI: 10.1109/TSE.2022.3187689. URL https://doi.org/10.1109/TSE.2022.3187689. 231

G. Kim, S. Hong, M. Franz, and D. Song. Improving cross-platform binary analysis using representation learning via graph alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 151163, New York, NY, USA, 2022a. Association for Computing Machinery. ISBN 9781450393799. DOI: 10.1145/3533767.3534383. URL https://doi.org/10.1145/3533767.3534383. 231

M. Kim, J. Park, and S. Moon. Irregular register allocation for translation of test-pattern programs. *ACM Trans. Archit. Code Optim.*, 18(1), dec 2021. ISSN 1544-3566. DOI: 10.1145/3427378. URL https://doi.org/10.1145/3427378. 121

M. Kim, J.-K. Park, and S.-M. Moon. Solving pbqp-based register allocation using deep reinforcement learning. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, page 230241. IEEE Press, 2022b. ISBN 9781665405843. DOI: 10.1109/CGO53902.2022.9741272. URL https://doi.org/10.1109/CGO53902.2022.9741272. 123, 147, 148, 150, 151, 152, 153

J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385394, July 1976. ISSN 0001-0782. DOI: 10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252. 180

B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–18, 2021. DOI: 10.1109/TITS.2021.3054625. 129, 317

G. Koch, R. Zemel, and R. Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML Deep Learning Workshop*, 2015. URL https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf. 185, 191, 197, 198

R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, page 11371143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1558603638. URL https://dl.acm.org/doi/10.5555/1643031.1643047. 105

K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):355–383, jul 2003. ISSN 1084-4309. DOI: 10.1145/785411.785416. 121

R. Kundel, L. Nobach, J. Blendin, W. Maas, A. Zimber, H. Kolbe, G. Schyguda, V. Gurevich, R. Hark, B. Koldehofe, and R. Steinmetz. Openbng: Central office network functions on programmable data plane hardware. *Int. J. Netw. Manag.*, 31(1), jan 2021. ISSN 1099-1190. DOI: 10.1002/nem.2134. URL https://doi.org/10.1002/nem.2134. 245

W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4): 323337, Dec. 1992. ISSN 1057-4514. DOI: 10.1145/161494.161501. URL https://doi.org/10.1145/161494.161501. 92

T. László and Á. Kiss. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009. URL https://www.inf.u-szeged.hu/~akiss/pub/fulltext/laszlo2009obfuscating.pdf. 215

C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society. ISBN 0769521029. DOI: 10.1109/CGO.2004.1281665. 19, 24, 26, 34, 88, 92, 123, 150

C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. DOI: 10.1109/CGO51591.2021.9370308. 92, 154, 173

L. Li, K. Jamieson, A. Rostamizadeh, K. Gonina, M. Hardt, B. Recht, and A. Talwalkar. Massively parallel hyperparameter tuning, 2018. URL https://openreview.net/forum?id=S1Y7OOlRZ. 204

X. Li, Y. Qu, and H. Yin. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 32363251, New York, NY,

USA, 2021. Association for Computing Machinery. ISBN 9781450384544. DOI: 10.1145/3460120.3484587. URL https://doi.org/10.1145/3460120.3484587. 60, 65, 88, 229, 230, 231, 232, 235

Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated graph sequence neural networks. In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05493. 166

Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli. Graph matching networks for learning the similarity of graph structured objects. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3835–3845. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/li19d.html. 235

Y. Li, D. Yuan, T. Zhang, H. Cai, D. Lo, C. Gao, X. Luo, and H. Jiang. Meta-learning for multi-family android malware classification. *ACM Trans. Softw. Eng. Methodol.*, 33(7), Aug. 2024. ISSN 1049-331X. DOI: 10.1145/3664806. URL https://doi.org/10.1145/3664806. 59

E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica. RLlib: Abstractions for distributed reinforcement learning. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3053–3062. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/liang18b.html. 165

R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *CoRR*, abs/1807.05118, 2018. URL http://arxiv.org/abs/1807.05118. 170, 204

Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI15, page 21812187. AAAI Press, 2015. ISBN 0262511290. DOI: 10.1609/aaai.v29i1.9491. 17

Linux-Foundation. Performance Counters for Linux. https://perf.wiki.kernel.org/index.php/Main_Page, n.d. [Online; accessed 29-Aug-2022]. 140

O. (Linux Foundation). ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx, 2017. [Online; accessed 11-Mar-2023]. 160

S. Liu, M. Dibaei, Y. Tai, C. Chen, J. Zhang, and Y. Xiang. Cyber vulnerability intelligence for internet of things binary. *IEEE Transactions on Industrial Informatics*, 16(3):2154–2163, 2020. DOI: 10.1109/TII.2019.2942800. 58, 182, 190

Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL http://arxiv.org/abs/1907.11692. 231

LLVM. LLVM Language Reference. https://llvm.org/docs/LangRef.html, 2018a. Accessed 2019-08-20. 19, 26, 55, 205

LLVM. LLVM Block Frequency Terminology. https://llvm.org/docs/BlockFrequencyTerminology.html, 2018b. Accessed 2019-08-09. 41

LLVM. LLVM BranchProbabilityInfo. https://llvm.org/doxygen/classllvm_1_1BranchProbabilityInfo.html, 2018c. Accessed 2019-08-09. 41

LLVM. LLVM Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html, n.d. [Online; accessed 29-Aug-2022]. 135

LLVM-Org. LLVM Test Suite. https://github.com/llvm/llvm-test-suite, n.d. Accessed 2021-08-25. 169

R. C. Lozano, M. Carlsson, F. Drejhammar, and C. Schulte. Constraint-based register allocation and instruction scheduling. In *Principles and Practice of Constraint Programming (CP)*, volume 7514 of *Lecture Notes in Computer Science*, pages 750–766. Springer, 2012. DOI: 10.1007/978-3-642-33558-7_54. 121

lua. lua. https://www.lua.org/source/, 2024. [version 5.4.4; Online; accessed 08-May-2024]. 192, 202

S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra. Aroma: Code recommendation via structural code search. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019. DOI: 10.1145/3360578. URL https://doi.org/10.1145/3360578. 12

Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary

code search. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL https://www.ndss-symposium.org/ndss-paper/vulhawk-cross-architecture-vulnerability-detection-with-entropy-based-binary-code-sear 60, 88, 230, 231, 232, 234, 235

A. Magni, C. Dubach, and M. F. P. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC'13, pages 11:1–11:11, 2013. DOI: 10.1145/2503210.2503268. URL https://doi.org/10.1145/2503210.2503268. 112

A. Magni, C. Dubach, and M. O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 455–466. ACM, 2014. DOI: 10.1145/2628071.2628087. 12, 13, 25, 96, 102, 112, 113, 114, 118, 119

Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *CoRR*, abs/1603.09320, 2016. URL http://arxiv.org/abs/1603.09320. 186

R. Mammadli, A. Jannesari, and F. Wolf. Static neural compiler optimization via deep reinforcement learning. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, Workshop on the LLVM Compiler Infrastructure in HPC, pages 1–11, 11 2020. DOI: 10.1109/LLVMHPCHiPar51896.2020.00006. 147

A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli. 59, 61, 234

P. A. Martínez, J. Woodruff, J. Armengol-Estapé, G. Bernabé, J. M. García, and M. F. P. OBoyle. Matching linear algebra and tensor code to specialized hardware accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC 2023, page 8597, New York, NY, USA, 2023. Association for

Computing Machinery. ISBN 9798400700880. DOI: 10.1145/3578360.3580262. URL https://doi.org/10.1145/3578360.3580262. 58, 190, 245

J. Martinez-Gil. Source code clone detection using unsupervised similarity measures. In *Software Quality as a Foundation for Security (SWQD)*, volume 505 of *Lecture Notes in Business Information Processing*, pages 21–37. Springer, 2024. DOI: 10.1007/978-3-031-56281-5_2. 182

L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni. SAFE: Self-attentive function embeddings for binary similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, volume 11543 of *Lecture Notes in Computer Science*, pages 309–329. Springer, 2019. DOI: 10.1007/978-3-030-22038-9_15. 59, 62, 88, 187, 191, 192, 205, 229, 230, 231, 232, 233, 235

Mbed-TLS. SSL Library mbed TLS/PolarSSL. https://tls.mbed.org/, 2009. [Online; accessed 16-March-2022]. 246

C. Mendis, A. Renda, D. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515. PMLR, 09–15 Jun 2019a. URL https://proceedings.mlr.press/v97/mendis19a.html. 122, 147, 258

