UNIVERSITY OF LEEDS

Towards Practical Machine Learning for Program Analysis and Optimisation

Huanting Wang

University of Leeds

School of Computer Science

Submitted in accordance with the requirements for the degree of

Doctor of Philosophy

October, 2025

Intellectual Property Statement

The candidate confirms that the work submitted is his own and that appropriate credit has been given where reference has been made to the work of others.

This copy has been supplied on the understanding that it is a copyright material and that no quotation from the thesis may be published without proper acknowledgement.

The right of Huanting Wang to be identified as the Author of this work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

© 2025 The University of Leeds and Huanting Wang.

Acknowledgements

I would like to express my gratitude to my supervisor and role model, Professor Zheng Wang, for his invaluable guidance and support. His mentorship has helped me grow not only as a PhD but also as a person. It has been an honour to work with him, and this will remain one of the most meaningful experiences of my life.

I would like to thank Zhanyong Tang, Shin Hwei Tan, Hugh Leather, and Chris Cummins, with whom I have collaborated on various projects that form parts of this thesis.

I would like to thank my colleagues and friends in both the U.K. and China.

I would like to thank my thesis examiners, Professor Karim Djemame and Professor Michael O'Boyle, for their valuable advice on this thesis.

I would like to thank my mother, Dongcui Zhu, my father, Kun Wang, and other family members for their constant encouragement and unconditional love.

Finally, I would like to thank my beloved, Guiting. Her patience, warmth, and unwavering support have been the quiet strength behind my journey. No words can truly express how deeply her love has meant to me.

Abstract

Machine learning (ML), such as supervised learning and deep reinforcement learning (DRL) techniques, has shown great potential in code-related tasks such as predicting code optimisation options and detecting software bugs. However, its practical use still faces multiple hurdles during the model design and deployment phases. This thesis addresses some of these challenges through three core contributions, aiming at making ML more practical and reliable for program analysis and optimisation.

The first contribution tackles a fundamental challenge in applying ML to code - how to represent programs. Our approach enables ML to combine static code information with dynamic symbolic execution traces to capture rich program semantics while using a learning-based approach to reduce the overhead of symbolic execution. This improved representation enabled the development of an effective ML-based bug detection tool that uncovered 55 unique code vulnerabilities from 20 real-world projects, leading to the assignment of 37 new Common Vulnerabilities and Exposures (CVEs).

The second contribution aims to lower the barrier to integrating ML into compiler development. We introduce a framework with a simple Application Programming Interface (API) that helps developers construct DRL systems for compiler optimisation. The framework adopts a meta-learning strategy combining DRL with multi-task learning to search ML architectures. We show that the ML solutions automatically assembled by our framework outperform those developed manually by independent experts across four code optimisation tasks.

The third contribution addresses the reliability issues of using trained ML models during deployment in the end-user environment. Our approach leverages statistical assessments to identify when an ML model will likely make incorrect predictions, enabling fallback strategies to maintain its robustness. We integrate our approach with 13 representative ML models across five code analysis and optimisation tasks, showing that our techniques can correctly identify an average of 97% of mispredictions.

The work presented in this thesis has resulted in three open-source tools, which we hope will support wider adoption of ML in software engineering tasks like code optimisation and bug detection.

Declaration

I declare that this thesis was composed by me, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material in this thesis has been published in the following papers:

- Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. 2024. **Combining Structured Static Code Information and Dynamic Symbolic Traces for Software Vulnerability Prediction**. *In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024) [1]* - Chapter 4 is based on this article;
- Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. **Automating reinforcement learning architecture design for code optimization**. *In Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC 2022) [2]* - Chapter 5 is based on this article;
- Huanting Wang, Patrick Lenihan and Zheng Wang. 2025. **Enhancing Deployment Time Predictive Model Robustness for Code Analysis and Optimization**. *In Proceedings of the International Symposium on Code Generation and Optimization (CGO 2025) [3] received a Distinguished Paper Award* - Chapter 6 is based on this article.

(Huanting Wang)

Contents

Abbreviations

A3C	Asynchronous Advantage Actor Critic	API	Application Programming Interface
AI	Artificial Intelligence	AST	Abstract Syntax Tree
AUC	Area Under the Curve	CFGs	Control Flow Graphs
CDF	Cumulative Distribution Function	CNNs	Convolutional Neural Networks
CP	Conformal Prediction	CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration	DDGs	Data Dependence Graphs
DFGs	Data Flow Graphs	DL	Deep Learning
DNNs	Deep Neural Networks	DQN	Deep Q-Networks
DRL	Deep Reinforcement Learning	FPR	False Positive Rate
GNNs	Graph Neural Networks	KNN	K-Nearest Neighbour
LLMs	Large Language Models	LSP	Language Server Protocol
LSTM	Long Short-Term Memory	MAB	Multi-Armed Bandit
MCTS	Monte Carlo Tree Search	MDP	Markov Decision Process
ML	Machine Learning	MLP	Multi-Layer Perception
MTL	Multi-task learning	PDCG	Program Dependency Control Graph
PPO	Proximal Policy Optimization	RL	Reinforcement Learning
RNNs	Recurrent Neural Networks	SARD	Software Assurance Reference Dataset
SGD	Stochastic Gradient Descent	SVM	Support Vector Machine

Chapter 1

Introduction

Over the past two decades, ML has emerged as a powerful method for a wide range of code-related tasks, including performance optimisation [4–8] and code analysis [1, 9, 10]. A growing body of research has shown that ML-based methods often outperform traditional approaches that rely on hand-crafted heuristics [11, 12].

ML has transformed how we approach code optimisation and analysis by automating decisions that required extensive manual effort. For example, ML-based vulnerability detectors can identify security bugs and vulnerabilities in minutes, tasks that previously took expert teams days or even weeks [13]. Similarly, ML models can recommend code transformations that improve performance, reducing the requirements for time-consuming trial-and-error tuning by developers [12]. These capabilities have led to growing interest in applying ML techniques to software engineering problems [14, 15].

Most ML-based solutions for program problems follow a two-phase process: design and deployment. In the *design phase*, developers use domain-specific datasets to find a suitable representation method and then construct an appropriate ML architecture to train the model for their specific tasks. In the *deployment phase*, the trained model is integrated into production systems to make predictions on those new, unseen programs. Each of these stages presents its own challenges in practice.

This thesis addresses some of the key challenges for ML-based program analysis and optimisation during design and deployment phases, through three core contributions. First, it demonstrates, for the first time, that combining static source code features with dynamic symbolic execution traces can improve learned program representations. Second, it introduces an automated framework to help compiler developers select appropriate reinforcement learning (RL) architectures for compiler tasks. Third, it shows how statistical assessments can be employed to identify when a trained model is likely to mispredict, thereby improving ML robustness during deployment. These contributions enhance model performance, reduce development effort, and lower the barrier to adoption by automating ML architecture design for code-related tasks.

In the next section, we describe the challenges this thesis targets in more detail.

1.1 Challenges for Applying Machine Learning to Code

To successfully apply ML to code optimisation and analysis tasks, three key challenges must be addressed, as described below.

1.1.1 Learning Program Representation

A key issue for applying ML to code-related tasks is how to best represent programs [16]. Early work in the area relied on hand-crafted features for program representation - for instance, counting the frequency of instructions of different types [17] or the number of function calls to routines like *free* and *malloc*. More recent approaches leverage the power of DL [9, 18, 19] to automatically learn representations from plain-text source code [20] or from static program structures such as control flow graphs (CFGs) and data flow graphs (DFGs) [21].

While static code representations enable DL models to capture token relationships, they often introduce ambiguity due to redundant code, complex structures, and long execution paths. These issues can lead to high false-positive rates and reduced accuracy in code analysis and optimisation. In addition, many existing methods rely solely on the sequential order of code statements presented in the plain-text source code, overlooking structured control and data flow information that is essential for capturing execution dependencies and alternative execution paths, such as function calls and branches. Expanding the input context can help models capture long-range dependencies [9, 22], but this comes at the cost of significantly increased memory usage [9], limiting the practicality of such approaches when applied to large-scale, real-world software projects.

To address these limitations, recent work [23, 24] has explored dynamic execution traces, which augment static representations with runtime behaviour by tracking variable changes through symbolic executions. While promising, these approaches use a random sampling strategy, which incurs significant overhead due to the cost of collecting dynamic execution traces.

To overcome these limitations, we propose a hybrid approach for learning program representations that combines static and dynamic views. Our method leverages static structural information, such as the Program Dependency Control Graph (PDCG), to ensure having a global view of all possible execution paths, while selectively applying symbolic execution to capture deeper program semantics and reduce ambiguity. To mitigate the overhead typically associated with dynamic tracing, we introduce a path selection module. For a given program, this module identifies the most relevant execution paths from its PDCG to guide symbolic execution. The resulting symbolic traces are then processed by a neural network to generate dynamic embeddings, which are combined with static code embeddings as program representation for downstream prediction tasks.

1.1.2 Expertise Barrier in Machine Learning

Introducing ML into domain-specific optimisation tasks presents new challenges for systems developers like compiler writers who may not have ML expertise. For example, when working with DL architectures, systems developers need to choose an appropriate neural network for the downstream tasks. This complexity increases further with the emergence of DRL architectures. For example, in DRL-based code optimisation, developers must choose and fine-tune several components: a discrete action space (e.g., compiler passes), a state representation (e.g., program features), and a reward function to assess outcomes. Some RL methods also require a transition function, adding further complexity, especially as these components must be tailored to different application workloads.

Although tools like CompilerGym [15], RLlib [25], and others [5, 6, 11, 26–28] offer reusable components, selecting the right configuration remains difficult. The optimal setup depends heavily on the specific task, and manual design - akin to neural architecture search - is time-consuming and error-prone.

To address this, we introduce an open-source framework that automates RL architecture design and parameter tuning for code optimisation. To use our tool, users provide a list of actions relevant to the optimisation problem and a measurement interface to report performance metrics, such as code size or speedup. Our system then automatically assembles an RL pipeline using modular components: pre-trained state encoders (e.g., Word2Vec [29], CodeBERT [30]), built-in reward functions (e.g., `RelativeMeasure`, `tanh`), transition models (e.g., probability matrices, Long-Short-Term Memory (LSTM) [31]), and RL algorithms (e.g., Proximal Policy Optimisation (PPO) [32] and Monte Carlo Tree Search (MCTS) [33]).

To tune RL architectures, our system uses customisable algorithms such as PPO [32] and MCTS [33], operating over user-defined reward, state, action, and transition components. Its modular design enables a rich space of parameterised RL configurations, allowing compiler developers to define customised setups with just a few lines of Python via a simple API.

1.1.3 Robustness at Deployment Time

ML models can be fragile. Small changes in hardware or the environment can significantly reduce decision accuracy and model robustness [34, 35]. This issue arises from the fact that ML models are typically built on the assumption that the training data accurately reflects the distribution of future test data. When the distributions of the training and test data no longer align, model performance degrades - an issue referred to as *data drift*.

Existing efforts to improve ML robustness for code-related tasks have primarily focused on enhancing learning efficiency or improving model generalisation during the design phase. These approaches include synthesising benchmarks to expand the training dataset [6], developing better program representations [9, 27], and combining multiple models to boost generalisation capabilities [9]. While these design-time methods are valuable, they are unlikely to account for all potential changes that may arise during deployment. Although some research has explored validating model assumptions [36] in the context of runtime scheduling, existing solutions often assume a specific ML architecture and therefore lack generalizability.

To further enhance the generalizability of ML models, it is essential to preserve and improve their robustness during deployment. To this end, we propose a statistical method to analyse the prediction probabilities given by underlying ML models and detect data drift. By using the detected drifting data(e.g. fine-tuning the models), we improve their robustness during deployment to retain 95% of the performance achieved during training. Building on this method, we introduce an open-source toolkit specifically designed for code optimisation and analysis tasks. Rather than replacing design-time solutions, the toolkit offers a complementary approach aimed at improving the robustness of ML systems after deployment. Its primary goal is to maintain the reliability of already deployed models in the face of changing data conditions and to support continuous improvements in real-world environments.

We adopt the prediction with rejections [37–40], which flags predictions that are likely to be unreliable. This mechanism enables the system to trigger corrective actions when data drift is detected. For example, an ML-based performance tuner can alert users when a prediction, such as a set of compiler flags for optimising a program, is unlikely to deliver good performance, prompting them to explore alternative search strategies [2, 41, 42]. Similarly, a bug detection model can raise warnings for potentially incorrect predictions, allowing human experts to verify and correct them.

By identifying and flagging likely mispredictions, the toolkit facilitates continuous learning in production. These misclassified or uncertain examples can be used as additional training data to iteratively refine the model, thereby enhancing long-term performance and reliability. Also, the toolkit offers a user-friendly Python interface for training and deploying supervised ML models, with built-in capabilities for detecting data drift post-deployment. Developers can easily integrate the toolkit into existing ML workflows by subclassing its abstract interface, typically requiring only a few dozen lines of code.

1.2 Contributions

This thesis aims to make ML practical for code analysis and optimisation. Its key contributions are:

- The first DL framework to combine structured code representations with symbolic execution for code analysis tasks such as vulnerability prediction. It incorporates an unsupervised active learning strategy to reduce symbolic execution overhead through efficient path selection. This work also provides two open datasets to support future research on integrating static and symbolic code information.

- A generic framework that automatically selects and tunes suitable ML models (e.g., RL architectures) for code optimisation tasks. It uses DRL as a meta-optimiser to simplify the integration of ML to code optimisation frameworks, supported by a comprehensive evaluation across four optimisation tasks.

- An open-source framework to enhance ML model reliability after deployment, supporting both classification and regression in code optimisation and analysis. It introduces a novel adaptive weighting and ensemble method for applying conformal prediction (CP) to detect data drift and mispredictions, validated through a large-scale empirical study across ML architectures and code-related tasks.

1.3 Publications

This thesis is in part based on ideas and results that have been previously described in publications. Our approach for combining static and dynamic information to learn program presentation, presented in Chapter 4, was described in:

- Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. 2024. **Combining Structured Static Code Information and Dynamic Symbolic Traces for Software Vulnerability Prediction**. *In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024).*

Chapter 5 describes a meta-optimiser-based approach to automating ML architecture search, which was previously published in:

- Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. **Automating Reinforcement Learning Architecture Design for Code optimization**. *In Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC 2022).*

Our approach for improving robustness at deployment time, as proposed in Chapter 6, was first published in:

- Huanting Wang, Patrick Lenihan, and Zheng Wang. 2025. **Enhancing Deployment Time Predictive Model Robustness for Code Analysis and Optimization**. *In Proceedings of the International Symposium on Code Generation and Optimization (CGO 2025).*

The experimental results in this thesis are reproductions of those in the above publications. What differentiates this work from prior publications is the addition of background material (Chapter 2) and a literature review (Chapter 3), which provide a more comprehensive overview of the relevant fields. The thesis also references new works published after the aforementioned papers, including those based on these publications.

1.4 Structure

The remainder of the thesis is organised as follows:

Chapter 2. provides the background. It defines key terminology and describes the ML techniques for program modelling and evaluation methods used in this work.

Chapter 3. reviews the relevant literature, divided into three categories: precise ML, automated ML, and robust ML.

Chapter 4. presents a new technique that combines static and dynamic information to model programs, using vulnerability detection as a case study.

Chapter 5. introduces a framework that automatically selects and tunes an appropriate DRL architecture for code optimisation tasks.

Chapter 6. presents techniques to improve the reliability of ML models at deployment time. Our approach enables the identification of data drift in code-related tasks during deployment.

Chapter 7. summarises the overall findings of the thesis, provides a critical review, and outlines potential directions for future research.

1.5 Summary

ML techniques have shown great potential in code optimisation and analysis by improving application performance (e.g., speedups) and reducing the manual effort required to develop optimisation heuristics. Realising these benefits, however, requires overcoming three key challenges: having effective program representations, designing appropriate model architectures, and ensuring model robustness after deployment. This thesis tackles these challenges through three technical contributions, resulting in three open-source frameworks. The next chapter introduces the technical background, and then we review related works. Subsequent chapters present our technical contributions and evaluation, concluding with a summary of the thesis's main contributions.

Chapter 2

Background

2.1 Introduction

This chapter provides an overview of the techniques and theories used in this thesis. Sections 2.2 to 2.4 describe some related ML techniques used in this thesis. Section 2.5 shows the evaluation methodologies used in this thesis. Section 2.6 concludes.

2.2 Machine Learning

ML is a subfield of artificial intelligence (AI) that focuses on developing algorithms and statistical models that enable computers to perform tasks. Instead of following hard-crafted instructions, ML systems learn patterns and relationships from a training dataset to make predictions, classifications, or decisions. The learning process typically involves training a model on the training dataset, allowing it to generalise from the data and perform well on unseen inputs. ML encompasses a range of techniques, including DL - supervised learning (where models learn from labelled data), unsupervised learning (which identifies hidden structures in unlabeled data), and RL (where agents learn optimal behaviours through trial and error). These methods have been successfully applied in diverse domains such as natural language processing, code analysis and code optimisation tasks.

2.2.1 Deep Learning

DL, a subset of ML, focuses on using neural networks to enable models to learn representations from training samples. Its strength lies in its ability to automatically extract complex patterns and features from raw input data, eliminating the need for manual feature engineering that is often required in traditional machine learning techniques. One of the key advantages of DL is its capability to handle large-scale datasets, making it highly effective for code optimisation [43–46] and analysis tasks [9, 27, 47–51].

2.2.2 Deep Reinforcement Learning

RL is an ML technique where an agent interacts with an environment to learn actions that maximise cumulative rewards over time [52]. The environment represents the problem to be solved, and the agent takes actions (e.g., selecting compiler options) based on its observations. The agent's goal is to improve its performance by refining its policy, which maps states to actions in a way that maximises the expected reward. Additionally, many RL algorithms use a value function to assess the quality of a given state, guiding the agent's decision-making process. While the policy tells the agent "what to do next," the value function answers "how good is this state?"

DRL combines RL with Deep Neural Networks (DNNs), enabling the agent to operate in high-dimensional and complex environments [53]. This combination allows DRL to approximate policies and value functions for tasks with large state and action spaces, making it highly suitable for complex applications like compiler optimisation. In these tasks, DRL can automate decisions such as selecting optimisation passes or tuning parameters to improve metrics like execution speed or code size.

Markov Decision Process

In this work, we model our reinforcement learning strategy using a Markov Decision Process (MDP) [54], a well-established framework for decision-making in environments where outcomes are partly random and partly under the control of the agent. An MDP is defined by a set of states, a set of actions, a transition function, and a reward function. The transition function defines the probability distribution over the next possible states, given the current state of the environment and the action taken by the agent. This transition function captures the stochastic nature of the environment, helping the agent predict the likelihood of moving to a particular state after taking a specific action.

The reward function estimates the expected immediate reward that the agent will receive when transitioning from one state to another, based on the action taken. This reward function serves as feedback, allowing the agent to learn which actions yield the highest cumulative reward over time. The goal of the agent in an MDP is to learn a policy, mapping from states to actions that maximises the expected cumulative reward, also known as the return, over an extended sequence of actions.

A key challenge in defining an MDP for real-world tasks lies in accurately modelling the transition and reward functions. These functions need to represent the complexity of the environment while being computationally tractable. In our work, we address this challenge by automatically selecting the most suitable candidate functions for both transitions and rewards. We achieve this by framing the policy search problem as a multi-armed bandit problem, where each "arm" represents a potential transition-reward function pair. This formulation allows the agent to explore different candidate functions systematically, balancing exploration and exploitation to find the most effective model for the environment.

By modelling the transition dynamics and reward structure as an MDP, our approach enables more informed decision-making and efficient policy learning. The multi-armed bandit approach to selecting these functions also ensures adaptability and robustness, reducing the need for manual intervention or hand-tuned functions. The result is an RL agent that can effectively operate in complex, uncertain environments by relying on a well-constructed MDP framework.

Multi-armed Bandit Problem

The effectiveness of an RL algorithm is highly dependent on selecting the appropriate components, such as the policy architecture, value function estimator, and exploration strategy. However, manually choosing these components is a non-trivial task [55]. The Multi-Armed Bandit (MAB) [56] problem is a classical framework in sequential decision making that models the trade-off between exploration and exploitation. It is inspired by the scenario of a gambler facing multiple machines, each with an unknown probability distribution of rewards. At each time step, the agent chooses one arm to pull and observes a reward, aiming to maximise the cumulative reward over time.

Formally, an MAB problem consists of K arms, each associated with an unknown reward distribution. The agent's goal is to select arms over a sequence of trials to minimise regret, defined as the difference between the reward of an optimal strategy and the one actually followed.

2.2.3 Algorithms of Machine Learning

K-Nearest Neighbour (KNN). KNN [57] is a simple, non-parametric classification and regression algorithm. It classifies a data point by looking at the K nearest points in the feature space and assigning the most common class label among them. The distance between data points is commonly computed using Euclidean distance:

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2} \tag{2.1}$$

The algorithm begins by selecting a value for the number of neighbours, K. Next, it computes the distance between the new data point and all points in the training set. After calculating the distances, the nearest K neighbours are identified by sorting these distances. For classification tasks, the algorithm assigns the most frequent label among the K neighbours. In the case of regression, it calculates the mean value of the neighbours to make a prediction.

K-Means. K-Means [58] is an unsupervised clustering algorithm that aims to partition data into K clusters by minimising the within-cluster variance. The algorithm is iterative, and the key steps and formulas are:

1. Initialize: Select K random centroids $\mu_1, \mu_2, \ldots, \mu_K$.

2. Assign: Assign each data point x_i to the nearest cluster based on the Euclidean distance:

$$\text{Cluster}(x_i) = \arg\min_k \|x_i - \mu_k\| \tag{2.2}$$

3. Update: Recompute the centroids for each cluster by calculating the mean of the points in that cluster:

$$\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i \tag{2.3}$$

4. Repeat: Repeat the assign and update steps until convergence, which occurs when the cluster assignments no longer change or the centroids stabilise.

K-Means is efficient for clustering large datasets but requires the number of clusters K to be pre-specified, which can be a limitation in some applications.

Transformer Network. Transformer [59] is a type of deep learning network designed to process sequential data by leveraging the self-attention mechanism. This mechanism computes attention scores between each pair of elements in a sequence, allowing the model to weigh the relative importance of different tokens. The key equations of the self-attention mechanism are as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.4}$$

where Q is the query matrix, K is the key matrix, V is the value matrix, and d_k is the dimension of the keys.

The Transformer model processes input through several layers of attention mechanisms and feedforward networks, each incorporating residual connections and layer normalisation. Initially, the input sequence is embedded into fixed-sized vectors, followed by the addition of positional encodings to provide information about the order of tokens. The model then applies multi-head attention, where the input passes through multiple attention heads to capture various relationships within the data. Afterwards, the output from the attention layer is processed by a fully connected feedforward network. Finally, the transformed sequence is used to perform classification or regression, depending on the task at hand.

2.3 Work Flow for Machine Learning based Program Modelling

2.3.1 Deep Learning based Program Modelling

The workflow for DL in program modelling typically involves several key steps. The first step is data collection and preparation. After gathering the data, preprocessing is performed to clean it by removing noise, handling missing values, and ensuring consistency. The dataset is then divided into training, validation, and test sets.