C. Mendis, C. Yang, Y. Pu, S. Amarasinghe, and M. Carbin. Compiler auto-vectorization with imitation learning. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, volume 32 of *NeurIPS'19*, 2019b. URL https://proceedings.neurips.cc/paper/2019/file/d1d5923fc822531bbfd9d87d4760914b-Paper.pdf. 12, 136, 150, 152, 319

T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013a. URL https://arxiv.org/abs/1301.3781. 13, 17, 106, 231

T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume*

2, NIPS'13, pages 3111–3119, 2013b. URL http://dl.acm.org/citation.cfm?id=2999792.2999959. 56

J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu. Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. *IEEE Transactions on Reliability*, 65(4):1647–1664, 2016. DOI: 10.1109/TR.2016.2570554. 58, 190

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/mniha16.html. 170

F. L. Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 144152, New York, NY, USA, 1973. Association for Computing Machinery. ISBN 9781450373494. DOI: 10.1145/512927.512941. URL https://doi.org/10.1145/512927.512941. 196

W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko. Polygeist: Raising c to polyhedral mlir. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–59, 2021. DOI: 10.1109/PACT52795.2021.00011. 173

L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, pages 1287–1293. AAAI Press, 2016. DOI: 10.1609/aaai.v30i1.10139. 12, 55, 236, 237, 239, 246, 247

S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4. 27, 28, 33, 37, 75, 92

N. Möller et al. Nettle: a low-level cryptographic library. https://www.lysator.liu.se/~nisse/nettle/nettle.html, 2001. [Online; accessed 22-June-2022]. 246

S. G. Nagarakatte and R. Govindarajan. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 ILP formulation. In *International*

Conference on Compiler Construction (CC), pages 126–140, 2007. DOI: 10.1007/978-3-540-71229-9_9. 121

N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89100, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. DOI: 10.1145/1250734.1250746. URL https://doi.org/10.1145/1250734.1250746. 16, 20, 62

Netronome. Netronome Agilio SmartNICs. https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf, 2017. [Online; accessed 22-June-2022]. 245

Nvidia. Nvidia Bluefield Data Processing Units. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2022. [Online; accessed 22-June-2022]. 245

NVIDIA. NVIDIA OpenCL SDK. https://developer.download.nvidia.com/compute/DevZone/OpenCL/html_x64/OpenCL_Basic_Topics.html, n.d. [Online; accessed 18-May-2020]. 104, 113

R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4206-3. DOI: 10.1145/2908812.2908918. URL http://doi.acm.org/10.1145/2908812.2908918. 246

OpenSSL. OpenSSL. https://www.openssl.org/source/, 2024. [version 3.0.2; Online; accessed 08-May-2024]. 246

M. Panchenko, R. Auler, B. Nell, and G. Ottoni. BOLT: A practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 2–14. IEEE Press, 2019. ISBN 9781728114361. DOI: 10.1109/CGO.2019.8661201. 58, 190

S. Panda, Y. Feng, S. G. Kulkarni, K. K. Ramakrishnan, N. Duffield, and L. N. Bhuyan. Smartwatch: Accurate traffic analysis and flow-state tracking for intrusion prevention

using smartnics. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 6075, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390989. DOI: 10.1145/3485983.3494861. URL https://doi.org/10.1145/3485983.3494861. 245

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. DOI: 10.5555/3454287.3455008. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf. 165

H. Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web: Interoperability, Usability, Applicability*, 8(3):489–508, 2017. DOI: 10.3233/SW-160218. 17

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011. URL http://jmlr.org/papers/v12/pedregosa11a.html. 208

K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020. URL https://arxiv.org/abs/2012.08680. 59, 62, 191, 229, 230, 231, 232, 235

D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu. How could neural networks understand programs? In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8476–8486. PMLR, 2021. URL https://proceedings.mlr.press/v139/peng21b.html. 60, 229, 230, 232, 235

J. Pennington, R. Socher, and C. D. Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. DOI: 10.3115/v1/D14-1162. 17

J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724, 2015. DOI: 10.1109/SP.2015.49. 65, 191, 230, 231

S. N. Pinku, D. Mondal, and C. K. Roy. On the use of deep learning models for semantic clone detection. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 512–524, 2024. DOI: 10.1109/ICSME58944.2024.00053. 182

M. Poletto and V. Sarkar. Linear scan register allocation. *ACM TOPLAS*, 21(5): 895913, sep 1999. ISSN 0164-0925. DOI: 10.1145/330249.330250. URL https://doi.org/10.1145/330249.330250. 92, 128

L.-N. Pouchet, C. Bastoul, and U. Bondhugula. PoCC: the polyhedral compiler collection. https://web.cs.ucla.edu/~pouchet/software/pocc, 2010. Accessed 2023-10-25. 173

L.-N. Pouchet, T. Yuki, et al. Polybench 4.2 benchmarks, 2018. http://sourceforge.net/projects/polybench/. 104, 142

M. Pradel and K. Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25, Oct. 2018. ISSN 2475-1421. DOI: 10.1145/3276517. URL http://doi.acm.org/10.1145/3276517. 56

Protobuf. Protocol Buffers. https://developers.google.com/protocol-buffers, n.d. [Online; accessed 29-Aug-2022]. 137, 159

PuTTY. PuTTY. https://www.chiark.greenend.org.uk/~sgtatham/putty/, 2024. [version 0.76; Online; accessed 08-May-2024]. 192, 202

A. Qasem, M. Debbabi, B. Lebel, and M. Kassouf. Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, pages 443–456, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700989. DOI: 10.1145/3579856.3582818. URL https://doi.org/10.1145/3579856.3582818. 187, 191, 192, 205, 229, 230, 231, 232, 233, 235

Y. Qiu, J. Xing, K.-F. Hsu, Q. Kang, M. Liu, S. Narayana, and A. Chen. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 772787, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. DOI: 10.1145/3477132.3483583. URL https://doi.org/10.1145/3477132.3483583. 236, 237, 245

F. M. Quintão Pereira and J. Palsberg. Register allocation by puzzle solving. *SIGPLAN Not.*, 43(6):216226, jun 2008. ISSN 0362-1340. DOI: 10.1145/1379022.1375609. URL https://doi.org/10.1145/1379022.1375609. 121, 147

M. Rabinovich, M. Stern, and D. Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017. URL https://arxiv.org/abs/1704.07535. 12

G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):14671471, Sept. 1994. ISSN 0164-0925. DOI: 10.1145/186025.186041. URL https://doi.org/10.1145/186025.186041. 92

F. Rastello and F. B. Tichadou. *SSA-based Compiler Design*. Springer Nature, 2022. 63

D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013. ISSN 0950-5849. DOI: 10.1016/j.infsof.2013.01.008. 182

V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3300-9. DOI: 10.1145/2676726.2677009. 55, 181

K. Redmond, L. Luo, and Q. Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis, 2018. 230, 232

H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. ISSN 00029947. URL http://www.jstor.org/stable/1990888. 11, 12, 180, 183, 191, 228, 236

R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.44. URL https://doi.org/10.1109/ICSE.2017.44. 56

N. Rosenblum, X. Zhu, and B. P. Miller. Who wrote this code? identifying the authors of program binaries. In *Computer Security – ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, pages 172–189. Springer, 2011. DOI: 10.1007/978-3-642-23822-2_10. 12

B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code Llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950. 54, 181, 259

A. Salem, S. Banescu, and A. Pretschner. Maat: Automatically analyzing virustotal for accurate labeling and effective malware detection. *ACM Trans. Priv. Secur.*, 24(4), July 2021. ISSN 2471-2566. DOI: 10.1145/3465361. URL https://doi.org/10.1145/3465361. 59

F. Schroff, D. Kalenichenko, and J. Philbin. FaceNet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 815–823, 2015. URL https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Schroff_FaceNet_A_Unified_2015_CVPR_paper.html. 185, 204

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017. 129, 170, 317

S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148, 2011. DOI: 10.1109/IISWC.2011.6114174. 104

R. Shah, V. Kumar, M. Vutukuru, and P. Kulkarni. Turboepc: Leveraging dataplane programmability to accelerate the mobile packet core. In *Proceedings of the Symposium on SDN Research*, SOSR '20, page 8395, New York, NY, USA, 2020. Association

for Computing Machinery. ISBN 9781450371018. DOI: 10.1145/3373360.3380839. URL https://doi.org/10.1145/3373360.3380839. 245

N. Shalev and N. Partush. Binary similarity detection using machine learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, PLAS '18, page 4247, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359931. DOI: 10.1145/3264820.3264821. URL https://doi.org/10.1145/3264820.3264821. 233, 235