The next step is DL model selection. Choosing an appropriate deep learning architecture is crucial and depends on the specific task. For instance, convolutional neural networks (CNNs) [60] are widely used for image classification, while Recurrent Neural Networks (RNNs) [61], including LSTM [62] networks, are better suited for handling sequential data such as time series or natural language. Transformer-based models are frequently applied to language-related tasks and form the foundation of large language models (LLMs).

Once the architecture is selected, the model is constructed by specifying the number of layers, types of neurons, activation functions (e.g., ReLU, Sigmoid)[63], and the loss functions, such as cross-entropy for classification or mean squared error for regression[64]. The model's parameters are optimised using algorithms such as Stochastic Gradient Descent (SGD) or Adam [65], and regularisation techniques like dropout or batch normalization [66] are employed to prevent overfitting.

The model is then trained by feeding the numerical vectors of the training data into the network to minimise the loss, while optimising its parameters through backpropagation with the help of optimisation algorithms. This process is iterative, with hyperparameters such as the learning rate and batch size fine-tuned throughout. A validation dataset is used during training to monitor performance and prevent overfitting. After training, the model is evaluated on a separate test dataset using appropriate performance metrics. For classification problems, metrics such as accuracy, precision, recall, and F1-score are commonly used, whereas for regression tasks, metrics like mean squared error are more relevant.

Finally, the trained model can be further tuned through model updating or the application of explainability techniques to ensure its suitability for deployment in real-world environments.

2.3.2 Deep Reinforcement Learning based Program Modelling

A typical DRL workflow begins with defining the environment, which encapsulates the program modelling task. This includes specifying the state space (e.g., program features, intermediate representations), action space (e.g., compiler optimisation passes or transformation decisions), and the reward function, which quantifies the performance improvement resulting from a particular action (e.g., execution speedup, reduced code size). The environment must accurately reflect the underlying structure and dynamics of the program optimisation or analysis problem.

Next, the DRL agent is configured. This includes selecting a neural network architecture to approximate the policy and value functions, choosing a suitable DRL algorithm (e.g., Deep Q-Networks (DQN) [67] or PPO [32]), and setting hyperparameters such as learning rate and discount factor. The agent's goal is to learn a policy that maps observed program states to actions that maximise cumulative rewards.

2.4 Robust Machine Learning

2.4.1 Credibility Evaluation

To detect unreliable prediction outcomes, a straightforward approach is to analyse the probability distribution given by an ML model. For instance, a high prediction probability for a specific label might suggest high confidence in the prediction. However, this alone does not fully capture the prediction's reliability, as ML models can produce skewed probabilities for rarely seen samples. Consider multi-class classification as an example. An ML model predicts the likelihood, r^i, that a given input belongs to each class (c_1 to c_n). However, if the input's pattern significantly differs from training samples, the model might assign a low probability to some classes (e.g., $r^1 \approx 0.0$ for class c_1). This can disproportionately inflate the probabilities of other classes (r^2 to r^n) as the sum of probabilities needs to equal 1.0 [68]. In this case, a high probability does not equate to high prediction confidence. Therefore, assessing the model's credibility requires an approach that evaluates how well the input aligns with the training data.

2.4.2 Statistical Assessment

Statistical assessments are used to evaluate the credibility and confidence of predictions in machine learning models. Unlike typical probabilistic evaluations, which assess the likelihood of a test sample belonging to a certain class or value in isolation, statistical assessments draw from historical data distributions. This broader approach helps answer questions like: *"How likely is the test sample to belong to a class compared to all other possible classes?"*

By framing the sample within the context of historical decisions and probabilistic distributions, statistical assessments provide a more robust measure of prediction confidence. For example, conformal prediction techniques quantify uncertainty by leveraging past data to make guarantees about the reliability of a prediction, rather than returning raw probability scores alone [69].

These methods are particularly useful in applications where understanding the uncertainty behind a prediction is critical, such as in healthcare [70], finance [71], or autonomous systems [34, 72]. In such domains, the consequences of incorrect predictions can be significant, and statistical assessments help to ensure a more trustworthy evaluation of how reliable a model's output is.

2.4.3 Conformal Prediction

CP [69, 73] is a statistical framework that, given a model g and a specified significance level, defines a prediction region that contains the true value with a certain probability. The core idea behind CP is to construct this prediction region based on the distribution of training data, taking into account noise and variability. Unlike point predictions that estimate a single value, CP offers a range, ensuring with high confidence that the true value lies within this region.

CP is designed to enhance prediction reliability by providing coverage guarantees- ensuring that the prediction interval contains the true value with a predetermined likelihood, based on the chosen significance level. By leveraging historical data and model residuals, CP adjusts the prediction region dynamically to maintain the desired confidence level, regardless of the complexity of the underlying model.

This method proves useful in applications where it is critical to quantify uncertainty, offering more reliable insights into the model's predictions. We utilise CP to evaluate the reliability and confidence of our model predictions, ensuring that the model not only performs well but also provides trustworthy prediction intervals.

2.4.4 P-value

The p-value [74], as derived through CP, plays a crucial role in evaluating the credibility of predictions and the confidence in that credibility. Specifically, we use the p-value to quantify the degree of evidence contradicting the null hypothesis. For example, when assessing the confidence of a classifier's prediction, the null hypothesis assumes no significant difference between the predicted class and any other class, as inferred from the model's probability distribution. Similarly, in gauging the credibility of a prediction, the null hypothesis posits that the prediction does not belong to a specified prediction region defined by CP.

The p-value is typically computed as:

$$p = \frac{\text{Number of conformal scores greater than or equal to the test score}}{\text{Total number of conformal scores}} \tag{2.5}$$

The conformal score measures how well a data point fits the underlying distribution learned by the model. A high p-value indicates that the observed data is not likely under the null hypothesis, providing strong evidence against it. In contrast, a low p-value suggests that the observed data are consistent with the null hypothesis, offering weaker evidence against it. In simple terms, a high p-value points to a more reliable prediction.

2.5 Evaluation Methodology

2.5.1 Cross-validation

In this work, we utilise K-Fold Cross-validation [75] as a robust strategy for evaluating the performance of our models. K-Fold Cross-validation is a technique used to assess the generalisation ability of a model by partitioning the dataset into K equal-sized subsets, or "folds." The model is trained on $K - 1$ of these folds and tested on the remaining fold. This process is repeated K times, with each fold serving as the test set exactly once.

The final performance metric is the average of the results obtained across all K iterations. This method provides a more reliable estimate of model performance compared to a simple train-test split, as it ensures that every data point is used for both training and testing. Additionally, it mitigates the risk of overfitting and provides insights into how well the model generalises to unseen data.

A key challenge in using K-Fold Cross-validation is the choice of K, which affects the bias-variance tradeoff. A lower K value may lead to higher variance, while a higher K value increases computational costs. In this work, we empirically determine the optimal K based on our dataset and computational constraints.

2.6 Summary

This chapter provides background on the DL and DRL techniques used in this thesis to develop precise, automated, and robust ML techniques for code optimisation and analysis tasks, as well as the methodologies used to evaluate their effectiveness. The following chapter surveys related work to our thesis.

Chapter 3

Related Work

3.1 Introduction

This chapter surveys the literature in areas relevant to this thesis. Section 3.2 reviews research on precise ML for program modelling. We take vulnerability detection as our case study. Section 3.3 discusses the literature on automated ML for program modelling on compiler and code optimisation. Section 3.4 surveys work related to ML robustness at deployment time for code-related tasks. Finally, Section 3.6 concludes the chapter.

3.2 Precise Machine Learning for Program Modelling

Our work on improving precise program modelling for vulnerability detection builds on foundations in DL, source code vulnerability analysis, and static symbolic execution. Table 3.1 shows several related SOTA approaches and our position relative to them.

3.2.1 Deep learning-based Vulnerability Detection

Our research on program modelling is part of the recent efforts in DL-based software vulnerability detection [47–51]. Prior studies primarily relied on static information like Abstract Syntax Tree (AST) and PDCGs.

Graph-Based Feature Representations

Prior researchers have applied DNNs to various graph-based program representations, such as ASTs [76, 77], CFGs, PDCGs [18, 50, 51], and Data Dependence Graphs

Table 3.1: Summary of related work on precise ML for program modelling (vulnerability detection).

Paper / Year	**Method**	**Features / Input**	**Dataset**	**Task**	**Limitations**
AST-based models [76, 77]	DNN	Abstract Syntax Trees	Source code corpora	Vulnerability detection	Limited semantic flow
CFG / PDG models [18, 50, 51]	GNN	Control/Data flow graphs	Benchmarks	Detecting control-flow issues	High complexity
DDG-based models [9]	GNN	Data dependence graphs	Code datasets	Vulnerability detection	Expensive to scale
Sequence-based models [1, 21]	RNN / LSTM	Function call sequences	Program traces	Detect vulnerabilities	Poor structural context
ContraFlow [78]	Transformer	Static value-flow paths	Vulnerability corpora	Path-based vuln. detection	Misses dynamic behaviour
Fuzzing + ML [79, 80]	Hybrid	Input mutation guided by ML	Benchmarks	Vulnerability discovery	Expensive cost
VUDDY [81]	DL	Function clone features	Large open-source repos	Clone detection	Misses dynamic information
This work	Hybrid	AST + dynamic traces	CVE, SARD and repos	Vulnerability detection	-

(DDGs) [9]. These graph structures capture key structural and semantic relationships in code, which are crucial for vulnerability detection. For instance, ASTs have been used to represent the hierarchical structure of programs, revealing programming patterns and code semantics that are vital for detecting vulnerabilities [77].

Recent works [82] have demonstrated the effectiveness of these representations in identifying software vulnerabilities, such as SQL injection and cross-site scripting vulnerabilities, by analysing sanitisation patterns in web applications. These studies [83] often utilise a combination of CFGs and DDGs to capture both control and data flow information, enhancing the detection of security weaknesses. Additionally, there has been increasing interest in leveraging more advanced models, such as Graph Neural Networks (GNNs) and tree-structured LSTMs, to better preserve the hierarchical information contained in ASTs and other graph-based representations.

Moreover, combining different graph-based representations, such as ASTs with CFGs and PDGs, has been shown to improve vulnerability detection accuracy by providing a more comprehensive understanding of code behaviour [84]. This approach has proven to be highly effective across a range of program analysis tasks.

Sequence-Based Feature Representations

Several studies [1, 21] have explored the use of sequential code entities to train DNNs for vulnerability detection. These methods typically utilise features such as system execution traces, function call sequences and sequences of statements representing data flows to capture flow-based patterns in the code.

For instance, one approach [1, 9] leveraged both static and dynamic features from call sequences and data flows to detect memory corruption vulnerabilities in operating system programs at the binary level. The authors extracted call sequences from disassembled binaries to generate static features and monitored program execution to gather dynamic features, which were then fed into DNN models, improving detection accuracy for vulnerable programs.

Moreover, RNNs [61] have been applied to learning the semantic meanings of vulnerabilities through sequential data. For example, Bi-LSTM [62] and gated recurrent units have demonstrated effectiveness in capturing long-term dependencies crucial for identifying vulnerabilities such as buffer overflows. These models analyse sequences of code statements to detect contextual patterns and interdependencies indicative of vulnerabilities.

Text-Based Feature Representations

Text-based feature representations have been widely explored in the context of software vulnerability detection. These approaches generally treat code as raw text and apply techniques such as tokenisation and embedding to convert source code into vector representations. For instance, recent studies [85] converted Java source files into token lists and used the N-gram model to generate token vectors. Feature selection techniques, such as the Wilcoxon rank-sum test, were then employed to reduce dimensionality. The authors of these studies demonstrated an average detection accuracy of 93%, with precision and recall rates as high as 97.6% and 89.26%, respectively.

Another study [86] applied CNNs to detect vulnerabilities at the assembly level in C programs. The researchers developed ”Instruction2vec” to map assembly instructions to fixed-length dense vectors [87]. The generated vectors were used as input for CNNs, showing promising results in detecting buffer overflows. These works highlight the efficacy of using text-based representations in capturing vulnerable code patterns by treating source code as sequential text.

Mixed Feature Representations

Mixed feature representations [88] combine different types of input-graph-based, sequence-based, and text-based representations to enhance the performance of vulnerability detection models. For example, a study on Android APK files combined token features with semantic features extracted from ASTs, which were then used as input for a deep fully connected network. This approach [89] outperformed several traditional machine learning algorithms, achieving an AUC of 85.98%.

Another approach [22] extended the concept of “code gadgets” by incorporating both data and control dependencies to capture “global” code semantics and potential vulnerabilities. Using two Bi-LSTM networks to learn different types of features at different levels of granularity, this method was shown to effectively identify code vulnerabilities and even detect 0-day vulnerabilities. Combining different types of feature representations provides a richer understanding of code behaviour, leading to improved detection accuracy across various tasks.

Our approach incorporates symbolic traces with static code information to capture deeper program semantics, enhancing vulnerability detection.

3.2.2 Symbolic Execution

Symbolic execution [90, 91] has become a widely used technique in program analysis, as it sidesteps the need for hand-crafted rules by using symbolic values to represent inputs and analysing their use over the execution tree of a program. This technique enables the exploration of multiple program paths by evaluating the effects of symbolic inputs across different execution states. Tools like KLEE [90] have demonstrated success in automatically generating test cases for programs, particularly in detecting bugs and vulnerabilities.

However, one of the primary challenges in symbolic execution is the state explosion problem, where language constructs such as loops, branches, and recursion can significantly increase the number of execution states to be analyzed [92]. This severely limits the scalability of symbolic execution for large programs [93]. To mitigate this, various approaches have been proposed. One notable method involves pruning the execution tree by applying heuristics that prioritise certain execution paths [91, 94], which helps to reduce the number of states that need to be explored.

Recent advances have also explored combining symbolic execution with other techniques to enhance its scalability. For example, hybrid approaches that integrate symbolic execution with fuzz testing have been shown to achieve better coverage by combining the strengths of both methods [95]. Other research has proposed the use of machine learning models to guide the symbolic execution process, leveraging learned patterns to predict which paths are more likely to uncover vulnerabilities [96].

Despite these advancements, the limitations of symbolic execution in handling large-scale software remain a key challenge, particularly for applications requiring extensive path coverage or deep program analysis. Ongoing research continues to explore ways to address these limitations, including using distributed computing to parallelise symbolic execution tasks [97].

3.2.3 Contrastive Learning

Contrastive learning has gained popularity for its ability to reduce the costs associated with annotating large-scale datasets [98]. Originally developed in the context of computer vision [99–101], this self-supervised learning technique has since been successfully applied to other domains, including natural language processing [102, 103], where it has proven effective in learning useful representations without the need for extensive labeled data.

In our work, contrastive learning is employed to tackle the challenge of label scarcity in Chapter 4 for the vulnerability detection case study. By leveraging this technique, we can train models using unsupervised learning, which reduces the reliance on manually labelled datasets. This is particularly beneficial in the context of code analysis tasks, where acquiring labelled examples can be both costly and time-consuming.

Recent research in contrastive learning has shown that this approach excels at learning meaningful representations by bringing similar samples closer together in the feature space while pushing dissimilar samples apart [104]. In the field of vulnerability detection, contrastive learning helps distinguish between vulnerable and non-vulnerable code by learning invariant features across different code patterns.

By adopting contrastive learning, our method not only mitigates the need for extensive manual annotation but also achieves high-performance results in detecting vulnerabilities, as models are trained to focus on the inherent structure and semantics of the code, even in the absence of labelled data. This ability to harness unsupervised learning makes contrastive learning a powerful tool for enhancing the efficiency and scalability of vulnerability detection systems.

3.2.4 Path Selection for Code Embedding

Several techniques utilise learning-based approaches for path selection in program analysis [78, 91]. Similar to ContraFlow [78], we employ unsupervised active learning to train a path selection network. ContraFlow is specifically designed to identify static value-flow paths that may trigger a vulnerability, relying exclusively on static code features to detect potential security risks.

In contrast, we extend beyond static analysis by integrating path selection with symbolic execution, thus reducing the overhead typically associated with symbolic execution. By incorporating both static and dynamic code features, we are able to generate more efficient and informative program representations. This hybrid approach addresses the limitations of purely static methods like ContraFlow, which, while effective in certain contexts, may miss vulnerabilities that require deeper execution-level analysis.

Our experimental results for precise program modelling in Sections 4.5.2 and 4.5.3 demonstrate that our approach not only reduces symbolic execution overhead but also outperforms ContraFlow in terms of accuracy and efficiency. By leveraging dynamic features alongside static analysis, our method offers a more robust solution for vulnerability detection, achieving higher performance across various datasets and scenarios.

3.2.5 Machine Learning for Software Engineering

Machine learning techniques have proven to be highly effective in various software development tasks [2, 4, 7, 11, 27]. Existing approaches have been applied to a range of development challenges, including fuzz testing [79, 80], code clone detection [19, 81, 105, 106], enhancing static analysis for vulnerability detection [107, 108], program repair [109], defect prediction [110, 111], attack detection [112], and processing vulnerability reports [17, 113, 114].

Fuzz testing has been significantly enhanced by machine learning. Ye et al. [79] proposed an automated fuzz testing approach that leverages machine learning to intelligently mutate inputs for exposing vulnerabilities. Similarly, AFL++ [80] incorporates machine learning techniques to guide the fuzzing process and improve coverage.

In the area of code clone detection, Kim et al. [81] introduced VUDDY, a scalable method for detecting functionally similar code fragments using deep learning. GraphCodeBERT [106] further enhances code clone detection by utilising a graph-based learning approach to model the structural properties of code, thus improving semantic clone detection accuracy.

Machine learning has also been applied to enhancing static analysis for vulnerability detection. MirChecker [107] leverages machine learning to detect vulnerabilities missed by traditional static analysis tools, particularly in complex Android middleware.

Tomassi et al. [108] proposed a machine learning-based approach to enhance static vulnerability analysis by learning from historical vulnerabilities to improve detection rates.

Program repair techniques have benefited from the integration of machine learning. Shariffdeen et al. [109] introduced a concolic program repair approach that combines symbolic execution and machine learning to automatically generate bug fixes by analysing potential repair actions.

Defect prediction is another area where machine learning has shown effectiveness. Zeng et al. [110] developed Deep Just-In-Time Defect Prediction, which uses deep learning to predict defects during software development. Li et al. [111] conducted a systematic review of deep learning applications in defect prediction, highlighting the ability of neural networks to model complex relationships within code to identify potential defects.

In attack detection, Zhang et al. [112] provided a comprehensive survey on the use of machine learning techniques for detecting cyber threats, highlighting anomaly detection and pattern recognition methods as highly effective.

ML has been utilised to process and prioritise vulnerability reports. Perl et al. [17] proposed VCCFinder, which uses machine learning to automatically categorise and rank vulnerability reports based on severity. RecDroid [113] automates the processing of Android vulnerability reports by recommending fixes using deep learning. Cooper et al. [114] further explored machine learning methods to streamline the analysis of vulnerability reports, reducing manual effort and improving report processing efficiency.

While our program modelling method builds on these past efforts, it differentiates itself by offering a unique approach to program analysis and vulnerability detection. Unlike many prior works that rely primarily on static analysis or supervised learning methods, we integrate symbolic execution with machine learning techniques to reduce overhead and improve program representation. Furthermore, by employing active learning for path selection and combining both static and dynamic code features, we address the limitations of traditional methods, offering improved accuracy and efficiency in vulnerability detection. This hybrid approach sets our method apart from previous studies, providing a more robust and scalable solution for software security tasks.

Table 3.2: Summary of related work on automated ML for code optimisation.

Paper	**Method**	**Features**	**Dataset**	**Task**	**Limitations**
ATLAS / FFTW [115, 116]	Auto-tuning	Loop tile sizes, FFT params	Kernels	Performance tuning	Manual design
Iterative compilation [117–123]	Evol./Bayes search	Compiler options	Compiler benches	Code optimisation	High search cost
Predictive modelling [124–128]	Regression/ML	Program features	Benchmarks	Compiler tuning	Needs feature eng.
Autophase [133]	DRL	Compiler phases	LLVM benches	Phase ordering	Training cost
Haj-Ali et al. [134]	RL	Loop transformations	Compiler workloads	Loop optimisation	Large training data
NeuroVectorizer [135]	RL	Scalar $\rightarrow$ vector code	CPU workloads	Vectorisation	Domain-specific
Decima [136]	DRL	Task DAGs	Pipelines	Scheduling	System-specific
Khadka et al. [137]	DRL	Memory patterns	Hetero. systems	Memory placement	Limited generalisation
NAS [138–140]	Neural Arch. Search	NN configs	ML workloads	Model optimisation	High compute cost
This work	Automated RL search	Modular RL	Code tasks	Code optimisation and analysis	-

3.3 Automated Machine Learning

Auto-tuning techniques have been widely applied to reduce expert involvement in performance optimisation tasks [4]. Early works such as ATLAS [115] and FFTW [116] illustrate the potential of searching for domain-specific optimisation parameters, such as loop tile size. In iterative compilation, evolutionary algorithms like genetic algorithms [117, 118] and other search strategies based on Bayesian optimisation [119–123], along with predictive modelling [124–128], have been successfully used to optimize performance. Recent advances have extended these auto-tuning techniques to DNNs for code generation [44, 129–131], image processing [43], and runtime tuning of operating system and processor parameters [132]. Our work builds on these foundations, applying auto-tuning to code optimisation as shown in Table 3.2.

3.3.1 Reinforcement Learning for Code Optimisation

In recent years, RL, particularly deep RL, has demonstrated remarkable success in domains such as game-playing [141, 142] and robotics [143]. Its application has extended to performance optimisation tasks, including compiler phase ordering [133], loop optimisation [134], vectorization [135], task scheduling [136], and memory placement [137]. These efforts typically rely on hand-tuned policies or feature engineering to devise effective search strategies for a given application domain.

Silver et al. [141] demonstrated the power of deep reinforcement learning by training AlphaGo, a deep RL-based agent that surpassed human performance in the complex board game Go, relying on a combination of supervised learning and reinforcement learning to refine its policy and value networks.

For performance optimisation in compilers, Autophase [133] applied deep RL to the task of compiler phase ordering by training an RL agent to select the optimal sequence of compiler optimisations based on the characteristics of the program being compiled. This approach replaced the manual design of heuristics with an automated policy learned through reinforcement learning, achieving significant performance improvements across different benchmarks.

In loop optimisation, Haj-Ali et al. [134] introduced an RL-based approach to optimise loop transformations. Their method employed deep reinforcement learning to automatically explore different loop transformation strategies, learning which optimisations result in the best performance for a given program, thereby removing the need for expert-tuned strategies.

NeuroVectorizer [135] extended RL techniques to the domain of vectorisation, where the goal is to convert scalar operations into vectorised instructions for modern processors. The authors used deep RL to predict effective vectorisation strategies based on program features, achieving significant speedups over traditional vectorisation techniques that rely heavily on manually crafted heuristics.

Task scheduling, a key component in high-performance computing, has also benefited from reinforcement learning. Decima [136] employed deep RL to dynamically schedule tasks in data processing pipelines, learning optimal scheduling policies that adapt to system workloads and hardware characteristics, significantly improving system throughput and reducing latencies.

Memory placement optimisation has similarly seen improvements through deep RL. Khadka et al. [137] proposed a deep RL approach to optimise memory placement for workloads running on heterogeneous memory systems. Their method learns optimal memory allocation policies, taking into account the memory access patterns of applications, and thereby reducing memory access times and improving overall system performance.

Given the vast diversity of workloads and the need for optimised solutions, automated RL architecture tuning has gained significant interest. Approaches like ours, which automate the selection and tuning of RL architectures, are essential for addressing the complexity and variety of tasks across different domains. Unlike previous efforts that manually tuned policies and features, we explore automated methods to optimise RL architecture for performance tasks, improving both the search efficiency and the accuracy of optimisation.

CompilerGym [15] is a machine learning platform for compiler optimisation, offering an OpenAI Gym-like environment [144] for users to explore compiler optimisation tasks. Our automated ML method utilises CompilerGym's API to define problems and complements the platform by automating the search and tuning of RL components. By integrating with CompilerGym, we enhance the automation of RL architecture design, addressing a critical gap in automating compiler optimisations. We also plan to incorporate this into future releases of CompilerGym, expanding the toolkit for compiler optimisation research.

3.3.2 Neural Architecture Search

Neural architecture search (NAS) is a rapidly growing area of research aimed at automating the design of neural networks by optimising the architecture for specific tasks. Notable works, such as those in [138–140], focus on reducing model size, improving execution time, or increasing accuracy. Inspired by NAS, our work extends this concept to RL architecture search, aiming to find an efficient RL architecture that can be applied to performance optimisation tasks, particularly in code optimisation. By drawing on NAS techniques, we enhance RL's ability to adapt to varying workloads and optimise performance across different domains.

Our work builds on auto-tuning techniques and leverages advances in reinforcement learning, neural architecture search, and machine learning platforms like CompilerGym. By automating RL architecture tuning, we lower the barrier to entry for developers and enable broader applications of RL in performance optimisation tasks. This automation enhances the scalability and efficiency of optimisation processes, offering a promising path forward in both compiler optimisation and other domains.

3.4 Robust Machine Learning

Supervised ML has proven to be a powerful tool for code analysis and optimisation [8, 9, 11, 21, 145–147]. However, supervised ML models typically rely on the assumption that training data will closely resemble future test data [148]. This assumption is often violated in deployment environments due to evolving hardware, changing workloads, and new inputs, leading to compromised ML model robustness [149–151].

Table 3.3: Summary of related work on robust ML for code-related tasks.

Paper	Method	Features	Dataset	Task	Limitations
ProgramL [8]	GNN	Graph-based program rep.	Benches	Program modelling	Design-time only
Wang et al. [9]	GNN	High-level graphs	Code corpora	Robust learning	Static-only
Data augmentation [152]	Code synthesis	Synthetic code	Benches	Robust training	Unrealistic code risk
Uncertainty [153]	Confidence measures	Pred. distributions	Various	Robust inference	Heuristic thresholds
MC Dropout [154]	Probabilistic approx.	Stochastic forward passes	Various	Uncertainty estimation	Extra inference cost
Deep ensembles [155]	Model ensemble	Multiple initialisations	Various	Uncertainty estimation	Training/inference cost
Confidence heads [156]	Aux. confidence nets	Learned confidence output	App-specific	Robust deployment	Arch.-coupled
CP [157, 158]	Model-agnostic CP	Nonconformity scores	Various	Reliable prediction	Real-time complexity
This work	CP + ML pipeline	Static rep.	Code tasks	Deployment robustness	–

3.4.1 Design-Time Robustness Techniques

To enhance ML robustness during the design phase, researchers have explored various techniques such as data augmentation through code synthesis [152], learning robust program representations [8, 9], and tuning ML model architectures for better generalization [2]. These methods contribute to the robustness of ML models by ensuring that the training data reflects a broader range of potential scenarios, improving generalisation to unseen data.

Data augmentation through code synthesis [152] focuses on generating additional training samples by synthetically creating new code examples. By artificially expanding the dataset, this technique exposes ML models to a more diverse set of inputs, helping them better generalise to edge cases and rare scenarios in real-world applications. The synthesis of code examples, often done using program synthesis techniques, can cover underrepresented programming constructs and patterns, thus increasing the resilience of models to unusual or unseen code during deployment.

Learning robust program representations [8, 9] has also been a key focus for enhancing robustness. Cummins et al. [8] introduced ProgramL, a machine learning framework that encodes programs as graph-based representations, enabling models to learn from program structures rather than purely from source code syntax. This shift from text-based to graph-based representations captures semantic relationships between code elements, allowing models to better understand and generalise across different coding styles and paradigms. Similarly, Wang et al. [9] emphasized the importance of learning more abstract, high-level representations of programs that are invariant to minor code changes, such as variable renaming or formatting differences. This approach improves model robustness by ensuring that small, irrelevant code changes do not significantly impact predictions.

Another important strategy for improving ML robustness is the tuning of model architectures for better generalization [2]. Wang et al. [2] demonstrated how careful tuning of neural network architectures-such as adjusting layers, activation functions, and hyperparameters-can lead to better generalization across unseen data. Automated architecture search techniques can also be employed to systematically explore a wide range of model configurations, selecting the ones that perform best across different validation datasets. This reduces the likelihood of overfitting and enhances the robustness of models in varying operational environments.

Together, these design-time techniques-data augmentation through code synthesis, learning robust representations, and architecture tuning-help ensure that ML models can handle a diverse range of inputs and remain reliable even in the face of unexpected scenarios.

While design-time approaches are essential for training robust models, they do not fully address challenges that arise in real-world deployment. In deployment environments, unforeseen changes in data distribution can occur, resulting in performance degradation despite robust training methods. Therefore, there is a need for approaches that extend robustness into the deployment phase.

3.4.2 Deployment-Time Robustness Solutions

Our method complements design-time efforts by focusing on improving robustness at deployment without altering the model architecture. Unlike traditional approaches that rely on collecting representative data from the operational environment before deployment [159], our tool assesses prediction reliability in real-time. This eliminates the need for extensive data collection, offering a scalable solution for dynamic environments.

Several recent works have proposed methods for quantifying prediction uncertainty to improve robustness. Common techniques include the use of entropy and mutual information to measure prediction confidence [153], or training DNNs to estimate confidence for specific applications [156]. These approaches aim to provide insights into the reliability of model predictions, allowing systems to act accordingly in uncertain scenarios and improving overall robustness.

Entropy-based methods calculate uncertainty by examining the distribution of predicted probabilities across all possible classes. In this context, higher entropy indicates greater uncertainty in the model's predictions, suggesting lower confidence in the output. Mutual information, on the other hand, quantifies how much uncertainty is reduced given additional information from the input data, effectively capturing the confidence of the model in its predictions based on both the data and model output [153]. These methods are widely applied in domains such as autonomous driving and medical diagnostics, where it is critical to know when the model is uncertain and could potentially make incorrect predictions.

Other approaches involve training DNNs specifically to estimate prediction confidence. For instance, researchers have designed network architectures that include auxiliary outputs explicitly trained to predict confidence scores alongside the main prediction task. These confidence scores are typically learned during training and help flag predictions with low confidence that may require human intervention or additional processing before being finalized [156]. Such methods have been deployed in safety-critical applications, such as self-driving cars or robotic systems, where mispredictions can lead to serious consequences if not caught early. These techniques, however, are tightly coupled with the specific DNN architectures used, limiting their generalizability to other model types.

To mitigate these limitations, recent advancements have explored model-agnostic techniques for uncertainty estimation, which can be applied across different machine learning models. One such technique is Monte Carlo dropout [154], where dropout is used at inference time to approximate Bayesian inference by sampling from the network multiple times. This results in an empirical estimation of uncertainty by observing the variability in predictions across different samples. While effective, Monte Carlo dropout is primarily tailored for neural networks and assumes the use of dropout layers during training.

Another model-agnostic approach is deep ensembles [155], which involves training multiple models with the same architecture but different initialisations and aggregating their predictions to estimate uncertainty. The variability between the outputs of these models gives a measure of the model's uncertainty, with more divergent predictions indicating greater uncertainty. This method can be applied to a variety of machine learning models, making it more flexible than techniques tied to specific architectures.

3.4.3 Conformal Prediction and Anomaly Detection

At its core, we integrate concepts from anomaly detection [39, 40, 160] and adversarial attack analysis [161] to detect mispredictions resulting from changes in input characteristics. By incorporating CP, our method ensures confident predictions while rejecting unreliable ones before they affect the system. This approach provides real-time assurance of prediction reliability, addressing the challenges posed by unpredictable data shifts during deployment.

Our work builds on existing CP methods, such as MAPIE [157] and PUNCC [158], but introduces key innovations. While traditional CP techniques use full calibration datasets and single nonconformity measures, our tool employs adaptive weighting and multiple nonconformity functions to enhance its ability to detect mispredictions. Additionally, we extend CP beyond its typical application in classification to support regression tasks, expanding its applicability across various problem domains.

By enhancing prediction reliability during deployment and extending CP to multiple use cases, our method offers a robust and scalable solution for improving ML model performance in dynamic environments. Its model-agnostic design and real-time reliability assessments make it an attractive option for addressing the inherent unpredictability of deployment environments, setting it apart from existing robustness techniques.

3.5 Position

Research on applying machine learning to software engineering and compiler optimisation has developed along several lines. Early efforts in bug detection focused on static or dynamic program analysis, relying heavily on rule-based heuristics or execution-driven testing. While effective in specific contexts, such approaches often lack adaptability to unseen bugs. More recently, ML-based methods have been proposed, which leverage program features to automatically learn patterns of faulty behaviour. However, many of these approaches consider static or dynamic information in isolation, limiting their generalisability.

In parallel, compiler optimisation has increasingly adopted machine learning, particularly reinforcement learning (RL) and deep reinforcement learning (DRL), to automate the search for optimisation strategies. Systems such as OpenTuner and AutoTVM demonstrate that RL agents can outperform traditional heuristic-based approaches. Nevertheless, such systems often remain difficult for developers to adopt, due to their complexity and lack of developer-oriented abstractions.

Finally, research into the reliability of ML models has emphasised uncertainty estimation, with methods such as Monte Carlo dropout, Bayesian neural networks, and ensemble learning. These approaches have been widely applied in safety-critical settings, but their computational overhead and limited adaptation to software engineering tasks restrict their broader applicability.

Table 3.1, 3.2 and 3.3 summarise representative prior work across these three research areas, contrasting them with the contributions of this thesis. The comparison is based on the underlying approach, application domain, and limitations, thereby positioning this research within the state of the art.

3.6 Summary

This chapter has surveyed the relevant literature in the field of practical ML for program modelling, with a particular focus on code-related tasks.

Chapter 4

Combining Structured Static Code Information and Dynamic Symbolic Traces for Program Modelling

This chapter presents a program modelling method designed to better represent programs, which is an important step toward improving ML performance. We use vulnerability detection as a case study to demonstrate the method's effectiveness on code-related tasks. The chapter is organised as follows. Section 4.2 highlights the limitations of program modelling methods that rely solely on static information. Section 4.3 explains how our approach is formulated. Section 4.4 then describes the experimental setup. Section 4.5 presents the experimental results, before Section 4.6 concludes the chapter.

4.1 Introduction

One of the most important challenges in applying ML to code is how to represent programs better. The success of a machine learning model is heavily dependent on having a suitable representation of the problem domain that can encode the essential information needed for the task at hand, such as vulnerability detection in our case [14]. In the context of DL-based code modelling, this requires constructing numerical vectors, or *embeddings*, that capture the important characteristics of the program source code or binary.

The vast majority of DL-based code modelling techniques rely on DNNs to learn program representations from static code, such as source code texts [47–49], ASTs

[76, 77], PDCGs [18, 50, 51], or a combination of these [9]. While static code information can capture all possible program execution paths, it can suffer from complex and ambiguous information due to redundant statements, complex data structures, and extensive execution paths in the source code. Like classical compiler analysis, this can lead to over-conservative decisions and a high false-positive rate[1], and a low true positive ratio [23] for automatic bug detection.

More recent approaches, like LIGER [24], attempted to use dynamic execution traces to learn program representation. These approaches utilise execution statements seen during profiling to represent static program information and track changes in program variables to capture dynamic program behaviour. By considering dynamic execution paths, symbolic traces provide precise information about dynamic program behaviour and reduce false-positive rates in code analysis. While promising, prior approaches have two limitations. Firstly, they solely rely on executed code statements seen for static code representation. This can suffer from poor coverage and overlook the structured data flow and dependence information available in the static program graph. This comprehensive data and control flow information is crucial for vulnerability detection, encompassing all possible execution paths. Secondly, they employ random sampling for dynamic tracing, which presents challenges when applying dynamic tracing methods to real-world software projects due to the expensive overhead of symbolic executions [162].

In this chapter, we ask the question, "*what if we could bring the best of static and dynamic code information in a single DL framework for code vulnerability detection?*". In response, we develop Concoction, a new DL system to combine static and dynamic code representation to detect software bugs and vulnerabilities at the source code level. Specifically, static code information, such as PDCG, offers a high-level view of all possible program behaviours and data flow information, which can mitigate the coverage issue of symbolic execution. On the other hand, symbolic execution traces on *a small set* of carefully selected execution paths can provide more precise information and deeper program semantics to disambiguate static code information. By integrating these two types of information, we avoid the computational overhead of running symbolic executions on every possible execution path while still leveraging the benefits of

[1] *False positive* occurs when the code does not contain a bug or vulnerability, but the detection model indicates otherwise. By contrast, a *false negative* occurs when the model fails to identify a true bug or vulnerability.

deep program semantics provided by dynamic executions.

CONCOCTION utilises a Transformer-based DNN architecture [59] to leverage static and dynamic code information. The model comprises a *representation component* and a *detection component*. The representation component maps static and dynamic program information into a joint embedding (or feature vector) for program representations. The detection component takes the program representation as input and predicts whether the input code contains a vulnerability. One of the key features of CONCOCTION is its ability to minimise the overhead of dynamic tracing. This is achieved by employing a path selection component during deployment to determine which execution paths from the PDCG should be chosen to collect symbolic execution traces. These traces are then fed into the representation component to generate dynamic embeddings, which are combined with the static embeddings and passed to the detection model for vulnerability prediction.

To overcome the challenge of limited numbers of labelled code samples, CONCOCTION combines supervised and unsupervised learning techniques. We first leverage unsupervised contrastive learning to pretrain the representation network. For this purpose, we employ the language masking method [163] and train the representation model on a dataset of 100K unlabeled C functions sourced from GitHub and open datasets. To generate additional training samples, we introduce a dropout-based contrastive learning component [164]. The contrastive loss function encourages the model to understand code semantics better by mapping similar samples closely and pushing dissimilar ones farther apart in the embedding space. Furthermore, we also extend unsupervised learning to train the path selection component using the same unlabeled training dataset. Once the representation component is trained, we remove the contrastive learning network and combine the representation layers with the detection component, creating an end-to-end model. This model takes joint embeddings of static and dynamic information as input and predicts whether the input code contains a bug. To train the end-to-end model, we use the learned weights of the representation model to initialize the corresponding layers of the model and fine-tune the entire architecture on a dataset of 14K labeled code samples obtained from public datasets, including the Software Assurance Reference Dataset (SARD) [165] and CVE [166].

We have implemented a working prototype of CONCOCTION[1]. Our implementa-

[1]Code and data are available at `https://github.com/HuantWang/CONCOCTION`.

```
1  extern void foo(void);
2  static int a[] = {0,0};
3  static int b(int c)
4  { return a[1] % c; } // false positive
5
6  int test() {
7      static int *a_ptr, c;
8      a_ptr = &(a[1]);
9      if (*a_ptr) { // unreachable branch
10         if (b(c)) {foo();}
11     }
12     return 0;}
```

Figure 4.1: This example contains a false "*division-by-zero*" issue at line 4 because the branch at line 10 will not be taken.

tion utilises KLEE [90] to generate the symbolic execution traces. We demonstrate the benefits of Concoction by applying it to C programs to detect function-level vulnerabilities from source code. We further integrate Concoction with fuzzing test techniques [167] to automatically generate bug-exposing test cases when a function is predicted to have a code vulnerability, aiming to minimise the effort of manual examination.

We evaluate Concoction by applying it to 20 diverse, real-life open-source projects that are not presented in our training dataset. We compare Concoction against 16 prior methods, including eleven state-of-the-art learning-based methods [9, 18, 24, 47, 49–51, 78, 106, 147, 168], two symbolic execution engines [90, 169], two static analysis tools [170, 171] and a fuzzing tool [80] for identifying security flaws. Experimental results show that Concoction consistently outperforms 16 competing methods across evaluation settings by discovering more code vulnerabilities with a lower false-positive rate. In less than 200 hours of automated concurrent testing runs, Concoction has uncovered vulnerabilities in all tested projects and successfully identified 54 software vulnerabilities, with 37 new CVE IDs assigned.

4.2 Motivation

Static code analysis techniques for bug detection can suffer from false positives (incorrectly flagging a bug). For example, Figure 4.1 shows a function `b()` which will not be invoked during execution because the static array `a` is initialised to zero by definition, and the condition at line 9 is evaluated to false.

However, in practice, this task is challenging even for modern compilers like GCC [172] and LLVM [173]. The difficulty stems from pointer aliasing analysis. Compilers often make conservative assumptions when they cannot definitively prove the value a pointer holds. Consequently, they fail to eliminate this unreachable branch. Similarly, DL-based bug detection approaches [9, 18, 47–51, 106] that rely on static code features predict that this example contains a "*division-by-zero*" vulnerability at line 4, which is a false positive.

Avoiding a *false positive* of Figure 4.1 would require capturing program semantics between variables, function calls and structured data. An approach that uses only static information may not accurately trace the data flow across function calls and data structures. Meanwhile, DL-based solutions based on static code information, such as AST or PDCG, suffer from the same issue.

Can we do better by combining static and dynamic information? This is the insight shared in this chapter. For this example, we can infer from symbolic execution traces that the function `b()` will not be executed due to the values in array a by inlining the callee function `b()` to `test()`. The static PDCG further reveals that `a` is an invariant in all possible execution paths. Combining static and dynamic information, we can observe that this branch is never taken, making a "*division-by-zero*" error impossible.

A natural question is: "*why not just rely on symbolic executions?*". In an ideal world where computation resources and symbolic execution overhead are not an issue, a symbolic execution engine will be able to identify the vulnerability of this example through exhaustive executions. However, this is often infeasible because exhaustively trying all possible execution paths is prohibitively expensive and potentially infinite. This example highlights the need to leverage static and dynamic program information for code vulnerability detection. Concoction is designed to offer this capability.

4.3 Concoction: A System for Precise ML for Program Modelling

Concoction is a DL framework for detecting software vulnerabilities in source code. In this work, we apply Concoction to identify bugs at the function level in C pro-

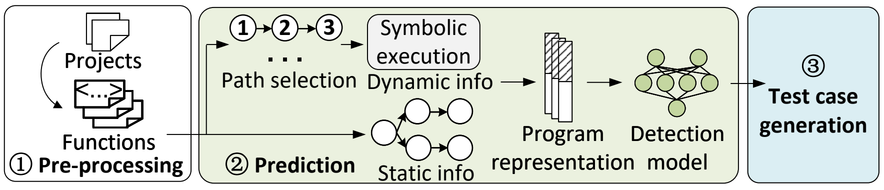


Figure 4.2: Workflow of Concoction during deployment.

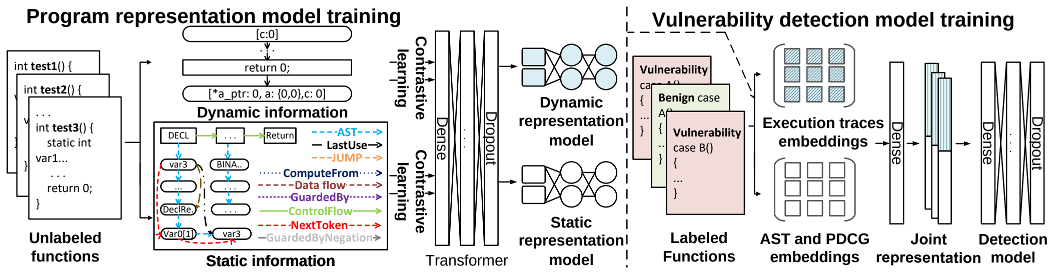


Figure 4.3: The Concoction DNN architecture and its training workflow.

grams. Specifically, our work focuses on detecting bugs and security flaws defined in the Common Weakness Enumeration (CWE) database [174]. In practice, Concoction can be integrated into an automated build system like Jenkins [175] to execute the vulnerability detection process as a background process when a new merge request is submitted. As these build systems often run overnight on a dedicated backend server, they do not affect the standard development activities and the overhead of Concoction should be acceptable to many developers.

The *key technical contributions* of Concoction include: (1) combining structured static code information and symbolic execution traces to learn program representation (Sec. 4.3.2), and (2) a learnable path selection component to reduce symbolic execution overhead (Sec. 4.3.4). Concoction builds upon prior foundations in enhanced AST, Transformer-based neural architectures, and contrastive learning (Sec. 4.3.3).

4.3.1 Overview of Concoction

Figure 4.2 depicts the workflow of using Concoction to detect function-level code vulnerabilities *during deployment*.

Pre-processing. Concoction uses a LLVM compiler plugin [173] to partition the project code into individual functions by inlining callee functions, relevant data structures, and global variables.

Prediction. Concoction extracts two types of information for each target function: (1) AST and PDCG from static source code, and (2) the symbolic execution traces of selected execution paths using a symbolic execution engine [90]. It employs a path selection component (Sec. 4.3.4) to identify critical paths to collect dynamic symbolic execution traces. The static and dynamic information produce static and dynamic embeddings through dedicated representation networks, which are then concatenated to create a joint representation to be used by the detection model for prediction.

Test case generation. When the model detects a potential vulnerability in the input function, it invokes a fuzzing engine to identify and expose the weakness by generating randomised test inputs for the function. As we only employ fuzzing for functions suspected to contain vulnerabilities, the fuzzing overhead is manageable, taking less than 12 hours for all fuzzed functions within a project.

4.3.2 The Concoction Architecture

Figure 4.3 shows the workflow of training the Concoction DL components for program representations and vulnerability detection.

Program representation. Our representation component uses two Transformer-based networks to map the input source code and symbolic execution traces into a numerical embedding vector. Then, a dense layer concatenates the embeddings generated by the two networks to a joint vector as the output. We set the embedding length of the static and dynamic embedding vectors to 100 dimensions, leading to a joint embedding vector of 200 dimensions. As in prior work [176], using a larger dimension does not yield better performance in our setting but may increase the training overhead.

Vulnerability detection. The detection component is a multi-layer perceptron (MLP) network that takes a joint embedding to predict the vulnerability. The architecture consists of a fully connected layer, a dropout layer with a rate of 0.1 using the default settings, and a sigmoid layer. Our current implementation only predicts if a function may contain a vulnerability and does not identify the type of vulnerability.

Extracting static code information

We use a parser built upon the Language Server Protocol (LSP) [177] to rewrite the variable names with a consistent naming scheme. This step handles syntactic variations in the programs. Next, we construct an enhanced AST, using the LSP, which contains the standard *syntax nodes*, i.e., nonterminals in the language grammar like an AST node for an `if` statement or function declaration, and *syntax tokens*, i.e., terminals like identifier names and constant values. Following [178], we also introduce eight additional types of edges to the AST, including *Child*, *Data and Control Flows*, *GuardedBy*, *Jump*, *ComputedFrom*, *NextToken* and *LastUse and LastLexicalUse*, shown in Figure 4.3.