Y. Shin and H. Sung. Hybrid register allocation with spill cost and pattern guided optimization. In *Languages and Compilers for Parallel Computing: 34th International Workshop, LCPC 2021, Newark, DE, USA, October 1314, 2021, Revised Selected Papers*, page 3349, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-99371-9. DOI: 10.1007/978-3-030-99372-6_3. URL https://doi.org/10.1007/978-3-030-99372-6_3. 147

Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice – automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. URL https://www.ndss-symposium.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware. 74

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017. DOI: 10.1038/nature24270. 129, 317

D. Simon, J. Cavazos, C. Wimmer, and S. Kulkarni. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization*, CGO'13, page 112, 2013. ISBN 9781467355247. DOI: 10.1109/CGO.2013.6495004. URL https://doi.org/10.1109/CGO.2013.6495004. 12, 122, 150

J. S. Sraw and K. Kumar. Using static and dynamic malware features to perform malware ascription, 2021. URL https://arxiv.org/abs/2112.02639. 59

D. Stenberg. cURL. https://curl.se/, 2024. [version 7.83.0; Online; accessed 08-May-2024]. 192, 202

M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the 2005 International Symposium on Code Generation and Optimization*, CGO'05, pages 123–134, March 2005. DOI: 10.1109/CGO.2005.29. 12, 94, 150

J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Center for Reliable and High-Performance Computing, 2012. URL http://impact.crhc.illinois.edu/Shared/Report/impact-12-01.parboil.pdf. 104, 113

Y. Sui, X. Cheng, G. Zhang, and H. Wang. Flow2vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov 2020. DOI: 10.1145/3428301. URL https://doi.org/10.1145/3428301. 147

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press. URL http://dl.acm.org/citation.cfm?id=2969033.2969173. 13

A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan. Intellicode compose: code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 14331443, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. DOI: 10.1145/3368089.3417058. URL https://doi.org/10.1145/3368089.3417058. 181, 260

O. Sykora, P. M. Phothilimthana, C. Mendis, and A. Yazdanbakhsh. Granite: A graph neural network model for basic block throughput estimation, 2022. URL https://arxiv.org/abs/2210.03894. 122

J. Taneja, A. Laird, C. Yan, M. Musuvathi, and S. K. Lahiri. Llm-vectorizer: Llm-based verified loop vectorizer. In *Proceedings of the 23rd ACM/IEEE International*

Symposium on Code Generation and Optimization, CGO '25, page 137149, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712753. DOI: 10.1145/3696443.3708929. URL https://doi.org/10.1145/3696443.3708929. 260

W. Tang, D. Chen, and P. Luo. Bcfinder: A lightweight and platform-independent tool to find third-party components in binaries. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 288–297, Los Alamitos, CA, USA, dec 2018. IEEE Computer Society. DOI: 10.1109/APSEC.2018.00043. URL https://doi.ieeecomputersociety.org/10.1109/APSEC.2018.00043. 58, 190

R. Team. *Radare2 Book*. GitHub, 2017. 205, 229

T. I. Team. https://github.com/openxla/iree, 2023a. Accessed 2023-11-13. 173

T. T. Team. https://github.com/openai/triton, 2023b. Accessed 2023-11-13. 173

Q. Tian, C. Lin, Z. Zhao, Q. Li, and C. Shen. Collapse-aware triplet decoupling for adversarially robust image retrieval. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 48139–48153. PMLR, 2024. URL https://proceedings.mlr.press/v235/tian24a.html. 204

M. Trofin, Y. Qian, E. Brevdo, and D. Li. RFC: MLGO Regalloc: learned eviction policy for regalloc. https://lists.llvm.org/pipermail/llvm-dev/2021-November/153639.html, https://lists.llvm.org/pipermail/llvm-dev/2021-November/153639.html, 2021a. [Online; accessed 08-May-2022]. 150, 152, 165

M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li. MLGO: a machine learning guided compiler optimizations framework. *CoRR*, abs/2101.04808, 2021b. URL https://arxiv.org/abs/2101.04808. 98, 146, 148, 150, 151, 152, 153, 165, 168, 255, 257

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936. DOI: 10.1112/plms/s2-42.1.230. 1

A. M. Turing. *Computing machinery and intelligence*. Springer, 1950. 1

J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):3142, jan 1976. ISSN 0004-5411. DOI: 10.1145/321921.321925. URL https://doi.org/10.1145/321921.321925. 65

L. van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008. URL http://jmlr.org/papers/v9/vandermaaten08a.html. 83

H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, pages 2094–2100. AAAI Press, 2016. DOI: 10.1609/aaai.v30i1.10295. 166

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. 13, 231, 319

M. Vechev and E. Yahav. Programming with "big code". *Found. Trends Program. Lang.*, 3(4):231–284, Dec. 2016. ISSN 2325-1107. DOI: 10.1561/2500000028. URL https://doi.org/10.1561/2500000028. 11, 56

Vector 35 Inc. Binary Ninja. https://binary.ninja/, 2024. [Online; accessed 20-Jan-2026]. 234

S. VenkataKeerthy. RFC: Enhancing MLGO Inlining with IR2Vec Embeddings. https://discourse.llvm.org/t/rfc-enhancing-mlgo-inlining-with-ir2vec-embeddings/86250, 2025a. [Online; accessed 20-Jan-2026]. Video: https://www.youtube.com/watch?v=ywe1-lI9H4k. 5, 251, 255

S. VenkataKeerthy. llvm-ir2vec: A Tool for IR2Vec Embedding Generation and Vocabulary Training. https://llvm.org/docs/CommandGuide/llvm-ir2vec.html, 2025b. [Online; accessed 20-Jan-2026]. 5, 251

S. VenkataKeerthy and S. Jain. ML-Compiler-Bridge: The Next 700 ML-Enabled Compiler Optimizations, Jan. 2024. URL https://doi.org/10.5281/zenodo.10574579. 177

S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant. IR2Vec: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.*, 17(4), Dec. 2020. ISSN 1544-3566. DOI: 10.1145/3418463. URL https://doi.org/10.1145/3418463. 7, 228

S. VenkataKeerthy, Y. Andaluri, S. Dey, R. Shah, P. Tammana, and R. Upadrasta. Packet processing algorithm identification using program embeddings. In *Proceedings of the 6th Asia-Pacific Workshop on Networking*, APNet '22, page 7682, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9781450397483. DOI: 10.1145/3542637.3542649. URL https://doi.org/10.1145/3542637.3542649. 7, 190, 257

S. VenkataKeerthy, S. Jain, A. Kundu, R. Aggarwal, A. Cohen, and R. Upadrasta. RL4ReAl: Reinforcement Learning for Register Allocation. In *CC 2023*, page 133144, New York, NY, USA, 2023b. Association for Computing Machinery. ISBN 9798400700880. DOI: 10.1145/3578360.3580273. URL https://doi.org/10.1145/3578360.3580273. 7

S. VenkataKeerthy, S. Jain, A. Kundu, R. Aggarwal, A. Cohen, and R. Upadrasta. RL4ReAl: Reinforcement Learning for Register Allocation, Jan. 2023c. URL https://doi.org/10.5281/zenodo.7575072. 149

S. VenkataKeerthy, N. Shah, A. Kundu, S. Jain, and R. Upadrasta. GeMS: Towards Generating Millions of SCoPs (Presentation-only). In *13th International Workshop on Polyhedral Compilation Techniques (IMPACT 2023, in conjunction with HiPEAC 2023)*, 2023d. URL https://www.impact-workshop.org/impact2023/slides/slides8.pdf. 8, 257

S. VenkataKeerthy, S. Jain, U. Kalvakuntla, P. S. Gorantla, R. S. Chitale, E. Brevdo, A. Cohen, M. Trofin, and R. Upadrasta. The next 700 ml-enabled compiler optimizations. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, CC 2024, page 238249, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705076. DOI: 10.1145/3640537.3641580. URL https://doi.org/10.1145/3640537.3641580. 7

S. VenkataKeerthy, S. Banerjee, S. Dey, Y. Andaluri, R. PS, S. Kalyanasundaram, F. M. Q. a. Pereira, and R. Upadrasta. Vexir2vec: An architecture-neutral embedding framework for binary similarity. *ACM Trans. Softw. Eng. Methodol.*, Mar. 2025. ISSN

1049-331X. DOI: 10.1145/3721481. URL https://doi.org/10.1145/3721481. Just Accepted. 7

O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems*, volume 29, pages 3637–3645. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/90e1357833654983612fb05e3ec9148c-Paper.pdf. 186

V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC 08. IEEE Press, 2008. ISBN 9781424428359. DOI: 10.1109/SC.2008.5222004. 112

F. Wang and Y. Shoshitaishvili. Angr - the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9, Los Alamitos, CA, USA, sep 2017. IEEE Computer Society. DOI: 10.1109/SecDev.2017.14. URL https://doi.ieeecomputersociety.org/10.1109/SecDev.2017.14. 60, 62, 229

H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 113, New York, NY, USA, 2022a. Association for Computing Machinery. ISBN 9781450393799. DOI: 10.1145/3533767.3534367. URL https://doi.org/10.1145/3533767.3534367. 60, 229, 230, 232, 235

H. Wang, Z. Tang, C. Zhang, J. Zhao, C. Cummins, H. Leather, and Z. Wang. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 129143, New York, NY, USA, 2022b. Association for Computing Machinery. ISBN 9781450391832. DOI: 10.1145/3497776.3517769. URL https://doi.org/10.1145/3497776.3517769. 151, 176

H. Wang, P. Ma, S. Wang, Q. Tang, S. Nie, and S. Wu. sem2vec: Semantics-aware assembly tracelet embedding. *ACM Trans. Softw. Eng. Methodol.*, 32(4), may 2023a. ISSN 1049-331X. DOI: 10.1145/3569933. URL https://doi.org/10.1145/3569933. 59, 60, 88, 183, 191, 229, 230, 231, 232, 233, 235

K. Wang, R. Singh, and Z. Su. Dynamic neural program embeddings for program repair. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. URL https://openreview.net/forum?id=BJuWrGW0Z. 56

Q. Wang, Z. Mao, B. Wang, and L. Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, Dec 2017. ISSN 1041-4347. DOI: 10.1109/TKDE.2017.2754499. 17, 61

S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 708–719, New York, NY, USA, 2016a. ACM. ISBN 978-1-4503-3845-5. DOI: 10.1145/2970276.2970341. URL http://doi.acm.org/10.1145/2970276.2970341. 12, 182

X. Wang, H. Zhao, and J. Zhu. Grpc: A communication cooperation mechanism in distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(3):7586, jul 1993. ISSN 0163-5980. DOI: 10.1145/155870.155881. URL https://doi.org/10.1145/155870.155881. 159

Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi. CodeT5+: Open code large language models for code understanding and generation. In H. Bouamor, J. Pino, and K. Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore, Dec. 2023b. Association for Computational Linguistics. DOI: 10.18653/v1/2023.emnlp-main.68. URL https://aclanthology.org/2023.emnlp-main.68/. 54, 181, 259

Z. Wang and M. OBoyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018. DOI: 10.1109/JPROC.2018.2817118. 94, 150

Z. Wang, K. Pierce, and S. McFarling. BMAT - a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000. URL https://jilp.org/vol2/v2paper2.pdf. 58, 65, 183, 190, 191, 229

Z. Wang, J. Zhang, J. Feng, and Z. Chen. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the AAAI Conference on Artificial Intelligence*,

volume 28, pages 1112–1119. AAAI Press, 2014. DOI: 10.1609/aaai.v28i1.8870. URL https://ojs.aaai.org/index.php/AAAI/article/view/8870. 17

Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003. PMLR, 2016b. URL https://proceedings.mlr.press/v48/wangf16.html. 97, 318

M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4): 352–357, 1984. DOI: 10.1109/TSE.1984.5010248. 88

S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 2(1-3):37–52, 1987. DOI: 10.1016/0169-7439(87)80084-9. 49

WolfCrypt. WolfCrypt Embedded Crypto Engine. https://www.wolfssl.com/products/wolfcrypt-2/, 2006. [Online; accessed 22-June-2022]. 246

J. Woodruff, J. Armengol-Estapé, S. Ainsworth, and M. F. P. O'Boyle. Bind the gap: compiling real software to hardware fft accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 687702, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. DOI: 10.1145/3519939.3523439. URL https://doi.org/10.1145/3519939.3523439. 245, 248

L. Wu, P. Cui, J. Pei, and L. Zhao. *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer Singapore, Singapore, 2022. 231

X. Wu, P. Paramasivam, and V. Taylor. Autotuning apache tvm-based scientific applications using bayesian optimization. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 2935, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707858. DOI: 10.1145/3624062.3626079. URL https://doi.org/10.1145/3624062.3626079. 259

Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MI-

CRO 27, pages 1–11, New York, NY, USA, 1994. ACM. ISBN 0-89791-707-3. DOI: 10.1145/192724.192725. URL http://doi.acm.org/10.1145/192724.192725. 27

T. Xavier, G. Oliveira, E. Lima, and A. Silva. A detailed analysis of the llvm's register allocators. In *2012 31st International Conference of the Chilean Computer Science Society*, pages 190–198, 2012. DOI: 10.1109/SCCC.2012.29. 128

Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426, May 2017. DOI: 10.1109/ICSE.2017.45. 56

X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 363376, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. DOI: 10.1145/3133956.3134018. URL https://doi.org/10.1145/3133956.3134018. 59, 65, 191, 229, 230, 231, 234, 235

H. Xuan, A. Stylianou, X. Liu, and R. Pless. Hard negative examples are hard, but useful. In *Computer Vision – ECCV 2020*, volume 12359 of *Lecture Notes in Computer Science*, pages 126–142. Springer, 2020a. DOI: 10.1007/978-3-030-58568-6_8. 204

H. Xuan, A. Stylianou, and R. Pless. Improved embeddings with easy positive triplet mining. In *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 2463–2471, Los Alamitos, CA, USA, mar 2020b. IEEE Computer Society. DOI: 10.1109/WACV45572.2020.9093432. URL https://doi.ieeecomputersociety.org/10.1109/WACV45572.2020.9093432. 204

H. Xue, G. Venkataramani, and T. Lan. Clone-hunter: Accelerated bound checks elimination via binary code clone detection. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, page 1119, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358347. DOI: 10.1145/3211346.3211347. URL https://doi.org/10.1145/3211346.3211347. 190

J. Yan, L. Tang, J. Li, X. Yang, W. Quan, H. Chen, and Z. Sun. Unisec: A unified security framework with smartnic acceleration in public cloud. In *Proceedings of the*

ACM Turing Celebration Conference - China, ACM TURC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450371582. DOI: 10.1145/3321408.3323087. URL https://doi.org/10.1145/3321408.3323087. 245

J. Yang, C. Fu, X. Liu, H. Yin, and P. Zhou. Codee: A tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering*, 48(07):2224–2244, jul 2022. ISSN 1939-3520. DOI: 10.1109/TSE.2021.3056139. 205, 229, 230, 231, 233, 235

S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi. Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 224–236, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society. DOI: 10.1109/DSN48987.2021.00036. URL https://doi.ieeecomputersociety.org/10.1109/DSN48987.2021.00036. 59, 62, 191, 229, 230, 233, 234, 235

S. Yang, C. Dong, Y. Xiao, Y. Cheng, Z. Shi, Z. Li, and L. Sun. Asteria-pro: Enhancing deep-learning based binary code similarity detection by incorporating domain knowledge, 2023. 229, 230, 235

F. Ye, S. Zhou, A. Venkat, R. Marcus, N. Tatbul, J. J. Tithi, N. Hasabnis, P. Petersen, T. Mattson, T. Kraska, P. Dubey, V. Sarkar, and J. Gottschlich. MISIM: A neural code semantics similarity system using the context-aware semantics structure, 2021. URL https://arxiv.org/abs/2006.05265. 236, 239

J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems*, volume 27, 2014. URL https://proceedings.neurips.cc/paper/2014/hash/375c71349b295fbe2dcdca9206f20a06-Abstract.html. 123

Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1145–1152, Apr. 2020. DOI: 10.1609/aaai.v34i01.5466. URL https://ojs.aaai.org/index.php/AAAI/article/view/5466. 59, 191, 229, 230, 231, 232, 235

R. Zhang, Z. Xu, C. Ma, C. Yu, W.-W. Tu, S. Huang, D. Ye, W. Ding, Y. Yang, and Y. Wang. A survey on self-play methods in reinforcement learning. *arXiv preprint arXiv:2408.01072*, 2024. URL https://arxiv.org/abs/2408.01072. 258

X. Zhang, W. Sun, J. Pang, F. Liu, and Z. Ma. Similarity metric method for binary basic blocks of cross-instruction set architecture. In *Proceedings 2020 Workshop on Binary Analysis Research*, 2020. URL https://www.ndss-symposium.org/wp-content/uploads/2020/04/bar2020-23002.pdf. 231

W. Zhu, H. Wang, Y. Zhou, J. Wang, Z. Sha, Z. Gao, and C. Zhang. kTrans: Knowledge-aware transformer for binary code embedding. *arXiv preprint arXiv:2308.12659*, 2023. URL https://arxiv.org/abs/2308.12659. 60, 229, 230, 232, 235

F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL https://www.ndss-symposium.org/ndss-paper/neural-machine-translation-inspired-binary-code-similarity-comparison-beyond-function-
59, 61, 86, 88, 191, 229, 230, 232, 233, 235

Zynamics. Bindiff7, 2024. URL https://www.zynamics.com/bindiff.html. [Online; accessed 17-June-2024]. 187, 191, 192, 205, 230, 231, 235

Appendices

Appendix A

IR2Vec: Seed Embeddings

A.1 List of Entities

The list of learned entities by IR2Vec is shown in Table A.1.