Extracting dynamic information

We use KLEE [90] to obtain symbolic execution traces. To pre-train the representation model (Sec. 4.3.3), we generate execution traces using a non-uniform random search heuristic to explore different execution paths. During deployment, symbolic traces are generated solely on the selected paths instead of a random search to minimise the overhead of symbolic executions (Sec. 4.3.4). We terminate symbolic execution after a configurable time limit (4 hours for collecting training data and 5 minutes when using the trained model). Subsequently, we combine different symbolic inputs and their corresponding reachable program paths as a sequence of execution traces to be fed into the dynamic embedding network (Figure 4.3).

4.3.3 Contrastive Pre-training

We use a bidirectional Transformer network [106] to learn static and dynamic program embeddings. We pre-train the static and dynamic embedding models separately on the same unlabeled dataset. Our pre-training dataset contains 100K C code snippets collected from GitHub and SARD, shown as Table 4.1. After training, we use the output of the last hidden layer of the embedding networks as the static or dynamic embedding vector. We employ contrastive learning to increase the dataset size and enhance the model's robustness.

Model inputs

We pair the source code text and flattened enhanced AST sequence, sending them to the static embedding network. The AST sequence is generated by traversing the AST in a breadth-first manner. During training, the static embedding model predicts masked tokens from either the source code or AST's data and control flow relations to generate contextual representations. Likewise, the dynamic embedding network takes

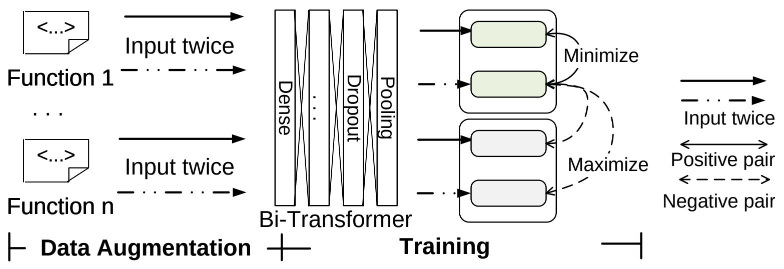


Figure 4.4: Contrastive learning for representation training.

symbolic execution traces and maps them to an embedding vector, capturing temporal dependencies and runtime behaviour of the program.

Training methodology

To enhance the model's robustness and generalisation ability, we employ dropout-based contrastive learning [164]. Our approach includes dense, dropout, and pooling layers, depicted in Figure 4.4. Specifically, we individually attach the contrastive learning component to the static and dynamic representation networks. Then, we train each combined network separately and remove them from the contrastive learning component.

Contrastive learning increases the training dataset size by adding noise to the data. Concoction randomly disables neurons in the representation network, generating various dropouts as shown in Figure 4.4. Specifically, it passes the code sample inputs x_i through the representation network twice with different dropout probabilities, resulting in two different embeddings as a "positive pair" for x_i. Another sample is paired with x_i to create a "negative pair". The contrastive learning component is then used to predict positive samples from negatives in a training mini-batch and calculate the loss. The training process aims to minimise the standard Noise Contrastive Estimate (NCE) loss function [104] to maximise the agreement between semantically similar pairs.

Training the end-to-end detection model

After training the static and dynamic embedding networks, we attach them to a dense layer to create joint embeddings for the detection network. This forms the final end-to-end architecture shown in the right part of Figure 4.3. The joint representation component serves as the encoder, and we initialise its weights using those obtained during pre-training. We train the end-to-end network using *labeled* data samples, which

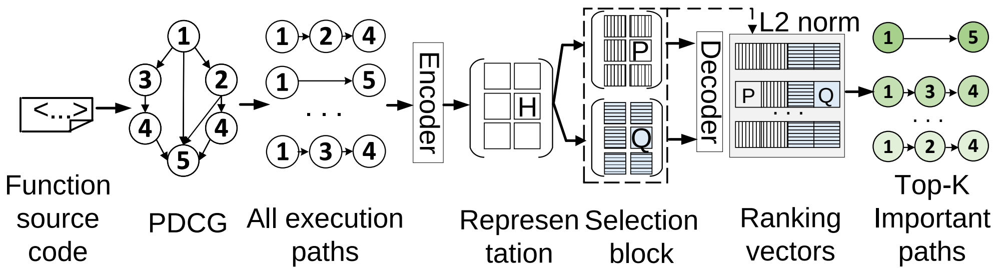


Figure 4.5: Rank and select paths for symbolic execution.

consist of 13,768 C code snippets from the CVE and SARD datasets. Each sample has a two-dimensional one-hot label indicating whether it contains a vulnerability. Vulnerable code samples are collected from open-source projects using the assigned CVE or SARD IDs, whereas benign samples are obtained from the patched version of the same project. For each training sample, we generate an enhanced AST and randomly sample symbolic traces (Sec. 4.3.2). We use the *pre-trained* representation component to generate the joint embedding as the program representation, which becomes the detection model's input. Our end-to-end model is trained to optimise the cross-entropy loss for classifications.

4.3.4 Path Selection for Symbolic Execution

After training the end-to-end model, we use the path selection component to choose significant paths for symbolic executions *during deployment*. This differs from execution traces collected during training, where we use a random sampling scheme to improve the training data size, as symbolic execution overhead is less of an issue for offline model training.

Overview of path selection

Figure 4.5 shows the workflow of choosing k most important paths from all possible execution paths of the target function. Like ContraFlow [78], we use unsupervised active learning to identify most representative paths for encoding programs [179], such that the number of paths for code embedding is reduced while important program semantics are well-preserved. Unlike ContraFlow, our goal is to select paths to collect symbolic execution traces. As we will show in Sec. 4.5.2 and 4.5.3, our approach outperforms ContraFlow.

Our goal. Given n execution paths $\boldsymbol{H} = [h_1, ..., h_n]$ collected from the PDCG of the test sample, our goal is to choose a subset of important paths to collect symbolic execution traces, where h_i is an embedding vector generated by our static representation model. The objective is to choose representative data points that can reconstruct most of the input PDCG information and preserve the unique features of the sample. In this work, we use the K-means clustering algorithm to model the path features by grouping data points into clusters in the embedding space. The path selection component is trained to find a sample subset that captures the input patterns and preserves the cluster structure of the data.

Execution path representation

For each test sample (function), we extract static execution paths from the PDCG. For each static execution path, we first eliminate irrelevant code (and AST nodes) and then feed the code segmentation to the trained static representation model (fine-tuned when training the end-to-end model) to generate embeddings for the code segments. This process is repeated for each static path, resulting in a matrix, $\boldsymbol{H}$, to serve as the input of the path selection component, where each matrix element is an embedding vector produced by our static representation model. Since path collection involves traversing the PDCG without program execution, the overhead is negligible.

Network structure

As our path selection component uses unsupervised learning (we do not have labels to tag if a path is important), using an encoder-decoder network to map the input into a latent space for path selection is a natural choice. We add a *selection block* between the encoder and decoder to select samples (or paths) from all input paths to be passed into the decoder. As the selection block is differentiable, the selection network is trainable.

Selection block. The selection block comprises two branches, each consisting of a single fully connected layer without bias and nonlinear activation functions. The first branch aims to identify a subset of paths ($\boldsymbol{H'}$) that can effectively approximate all paths from the input matrix ($\boldsymbol{H}$). The second branch initially clusters the data in the latent space and then selects a sample subset approximating the resulting cluster centroids.

Unsupervised active learning

For all input paths of a test sample, our approach constructs two coefficient matrices, $\boldsymbol{Q}$ and $\boldsymbol{P}$, where each matrix element is a d-dimensional embedding vector produced by the encoder. The matrix $\boldsymbol{Q}$ is constructed by the first branch of the selection block to approximate the input matrix ($\boldsymbol{H}$), while the matrix $\boldsymbol{P}$, constructed by the second branch of the selection block, aims to preserve the cluster structure when applying K-means to the latent space learned by the encoder.

Training objectives. Our active learning process refines the matrices to maximize the distance between (1) $\boldsymbol{H}$ and matrix $\boldsymbol{Q}$, (2) the cluster centroid matrix $\boldsymbol{C}$ obtained using K-means clustering on $\boldsymbol{H}$, and the reconstructed matrix $\boldsymbol{P}$, and (3) $\boldsymbol{H}$ and the decoder outputs.

Loss function. Our overall loss function is defined as $Min\ \ell = \alpha\ell_a + \beta\ell_b + \ell_c$, where α and β are tradeoff parameters [179]. The terms in the loss function are:

$$\ell_a = ||\phi(H) - \phi(H)Q||_F^2 + \gamma||Q||_2 \tag{4.1}$$

$$\ell_\mathrm{b} = ||C - \phi(H)P||_F^2 + \eta||P||_2 \tag{4.2}$$

$$\ell_c = ||\phi(H) - G||_F^2 \tag{4.3}$$

Eq. 4.1 , ℓ_a: Approximates input patterns by projecting latent features $\phi(H)$ with Q, with regularisation on Q controlled by γ, corresponding to objective (1). The value of ε represents either 2 (the ℓ_2 norm) or F (the Frobenius norm) for the normalisation function $||\cdot||_\varepsilon$, which is used to measure the informativeness of each feature. $\phi(H)$ is a nonlinear transformation that maps input paths H to a new latent representation, and the tradeoff parameter γ controls the balance between the reconstruction loss and the regularisation term.

Eq. 4.2 , ℓ_b: Represents the cluster reconstruction loss, aligning latent features with centroids C, with regularisation on P controlled by η. This corresponds to objective (2), with the tradeoff parameter η.

Eq. 4.3 , ℓ_c: Denotes the decoder reconstruction loss, enforcing $\phi(H)$ to reconstruct back to G. This corresponds to the reconstruction loss of the encoderâ€"decoder model, which represents the third objective. $\boldsymbol{G}$ denotes the decoderâ€™s output for a given input $\phi(H)$.

Training process. We iteratively train the selection component on the unlabeled CONCOCTION training data. Firstly, we pre-train the encoder and decoder without considering the selection block. After that, we perform K-means on the encoder output and consider the obtained K cluster centroids as the centroid matrix $\boldsymbol{C}$ for subsequent sample selection. The number of clusters (K) is determined automatically using the Bayesian information criterion (BIC) [180]. Finally, we use the pre-trained parameters to initialise the encoder and decoder, batch all data, and minimise the overall loss function using the Adam optimiser with a 0.001 learning rate.

Path selection during deployment

Once the selection component is trained on the CONCOCTION training data, we can obtain two reconstruction coefficient matrices $\boldsymbol{Q}$ and $\boldsymbol{P}$. To use the selection component, we normalise the columns of $\boldsymbol{Q}$ and $\boldsymbol{P}$ using L2-norm and convert the values to $[0, 1]$. This produces two ranking vectors $\hat{q}, \hat{p} \in \mathbb{R}^n$, which we merge and sort in descending order to identify K top-ranked paths. Parameter K can be flexibly set by the user. In this work, we set K to be 30%, sufficient to cover important paths of vulnerable functions in our *training* dataset. If the number of paths of the target code is less than 10, we consider all execution paths as the overhead of symbolic executions is small.

Symbolic execution for chosen paths

We extended KLEE [90] to cover the selected paths of the target function. To do so, we use a compiler-based pass to insert a callback function into the target program to guide KLEE to skip paths not selected by our path selection component. We also use a script to record the addresses, sizes, and names of all variables of the target function during symbolic execution. The script also generates a test driver program for KLEE to facilitate the execution of KLEE.

By skipping unwanted execution traces early, we can manage the overhead of symbolic execution effectively. We terminate symbolic execution after a configurable threshold (5 minutes in this work) during evaluation. The symbolic inputs and corresponding reachable code paths produce execution traces that we pass to the dynamic representation model to generate dynamic embeddings. Note that our approach guarantees that there are always symbolic execution traces generated for the test function.

Targeted fuzzing for test case generation

When our detection model identifies a potentially buggy function that symbolic execution fails to expose, we use AFL++ [80] (an AFL extension) for fuzzing functions predicted to be vulnerable. The idea is to generate bug-exposing test cases to help developers analyse and verify the identified vulnerabilities. In this work, we use the AFL++ partial instrumentation mode to guide AFL++ towards targeting functions predicted by CONCOCTION to have vulnerabilities. To this end, we provide AFL++ with the project's source code, build scripts, vulnerable functions identified by CONCOCTION, and seed program inputs to be mutated using the default AFL++ configurations. We then ask AFL++ to instrument the compiled assembly code and mutate the seed test inputs to cover the target functions. Fuzzing is terminated if AFL++ detects a program crash or exceeds a 12-hour runtime (which may involve fuzzing multiple functions together). We then manually verified and reported the issue to developers with information to reproduce the issue. If AFL++ does not trigger any crash, we manually examine the predicted function to check for a vulnerability and file an issue report for each confirmed bug.

4.4 Experimental Methodology

We implemented CONCOCTION in 10+K lines of Python and 5K lines of C/C++ code. Our DL model is implemented using PyTorch (ver. 1.13). We use Joern (ver. 1.1) [181] to construct the AST for static code representation and KLEE (ver. 2.1) to collect the symbolic execution traces. We train and test CONCOCTION models and all baselines on a multi-core server with two 32-core AMD EPYC 7532 CPUs at 2.40GHz and an NVIDIA 2080Ti GPU. The server has 128GB of RAM and runs Ubuntu 18.04 with the Linux kernel version. 5.4.

4.4.1 Workloads

Open-source projects. We applied CONCOCTION to 20 open-source projects from various domains. These projects, listed in Table 4.3, were chosen because they are widely used, have been used in related work [182], or have active development teams. We stress that none of these projects was used to train CONCOCTION, and we tested the latest version of each project at the time of testing. We also note that bugs discovered

Table 4.1: Open datasets used in training and evaluation

Source	#Projects	Versions	#Samples	#Vulun. samples
SARD	/	/	30,954	5,477
CVE	Jasper	v1.900.1-5, v2.0.12	24,330	663
	Libtiff	v4.0.3-9	4,896	558
	Libzip	v0.10, v1.2.0	5,618	49
	Libyaml	v0.1.4	27,773	41
	SQLite	v3.8.2	1,794	31
	ok-file-fromats	203defd	1,014	25
	libpng	v1.2.7, v1.5.4, v1.6.0	954	12
	libming	v0.4.7-8	1,104	15
	libexpat	v2.0.1	1,051	13

Table 4.2: Open-source projects with known CVEs

No.	Project	Versions	#Lines of code	#vuln.
1	SQLite	v3.30.1, v3.8.2	242K	13
2	Libtiff	v4.0.9	140K	10
3	Libpng	v1.2.7, v1.5.4	32K	12

by Concoction were previously unreported at the time of testing; hence, there were no data leakage issues.

Open datasets. In Sec. 4.5.2 and 4.5.3, we compare Concoction with prior work on three datasets used by the prior work for evaluation purposes. In Sec. 4.5.2, we use samples from SARD [165] and CVE datasets (see Table 4.1). In Sec. 4.5.3, we evaluate Concoction on three open-source projects that have known CVEs (see Table 4.2). We use three-fold cross-validation to evaluate all approaches on the above datasets, where samples are split at the project level, and samples of a test project are excluded from the training dataset.

Data collection and workload characteristics. Our program representation model,

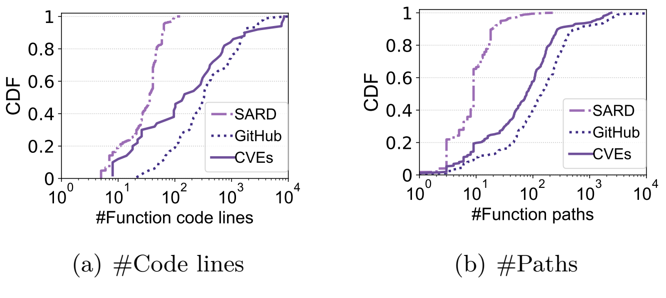


Figure 4.6: Cumulative Distribution Function (CDF) for the (log-scale) number of lines (a) and execution paths (b) of our test samples.

Concoction, was trained and tested on a dataset of over 100K functions from SARD and 9 large C-language open-source projects (Table 4.1). Four security researchers, Jie Wang, Hejun Fang, Yuzhe Liu, and Rongze Xu, devoted approximately 600 man-hours to manual labeling and cross-verification to collect these samples. Additionally, we spent 200+ machine hours extracting dynamic and static information using KLEE. Figure 4.6 shows the cumulative distribution functions (CDF) of the number of lines and execution paths in the test samples in Tables 4.1 and 4.3. The SARD dataset consists mainly of short functions, where over 50% have <40 lines of code and 4 paths, leading to high detection accuracy with baseline methods. However, the functions from the CVE dataset and open-source projects are much larger, with over 50% containing $\geq$ 400 lines of code (up to 10K) and 128 paths (up to 12K). This increased complexity in the CVE dataset and open-source projects reduces the accuracy and recall for our baselines compared to SARD.

4.4.2 Competing Baselines

We evaluate Concoction by comparing it with 16 prior methods. These include (1) eleven state-of-the-art DL-based models, (2) two symbolic execution tools, (3) one fuzzing tool that our approach relies on, and (4) two static analysis tools. Before running the DL baselines on the same datasets, we ensure that our evaluation setup achieves results comparable to those reported in their source publications for a fair comparison.

DL models based on static information. We compared Concoction against eleven DL models that use static code information. These include Vuldeepecker [47], which utilises a BiLSTM architecture, as well as Funded [9], Devign [18], ReVeal [50], and ReGVD [51], which employ a variant of graph neural networks to learn program representations. We also compared against LineVul [147], LineVD [168], CodeXGLUE [49] and GraphcodeBERT [106], which use the Transformer architecture, and ContraFlow [78] which utilises contrastive learning to represent the code, followed by an LSTM architecture to identify vulnerabilities.

DL models based on dynamic information. LIGER [24] is a closely related work that learns program representations from symbolic execution traces. However, unlike our approach, LIGER uses a random sampling method for collecting symbolic traces and does not utilise structured data flow and dependence information from static source code. Additionally, it employs a multi-tier RNN architecture, whereas CONCOCTION uses the Transformer architecture. For our evaluation, we used the open-source implementation of LIGER, adapting it for vulnerability detection and training it on the same datasets as CONCOCTION.

Static tools. We compare CONCOCTION with two representative static analysis tools: CODEQL [170] and INFER [171], by using the default, recommended configurations of the tools.

Symbolic execution engines. We also compare CONCOCTION to two state-of-the-art symbolic execution tools, including KLEE [90] and MOKLEE [169]. The latter is designed to reduce the overhead of symbolic executions by allowing symbolic executions to run on previous paths while continuing to explore new paths.

Fuzzing tool. As noted in Sec. 4.3.4, CONCOCTION uses AFL++ to fuzz predicted buggy functions and generate bug-exposing tests. We compare CONCOCTION with the native AFL++, which tests the entire program without CONCOCTION's guidance. For a fair comparison, both AFL++ and CONCOCTION use the same seed program inputs.

4.4.3 Evaluation Methodology

We applied CONCOCTION to 20 open-source projects and 14K function-level source code samples (vulnerable and benign). Our evaluation is designed to answer the following research questions:

RQ1: Does combining static and dynamic information help detect code vulnerabilities in real-life open-source projects (Sec 4.5.1)?

RQ2: How does CONCOCTION compare with prior approaches in detecting function-level vulnerabilities (Sec. 4.5.2 and 4.5.3)?

RQ3: How do individual components of CONCOCTION contribute to its overall performance (Sec. 4.5.4)?

Evaluation metrics. We consider four *higher-is-better* statistical metrics: *accuracy*, *precision*, *recall* and the *F1 score*. Accuracy is computed as the ratio of correctly labelled cases to the total test cases. Precision is the ratio of correctly predicted samples to the total number of samples predicted to have the same label. It answers the question, "*Out of all the samples predicted to contain a vulnerability, how many are correct?*" High precision indicates a low *false-positive* rate, meaning that a lower proportion of the samples predicted to have bugs are bug-free. Recall is the ratio of correctly predicted samples to the total number of test samples belonging to a class. It answers questions like "*Of all the vulnerable test samples, how many are actually predicted to be vulnerable?*". High recall suggests a low *false-negative* rate. Finally, the F1 score is the harmonic mean of Precision and Recall, calculated as $2 \times \frac{Recall \times Precision}{Recall + Precision}$. It is useful when the test data has an uneven label distribution.

4.5 Experimental Results

4.5.1 Detect Vulnerabilities in Large-scale Testing

This subsection quantifies CONCOCTION's ability to detect function-level code vulnerabilities in the 20 projects listed in Table 4.3. For ethical considerations, we first contacted the developers through a private email for vulnerabilities that are likely exploitable (including all those with a CVE ID assigned), and followed their advice.

Vulnerability count

Table 4.3 reports the distribution of our submitted vulnerability reports across the tested projects. In total, we have submitted 54 reports, and 53 were confirmed by developers. At the time of submission, 27 vulnerabilities have been fixed, with 37 new, unique CVE IDs assigned and 17 CVE applications pending.

Vulnerability types

Table 4.4 categorises the vulnerabilities found by CONCOCTION[1]. The top three security flaw-related categories are presented here. The "other types" category includes six types of vulnerabilities: 'allocation-size-too-big', 'out-of-memory', 'use-after-free', 'memcpy-param-overlap', 'illegal-memory-access', and 'DEADLYSIGNAL'. Of all the detected vulnerabilities, 62.3% are *buffer-overflow* related, covering both *heap* and *stack-buffer-overflow*. CONCOCTION's ability to detect buffer-overflow vulnerabilities comes from its

[1]A full list can be found at `https://github.com/HuantWang/CONCOCTION/blob/main/vul_info/README.md`.

Table 4.3: Vulnerability statistics for each tested project.

Projects	**Versions**	**Release date**	**#Stars**	**#Lines of code**	**#Sub-mitted**	**#Confirmed**	
						#Verified	**#Fixed in verf.**
Linux Kernel	v6.1-rc5, v6.1-rc4	Apr. 2023	160k	30M	2	2	0
assimp	v5.1.4	Sep. 2023	9.7k	374k	6	6 (6 CVE)	0
Image-Magick	v7.0.11-5	Jul. 2023	10.2k	42k	1	1 (1 CVE)	1
lepton	v1.0-1.2.1	Feb. 2023	5k	80k	1	1 (1 CVE)	1
zydis	770c320	Apr. 2023	3.0k	80k	4	4	4
openEXR	v2.2.0	Jul. 2023	1.5k	246k	1	1 (1 CVE)	1
openjpeg	a44547d	Apr. 2023	902	124k	1	1 (1 CVE)	1
Leanify	b5f2efc	Dec. 2022	801	61k	2	2	2
astc-encoder	v3.2k	Jun. 2023	885	148k	3	3 (2 CVEs)	3
AudioFile	004065d	Apr. 2023	355	7k	1	1 (1 CVE)	1
xlsxio	af485eb	Nov. 2022	231	9k	1	1	1
mediancut-posterizer	v2.1	Feb. 2023	203	1.8k	1	1 (1 CVE)	0
ELFLoader	34fd7ba	May 2022	203	3k	4	4	0
pdftojson	94204bb	Oct. 2017	138	148k	3	2 (2 CVEs)	0
epub2txt2	71dc41	Jun. 2022	153	10k	1	1 (1 CVE)	1
deark	v1.6.2	Jul. 2023	136	154k	3	3 (1 CVE)	3
ok-file-formats	203defd	Sep. 2021	136	15k	7	7 (7 CVEs)	7
sqlcheck	391ae84	Mar. 2022	2.3k	4.5k	4	4 (4 CVEs)	0
packJPG	v2.5k	Apr. 2020	151	11k	7	7 (7 CVEs)	0
json2xml	v3.14.0	Nov. 2023	88	2.9k	1	1 (1 CVE)	1
Total	/	/	/	/	**54**	**53 (37 CVEs)**	**27**

Table 4.4: Top-3 types of issues found by Concoction.