Table A.1: List of Entities

Opcodes					
add	extractelement	fptoui	landingpad	sdiv	switch
alloca	extractvalue	fptrunc	load	select	trunc
and	fadd	fsub	lshr	sext	udiv
ashr	fcmp	getelementptr	mul	shl	uitofp
atomicrmw	fdiv	icmp	or	shufflevector	unreachable
bitcast	fence	insertelement	phi	sitofp	urem
br	fmul	insertvalue	ptrtoint	srem	xor
call	fpext	inttoptr	resume	store	zext
cmpxchg	fptosi	invoke	ret	sub	

Types	**Arguments**
floatTy	CONST
integerTy	FUNCTION
pointerTy	LABEL
structTy	PTR
vectorTy	VAR
voidTy	

A.2 List of analogies

The list of analogies which were used for experimentation is shown below in Tables A.2, A.3, and A.4.

Table A.2: List of Analogies – Based on Arguments and Inherent Semantic Relations

Based on Arguments	Based on Inherent Semantic Relations
phi : variable :: load : pointer	and : or :: mul : shl
call : function :: ret : constant	shl : lshr :: mul : udiv
call : function :: ret : variable	shl : ashr :: mul : sdiv
load : pointer :: store : variable	call : ret :: switch : label
load : pointer :: store : constant	function : ret :: switch : br
alloca : variable :: cmpxchg : pointer	load : store :: landingpad : invoke
inttoptr : pointer :: ptrtoint : variable	invoke : landingpad :: function : call
ptrtoint : pointer :: inttoptr : variable	inttoptr : ptrtoint :: trunc : zext
pointerty : pointer :: integerTy : variable	inttoptr : ptrtoint :: trunc : sext
pointerty : pointer :: integerTy : constant	

Table A.3: List of Analogies – Based on Operands and Operation Type

Based on Operands	Based on Operation Type
fadd : fsub :: add : sub	fcmp : floatTy :: icmp : integerTy
add : fadd :: sub : fsub	uitofp : integerTy :: fptoui : floatTy
fdiv : sdiv :: fmul : mul	inttoptr : integerTy :: ptrtoint : pointerTy
add : sdiv :: fadd : fdiv	insertvalue : structTy :: insertelement : vectorTy
add : udiv :: fadd : fdiv	phi : integerTy :: fadd : floatTy
trunc : fptrunc :: icmp : fcmp	extractelement : vectorTy :: extractvalue : structTy
urem : srem :: udiv : sdiv	fpext : floatTy :: sext : integerTy
zext : fpext :: trunc : fptrunc	
trunc : sext :: fptrunc : fpext	

A.3 Proof of Unique Solution for Circular Dependencies

In Section 3.5, we established that circular dependencies in Flow-Aware encodings give rise to a system of linear equations of the form $(\mathbb{I} - W_a \cdot \mathbf{A}) \cdot \mathbf{x} = \mathbf{b}$, and stated that the weight hierarchy constraint guarantees a unique solution (Theorem 3.1). Here, we provide the formal proof and generalize the result to arbitrary dependency structures.

Table A.4: List of Analogies – Based on Return Type

Based on Return Type
shufflevector : vectorTy :: getelementptr : structTy
shufflevector : vectorTy :: getelementptr : integerTy
shufflevector : vectorTy :: getelementptr : voidTy
shufflevector : vectorTy :: getelementptr : floatTy
shufflevector : vectorTy :: getelementptr : pointerTy
load : pointerTy :: store : voidTy
trunc : integerTy :: fptrunc : floatTy
ptrtoint : integerTy :: inttoptr : pointerTy
add : integerTy :: fadd : floatTy
fsub : floatTy :: sub : integerTy
fmul : floatTy :: mul : integerTy
fdiv : floatTy :: sdiv : integerTy
fdiv : floatTy :: udiv : integerTy
srem : integerTy :: shufflevector : vectorTy
zext : integerTy :: fpext : floatTy
sext : integerTy :: fpext : floatTy
fptosi : floatTy :: sitofp : integerTy

A.3.1 Circular Dependency Degree

Recall from Definition 3.4 (Section 3.5) that the *maximum circular dependency degree* $d_{\max}$ is the largest number of unknown instruction embeddings that appear among the reaching definitions of any single instruction in the cycle. In the common case, circular dependencies are pairwise ($d_{\max} = 1$). More complex structures ($d_{\max} \geq 2$) can arise when multiple write instructions on different control-flow branches target the same non-promotable memory location within a loop.

A.3.2 Formal Statement and Proof

Theorem A.1 (Sufficient Condition for Unique Solution)**.** *Let $d_{\max}$ be the maximum circular dependency degree (Definition 3.4). If*

$$W_a \cdot d_{\max} < 1 \tag{A.1}$$

then the system of linear equations arising from circular dependencies always has a unique solution.

Proof. As illustrated in Equation 3.4, each circularly-dependent instruction embedding $[\![\mathbf{I}_i]\!]$ satisfies an equation of the form:

$$[\![\mathbf{I}_i]\!] \;=\; \mathbf{b}_i \;+\; W_a \sum_{I_j \in \mathcal{C}_i} [\![\mathbf{I}_j]\!]$$

where $\mathbf{b}_i$ denotes the known terms—the weighted opcode and type embeddings, and the embeddings of non-circular arguments—and $\mathcal{C}_i \subseteq \{I_1, I_2, \ldots, I_n\}$ is the set of instructions whose embeddings appear among the circular reaching definitions of instruction I_i, with $|\mathcal{C}_i| \leq d_{\max}$.

Moving the unknown terms to the left-hand side, each equation becomes:

$$[\![\mathbf{I}_i]\!] \;-\; W_a \sum_{I_j \in \mathcal{C}_i} [\![\mathbf{I}_j]\!] \;=\; \mathbf{b}_i$$

Writing all n such equations together in matrix form gives $(\mathbb{I} - W_a \cdot \mathbf{A}) \cdot \mathbf{x} = \mathbf{b}$, where $\mathbb{I}$ is the $n \times n$ identity matrix, $\mathbf{A}$ is a binary matrix with $A_{ij} = 1$ if and only if $I_j \in \mathcal{C}_i$ (and $A_{ii} = 0$ for all i, since an instruction's embedding equation never contains itself on the right-hand side), $\mathbf{x}$ is the vector of unknown embeddings, and $\mathbf{b}$ is the corresponding vector of known terms. The system has a unique solution if and only if the coefficient matrix $(\mathbb{I} - W_a \cdot \mathbf{A})$ is non-singular (i.e., invertible).

We now show this is the case. Let $\mathbf{M} = W_a \cdot \mathbf{A}$. Consider an arbitrary row i of $\mathbf{M}$ (for any $1 \leq i \leq n$). Because $\mathbf{A}$ is a binary matrix with zero diagonal, the entry $M_{ij} = W_a \cdot A_{ij}$ equals W_a when $I_j \in \mathcal{C}_i$ and 0 otherwise. Hence the number of non-zero entries in row i is exactly $|\mathcal{C}_i|$, each equal to W_a, so

$$\sum_{j=1}^{n} |M_{ij}| \;=\; W_a \cdot |\mathcal{C}_i| \;\leq\; W_a \cdot d_{\max} \;<\; 1 \qquad \text{for every } 1 \leq i \leq n \tag{A.2}$$

where the first inequality follows from $|\mathcal{C}_i| \leq d_{\max}$ by definition and the second from the hypothesis $W_a \cdot d_{\max} < 1$.

The matrix $(\mathbb{I} - \mathbf{M})$ has diagonal entries equal to 1 and off-diagonal entries of absolute value at most W_a. Equation A.2 shows that in every row, the diagonal entry $(= 1)$ strictly exceeds the sum of the absolute values of all off-diagonal entries $(\leq W_a \cdot d_{\max} < 1)$. That is, $(\mathbb{I} - \mathbf{M})$ is *strictly diagonally dominant.*

To show non-singularity, we prove by contradiction that $(\mathbb{I} - \mathbf{M})\,\mathbf{x} = \mathbf{0}$ has no non-trivial solution. Suppose $\mathbf{x} \neq \mathbf{0}$ satisfies $(\mathbb{I} - \mathbf{M})\,\mathbf{x} = \mathbf{0}$, which means $\mathbf{x} = \mathbf{M}\,\mathbf{x}$, i.e., every component satisfies $x_i = \sum_{j=1}^{n} M_{ij}\, x_j$. Among all components, pick the one with the largest absolute value; call it x_k (so $|x_k| \geq |x_j|$ for all j, and $|x_k| > 0$ since $\mathbf{x} \neq \mathbf{0}$). Then:

$$
\begin{aligned}
|x_k| &= \left|\sum_{j=1}^{n} M_{kj}\, x_j\right| && \text{(from } x_k = \textstyle\sum_j M_{kj}\, x_j\text{)} \\
&\leq \sum_{j=1}^{n} |M_{kj}|\, |x_j| && \text{(triangle inequality)} \\
&\leq |x_k| \sum_{j=1}^{n} |M_{kj}| && \text{(since } |x_j| \leq |x_k| \text{ for all } j\text{)} \\
&\leq |x_k| \cdot W_a \cdot d_{\max} && \text{(by Equation A.2)} \\
&< |x_k| && \text{(since } W_a \cdot d_{\max} < 1\text{)}
\end{aligned}
$$

We arrive at $|x_k| < |x_k|$, a contradiction. Therefore, no non-trivial solution exists, $(\mathbb{I} - \mathbf{M})$ is non-singular, and the system has a unique solution.[1] □

Example A.3.1: Proof applied to the running example

For the circular dependency in Equation 3.4, the two unknowns are $[\![\mathbf{I_5}]\!]$ and $[\![\mathbf{I_7}]\!]$ (so $n = 2$). The dependency matrix is $\mathbf{A} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, since I_5 depends on I_7 and vice versa. Each row has exactly one non-zero entry, so $d_{\max} = 1$. With $W_a = 0.2$, we get $W_a \cdot d_{\max} = 0.2 < 1$, and the coefficient matrix $\mathbb{I} - 0.2 \cdot \mathbf{A} = \left[\begin{smallmatrix} 1 & -0.2 \\ -0.2 & 1 \end{smallmatrix}\right]$ has determinant $1 - 0.04 = 0.96 \neq 0$, confirming a unique solution.

Observation A.2 (Simple Cycles and LLVM IR)**.** *In practice, circular dependencies in LLVM IR are almost always pairwise ($d_{\max} = 1$). This is a structural consequence of two properties of LLVM IR:*

1. ***SSA form:*** *Each SSA register has exactly one static definition, so circular dependencies can only arise through memory operations (store → load → store chains writing to the same location).*
2. ***`mem2reg` promotion:*** *The standard `mem2reg` pass promotes stack variables to SSA registers with phi nodes, eliminating most store/load patterns. Only non-promotable memory*

[1] This argument is equivalent to invoking the *Levy–Desplanques theorem*: every strictly diagonally dominant matrix is non-singular.

operations (address-taken variables, globals, heap allocations, array/struct accesses) survive.

For $d_{\max} > 1$ to arise, multiple write instructions on different control-flow branches must target the same non-promotable memory location within a loop—a rare pattern in typical programs. When $d_{\max} = 1$, the condition (Equation A.1) reduces to $W_a < 1$, which is directly guaranteed by the weight hierarchy constraint (Equation 3.2).

Appendix B

VexIR2Vec: Seed Embeddings

B.1 List of Analogies

In Table B.1, we show the complete list of analogies that we use for evaluating the vocabulary $\mathcal{V}_{lookup}$ in Section 4.5 on page 83.

We are able to correctly answer both correct syntactic and semantic analogies. Intrinsic syntactic analogies are like add : addf :: sub : subf, GetI: PutI :: Get : Put, while semantic analogies like shl : mul :: shr : divmod and or : and :: shr : shl.

Table B.1: List of Analogies

List of Analogies	
getI : putI :: get : put	get : load :: put : store
get : put :: load : store	get : put :: load : wrtmp
store : variable :: put : register	get : register :: store : constant
put : register :: load : variable	put : register :: store : constant
orv : or :: xorv : xor	or : orv :: shr : shrnv
or : and :: orv : andv	or : and :: shr : shl
and : or :: add : sub	mul : divmod :: and : or
shr : shl :: shrnv : shlnv	reinterpif : reinterpfi :: convif : convfi
get : register :: geti : constant	put : register :: puti : register
sub : subv :: add : addv	subf : addf :: subfv : addfv
add : addf :: sub : subf	addf : addfv :: subf : subfv

(Continued on next page)

List of Analogies

add : sub :: addf : subf
add : sub :: addfv : subfv
add : addf :: mul : mulf
add : addf :: mul : mull
add : sub :: mul : div
add : sub :: mul :: divmod
addf : subf :: mulf : divf
addv : subv :: addf : subf
addv : subv :: addfv : subfv
mulf : divf :: mulfv : divfv
shl : mul :: shr : divmod
shl : mul :: sar : divmod
shl : mul :: shr : div
shl : mul :: sar : div
add : sub :: mul : shr
add : sub :: mul : sar
add : integer :: addf : float
subf : float :: sub : integer
mul : integer :: mulf : float
divf : float :: divmod : integer
add : addv :: integer : vector
integer : sub :: vector : subv
divfv : divf :: vector : float
subv : subfv :: vector : float
addv : vector :: addfv : float
mulv : vector :: mulfv : float
add : integer :: addfv : vector
add : integer :: addfv : float
sub : subfv :: integer : vector
sub : subfv :: integer : float
mul : integer :: mulfv : float
mul : integer :: mulfv : vector
divmod : integer :: divfv : vector
divmod : integer :: divfv : float
orv : vector :: or : integer
not : integer :: notv : vector
andv : vector :: and : integer
shrnv : vector :: shr : integer
shl : integer :: shrnv : vector
xor : xorv :: integer : vector
or : andv :: integer : vector
not : integer :: negf : float
cmple : cmplt :: cmplefv : cmpltfv
geti : integer :: get : register
ext : integer :: extf : float
extf : float :: extv : vector
ext : integer :: extv : vector
hlextv : ext :: htruncv : htrunc
trunc : integer :: truncv : vector
truncv : vector :: truncf : float
htrunc : trunc :: hlext : ext
dirty : function :: if : store
if : variable :: dirty : function
if : exit :: put : register
store : variable :: put : register
load : variable :: get : register
get : register :: store : variable
put : register :: load : constant
maxv : maxfv :: minv : minfv
maxv : minv :: addv : subv
maxfv : minfv :: addfv : subfv
if : else :: get : put
or : and :: ext : trunc
ext : trunc :: get : put

(Continued on next page)

List of Analogies	
sqrtf : float :: sqrtfv : vector	mulv : vector :: mul : integer

Appendix C

Learning Paradigms for ML-Driven Compiler Optimizations

C.1 Learning Paradigms

Applying ML models to compiler optimizations requires selecting appropriate learning paradigms based on the characteristics of the optimization problem. The most commonly used paradigms are supervised learning and reinforcement learning.

C.1.1 Supervised Learning

Supervised learning involves training a model to map inputs to outputs using a labeled dataset comprising input-output pairs. The model iteratively minimizes the error between its predictions and the actual outputs, effectively learning to generalize from the training data.

In the context of compiler optimizations, supervised learning is used to model problems where labeled datasets consisting of input programs and their corresponding ground truth are available. Supervised learning is preferred in scenarios where:

- The optimization decisions can be determined *a priori* for a given input program.
- The optimization can be modeled as a single decision problem without dependencies on prior predictions.

- The optimization decisions do not involve complex interactions between the decisions.
- The optimization decisions are deterministic and do not have uncertainties or a large exploration space.

For example, predicting performance characteristics of a program (execution time, throughput, etc.) aligns with these criteria and can be modeled using supervised learning. Similarly, given a program and a heterogeneous architecture, predicting the optimal device to offload the computation can be modeled using supervised learning.

C.1.2 Reinforcement Learning

Reinforcement Learning (RL) is a learning paradigm where an *agent* learns to take actions by interacting with an *environment*. The agent learns a policy to determine the best possible action based on its *observation* from the environment. Depending on the quality of the action, the environment provides positive or negative rewards to guide learning. This process repeats until the agent learns an optimal policy that maximizes the cumulative reward. With the evolution of deep learning, methods like PPO [Schulman et al., 2017] that use gradient-based approaches to learn policies have become prevalent.