Category	**#Submitted**	**#Confirmed**	**#Fixed**	**#Dyn-related**
buffer-overflow	33	33	20	23
segmentation-violation	6	6	1	5
memory-leaks	4	3	1	3
other types	11	11	5	6
Total	**54**	**53**	**27**	**37**

capability to reason about input value change ranges by combining static code structures and carefully selected symbolic traces. During testing, Concoction discovered six vulnerabilities (11.3%) related to *SEGV (segmentation violation)*, which were later confirmed by developers. Concoction identifies *SEGV* by inferring and verifying bounds on the variable value of array references. Additionally, Concoction submitted four vulnerabilities (7.5%) related to *memory-leaks*. The combination of static and dynamic code features enables Concoction to infer this type of vulnerability by correlating the allocated and released memory buffer sizes within the test function. Listing 4.1 shows an example of a *heap-buffer-overflow* vulnerability detected by Concoction, caused by an incomplete bounds-checking pattern. Meanwhile, Listing 4.2 presents a *SEGV* example uncovered by Concoction.

Concoction detected examples

We present several examples of Concoction-detected vulnerabilities, covering three types of memory-related security flaws. As DL models generally work as a black box [183], to understand Concoction's workings and the vulnerability's root cause, we compare the original buggy code with the developer-generated patch after reporting the issue.

Heap buffer overflow. Listing 4.1 shows `CVE-2022-26181`, a *heap-buffer-overflow* vulnerability identified by Concoction. The vulnerability stems from the value of the `data` variable. By making the `data` variable symbolic, Concoction allows the DL model to infer that when `data` is not null, the execution trace consistently reaches line 6. The patch mitigates this by adding a crucial bounds-check assertion on line 5. This assertion constrains the value read from `data[-1]` to a small, safe range ($\leq$ `0x10`), thus preventing an attacker from using a large value to corrupt the `data` pointer and gain control over memory deallocation. Concoction successfully identifies the incomplete bounds checking, which may overlook certain non-compliant inputs.

```
1     void aligned_dealloc(unsigned char *data) {
2       if (!data) return;
3       // function always_assert(condition) is used to catch exception when
          condition is ture
4   +   always_assert(((size_t)(data-0) & 0xf) == 0);
5   +   always_assert(data[-1] <= 0x10);
6       data -= data[-1];
7       custom_free(data);}
```

Listing 4.1: Patch for CVE-2022-26181, a `heap-buffer-overflow` vulnerability in the Lepton project.

Segmentation violation. Listing 4.2 shows a *segmentation-violation* vulnerability resulting from an out-of-bounds read. The issue arises when the value of `dctx->dcmpri.len` exceeds `dctx->dcmpri.f->len`, causing a reference to a memory location beyond the allocated buffer boundary. The patch resolves the issue by introducing a classic precondition check. On lines 3-4, it "clamps" the requested length to the buffer's maximum size, ensuring the subsequent loop never reads out of bounds, regardless of the input. CONCOCTION could not discover this vulnerability without the symbolic trace input.

```
1    dctx->dcmpri.f->len = sizeof(dctx->dcmpri.f->rcache);
2    buf = dctx->dcmpri.f->rcache;
3  + if(dctx->dcmpri.len > dctx->dcmpri.f->len)
4  + { dctx->dcmpri.len = dctx->dcmpri.f->len; }
5  ... // pass variable dctx->dcmpri.len to buf_len
6    for(i=0; i<buf_len; i++) {...
7          b = buf[i];
8          ...}
```

Listing 4.2: Patch for a *segmentation-violation* vulnerability that reassigns `dctx->dcmpri.len` to avoid memory overflow.

Memory leak. Listing 4.3 shows `CVE-2021-3574`, a *memory leak* vulnerability discovered by CONCOCTION. This issue can be detected by inspecting the execution trace from the memory allocation size of `samples_per_pixel` to the memory-free size of `MaxPixelChannels`. The patch fixes the vulnerability by adding an early check. If the input size is invalid, it performs necessary cleanup (e.g., `TIFFClose`) and throws an exception to exit *before* the vulnerable state is reached, thus preventing the leak. CONCOCTION detects this vulnerability by learning to compare the sizes of these two variables because `memory leaks` are typically caused by allocating more memory than required and subsequently freeing less.

```
  malloc (samples_per_pixel) // malloc a buffer with size equal to
      variable samples_per_pixel
  ...
+ if (samples_per_pixel > MaxPixelChannels)  {
+     TIFFClose(tiff);
+     ThrowReaderException(CorruptImageError,
+     "MaximumChannelsExceeded");}
  ...RelinquishMagickMemory(MaxPixelChannels) // free the buffer with size
       equal to variable MaxPixelChannels
```

Listing 4.3: Patch of CVE-2021-3574 for fixing a `memory leak` vulnerability in the ImageMagick project.

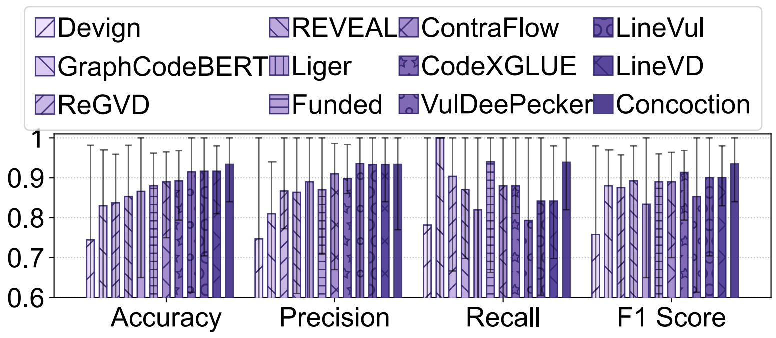


Figure 4.7: Evaluation on standard vulnerability databases. Min-max bars show performance across vulnerability types.

Importance of bugs found.

It is difficult to assess the importance of the bugs we found. Still, we found some evidence to show their importance: (1) some of the bugs we found were also reported by other application users later, indicating that the issues we identified are relevant and have occurred in real-world use cases; (2) most of our newly reported issues were confirmed and fixed by the developers demonstrates their importance; (3) developers promptly welcomed and resolved 14 of our reported issues within 48 hours, showing the importance of these issues.

4.5.2 Comparison on Open Datasets

SARD dataset

Figure 4.7 reports four "higher-is-better" metrics (Sec. 4.4.3) achieved by Concoction and the baselines on the SARD dataset (Table 4.1). The min-max bar shows variances across cross-validation runs. Concoction outperforms other methods in all metrics and has the most reliable performance across cross-validation runs, with the narrowest min-max bar. While LineVul and LineVD achieve high precision (low false-positive rate) similar to Concoction, they have lower Recall and miss some vulnerable cases. For example, LineVD only detects 70.3% of the CWE-126 vulnerability typed test cases, whereas Concoction detects all. Other baselines show low detection accuracy. Concoction achieved 100% recall in detecting certain vulnerability types like CWE-416 and CWE-789. Other methods, in contrast, failed to detect all of these vulnerabilities. Notably, Concoction is highly effective in detecting the `use-after-free` vulnerability by leveraging dynamic traces and static code structures to infer the use

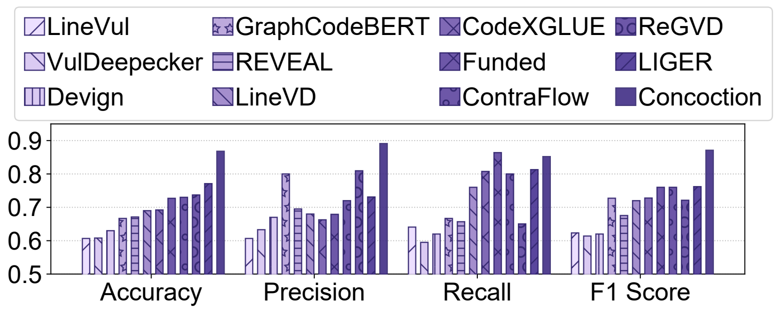


Figure 4.8: Evaluation on the CVE dataset.

of pointers.

CVE dataset

As explained in Sec. 4.4.1, test samples in the CVE dataset (Table 4.1) are more complex than the SARD dataset. As such, it is more challenging to achieve good performance. However, Concoction outperforms all other methods across all evaluation metrics, shown as Figure 4.8. Thanks to the carefully selected execution traces, Concoction can track changes in program states and variables (shown in examples in Sec. 5.1). This information enhances precision by reducing the false positive rate and helps discover more vulnerabilities with a higher true positive rate than static information alone, resulting in a higher recall. Among the baseline methods, LIGER performs best, but its F1 score is 10.4% lower than that of Concoction. This shows that Concoction strikes a better balance between false and negative positives, leveraging the advantages of structured static source code information.

4.5.3 Comparison on Known CVEs

We compare Concoction to the baselines on three open-source projects listed in Table 4.2. These projects contain 35 CVEs reported by independent users, which were also used by prior work [184, 185]. We apply all methods to functions associated with a CVE and use the reported CVEs to compute evaluation metrics (Sec. 4.4.3). To ensure a fair comparison, we train all methods, including Concoction, on the same training dataset, but we exclude these projects from the training data. For the dynamic methods listed in Sec. 4.4.2, we allocate 200 hours of machine time for each project.

Table 4.5: The number of vulnerabilities found by different methods for the projects in Table 4.2. The "#vuln." column shows the total vulnerabilities across all tools for a category.

Categories	**Approaches**	**#vuln.**
Static analysis tools	Infer [171], CodeQL [170]	5
Symbolic execution engines	KLEE [90], MoKLEE [169]	6
Fuzzing tool	AFL++ [80]	8
DL based on static code information	VulDeePecker [47], Funded [9], Devign [18], ReVeal [50], ReGVD [51], LineVul [147], LineVD [168], CodeXGLUE [49], GraphcodeBERT [106], ContraFlow [78]	22
DL based on dynamic information	LIGER [24]	16
	Concoction	31

Vulnerabilities identified

Table 4.5 demonstrates Concoction's advantages over other methods in identifying vulnerabilities across the three open-source projects evaluated. Concoction achieved 100% precision and 89% recall, correctly detecting 31 out of 35 confirmed CVEs. Additionally, Concoction identified all issues found by other methods and uncovered 9 additional vulnerabilities that others missed. Among DL-based static methods, ReGVD had the second-best and highest recall rates, detecting 21 vulnerabilities. However, they struggled to identify 14 vulnerabilities caused by API parameter misuse.

Static tool baselines, CodeQL and Infer, can only detect five vulnerabilities at their best (5 for CodeQL and 0 for Infer) because they rely on hand-crafted rules with limited coverage. They missed all vulnerabilities related to "*CWE-754: Improper Check for Unusual or Exceptional Conditions*" because it was not in their rule sets. Dynamic methods using symbolic execution and fuzzing tools have a low recall of 0.26 in our evaluation due to limited path coverage within the testing time (12 hours). MoKLEE found 2 more bugs than KLEE (which found 4) in this context. Without Concoction's DL component, native AFL++ detected 8 vulnerabilities in 12 hours, while Concoction improved efficiency by finding 31 CVEs within the same test time by guiding the fuzzing engine to focus on potentially buggy code paths.

Though highly effective, Concoction missed four vulnerabilities (one example in Sec. 4.5.3) due to incomplete vulnerable execution traces during feature extraction. Addressing this limitation could involve extending symbolic execution time and improving the path selection model or the number of paths selected.

Case missed by baselines

Listing 4.4 shows `CVE-2020-35523`, an `integer overflow` vulnerability identified by CONCOCTION but missed by all baselines. The patch shown in the listing validates this by implementing exactly that missing check, preventing the dangerous computation entirely. Other methods missed this bug because they mainly rely on static information, such as tokens typically associated with `integer overflow` (e.g., `malloc`), which is insufficient in this case. CONCOCTION detects this vulnerability by symbolising the `tw` and `w` variables, allowing the DL model to infer the absence of a corresponding `INT32` bounds check, which aligns with the pattern of `integer overflow`.

```
  if (flip & FLIP_VERTICALLY) {
+   if ((tw + w) > INT_MAX) {
+     TIFFErrorExt(...);//
+     return (0);}
    y = h - 1;
    toskew = -(int32)(tw + w);
```

Listing 4.4: An `integer overflow` vulnerability in Libtiff.

Case missed by Concoction

List 4.5 shows `CVE-2020-35523`, a `memory leak` vulnerability missed by CONCOCTION and all other baselines. The issue is caused by the function directly returning without closing the input file handle `in`, causing resource leakage. The patch, highlighted on line 6, rectifies this by inserting the necessary call to `TIFFClose(in)` immediately before the `return` statement. This ensures that the acquired resource is properly released on all execution paths, thereby fixing the leak. CONCOCTION missed this because the memory leakage is introduced within the file handle data structure, but CONCOCTION does not learn such patterns from the training dataset. This can be improved by expanding the training dataset to encompass a broader range of patterns.

```
1     extern int optind;
2     in = TIFFOpen(argv[optind], "r");
3     ...
4     if (...) {
5       fprintf(...);
6   +   (void) TIFFClose(in);
7       return (-1);}
```

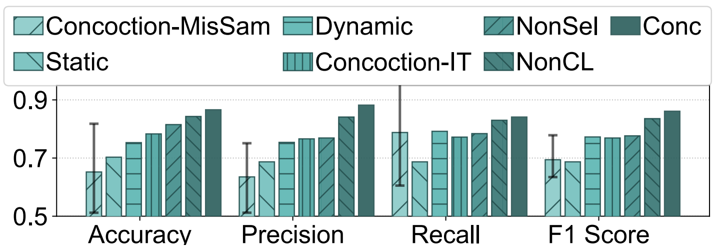


Figure 4.9: CONCOCTION variants on CVE dataset.

Listing 4.5: A `memory leak` vulnerability in Libtiff that occurs because the file handle `in` is not released.

4.5.4 Ablation Study

DL model implementation choices

We conduct an ablation study [186] on CONCOCTION using the CVE dataset. The study includes the following variants: Static (using enhanced AST, Sec 4.3.2), Dynamic (utilizing randomly sampled symbolic traces with 30 minutes of symbolic execution for each project, NonCL (without the contrastive learning module, Sec.4.3.3), NonSel (omitting the path selection module by using randomly sampled symbolic execution traces with static code information, Sec 4.3.4), and Conc (the complete CONCOCTION implementation).

The results are given in Figure 4.9. Using only static or dynamic representations is insufficient for accurately modelling program structures, with F1 scores of 68.7% and 77.2% for each variant, respectively. In our approach, we employed dropout-based contrastive learning as data augmentation for training our representation model (Sec.4.3.3). This helps extend our training set and mitigates overfitting [187]. Removing the contrastive learning component led to a 3.7% decrease in the F1 score, reaching 82.4% compared to the full model. Additionally, removing the path selection method resulted in an F1 score drop to 77.6% since random sampling may not capture crucial path information within a given budget.

Sensitivity analysis

To test the sensitivity of CONCOCTION on mislabeled training samples, we introduce mislabeled samples that account for from 20% to 80% of the training samples into the

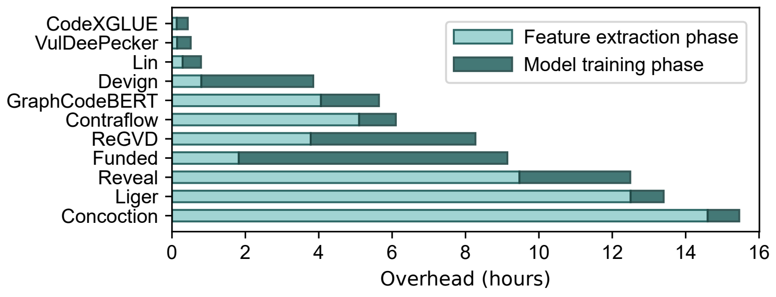


Figure 4.10: Training overhead of different DL methods.

training dataset of Concoction, leading to a variant of Concoction-MisSam. Similarly, we randomly remove some symbolic execution traces selected by Concoction to simulate a scenario where some traces are missing. This led to another implementation variant named Concoction-IT. The performance of Concoction-MisSam on Figure 4.9 shows that mislabeled training samples can harm performance, where the F1 score of Concoction-MisSam drops from 78.0% to 64.3%. This is expected, as machine learning techniques can suffer from noisy and mislabeled training data [188, 189]. However, the impact of mislabeled samples can be mitigated by increasing the training dataset and using data cleaning methods [190, 191], which are orthogonal to our approach. Similarly, missing execution traces can also negatively impact the performance, where the F1 score of Concoction decreases from 86.1% to 77.1% in Concoction-IT, which is still at least 2.1% higher than other DL baselines that rely on static code information. Missing symbolic execution traces is likely to happen when testing external libraries where the tool has no access to the source code. This issue is beyond the capability and scope of a source code-level detection tool. In the worst case, where all symbolic traces are missing, our DNN model can still use static information to detect bugs, albeit less efficiently.

Training and deployment overhead

Figure 4.10 compares training overhead for various DL-based methods. It includes one-off time spent on feature extraction (e.g., AST and symbolic executions) on training samples and iterative training time using labelled samples from Table 4.1. Training terminates when the loss does not improve within 20 consecutive epochs or meets the termination criteria specified in the baselines' source publication. The experiment was conducted on a multi-core server using a desktop-level NVIDIA 2080 Ti GPU. VulDeePecker, CodeXGLUE, and Lin *et al.* achieved the shortest training overhead, relying mainly on sequence neural networks like Bi-LSTM. However, they have a low F1 score, indicating a limited ability to capture complex code structures. More advanced models that use ASTs required longer feature extraction and training times but showed higher accuracy during evaluation. Additionally, LIGER and Concoction incur a more expensive feature extraction time due to collecting dynamic runtime information by symbolic execution. It is important to note that model training is performed offline and is a *one-off* cost.

During deployment, Concoction can complete predictions within minutes, and the fuzzing tool may take several hours to generate a vulnerability-exposing test case. Since Concoction can be integrated with a parallelised overnight build system, the deployment overhead should be acceptable for many software developers.

4.6 Summary

In this chapter, we have presented Concoction, a new DL system for detecting vulnerabilities at the source code level. It utilises structured static code features and dynamic symbolic execution traces to learn program representations, enabling accurate prediction of bugs. We train Concoction by combining unsupervised and supervised learning and minimising the overhead of symbolic executions by using a path selection network. We apply Concoction to detect bugs and vulnerabilities for C programs from 20 open-source projects. In 200 hours of automated concurrent test runs, Concoction successfully detected vulnerabilities in all tested projects, discovering 54 unique vulnerabilities and yielding 37 new, unique CVE IDs. Compared to 16 previous methods, Concoction finds more vulnerabilities with higher accuracy and a lower false positive rate.

Chapter 5

Automated Machine Learning for Program Modelling

The previous chapter presents a method to better represent programs to improve precise program modelling. In this chapter, we present a framework for automated DRL component design and implementation. We conduct a large-scale evaluation across four code optimisation tasks to demonstrate how our approach lowers the barrier between ML expertise and compiler programmers. The chapter is organised as follows. Section 5.2 discusses the current challenges in applied ML caused by the expertise barrier. Section 5.3 explains how our approach is formulated. Section 5.4 then describes the experimental setup. Section 5.5 presents the experimental results, before Section 5.6 concludes the chapter.

5.1 Introduction

In recent years, we see a growing interest in the research community and industry [15, 192] in using RL and DRL to tackle a wide range of code optimisation problems [133–135, 137, 193]. Developing a deep RL solution for code optimisation requires choosing and parameterizing several components: (i) a discrete set of actions or transformations that can be applied to a program, such as passes in a compiler; (ii) a state function that can summarize the program after each action as a finite feature vector, and (iii) a reward function that reports the quality of the actions taken so far. Some RL algorithms may also allow the selection and further parameterisation of a transition function, which governs the choice of actions to be applied in each state. Moreover, parameters of individual RL components need to be tuned on benchmarks.

Efforts have been made to provide RL algorithms and high-level APIs for action definitions [15, 25], models for program state representation [5, 11, 26, 27], and tools for training benchmark generation [6, 15, 28]. While these recent works have lowered the barrier for integrating RL techniques into compilers, compiler engineers still face a major hurdle. As the right combination of RL exploration algorithms and their state, reward and transition functions and parameters highly depend on the optimisation task, developers must carefully choose the RL architecture by finding the right RL component composition and their parameters from a large pool of candidate RL algorithms, machine-learning models and functions. This process currently requires testing and manually analysing a large combination of RL components. Experience in the field of neural architecture search shows that doing this by hand is an expensive and non-trivial process [138].

This chapter presents SUPERSONIC[1], an open-source framework to automate the RL architecture search and parameter tuning process to make it easier to integrate RL into compilers. To use SUPERSONIC, the compiler developer provides the action list according to the problem being tackled and a measurement interface to report metrics like code size or speedup. SUPERSONIC then automatically assembles an RL architecture for the targeting optimisation from an extensible set of built-in RL components. The SUPERSONIC RL components include pre-trained state functions, such as `Word2Vec` [29] and `CodeBert` [30]. It provides candidate reward functions like `RelativeMeasure` and `tanh` to compute the reward based on the metric given by the measurement interface. The state-transition function can be selected from a further set of predefined transition functions, such as a transition probability matrix or LSTM [31]. Finally, SUPERSONIC takes a customizable set of predefined RL algorithms, like PPO [32] and MCTS [33], which may be driven by any of the chosen reward, state, actions and transition function. This creates a large space of possible parameterised RL architectures, which can be defined by the compiler developer with a few lines of Python using an easy-to-use API.

Armed with this space of RL architecture choices, SUPERSONIC will automatically and efficiently search it to find the one that gives the best results over a sample of user-provided *training* benchmarks. The search is accomplished by a deep RL-based *meta-optimiser* [194, 195] that is designed to be generalizable to any optimisation tasks. Once the meta-optimiser has selected the *client RL* architecture and its parameters,

[1]Available at: `https://github.com/HuantWang/SUPERSONIC`

the work of SUPERSONIC is over. As an output, SUPERSONIC stores the tuned client RL as serialised objects. It provides an API for a compiler or performance tuner to use the stored RL to drive the code transformation pass for *new, unseen* programs. On an unseen program, the client RL may be used to search for the actions that yield the best reward. Alternatively, it can be further generalised by training on additional benchmarks so that actions for new programs can be chosen *directly* from the policy without further search at the deployment time. SUPERSONIC aims to automatically make the client RL as good as possible for the compiler developer's needs.