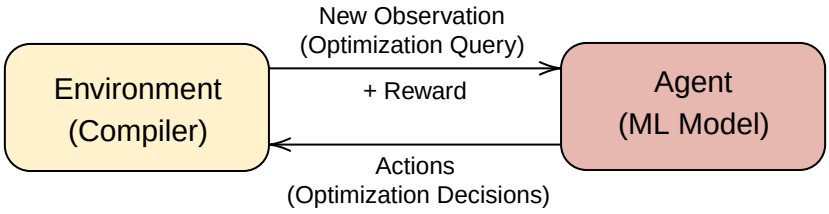


Figure C.1: Reinforcement Learning: Interaction between Agent and Environment

Depending on the problem formulation, there can be single or multiple agents. When the problem involves multiple agents, it is called *Multi-Agent Reinforcement Learning* (MARL). If agents work together toward a common goal, learning is cooperative; if they compete, it is competitive. MARL has achieved success in gaming [Silver et al., 2017], robotics [Kalashnikov et al., 2018], navigation [Almasan et al., 2020], and autonomous driving [Kiran et al., 2021].

In the context of compiler optimizations, the environment is typically the compiler infrastructure, and the agent is the ML model making optimization decisions. Actions

correspond to optimization choices, rewards reflect performance improvements, observations represent input programs, and states correspond to the program state after successive optimization actions.

RL is preferred in scenarios where:

- Deterministic ground truths are difficult or infeasible to establish.
- Optimization requires a sequence of interdependent decisions.
- Decisions involve complex interactions or large exploration spaces.
- Some decisions are constrained by correctness requirements.

Most compiler optimizations involve exploring a large space of decisions or are inherently sequential, making RL a suitable choice [Amarasinghe, 2020]. Additionally, RL frameworks can enforce constraints via *action masks* [Wang et al., 2016b], which restrict the agent to valid actions at each decision point.

For example, determining the optimal vectorization factor for a loop requires exploring a large space of factors, and exhaustive profiling is infeasible. Similarly, register allocation involves sequences of interdependent decisions (allocation order, spilling, splitting) that can be naturally modeled as an RL problem.

C.2 ML Models

Various machine learning models have been employed in compiler optimizations, ranging from classical models to advanced neural network architectures. The choice depends on the nature of the optimization problem, input data type, and available computational resources.

C.2.1 Classical Machine Learning Models

Classical models such as decision trees, random forests, support vector machines (SVMs), k-nearest neighbors (kNN), and gradient boosting classifiers have been extensively used in early ML-driven compiler optimizations [Fursin et al., 2011; Cavazos et al., 2007; Grewe et al., 2013].

These models are suitable when input data can be represented as fixed-size vectors. They are computationally efficient, require less training data, and when feature-based, offer interpretable predictions. However, they are limited in capturing complex relationships and are not suitable for sequential data.

C.2.2 Sequential Models

Sequential models like Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, Gated Recurrent Units (GRUs), and Transformers are suited for problems involving sequential data.

RNNs Effective for short sequences but suffer from vanishing gradients, limiting their ability to capture long-range dependencies.

LSTMs and GRUs Address the vanishing gradient problem and model long sequences effectively [Hochreiter and Schmidhuber, 1997].

Transformers Use self-attention to capture long-range dependencies efficiently [Vaswani et al., 2017].

However, these models (particularly transformers) are resource-intensive, requiring significant compute, memory, and training data.

In compiler optimizations, sequential models are used for modeling instruction sequences [Ben-Nun et al., 2018] rather than fixed-size vector representations.

C.2.3 Graph Neural Networks

Graph Neural Networks (GNNs) operate on graph-structured data. Training involves message passing between nodes, propagating information across the graph. GNNs can annotate nodes and edges with types and properties.

Resource requirements and training times for GNNs grow rapidly for larger or densely connected graphs.

In compiler optimizations, GNNs are widely used [Cummins et al., 2021; Mendis et al., 2019b] when inputs are modeled as graphs (control flow graphs, program dependence graphs, etc.).

Appendix D

Binary Similarity with VexIR2Vec

This chapter contains additional tables and data that can be of interest to readers interested in reproducing the results in this dissertation but that are not essential to understanding its core ideas.

D.1 Description of Dataset

Details and statistics of our dataset for our binary similarity experiments are shown in Table D.1. As it can be seen, the number of functions differs across the architectures due to the impacts of optimizations and the differences in compilers. Coreutils and PuTTY form a significant portion of the dataset. The binary sizes vary from about $9KB$–$7.6MB$.

D.2 Peepholes

In Tables D.2 and D.3, we show the average number of basic blocks (vertices), the average number of edges, and the average length of the longest straight-line path in the Control-Flow Graph. These values are averaged across all functions of all binaries in our dataset (Section 10.4 on page 202). Additionally, we also show the average number of peepholes obtained for different values of k (with c set to 2).

Table D.1: Description of dataset

Projects	Arch	#Functions	#Functions in Test set	GT pairs	#Binaries	Binary sizes
Coreutils	ARM x86	751K 843K	128K 113K	98K	13,320	9.7 KB - 1.3 MB
Diffutils	ARM x86	44K 46K	7K 6K	5K	702	9.7 KB - 1.5 MB
Findutils	ARM x86	61K 71K	5K 4K	4K	690	9.8 KB - 2.2 MB
cURL	ARM x86	9K 5K	9K 5K	5K	60	762.8 KB - 1.1 MB
Lua	ARM x86	40K 41K	40K 41K	23K	120	554.3 KB - 2.3 MB
PuTTY	ARM x86	436K 428K	436K 428K	325K	780	9.2 KB - 7.6 MB
Gzip	ARM x86	4K 4K	4K 4K	3K	60	205 KB - 547.1 KB

There is a decreasing trend in the number of peepholes with the increase in values of k. However, k being the maximum number of basic blocks in a peephole, there is little to no difference in the number of peepholes generated when k matches the number of basic blocks. (Notice the entries corresponding to $k = 36$ to $k = 144$.)

In practice, the number of peepholes generated is typically less than the worst-case defined by Theorem 4.1 on page 72, and tends to a value close to $c|V|/2$ (especially when $k > 1$). Beyond a threshold, k does not appear to impact the number of peepholes generated.

D.3 Cross-Optimization Binary Diffing

This section provides additional results on the cross-optimization binary diffing experiment described in Section 10.5.1 on page 208.

In Table D.4, we show the detailed result of Cross-Optimization binary diffing involving a comparison between O1 and O3 optimization levels. As it can be observed, VexIR2Vec (V2V) achieves the highest precision and recall scores across all configurations.

Table D.2: Peepholes Trend with varying k

Compiler	Avg. block_count	Avg. edge_count	Avg. Longest Path in DAG
x86-clang-8-O0	22.23	28.80	7.57
x86-clang-8-O1	14.82	18.91	6.42
x86-clang-8-O2	25.92	35.39	9.50
x86-clang-8-O3	27.54	38.17	9.47
x86-clang-8-Os	20.16	26.68	7.64
x86-gcc-8-O0	16.78	21.31	6.82
x86-gcc-8-O1	21.51	28.34	7.74
x86-gcc-8-O2	20.63	27.19	7.75
x86-gcc-8-O3	27.57	37.33	10.24
x86-gcc-8-Os	19.21	25.16	7.10

Table D.3: Peepholes Trend with varying k

Compiler	Average number of Peepholes										
	k=1	k=3	k=5	k=7	k=10	k=12	k=36	k=64	k=72	k=100	k=144
x86-clang-8-O0	44.45	26.42	22.98	21.65	20.94	20.73	20.10	20.05	20.03	20.92	20.06
x86-clang-8-O1	29.63	18.59	16.65	15.99	15.66	15.52	15.36	15.43	15.41	15.72	15.42
x86-clang-8-O2	51.84	31.36	27.34	25.96	24.96	24.75	24.25	24.20	24.25	25.10	24.29
x86-clang-8-O3	55.08	33.38	29.24	27.51	26.73	26.44	25.94	25.93	25.98	26.82	26.00
x86-clang-8-Os	40.32	24.64	21.72	20.75	20.18	20.04	19.65	19.73	19.76	20.26	19.84
x86-gcc-8-O0	33.56	20.82	18.37	17.53	17.04	16.92	16.64	16.66	16.65	17.12	16.71
x86-gcc-8-O1	43.03	26.55	23.51	22.42	21.76	21.48	21.27	21.22	21.15	21.83	21.30
x86-gcc-8-O2	41.25	25.36	22.20	21.10	20.51	20.24	20.12	20.02	20.03	20.60	19.99
x86-gcc-8-O3	55.13	33.51	29.13	27.50	26.40	26.22	25.71	25.67	25.72	26.62	25.59
x86-gcc-8-Os	38.42	23.45	20.51	19.58	18.83	18.68	18.44	18.56	18.50	19.01	18.48

Table D.4: Cross-Optimization Binary Diffing - O1 Vs. O3. Missing values indicate timeout.