As this client RL search and tuning process is automated and requires little RL expertise, SUPERSONIC further reduces the difficulties for integrating RL into compilers by replacing compiler developer time with machine hours. This client RL search and tuning process is a *one-off* cost performed offline. The compiler end-users (e.g., application developers) will not experience this process - they will use the shipped RL like any other feedback directed compiler passes [196].

We evaluate SUPERSONIC by applying it to four optimisation tasks: Halide schedule optimisation [43], neural network code generation [44], compiler phase ordering [45] for code size reduction and superoptimisation [46]. Each of the tasks has a large number of combined optimisation options, so it is non-trivial to design a good search strategy. We compare SUPERSONIC against eight tuning methods developed by independent researchers, including search-based strategies specifically designed for the targeting problem [197–200], generic tuning frameworks like OpenTuner [201] and CompilerGym [15], and hand-tuned RL solutions for the relevant task [202]. Our extensive evaluation shows that SUPERSONIC consistently gives better overall performance than alternative methods across tasks during deployment. We show that the client RL given by SUPERSONIC converges fast, and it can start producing better code than competing search methods by using on average 1.75x less search time (up to 100x) for new programs during deployment.

5.2 Problem Definitions

SUPERSONIC automates RL component searching and parameter tuning. Within SUPERSONIC, a *client RL* consists of an exploration algorithm to choose actions (e.g., compiler options), a reward function for computing the expected cumulative reward based on past observations of the environment (e.g., execution time after applying a

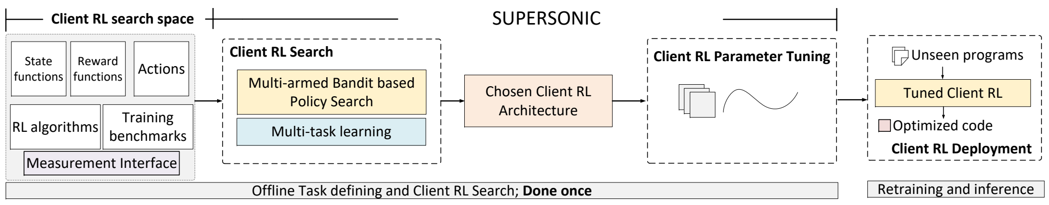


Figure 5.1: Overview of SUPERSONIC components. This framework enables developers to express the optimisation space. It automatically searches for the optimal client RL architecture to be used for inference during deployment.

transformation), a method for modelling the environment or program state (e.g., a DNN or a linear function), and an action list provided by the user (e.g., legitimate code transformation options). Depending on the exploration algorithm, this can also include a state transition function to compute the probability of going from one state to another. Each of the components can be chosen from a pool of SUPERSONIC built-in candidate methods, and the combination of these components can result in a large policy search space. SUPERSONIC is designed to automatically find and tune the right combination of RL components and their hyperparameters. It complements existing RL platforms like CompilerGym [15] and RLLib [25], by helping the compiler developers to choose and optimise a suitable RL algorithm.

5.3 Supersonic: A System for Automated ML for Program Modelling

5.3.1 Overview

Figure 5.1 gives an overview of SUPERSONIC. At the core of SUPERSONIC is a meta-optimiser that builds upon MAB and deep RL techniques. Given a client RL search space defined by the SUPERSONIC Python API, the meta-optimiser searches for a suitable RL component configuration for an optimisation task. It then automatically tunes a set of tunable hyperparameters of the chosen components. The tuned client RL can then be used to optimise unseen programs through inference (including potential retraining), which is outside the scope of SUPERSONIC. The search space definition and RL client architecture search and parameter tuning are a *one-off offline* process, which is within the scope of SUPERSONIC.

Implementation. We implement Supersonic in Python and use gRPC for distributed communications. Supersonic builds upon CompilerGym [15] and RLlib (Ray) [25] by utilising their APIs for task definitions and RL algorithm implementations. Supersonic currently supports 23 RL algorithms from RLLib [25] and 10 pre-trained DNNs and functions for representing the program state or computing the reward. Compiler developers may choose a subset, add their own, or include all (the default) supported RL algorithms and models for the meta-optimiser to search over.

Task definition. The compiler developer first defines the optimisation problem by creating an RL policy interface (Figure 5.2). The definition includes a list of client RL components for the meta-optimiser to search over.

Client RL search. Calling to `policy_search()` invokes Supersonic meta-optimiser, where the developer can also limit the number of trials spent on client RL searching.

Client RL parameter tuning and deployment. After choosing an RL architecture, the meta-optimiser will fine-tune a set of model-specific hyper-parameters of the selected client RL (see also Table 5.1). Hyperparameter tuning is performed on the training benchmarks. The tuned client RL and its parameters are saved, which can be shipped with a compiler to optimise unseen programs at deployment time.

Measurement engine. The measurement engine evaluates a code transformation option using a user-supplied interface (line 24 in Figure 5.2). Measurements are used during client RL search and tuning, as well as deployment phases, to obtain feedback for a chosen optimisation action.

5.3.2 Task Definition

The user-defined client RL search space typically includes candidate functions (or models) for representing the environment state, objective functions for computing the reward, and the set of possible actions that can be taken from a given state. This search space definition can optionally include a chosen set of RL exploration algorithms and transition functions to be used by a client RL algorithm. By default, Supersonic automatically search over all supported RL algorithms where each algorithm has a default transition function. Furthermore, the compiler engineer also needs to provide a `run` function, which provides the measurement of an action to compute the reward. These are implemented in a small Python program to interface with the Supersonic

```python
import SuperSonic as ss
from SuperSonic.stateFunctions.models import *
...
statefs = [Word2Vec(...),Doc2Vec(...),CodeBert(...),ActionHistory(...)]
tranfs = [DNN(...),CNN(...),LSTM(...)]
rewards = [RelativeMeasure(...),tanh(...)]
rl_algs = [MCTS(...),PPO(...),DQN(...),QLearning(...)]
actions = [Init(...)]
...

class SuperOptimizer(ss.PolicyInt):
  def __init__(self, statefs, tranfs, actions, rewards, rl_algs):
      self.PolicySpace = {
        "StatList": statefs,
        "TranList": tranfs,
        "ActList": actions,
        "RewList": rewards,
        "AlgList": rl_algs,
      }
      self.search_engine = SearchEngine(self.PolicySpace)

  def run(self):
    #user code for compilation and execution
    ...
    return Result(time=run_result['time'])

if __name__=='__main__':
  opt = SuperOptimizer(statefs, tranfs, actions, rewards, rl_algs)
  policy = opt.policy_search(training_benchmark_list, num_of_trials=100)
```

Figure 5.2: Simplified tasks definition for superoptimization.

API. The definition code is similar to the programming environment of mainstream auto-tuners like OpenTuner and CompilerGym, allowing the developer to quickly port their code to use the Supersonic search and tuning components.

Figure 5.2 gives a simplified example that defines the client RL search space for superoptimizaton [46, 203]. This example specifies candidate methods for representing the environment state, functions for computing rewards, the definition for action space, and a chosen set of client RL algorithms. The `run` function implements the measurement interface, including user code for compiling and executing the program for a given code transformation action.

Finally, the program invokes the policy search API by passing in a list of training benchmarks and the number of search trials. We stress that the measurement interface defines how to compile and execute a test program, but the target program can be of any language.

5.3.3 Client RL Search

Given a client RL search space, the Supersonic meta-optimiser automatically finds a suitable client RL architecture (i.e., <state function, transition function, reward function, RL algorithm> and potentially a value function) from training benchmarks. We formulate the client RL search problem as an MAB problem that is solved using a parallel DRL algorithm. Deep RL can reuse knowledge from other tasks to speed up the search. In contrast, evolutionary algorithms have to search from afresh. Our work is the first to formulate client RL tuning for code optimisation as a MAB problem and employs deep RL as a meta-solver.

Our meta-optimiser is a variant of the Asynchronous Advantage Actor Critic (A3C) algorithm [204, 205]. A3C is a distributed algorithm, where multiple workers independently update a global value function - hence "asynchronous". We chose this algorithm because it has been shown to be effective in other RL application domains [205] and permits us to develop a parallel policy search engine. Specifically, the meta-optimiser consists of two RL models, an *actor* for computing an action based on observation and a *critic* for estimating a reward value. In a nutshell, the actor is a policy RL that takes as input the environment state and outputs the best action (a policy architecture in our context). The actor essentially controls how the meta-optimiser chooses a candidate client policy to try out. By contrast, the critic is value-based RL that evaluates the action by adjusting its value function to estimate the maximum future reward based on the historical observations obtained from training benchmarks. As time passes, the actor is learning to produce better actions, and the critic is getting better at evaluating those actions.

Like [205], we implement the actor and the critic using a stacked neural network consisting of a ResNet CNN [206] that is followed by an LSTM RNN [31]. We use the output of the LSTM to update the policy function of the actor and the value function of the critic. Input to the actor model is a 1-dimensional history vector containing the last 20 actions (policies) that the meta-optimiser has tested. Input to the critic model is a history vector plus a cumulative reward averaged across benchmarks, computed using an Area under the Curve (AUC) function.

Client RL searching strategy. At each of the n trials, the meta-optimiser obtains an action (i.e., a client policy architecture to test) from the actor model. The meta-optimiser uses the user-provided measurement function to obtain the observation, which is then used to compute the current reward. The current reward and the environment state are passed to the critic model to update its value function. The value function of the critic also estimates the future reward, which is given to the actor to update its policy network. By default, the meta-optimiser runs each client RL for 50 exploration steps during a trial to allow it to converge before taking the observation. We use the 20 most recently chosen policy architectures as the state. We then measure the area under the reward curve of the recently chosen 20 policies to compute the current reward. After the meta-optimiser performs n trials on training benchmarks, we check the latest 20 actions chosen by the actor. We then use the most frequently chosen policy architecture as the outcome of the policy search. Our intuition is that the actor would be more efficient in picking good policy architectures towards the end of the search and would choose the optimal policy as the action more often. This search process also implicitly models the learning time of a candidate client RL. A client RL is chosen because it can learn quickly to give good results on training benchmarks within the given time.

Failure during client RL search. In this work, we did not observe failure in combining RL components in our case studies. When using a client RL, failures can happen - the underlying compiler (driven by RL) may fail due to e.g., invalid combinations of compiler flags or compiler bugs. These are automatically handled by RL which will avoid trying these options in future iterations. Furthermore, while SUPERSONIC could miss an RL architecture that can be improved through fine-tuning, we have performed a large-scale search on our case studies and did not observe this. This issue can also be mitigated by performing RL-architecture search and parameter fine-tuning in a single process.

Multi-task learning. If multiple optimisation tasks are defined, we then use multi-task learning (MTL) to find an individual policy for each task given a total budget of n trials. SUPERSONIC provides a distributed, parallel meta-optimiser, building on top of an open-source MTL framework [205]. For each optimisation task, it creates an

Table 5.1: Example tunable parameters

Algorithms	**Parameters**
Common param.	Batch size for workers; Train batch size; Minibatch size; learning rate
MCTS	Dirichlet noise and epsilon; Puct coefficient; Simulation times; Loss temperature
PPO	Entropy coefficient; Adam optimiser step size; GAE estimator parameter; Policy ratio clipping
DQN	#atoms; Discrete supports; Adam epsilon; Clip gradients
Q-Learning	Behavior Cloning Pretraining numbers; Q-Learning loss temperature; Lagrangian threshold; Min Q weight multiplier

environment and assigns a meta-optimiser instance to the environment, so that client RL search for different optimisation tasks can be performed in different, potentially distributed environments. In our evaluation, we implement an execution environment in a Docker container. Supersonic realises different environments and their associated actors as parallel workers that can run on a single machine or multiple distributed machines. An issue of using a standard MTL algorithm is that the learner is likely to give more resources (e.g., #trials, time or machines) to tasks with higher rewards, leading to unfair resource allocation among tasks. To address this issue, we use a regularisation mechanism, similar to [207], by adding a normalisation layer to the actor and the critic network.

5.3.4 Client RL Parameter Tuning

During the client RL search stage, Supersonic uses the default hyperparameter and pre-trained models. After a client RL architecture is chosen, the Supersonic meta-optimiser uses training benchmarks to fine-tune a set of common and algorithm-specific hyperparameters. Each Supersonic built-in DNN model also has a standard training API. Therefore, the meta-optimiser also uses the measurements and observations generated during parameter tuning to fine-tune the relevant DNN models, e.g. for state representation. Table 5.1 gives some of the example hyperparameters supported by Supersonic. We note that the user does not need to explicitly supply these parameters because they are known to Supersonic. Our parameter tuning method is a parallel population-based training algorithm [208] from RLlib. Like client RL search, the user can also specify how many trials can be spent on parameter tuning. Once the budget is used up, the best-found parameter setting is returned. Our evaluation applies cross-validation to ensure the tuned RL is always tested on new, previously unseen benchmarks.

5.3.5 Client RL Deployment

Finally, the tuned client RL is saved as serialised objects. The chosen hyperparameters and action space are stored in JSON files. SUPERSONIC provides APIs to load and reuse the stored objects to optimise any new program. To apply a tuned RL, SUPERSONIC creates a session to apply a standard RL loop to optimise the input program by using the chosen RL exploration algorithms to select an action for a given state. For example, the state could be a vector recording the last n compiler options added into the compiler flags or a DNN (see also Section 5.5.5).

5.3.6 Measurement Engine

For a given optimisation option, the measurement engine invokes the user-supplied `run` function to compile and execute the program in the target environment. The user function reports the result for each execution, which is stored in a result database implemented using SQLLite. The database also holds information obtained during the search, including the action history, reward, and execution outputs for each benchmark. The client RL gathers the result of an action by querying the result database. Decoupling the RL exploration and measurement allows the parallel execution of measurement and the RL agent, possibly across different machines. Parallel execution can reduce the measurement cost, which often dominates the auto-tuning process.

5.4 Experimental Methodology

We evaluate our approach by applying it to four code optimisation tasks and comparing it against eight tuning methods, including hand-tuned RL solutions. Table 5.2 summarises our evaluation setup, including the search space size and competing methods for each case study. We note that the overhead of RL is dominated by gathering feedback from the environment through, e.g., compiling and executing the program. While case studies 3 and 4 have a larger search space than case studies 1 and 2, obtaining

Table 5.2: Case studies in our evaluation

Use cases	**#Bench.**	**Competing Methods**	**Search space**
C1: Optimizating image pipeline	10	Halide master [198], auto-schedulers [199], HalideRL [209], OpenTuner	$7^3 * 2^7 \sim 7^3 * 3^7$
C2: Neural network code optimisation	5	AutoTVM [200], Chameleon[202], OpenTuner	$5 * 10^3 \sim 20 * 10^3$
C3: Code size reduction	43	CompilerGym [15], OpenTuner	$123^{60} \sim 123^{150}$
C4: Superoptimisation	40	STOKE [197], OpenTuner [201]	$9^{100,000} \sim 9^{16,000,000}$

Table 5.3: Candidate state functions and reward functions

		Case Studies: 1	2	3	4
State func.	Word2Vec [210]				
	Doc2vec [211]		✓		✓
	CodeBert [30]				
	Manual features (e.g. LLVM IR representation from [212])	✓			
	Action History				
	Hash of Action History			✓	
Reward func.	Relative measure (e.g. speedup, code size reduction ratio)	✓	✓		
	function output (e.g. tanh())			✓	✓

feedback incurs lower overhead in case studies 3 and 4 compared to 1 and 2, leading to an overall faster search time for case studies 3 and 4.

5.4.1 Case Study 1: Optimising Image Pipelines

The problem. This task aims to improve the optimisation heuristic of the Halide compiler framework for image processing [43]. A Halide program separates the expression of the computation kernels and the application processing pipeline from the pipeline's *schedule*. Here, the schedule defines the order of execution and placement of data on the hardware. The goal of this task is to automatically synthesise schedules to minimise the execution time of the benchmark.

Methodology. This task builds upon Halide version 10. Our evaluation uses ten

Halide applications that have been heavily tested on the Halide compiler. We measure the execution time of each benchmark when processing three image datasets provided by the Halide benchmark suite. The benchmarks are optimised to run on a multi-core CPU.

Competing methods. We compare Supersonic against four prior methods designed for optimising Halide schedules. These include two Halide built-in auto-scheduling algorithms (Halide master [198] and auto-scheduler [199]), a hand-tuned RL method (HalideRL) [209], and OpenTuner. We show speedups over Halide master.

Actions. Each Halide program comes with a scheduling template that defines an n stages schedule. We can apply optimisations like loop tiling and vectorisation to each stage. We apply four actions to construct a n-stage scheduling sequence. These include adding or removing an optimisation to the stage and decreasing or increasing the value (by one) of an enabled parameterised option.

5.4.2 Case Study 2: Neural Network Code Generation

The problem. This task targets DNN back-end code generation to find a good schedule. e.g., instruction orders and data placement, to reduce execution time on a multi-core CPU.

Methodology. This study is conducted within the TVM compiler v 0.8 [44]. We use 5 CNN kernels where their schedule optimisation space is defined by the TVM developer.

Competing methods. We compare our approach against four TVM built-in tuning strategies, including random search, genetic algorithms, grid-based search and XGBoost-based search. In addition to these, we also compare our approach to OpenTuner and Chameleon [202] - a recently proposed, hand-tuned RL method designed for TVM. We show the improvement over the TVM compiler (TVMC) without schedule optimisation.

Actions. Each TVM benchmark comes with a schedule template that defines a set of tuning knobs like loop tiling parameters. We consider four actions in this task: adding or removing a knob to the schedule sequence, and decreasing or increasing the parameter value (by one) of a parameterized knob in the optimisation sequence. The number of tuning configurations varies across benchmarks (Table 5.2).

5.4.3 Case Study 3: Code Size Reduction

The problem. This task is concerned with determining the LLVM passes and their order to minimise the code size.

Methodology. Following the setup of CompilerGym, we compute the code size reduction by measuring the ratio of LLVM IR instruction count reduction over the LLVM -Oz code size optimisation option. This metric is platform-independent and deterministic. Note that the IR instruction count strongly correlates to the binary size - fewer instructions typically lead to a smaller binary. In this evaluation, we use 43 benchmarks: 23 from the CBench suite [213] and 20 single-source benchmarks from the LLVM test suite [214].

Competing methods. We compare our approach against OpenTuner and the Greedy and Random search strategies for code size reduction implemented by CompilerGym. The CompilerGym Greedy algorithm has a threshold, e, for controlling how often the algorithm switches between random and greedy searches, with $e = 0$ for a solely greedy strategy and $e = 1$ for a purely random algorithm. We set e to 0.1, which produces comparable results as CompilerGym developers reported on their platforms.

Actions. We consider all the 123 semantics-preserving passes of LLVM. The RL agent determines which pass to be added into or removed from the current compiler pass sequence. Note that the length of the compiler pass sequence is unbounded. An LLVM pass can appear multiple times in the pass sequence and be inserted before or after any pass.

5.4.4 Case Study 4: Superoptimisation

The problem. This classical compiler optimisation task finds a valid code sequence to maximise the performance of a loop-free sequence of instructions [46, 203]. Superoptimisation is an expensive optimisation technique as the number of possible configurations grows exponentially as the instruction count to be optimised increases.

Methodology. In this task, we apply SUPERSONIC to find a client RL for STOKE, the state-of-the-art superoptimiser [197]. Given a set of test cases consisting of input-output pairs and a subset of x86-64 instructions, STOKE synthesises a program (at the assembly code level) that uses these instructions and agrees with the test cases. We use all the 25 benchmarks from the STOKE Hacker dataset. Additionally, we also extract 15 loop-free and frequently-executed functions from SPEC CPU 2017 and the LLVM test suite, 10 from SPEC and 5 from the LLVM test suite. The seed input to the performance tuner is the assembly code generated by compiling the C code with the -O0 compiler option. We use STOKE's mutation engine to modify the target instructions and its equivalent testing method to verify if the transformed code satisfies the test cases. We also manually verify the correctness of the best-performing version found by each method.

Competing methods. We compare SUPERSONIC to the Markov Chain Monte Carlo-based STOKE search technique and OpenTuner. We test all schemes on LLVM and GCC and use -O3 as the baseline. *As noted in the STOKE document, the STOKE implementation is not mature enough to improve -O3 code.* However, as we will show later, SUPERSONIC can deliver noticeable improvement over -O3 on certain test cases, demonstrating the potential of auto-tuning techniques.

Actions. We consider all the instruction-level transformations supported by STOKE. These include replacing the opcode and operand of an instruction as well as inserting, replacing and swapping instructions.

5.4.5 Client RL Architecture Search Space

We consider all the SUPERSONIC-supported RL algorithms when searching the client RL. SUPERSONIC chooses to use PPO for code size reduction and MCTS for the other three tasks. Table 5.3 lists the state and reward functions considered in each task, where we highlight the SUPERSONIC chosen function using a check mark. As can be seen from the table, no RL algorithm dominates our case studies - the best algorithm depends on the optimisation task. However, the Supersonic-chosen RL algorithm generalises well to input test programs of an optimisation task.

5.4.6 Hardware and Software Platforms

For case studies 1, 2 and 4, we use two multi-core servers to evaluate the resulting code. The first server has 2x 26-core Intel Xeon 8179M CPU running at 2.40GHz, and the second server has 2x 32-core AMD EPYC 7532 CPU at 2.4GHz. Both servers have 128GB of RAM and run Ubuntu 20.04 with Linux kernel v5.4. In our evaluation, we run the SUPERSONIC meta-optimiser on the AMD server and use the chosen client RL to optimise the target task on both machines. We run the relevant deep learning models on an NVIDIA RTX 2080 Ti GPU. We use LLVM v10 as the backend compiler to generate the executable binary in our evaluation. For superoptimization, we also test our approach on GCC v11.2.

5.4.7 Performance Report

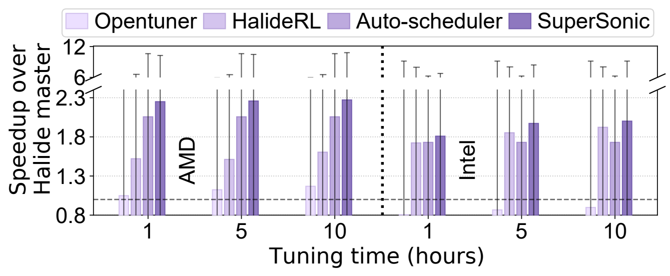
Cross-validation. We use 3-fold cross-validation to evaluate our approach. This means we partition the benchmarks into three groups (folds). We perform client RL search and tuning on two folds of the benchmarks and then test the tuned RL architecture on the remaining benchmarks. We repeat this procedure three times to test each of the three folds in turn. Unless stated otherwise, we exclude the time spent on client RL search and tuning from the overhead of deployment-time performance optimisation, because client RL search and tuning is a *one-off* cost performed offline. This cost is incurred only once during training, and the tuned RL architecture can then be applied to many new programs without repeating the tuning-search overhead. However, *we give the same amount of search time/iterations for all methods when optimising a test program.*

Runtime measurement. To measure the runtime of the resulting binary, we run each benchmark at least 100 times on an unloaded machine. For each benchmark, we also compute the 95% confidence interval bound and increase the number of profiling runs if the interval is greater than 2%. We report the geometric mean across runs. We also show the performance variances across benchmarks, compilers and cross-validation settings as min-max bars on the diagram.

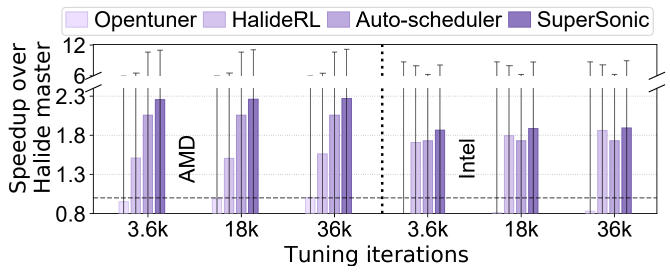
Client RL search. In our evaluation, we apply MTL to perform RL search for all four optimisation tasks simultaneously on a single server, with a total search budget of 100 trials.

5.5 Experimental Results

In this section, we first present the case study results, finding that SUPERSONIC outperforms hand-crafted strategies in each task. We then provide an analysis of SUPERSONIC's working mechanisms, showing SUPERSONIC can accelerate the deployment-time search by 1.75x. Note that the tuning time is proportional to the number of tuning iterations, but the tuning time per iteration can vary between evaluated methods. Spe-



(a)



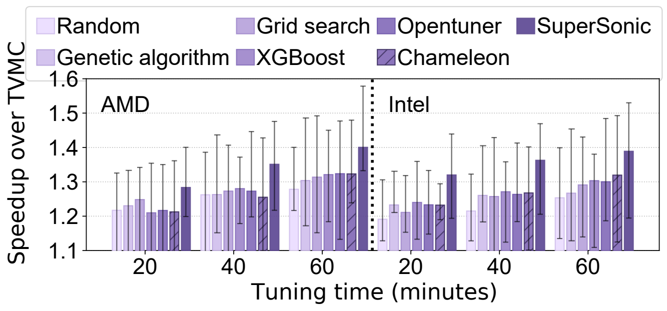
(b)

Figure 5.3: Performance to expert-tuned Halide schedules under different search time (a) and iteration (b) constraints. SUPERSONIC gives the overall best performance than other auto-tuning methods.

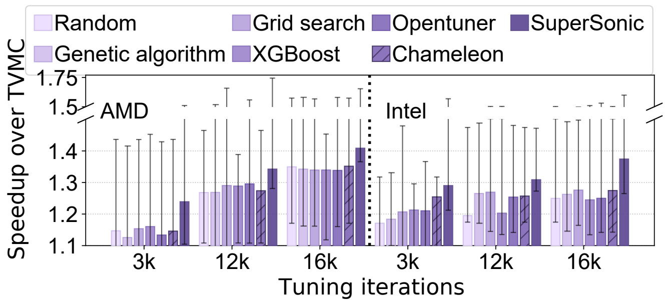
cifically, the SUPERSONIC-chosen client RL can perform, on average, 1, 3, 24 and 10 search iterations per second for case studies 1, 2, 3 and 4, respectively, on our evaluation platforms.

5.5.1 Case Study 1: Optimizing Image Pipelines

Figure 5.3 reports speedup over the Halide master scheduler [198]. SUPERSONIC gives noticeable performance improvement on the AMD platform. It also manifests larger advantages when the search time is limited. On certain benchmarks, SUPERSONIC is able to give over 11x speedup with a correctly optimised code. The tree-based auto-scheduler gives a high speedup of over 10x for a single benchmark, but it has a lower mean performance improvement across benchmarks compared to SUPERSONIC. On average, SUPERSONIC delivers a 1.5x improvement over HalideRL, a manually tuned RL strategy. Note that HalideRL uses PPO as the exploration algorithm and the sequence of already applied schedules as the state function. By contrast, SUPERSONIC determines



(a)



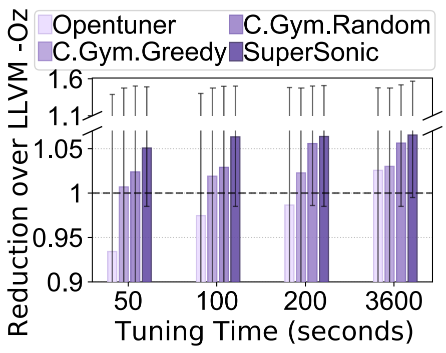
(b)

Figure 5.4: Speedup for DNN code generation under different search time (a) and iteration (b) constraints. The min-max bar shows the range across benchmarks. SuperSonic outperforms all competing methods on both platforms.

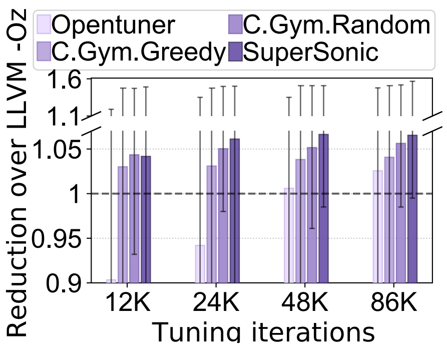
that for this task, using MCTS as the exploration algorithm and a hash function of the applied schedules as the state function is better than HalideRL. MCTS compares complete schedules through simulations and looks ahead before making intermediate scheduling optimisations, leading to a better result. Overall, SuperSonic outperforms all alternative search techniques on all but two benchmarks on both platforms.

5.5.2 Case Study 2: Neural Network Code Generation

Figure 5.4 reports the performance improvement over the default schedules. RL-based methods (Chameleon and SuperSonic) can generate better code than TVM's evolutionary or predictive modelling-based search techniques. While random and grid-based search can significantly improve one benchmark (the top point of their min-max), their performance is not robust and can give poor performance for other benchmarks. By contrast, SuperSonic delivers more robust performance by giving no slowdown on the



(a)



(b)

Figure 5.5: Code size reduction over LLVM -Oz under different search time (a) and iteration (b) constraints. SUPERSONIC outperforms alternative methods.

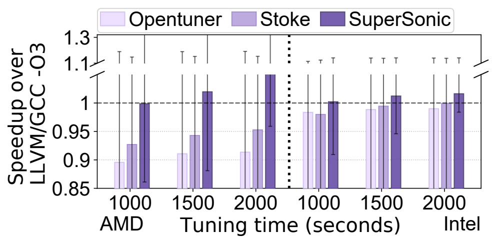
Table 5.4: Important compiler passes for code size reduction and how often a pass is chosen by a search method.

	Opentuner	**C.Gym.Greedy**	**C.Gym.Random**	**Supersonic**
simplifycfg	0.84%	3.29%	1.23%	**10.78%**
early-cse-memssa	0.78%	0.18%	0.92%	**6.80%**
gvn	0.69%	1.83%	0.97%	**5.98%**
instcombine	0.75%	0.12%	1.17%	**8.99%**
newgvn	0.74%	2.90%	1.02%	**7.66%**
others	**96.21%**	91.67%	94.69%	59.79%

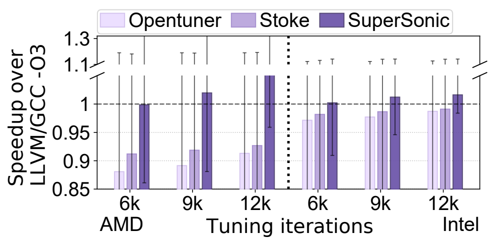
AMD platform and only a minor slowdown over XGBoost on two benchmarks on the Intel platform. SUPERSONIC improves Chameleon, the second-best-performing method, by up to 1.22x, improving the default schedules by up to 1.74x.

5.5.3 Case Study 3: Code Size Reduction

Figure 5.5 reports the code size reduction conducted on the Intel platform using LLVM. We note that an average code size reduction of 4% is considered to be significant [215–217]. SUPERSONIC improves -Oz for all but two test benchmarks. It delivers the best reduction for 90% of the test benchmarks. For those benchmarks where SUPERSONIC does not deliver the best reduction, the difference between SUPERSONIC and the best-performing method is small, less than 1%. On average, SUPERSONIC gives the highest mean code size reduction, improving LLVM -Oz by up to 1.57x.



(a)



(b)

Figure 5.6: Speedup over LLVM/GCC -O3 for superoptimisation under different search time (a) and iterations (b).

Table 5.4 lists the top-5 most frequently-appeared LLVM passes in the compiler pass list that give the best code size reduction. Because these passes are often included in a compiler sequence that offers a high code size reduction, they are likely to be important for reducing code size. The table also shows how often a search method chooses a pass as an action when optimising a benchmark. Compared to other methods[1], SUPERSONIC picks an important pass more often during the search, suggesting that it learns the importance of optimisation passes.

5.5.4 Case Study 4: Superoptimisation

Figure 5.6 shows the superoptimisation results using LLVM and GCC. SUPERSONIC outperforms Stoke and OpenTuner on most of the test benchmarks with a higher mean speedup. Increasing the tuning iterations (and the search time) during deployment can improve the search performance because it allows the search algorithm to explore the optimisation space better and uses feedback to improve its search strategy. Although the STOKE developers note that their mutation engine is not mature enough to outperform LLVM/GCC -O3, we demonstrate that RL can deliver noticeable improvement by better exploring the optimisation space (up to 1.34x).

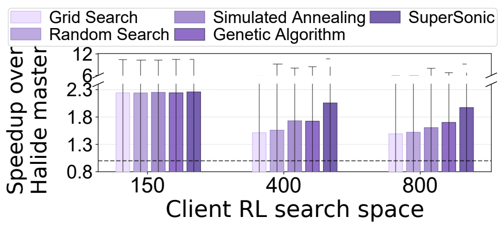
Figure 5.7 shows a kernel from STOKE Hacker benchmark dataset, compiled by LLVM -O3, and the best-performing version found by SUPERSONIC. The SUPERSONIC code for the `_Z3p23i` kernel is 18 lines shorter, 1.2x faster than -O3. This is one of the many examples where the Supersonic-driven superoptimisation generates faster code

[1]In theory, with a sufficiently larger number of samples, each pass will have a 1/123 chance to be chosen by C.Gym.Random in Table 5.4.

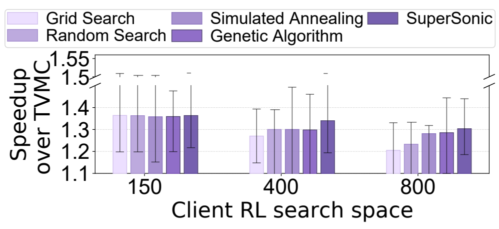
```
.L_4004e0_LLVM_O3:             .L_4004e0_SuperSonic:
  movl    $0x60106f,%eax         movl    $0x601191,%edi
  movl    $0x601191,%edi         movq    $0x400990,%rcx
  movl    $0x400700,%edi         movl    $0x400700,%edi
  pushq   %rax                   pushq   %rax
  nopl    0x0(%rax,%rax,1)       nopl    0x0(%rax,%rax,1)
  ...                            ...
  callq   4008f0 <_Z3p23i>       callq   4008f0 <_Z3p23i>

._Z3p23i:                      ._Z3p23i:
  movl    %edi,%eax              popcntl %edi,%ecx
  shrl    %eax                   movq    %rcx,%rax
  andl    $0x55555555,%eax       retq
  subl    %eax,%edi
  movl    %edi,%eax
  andl    $0x33333333,%eax
  shrl    $0x2,%edi
  andl    $0x33333333,%edi
  addl    %eax,%edi
  movl    %edi,%eax
  shrl    $0x4,%eax
  addl    %edi,%eax
  andl    $0xf0f0f0f,%eax
  movl    %eax,%ecx
  shrl    $0x8,%ecx
  addl    %eax,%ecx
  movl    %ecx,%eax
  shrl    $0x10,%eax
  addl    %ecx,%eax
  movzbl  %al,%eax
  retq
  nopl    0x0(%rax,%rax,1)
```

Figure 5.7: SUPERSONIC finds a better program (right) over LLVM -O3 (left) for a STOKE Hacker kernel (`_Z3p23i`) that counts the number of bits, by using the `popcntl` instruction.

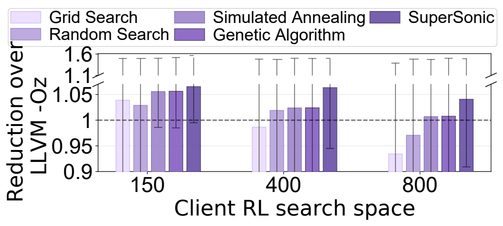
over -O3.



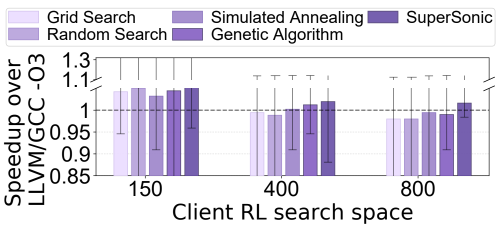
(a) Optimizing Image Pipelines



(b) Neural Network Code Generation



(c) Code size reduction



(d) Superoptimisation

Figure 5.8: Comparing the performance of a client RL chosen by different search algorithms during the client RL search stage. We vary the number of client RL combinations by varying the number of candidate RL components. The SUPERSONIC chosen client RL gives the best overall performance during deployment.

5.5.5 Further Analysis

Compare to Other Client RL Search Strategies

This experiment compares SUPERSONIC's deep RL-based meta-optimiser against four widely used parameter search techniques: grid search, simulated annealing, random search, and genetic algorithms. We set the number of search iterations to 100 for all search algorithms. We also vary the search space by adding more candidate RL components, where the number of RL component combinations varies between 150 and 800. For a chosen RL client, we follow the same parameter fine-tuning process to fine-tune the selected RL components (Section 5.3.4). Figure 5.8 compares the performance of the client RL chosen by different search algorithms during the RL client search stage. As can be seen from the diagram, all client RL give an average improvement over the baseline. SUPERSONIC finds a better client RL during the limited client RL search budget, leading to overall better performance across search space sizes and case studies. We note that the limited client RL search budget restricts the performance of finding

Table 5.5: Search overhead required by Supersonic to exceed the performance given the best-performing alternative

Use cases	% of search time (& raw numbers)			% of iterations (& raw numbers)		
	MIN	**GeoMean**	**MAX**	**MIN**	**GeoMean**	**MAX**
Case study 1	3.6 (21 mins)	39.6 (4 hours)	72.1 (6 hours)	3.9 (1,425)	44.0 (15,821)	75.3 (27,141)
Case study 2	1.0 (1 min)	39.1 (24 mins)	74.1 (45 mins)	4.2 (885)	32.3 (3,885)	68.8 (8,256)
Case study 3	1.5 (3 sec)	31.0 (61 sec)	61.1 (2 mins)	2.1 (738)	29.7 (10,714)	83.5 (30,068)
Case study 4	1.1 (22 sec)	32.8 (11 mins)	42.3 (14 mins)	1.2 (1,478)	36.5 (43,745)	46.9 (56,387)

good client RL with a large search space (e.g., when the number of RL component combinations is 800). Increasing the search budget will allow the meta-optimiser to find a better client RL to improve the resulting performance.

Deployment-stage Search Time Relative to the Best-performing Alternative

Table 5.5 shows the relative search time and iteration count required by Supersonic-tuned client RL to exceed the results given by the best-performing competitive scheme. The table shows the minimum, average, and maximum search overhead across test benchmarks. On average, the RL architecture found by Supersonic converges faster, requires less than 39.6% of the search time was used by the best-performing alternative search algorithm to deliver better results. This means the RL architecture chosen by Supersonic can start delivering a better optimised code with, on average, 1.75x less search time (up to 100x) compared to the best-performing alternative tuning algorithm that runs longer. While search methods like genetic algorithms have no upfront training cost, they must search each time afresh. Supersonic performs a one-off offline RL search and tuning, but the tuned client RL significantly reduces the search cost for any *new* programs after that.

Chosen RL Components

Table 5.3 shows that the client RL components chosen by Supersonic vary from one task to another. We also observe that the optimal RL found by Supersonic is consistent across cross-validation runs for a given task. This means the client RL architecture can generalise across inputs of a given task. We notice that using MCTS with DNNs for state representation gives good performance for three out of the four tasks, but it is less effective for code size optimisation compared to PPO. Using MCTS for code reduction gives an average reduction of 0.6% instead of 6.5% delivered by PPO. This is perhaps due to a large number of discrete actions (i.e., the compiler passes) in the optimisation space, which requires more search time to learn a good value function to efficiently guide the MCTS simulation.

5.6 Summary

In this chapter, we have presented Supersonic, a framework for building RL-based performance tuners. Supersonic provides the capability to automate the process of designing and tuning RL algorithm structures. We evaluate Supersonic by applying it to four different optimisation tasks. Experimental results show that the RL architecture found by Supersonic delivers better overall performance than alternative search techniques, including hand-tuned RL strategies.

Supersonic supports mainstream and emerging RL programming environments like RLlib and CompilerGym. It is designed to provide customizable interfaces to allow developers to introduce new algorithms and methods to be used in the RL policy architecture. As the community provides more models and methods, Supersonic will be able to explore a more comprehensive policy search space. As a result, there will be less of a need to create domain-specific methods for each project. In the long term, we hope that Supersonic will work out-of-the-box for most performance developers once they have defined their optimisation tasks.

Chapter 6

Robust Machine Learning for Program Modelling

In the previous two chapters, we demonstrated how to make ML program modelling more precise and how to lower the barrier to applied ML. In this chapter, we present a framework that uses statistical analysis to detect drifting data and improve model robustness during deployment. The chapter is organised as follows. Section 6.2 illustrates how *data drift* affects the performance of underlying models during deployment. Sections 6.3 and 6.4 explain how our approach is formulated. Section 6.5 then describes the experimental setup. Section 6.6 presents the experimental results, followed by Section 6.7, which concludes the chapter.

6.1 Introduction

During ML deployment time, small changes in hardware or application workloads can reduce decision accuracy and model robustness [218]. This often arises from "*data drift*" [219, 220], where the training and test data distributions no longer align. In code optimisation, this can result from changes in workload patterns, runtime libraries, or hardware micro-architectures. It is a particular challenge for ML-based performance optimisations, where obtaining sufficient performance training data is difficult [221].

Existing efforts to enhance ML robustness for code optimisation have predominantly focused on improving the learning efficiency or model generalisation during *the design time*. These approaches include synthesising benchmarks to increase the training data size [7, 152, 222], finding better program representations [8, 9, 145, 223], and combining multiple models to increase the model's generalisation ability [9, 224, 225]. While important, these design-time methods are unlikely to account for all potential changes during deployment [149]. Although there has been limited exploration into the validation of model assumptions [36] for runtime scheduling, existing solutions assume a specific ML architecture and lack generalizability.

We introduce PROM, an open-source toolkit designed to address data drift *during deployment*, specifically targeting code optimisation and analysis tasks. PROM is not intended to replace design-time solutions but to offer a complementary approach to improve ML robustness during deployment. Its primary objective is to ensure the reliability of an already deployed system in the face of changes and support continuous improvements in the end-user environment. To this end, PROM offers a Python interface for training and deploying supervised ML models, focusing on detecting data drift post-deployment. Model and application developers can integrate PROM into the ML workflow by implementing its abstract class, usually requiring just a few dozen lines of code.

PROM adopts the emerging paradigm of *prediction with rejections* [37–40], which identifies instances where predictions may be inaccurate, allowing for corrective measures when data drift occurs. For instance, an ML-based performance tuner can notify users when the model prediction, like the compiler flags to be used for a given program, might not yield good performance, prompting them to use alternative search processes [2, 41] to find better solutions [42]. Likewise, a bug detector can alert users to potential false positives for expert inspection. Essentially, this capability allows using alternative metrics when predictive model performance deteriorates. By flagging likely mispredictions, PROM supports continuous learning by using these mispredicted instances as additional training samples to enhance model performance in a production environment.

To evaluate whether a predictive model may mispredict a test input, PROM computes the *credibility* and *confidence* scores of the prediction during deployment. Credibility measures the likelihood that a prediction aligns with the learned patterns. High credibility means the test input is highly consistent with the training data, suggesting the model's prediction is likely to be reliable. Conversely, the confidence score estimates the model's certainty in its prediction. PROM uses the two scores to determine whether the model's outcome should be accepted or requires further investigation. Our intuition is that a prediction is reliable only if the model shows high confidence in its predictions and these predictions, along with the model's confidence level, are credible.

Prom employs CP theory [69] to assess the test input's *nonconformity* to compute the confidence and credibility scores of a prediction. Nonconformity is measured by comparing the test input against samples from a *calibration dataset* held out from the model training samples. The idea is to observe the ML model's performance on the calibration set and then evaluate the test input's similarity (or "strangeness") to the calibration samples.

Prom draws inspiration from recent advances in applying CP to detect drifting samples in malware prediction [37, 39] and wireless sensing [40]. While CP has shown promise in these domains, its effectiveness for code optimisation tasks remains unclear. This chapter presents the first application of CP to code optimisation tasks like loop vectorisation and neural network code generation. Doing so requires addressing several key limitations of existing approaches. One major drawback of prior methods is that they rely on the entire calibration dataset to compute the nonconformity of test samples, which is ill-suited for code with diverse patterns. For example, if we want to estimate the model's accuracy on a computation-intensive program, including many calibration samples with different characteristics (e.g., memory-bound benchmarks) can mislead and bias the credibility estimation. Furthermore, previous solutions employ a monolithic nonconformity function that lacks robustness across different ML models and tasks. They also do not support regression methods and usually require changing the underlying ML model [40]. Prom is designed to overcome these pitfalls.

Unlike prior work [37, 39, 40], Prom adopts an adaptive scheme to measure a test sample's nonconformity. Instead of using the entire calibration dataset, Prom dynamically selects a subset of calibration samples with similar characteristics to the test sample in the feature space defined by the ML model. When computing the nonconformity score, Prom assigns different weights to these selected samples based on their distance from the test sample, giving higher weight to closer samples. This scheme allows Prom to construct a calibration set that closely matches the test sample distribution, thereby improving nonconformity estimation accuracy.

Prom improves conformity reliability by using multiple statistical functions to compute nonconformity scores and applying a majority voting scheme to approve or reject predictions. It is extensible, allowing easy addition of new nonconformity functions. Prom also supports regression by combining CP with clustering algorithms. Unlike [40], it uses a model-free approach instead of learning a probabilistic classifier for data drift detection.

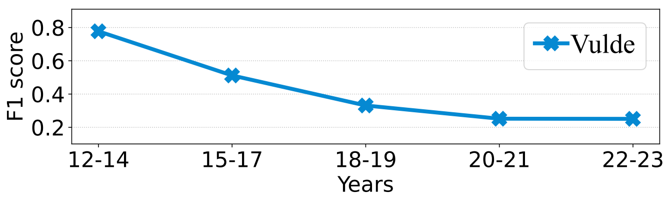
As a potential mitigation of data drift, we showcase that PROM enables a learning-based method to enhance its robustness and maintain reliable performance over time. This is achieved by adopting incremental learning [226, 227] to retrain a deployed model using drifting samples from the production environment. Depending on the specific application, one can relabel a few test samples flagged by PROM and use the relabelled data to retrain the model, by only seeking feedback and user intervention on instances showing data drift. We stress that incremental learning is just one of the possible remedies, but such mitigations hinge on accurately detecting drifted samples - the main focus of this work.