	Precision						Recall					
	BinDiff	**DBD**	**SAFE**	**OPC**	**BinFinder**	**V2V**	**BinDiff**	**DBD**	**SAFE**	**OPC**	**BinFinder**	**V2V**
						ARM - Clang12						
Coreutils	0.32	0.46	0.28	0.47	0.42	**0.57**	0.20	0.58	0.22	0.62	0.56	**0.76**
cURL	0.44		0.51	0.71	0.68	**0.82**	0.42		0.34	0.82	0.78	**0.94**
Diffutils	0.51	0.45	0.40	0.62	0.55	**0.71**	0.37	0.60	0.29	0.80	0.71	**0.93**
Findutils	0.51	0.52	0.38	0.54	0.51	**0.68**	0.30	0.68	0.31	0.67	0.63	**0.83**
Gzip	0.44	0.34	0.43	0.53	0.50	**0.65**	0.40	0.77	0.30	0.68	0.64	**0.83**
Lua	0.34		0.31	0.24	0.27	**0.47**	0.33		0.20	0.38	0.43	**0.74**
PuTTY	0.29		0.24	0.28	0.29	**0.41**	0.21		0.24	0.38	0.39	**0.54**
						ARM - GCC8						
Coreutils	0.32	0.40	0.34	0.60	0.56	**0.67**	0.22	0.62	0.27	0.69	0.65	**0.77**
cURL	0.39		0.46	0.84	0.71	**0.91**	0.31		0.35	0.89	0.77	**0.97**
Diffutils	0.44	0.48	0.55	0.74	0.73	**0.83**	0.35	0.81	0.39	0.84	0.83	**0.94**
Findutils	0.41	0.43	0.43	0.63	0.61	**0.76**	0.28	0.68	0.36	0.69	0.67	**0.83**
Gzip	0.36	0.44	0.51	0.56	0.73	**0.8**	0.31	0.77	0.36	0.64	0.84	**0.91**
Lua	0.31		0.29	0.35	0.41	**0.59**	0.30		0.22	0.44	0.51	**0.74**
PuTTY	0.28		0.23	0.31	0.45	**0.54**	0.20		0.26	0.35	0.43	**0.6**
						x86 - Clang12						
Coreutils	0.40	0.21	0.32	0.46	0.39	**0.59**	0.36	0.67	0.25	0.61	0.52	**0.79**
cURL	0.60		0.57	0.73	0.69	**0.83**	0.60		0.41	0.84	0.79	**0.95**
Diffutils	0.62	0.25	0.46	0.63	0.51	**0.7**	0.62	0.83	0.33	0.82	0.67	**0.91**
Findutils	0.52	0.24	0.43	0.53	0.51	**0.69**	0.50	0.74	0.35	0.65	0.64	**0.85**
Gzip	0.40	0.17	0.52	0.47	0.48	**0.68**	0.38	**0.93**	0.37	0.60	0.62	0.88
Lua	0.40		0.34	0.24	0.23	**0.48**	0.39		0.24	0.38	0.36	**0.74**
PuTTY	0.34		0.27	0.23	0.27	**0.41**	0.33		0.27	0.30	0.35	**0.53**
						x86 - GCC8						
Coreutils	0.46	0.30	0.33	0.58	0.52	**0.67**	0.43	0.62	0.28	0.66	0.59	**0.76**
cURL	0.48		0.56	0.76	0.75	**0.89**	0.62		0.41	0.82	0.81	**0.96**
Diffutils	0.58	0.36	0.51	0.76	0.69	**0.82**	0.61	0.85	0.37	0.87	0.79	**0.93**
Findutils	0.49	0.28	0.43	0.58	0.61	**0.71**	0.47	0.70	0.36	0.66	0.69	**0.81**
Gzip	0.39	0.22	0.53	0.46	0.61	**0.78**	0.39	0.87	0.39	0.53	0.71	**0.9**
Lua	0.36		0.33	0.37	0.38	**0.55**	0.36		0.26	0.46	0.47	**0.69**
PuTTY	0.33		0.25	0.30	0.33	**0.5**	0.32		0.29	0.33	0.36	**0.54**

Table D.5: Cross-Optimization Binary Diffing - O0 Vs. O2. Missing values indicate timeouts.

	Precision						Recall					
	BD	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**	**BD**	**DBD**	**SAFE**	**OPC**	**BF**	**V2V**
						ARM - Clang12						
Coreutils	0.08	0.18	0.11	0.34	0.35	**0.58**	0.05	0.10	0.09	0.45	0.45	**0.76**
Diffutils	0.41	0.27	0.19	0.44	0.48	**0.67**	0.30	0.22	0.14	0.58	0.63	**0.89**
Findutils	0.34	0.32	0.13	0.40	0.52	**0.69**	0.22	0.17	0.11	0.48	0.62	**0.82**
Gzip	0.05	0.17	0.20	0.33	0.43	**0.61**	0.04	0.18	0.13	0.43	0.56	**0.80**
Lua	0.17		0.04	0.14	0.21	**0.41**	0.16		0.02	0.21	0.33	**0.65**
PuTTY	0.07		0.04	0.06	0.23	**0.29**	0.05		0.04	0.07	0.30	**0.39**
cURL	0.07		0.14	0.46	0.58	**0.67**	0.06		0.10	0.53	0.67	**0.77**
						ARM - GCC8						
Coreutils	0.14	0.18	0.20	0.29	0.40	**0.55**	0.10	0.12	0.15	0.41	0.56	**0.76**
Diffutils	0.25	0.22	0.34	0.40	0.41	**0.65**	0.18	0.14	0.23	0.55	0.63	**0.90**
Findutils	0.10	0.28	0.26	0.33	0.50	**0.65**	0.07	0.12	0.20	0.42	0.64	**0.83**
Gzip	0.23	0.21	0.37	0.18	0.48	**0.55**	0.22	0.16	0.25	0.24	0.64	**0.72**
Lua	0.23		0.16	0.13	0.30	**0.40**	0.23		0.12	0.19	0.44	**0.58**
PuTTY	0.10		0.10	0.06	0.27	**0.36**	0.07		0.10	0.08	0.36	**0.48**
cURL	0.23		0.25	0.12	0.15	**0.57**	0.19		0.19	0.58	0.61	**0.75**
						x86 - Clang12						
Coreutils	0.16	0.12	0.24	0.34	0.34	**0.53**	0.12	0.34	0.19	0.46	0.44	**0.70**
Diffutils	0.48	0.17	0.33	0.46	0.45	**0.68**	0.45	0.60	0.23	0.61	0.59	**0.90**
Findutils	0.32	0.17	0.32	0.37	0.45	**0.67**	0.28	0.53	0.26	0.46	0.55	**0.82**
Gzip	0.24	0.11	0.35	0.37	0.36	**0.62**	0.20	0.76	0.25	0.48	0.48	**0.81**
Lua	0.23		0.17	0.10	0.22	**0.39**	0.21		0.12	0.15	0.34	**0.59**
PuTTY	0.11		0.12	0.06	0.21	**0.30**	0.10		0.12	0.07	0.28	**0.41**
cURL	0.30		0.37	0.41	0.57	**0.73**	0.25		0.27	0.47	0.66	**0.84**
						x86 - GCC8						
Coreutils	0.21	0.14	0.30	0.32	0.36	**0.51**	0.19	0.31	0.23	0.45	0.51	**0.72**
Diffutils	0.29	0.14	0.39	0.43	0.42	**0.60**	0.31	0.42	0.26	0.61	0.60	**0.86**
Findutils	0.36	0.13	0.32	0.32	0.48	**0.62**	0.32	0.34	0.25	0.41	0.62	**0.81**
Gzip	0.35	0.14	0.38	0.22	0.44	**0.55**	0.36	0.53	0.27	0.30	0.59	**0.73**
Lua	0.35		0.19	0.12	0.30	**0.39**	0.37		0.14	0.18	0.44	**0.57**
PuTTY	0.23		0.15	0.07	0.25	**0.36**	0.21		0.15	0.10	0.33	**0.49**
cURL	0.20		0.40	0.56	0.63	**0.79**	0.19		0.29	0.64	0.72	**0.91**

Index