We demonstrate the utility of PROM by applying it to 13 representative ML methods developed by independent researchers for code optimisation and analysis [8, 9, 11, 21, 49, 135, 145–147, 228–230]. Our case studies cover five problems, including heterogeneous device mapping [8, 11, 145, 229], GPU thread coarsening [11, 145, 228], loop vectorisation [145, 230], neural network code generation [146] and source-code level bug detection [9, 21, 49, 147]. Experimental results show that PROM can successfully identify an average of 96% (up to 100%) of test inputs where the underlying ML model mispredicts with a false-positive rate[1] of less than 14%. By employing incremental learning, PROM substantially enhances prediction performance in the operational environment, allowing the deployed model to match the performance achieved during its design phase. Notably, this improvement is achieved with minimal user intervention or profiling overhead. In our evaluation, PROM requires relabelling fewer than 5% of the samples identified as drifted by PROM, which are then used to update the model through retraining.

6.2 Motivation

As a motivating example, consider training an ML model to detect source code bugs. This pilot study uses the state-of-the-art method VULDEEPECKER [21], which employs an LSTM network. Following the original setup, we train and test the models on labelled samples from the CVE dataset, using open-source tools and ensure results comparable to those in the source literature.

[1]This occurs when the ML model gives an accurate prediction, but PROM believes otherwise.



(a) Data drift leads to deteriorating performance in vulnerability detection.

```
1  static sftp_parse_attr_3(...) {
2  ssh_string name = NULL;
3  ...
4  if ((name = buffer_get_ssh_string(buf))...)
5      {...}
6  ssh_string_free(name);
7  ...
8  ssh_string_free(name);}
```

(b) CVE-2012-4559: A "*double-free*" for `name` at lines 6 and 8.

```
1  #define NUMT 100
2
3  CURLcode curl_easy_cleanup(...){
4  sts = ...;
5  if(sts) {...
6     hsts_free(sts);}}
7  ...
8  static void* pull_one_url(...){
9  for (i = 0; i < NUMT; i++) {
10     CURL* curl = ...;
11     ...
12     curl_easy_cleanup(curl); }}
13
14 int main(...){...
15 for (i = 0; i < NUMT; i++) {
16     pthread_create(pull_one_url,...);
17 ...}
```

(c) CVE-2023-27537: A potential "*double-free*" due to multiple concurrent threads can invoke `hsts_free` at line 6 at the same time.

Figure 6.1: Motivation example: impact of data drift on ML models for code vulnerability detection.

Figure 6.1(a) shows what happens if we train the models using CVE data collected between 2012 and 2014 and then apply them to real-life code samples developed between

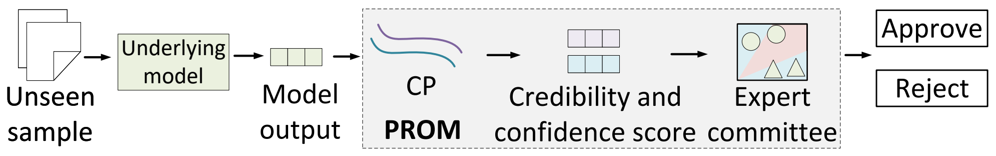


Figure 6.2: Workflow of PROM during deployment.

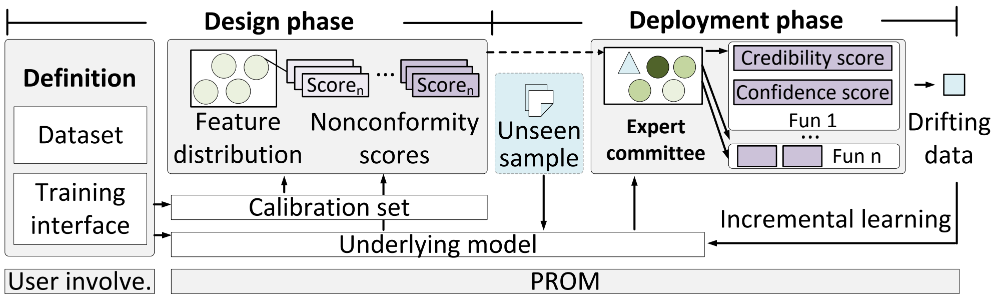


Figure 6.3: At design time, PROM splits the training data into training and calibration sets. During deployment, it calculates credibility and confidence scores, using majority voting to detect drifting samples. These samples can then be labelled for model updates via offline incremental training.

2015 and 2023. This mimics a scenario where the code and bug patterns may evolve after a trained model is deployed. The trained model achieves an F1 score of more than 0.8 (ranging from 0 to 1, with higher being better) when the test and training data are collected from the same time window. However, the F1 score drops to less than 0.3 when tested on samples collected from future time windows. This shows that data drift can severely impact model performance, which was also reported in prior studies [9, 72].

As an example of code pattern changes, Figures 6.1(b) and 6.1(c) show two cases of "*double-free*" vulnerabilities, one from 2012 and one from 2023. Earlier cases were simpler, where the same memory was freed twice (e.g., the buffer `name` is freed on lines 6 and 8). A model trained on these samples is unlikely to detect the more complex later case in Figure 6.1(c), where a "double-free" occurs due to concurrent threads calling the same buffer-free routine. Because it is difficult to collect a training dataset which covers all possible code patterns seen at deployment time, data drift can happen in the real-life deployment of ML models for code-related tasks.

```python
from prom import ModelInterface

class ModelDefinition(ModelInterface):
    def __init__(self, model, cali_dataset):
        # Setting the underlying model
        self.model = model
        self.calibration_data = cali_dataset
        super().__init__()

    def data_partitioning(self, dataset, calibration_ratio=0.1):
        # Splitting the training data into training and calibration sets
        ...
        return training_data, calibration_data

    def predict(self, X, significant_level=0.1):
        ...
        # Underlying model prediction function also returns a
            probabilistic vector
        pred, probability = self.model.predict(X)
        #Call the expert committee
        drifted = self.classifier(probability, significant_level)
        return pred, drifted

    def feature_extraction(self, X):
        # Convert the model input into a feature vector
        ...
        return self.model.feature_extraction(X)

if __name__ == '__main__':
  model = ModelDefinition(mymodel, dataset)
  pred, drifted = model.predict(sys.argv[1])
```

Figure 6.4: Simplified code template of PROM.

6.3 PROM: A Robust ML System for Program Modelling

Figure 6.2 illustrates how PROM can enhance learning-based methods *during deployment*. Users of PROM are ML model developers. PROM requires no change to a user model's structure and working mechanism. We refer to the user model as the "*underlying model*".

User involvement. For a given input, the underlying model works as it would without PROM during inference. Users of PROM need only provide the model training dataset and the training interface to PROM.

Added values of Prom. PROM serves as an open-source framework that provides: an adaptive scheme for constructing the calibration set; a collection of conformal prediction methods that take as input the intermediate results (e.g., probabilities of each class label produced by the underlying model) to compute a credibility and confidence score, which is used to suggest whether to accept the prediction outcome; and an ensemble approach to detect mispredictions.

Scope. PROM goes beyond a standard CP library [157, 158] and is not limited to compiler optimisation. The methodology applies more broadly to program analysis and software engineering tasks where data scarcity or distributional shifts hinder robust model training. Examples include vulnerability detection, automated program repair, performance prediction, and software testing. It addresses a key challenge in ML for insufficient-data tasks by offering an orthogonal approach to enhance model reliability during deployment. This generality makes PROM a valuable tool not only for ML in compilers but also for a wider range of program modeling scenarios.

6.3.1 Implementation

We implemented PROM as a Python package, offering an API for use during both the ML model design and deployment phases. PROM provides interfaces to assess framework setup (Sec. 6.4.2), automatically searches for hyperparameter settings on the training and calibration datasets, and provides examples to showcase its utilities. These include all the case studies and ML models used in this work (Sec. 6.5) and simpler examples for beginners. PROM supports classification and regression methods built upon classical ML methods (e.g., support vector machines) and more recent deep neural networks. Figures 6.3 and 6.4 provide an overview of PROM's role during the model design and deployment phases, described as follows.

Model Design Phase

As depicted in Figure 6.4, using PROM requires overwriting a handful of methods in PROM's `ModelDefinition` class and exposing the underlying model's internal outputs.

Training data partitioning. PROM is based on split CP [69, 73], which divides the training data into a "*training dataset*" and a "*calibration dataset*". PROM uses the calibration dataset to detect drifting test samples during deployment. By default, it randomly sets aside 10% of the training data (up to 1,000 samples) for calibration, a method shown to be effective in prior work [73, 231]. PROM also offers a way to assess the suitability of the calibration dataset (Sec. 6.4.2), or users can provide their own holdout calibration dataset.

Changes to user models. For classification tasks, the user model should implement a prediction function (line 15) that returns both a prediction and a probability vector. Most ML classifiers already associate probabilities with each class, which can be easily accessed. Popular frameworks like scikit-learn, PyTorch, and TensorFlow directly provide probability distributions. For example, scikit-learn's `predict_proba` method offers probabilistic values for 33 common ML models. In neural networks, probabilities can usually be extracted from the hidden layer before the output. For regression models [73, 232], PROM applies a similar approach.

Feature extraction. The user needs to provide a feature extraction function to convert the model input into a feature vector of numerical values. For example, this could be a neural network to generate embeddings of the input [30, 233] or a function to summarise the input programs into numerical values like the number of instructions [2]. Since most ML models already require this function, this requirement should not incur additional engineering effort.

Process calibration dataset. The user model is trained *outside* the PROM framework using any method the user deems appropriate. The trained model is loaded and passed as a Python object to PROM. With the calibration dataset and the trained user model, PROM *automatically* preprocesses the calibration dataset offline before deploying the ML model (Sec. 6.4). This is done by applying the learned model to each calibration sample and using a nonconformity function described in Sec. 6.4.1 to calculate a score. This score reflects how 'strange' or '*non-conforming*' each calibration example is compared to the learned model.

Significant level. The user can set a *significant level*, $1 - \epsilon$, to determine the severity of data drifts. A smaller ϵ can reduce the probability of misprediction by PROM but can lead to a higher false positive rate. By default, PROM sets ϵ to 0.1.

Overwrite the prediction function. As a final step, the model developer needs to overwrite a prediction function (`predict`) to return the model's prediction for a test input. The original prediction function can be invoked as a subroutine, and the PROM prediction function is used during deployment.

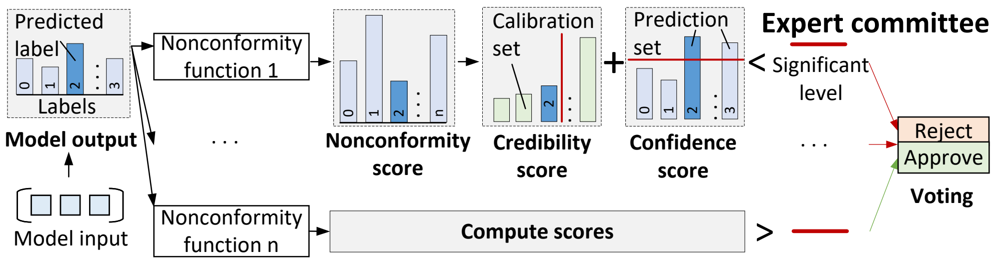


Figure 6.5: Prom integrates multiple nonconformity functions that vote to reject or approve the ML prediction.

Model Deployment Phase

During deployment, the user model functions as usual, taking a test sample and making a prediction. The difference with Prom is that it also suggests whether to accept or reject the prediction. The user can use this outcome to identify mispredictions and provide ground truth, and Prom will use these relabelled drifting samples to update the model.

6.4 Methodology

As shown in Figure 6.5, during deployment, Prom uses multiple (default: 4) nonconformity functions to independently compute the prediction's credibility and confidence scores. These scores are compared to a pre-defined significance level, $1 - \varepsilon$ (Sec. 6.3.1), to decide whether to accept the prediction. If both scores fall below the threshold, the test sample is flagged as drifting. The results are then aggregated using majority voting, where each nonconformity function (expert) decides whether the prediction should be accepted, forming an ensemble "expert committee".

6.4.1 Nonconformity Measures

Prom computes the p-value of a prediction using nonconformity functions, which are then used to derive credibility and confidence scores. Previous work on using CP to detect drifting samples [37, 39, 40] focuses primarily on classification tasks and does not extend to regression. Prom is the first framework to support both classification and regression for data drift detection. Additionally, prior methods only consider the predicted label, ignoring the probability distribution across labels, whereas Prom accounts for probabilities across all labels in classification tasks.

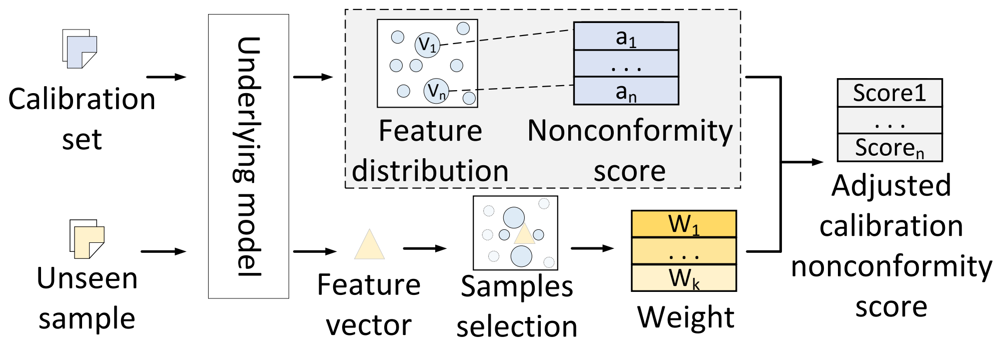


Figure 6.6: PROM dynamically selects a subset of the holdout calibration dataset to assess the test input's nonconformity. Each chosen sample is weighted based on its Euclidean distance to the test sample in the feature space, and these weights are used to adjust the nonconformity score of the chosen calibration samples.

Nonconformity functions

PROM integrates multiple ready-to-use nonconformity functions, and the choice of nonconformity functions can be customised by passing a list to the relevant PROM interface. By default, PROM uses 4 nonconformity functions: *LAC* [234], *TopK* [235], *APS* [236] and *RAPS* [235]. Other nonconformity functions can be easily incorporated into PROM by implementing an abstract class.

For regression tasks, our nonconformity functions compute the nonconformity score using the residual error between the prediction and the ground truth. Since we do not have the ground truth during deployment, we approximate it using the k-nearest neighbour algorithm [237, 238]. This approximation is based on the null hypothesis that the test sample is similar to those encountered during design time.

Specifically, PROM finds the k-nearest neighbors (we set k to be 3 in this work), denoted as $N_k(n+1)$, of s_{n+1}. The distance is measured by computing the Euclidean distance [239] between the test sample s_{n+1} and calibration samples on the feature space. We then approximate the true value of s_{n+1} by averaging the distance of k-nearest neighbors, $y_{s_{n+1}} = \frac{1}{k}\sum_{i\in N_k(n+1)} y_{s_i}$. The estimated value is then passed to a regression-based nonconformity function to compute the nonconformity score of the test sample. Essentially, we approximate the ground truth by assuming the samples seen at the design time are sufficient to generate an accurate prediction. If this assumption is violated due to drifting test samples, it will likely result in a large residual error (and a greater nonconformity score).

Computing p-value

PROM uses a *p-value* to assess whether a test sample s fits within the prediction region defined by the calibration dataset, which reflects the training data distribution. To compute the p-value, a subset of calibration samples is selected, their nonconformity scores are adjusted, and these scores are used to derive the p-value for the model's prediction.

Calibration nonconformity scores. As shown in Figure 6.6, PROM dynamically selects a subset of calibration samples to adjust the nonconformity score. Specifically, it computes the Euclidean distance between each calibration sample and the test input based on their feature vectors, sorting the samples by distance. By default, the closest 50% of calibration samples are selected. If the dataset contains fewer than 200 samples, all of them are selected; a threshold that can be configured via the PROM API. This nearest subset of calibration samples is chosen to estimate the nonconformity of the test data relative to the training data. These distances are also used as weights to adjust nonconformity scores.

For a calibration dataset with n samples, $(a_1, a_2, \ldots, a_n)$, PROM computes nonconformity scores and feature vectors $(v_1, v_2, \ldots, v_n)$ offline. For a new test sample s_{n+1}, it extracts the feature vector v_{n+1}, calculates the distances to calibration samples, and selects K samples. The weight w_i for each selected sample i is given by:

$$w_i = \exp\left(-\frac{\|v_i - v_{n+1}\|^2}{\tau}\right), i \in \{1, ..., k\} \tag{6.1}$$

where $\|v_i - v_{n+1}\|^2$ is the l2-norm, and τ is a temperature hyperparameter (default 500). This weight is used to adjust the nonconformity score: $a_i = w_i \times a_i$.

P-value for classification. After selecting the calibration samples and adjusting their nonconformity scores, PROM calculates the p-value for each test sample. First, it determines the nonconformity score $a_{n+1}^{y^p}$ for the predicted outcome y^p. Then, it evaluates the similarity of the test sample to the chosen calibration samples to compute the p-value, p_{s_i}, as:

$$p_{s_i} = \frac{\text{COUNT}\left\{i \in \{1, ..., n\} : y_i = y^p \text{ and } a_i^{y^p} \geq a_{n+1}^{y^p}\right\}}{\text{COUNT }\{i \in \{1, ..., n\} : y_i = y^p\}} \tag{6.2}$$

This counts the proportion of calibration samples with the predicted label y^p whose nonconformity scores are $\geq$ than the test sample's score. A low p-value (near $1/n$)

suggests high nonconformity, meaning the test sample is significantly different from the training samples. A high p-value (close to 1) indicates strong conformity, showing the test sample closely matches the calibration samples for label y^p.

P-value for regression. We extend classification p-values to regression tasks by generating labels in the calibration dataset using K-means clustering [240]. Specifically, we partition the calibration set into K clusters, $y_1, y_2, \ldots, y_K$, based on the feature vectors of each sample. The optimal number of clusters (K) is determined using the Gap statistic method [241], which compares the within-cluster sum of squares from K-means to that of random clustering over K values from 2 to 20. The Euclidean distance between feature vectors is used as the clustering metric. A larger gap indicates better clustering quality, and Prom selects the K with the highest gap. During deployment, test sample labels are assigned based on the nearest neighbour in the feature space. Prom then computes the *p-value*, as in classification, using Equation 6.2 during both design and deployment.

6.4.2 Initialization Assessment

Prom provides a Python function to evaluate whether the framework is properly initialised at design time after obtaining the calibration dataset (Sec. 6.3.1) and the trained underlying model. This is achieved by computing the coverage rate by performing cross-validation on the holdout calibration dataset. Specifically, Prom automatically splits the calibration dataset R times ($R = 3$ by default) into two *internal* datasets for calibration (80%) and validation (20%). It then applies the trained model to the internal validation set and calculates the coverage as:

$$\frac{1}{R}\sum_{j=1}^{R}\frac{1}{n_{\text{val}}}\sum_{i=1}^{n_{\text{val}}}\mathbb{1}\left\{y_i^{(\text{val})} \in C\left(x_i^{(\text{val})}\right)\right\} \approx 1-\varepsilon \tag{6.3}$$

where n_{val} is the size of the validation set, $y_i^{(\text{val})}$ is the ground truth of the ith validation example, and $C(x_i^{(\text{val})})$ is the prediction region of the ith validation example computed by Prom using the calibration data. The coverage ratio should be approximately the pre-defined significant level, $1 - \varepsilon$, with minor fluctuations in deviation [73]. A large deviation indicates an ineffective initialisation, which usually stems from a poorly trained or designed underlying model. In this case, Prom will alert the users when the deviation is more than 0.1, enabling them to enhance the underlying model or adjust the significance level during the design time.

A parameter selection function with a grid search algorithm is provided to help users set the optimal parameters automatically, such as the significant level and cluster size (Sec. 6.4.1). After evaluating the candidate parameters on the validation dataset, PROM will save the selected parameters and use them to predict the confidence in the underlying model at deployment time.

6.4.3 Credibilty and Confidence Evaluation

Credibility score. For each nonconformity function, we use the p-value (Sec. 6.4.1) computed for the predicted class as the credibility score. The higher the p-value is, the more likely the test sample is similar to the training-time samples, hence a higher credibility score.

Confidence score. PROM estimates the confidence score by evaluating the statistical significance of the prediction using a Gaussian function, $f(x) = e^{-\frac{(x-1)^2}{2\times c^2}}$, where c (default 3) is a constant, and x is the *prediction set size* for the test sample. The prediction set includes labels likely associated with the test sample, where the nonconformity score exceeds the significance level, $1-\varepsilon$. An empty set suggests the test sample is not linked to any known class, while multiple labels indicate uncertainty, resulting in a low confidence score. As with the credibility score, the prediction set is built from the p-value (Sec. 6.4.1). Regression tasks apply the same approach, using the labels introduced by clustering (Sec. 6.4.1). According to our *prediction with rejections* strategy, a sample is flagged as drifting if both scores fall below the significance level.

6.4.4 Improve Deployment Time Performance

PROM can enhance the performance of deployed ML systems through incremental learning [226, 227]. For example, suppose a predicted compiler option is likely to be sub-optimal. In that case, the compiler system can use auto-tuning to sample a larger set of configurations to find the optimal one. The idea is to apply other (potentially more expensive) measures to drifting samples. The ground truths can then be added back to the training dataset of the underlying model in a feedback loop for *offline* retraining. Since model retraining occurs only during instances of data drift, it reduces the overhead associated with the collection of training data.

As we will show later, updating a trained model with up to 5% of identified drifting samples significantly enhances robustness post-deployment. The goal is not to reduce

Table 6.1: Case studies and their setups

#Use cases	#Test methods	Models	Tasks
C1: Thread Coarsening	Magni *et al.* [228]	MLP	Class.
	DeepTune [11]	LSTM	
	IR2Vec [242]	GBC [243]	
C2: Loop vectorisation	K.Stock *et al.* [230]	SVM	Class.
	DeepTune [11]	LSTM	
	Magni *et al.*	MLP	
C3: Heterogeneous Mapping	DeepTune [11]	LSTM	Class.
	ProGraML [8]	GNN	
	IR2Vec [242]	GBC	
C4: Vulnerability Detection	VulDeePecker [21]	Bi-LSTM	Class.
	CodeXGLUE [49]	Transformer	
	LineVul [147]		
C5: DNN Code Generation	Tlp [146]	BERT [233]	Reg.

training time but to provide a framework for assessing robustness. Without such a system, frequent retraining or risking performance degradation is required. In code optimisation tasks, the main expense is labelling data, not training, and by focusing only on mispredicted samples (e.g. for relabelling), our approach reduces labelling overhead and shortens retraining time. By filtering out mispredictions, Prom detects ageing models and supports implementing corrective methods. This, in turn, will improve user experience and trust in ML systems.

6.5 Experimental Methodology

Evaluation methodology. As shown in Table 6.1, we apply Prom to detect drifting samples across 5 case studies, covering 13 representative ML models for classification and regression. We faithfully reproduced all methods following the methodologies in their source publications and used available open-source code. We adhered to the original training methods to ensure *comparable design-time results.*

Introduce data drift. We introduce changes by separating the training and testing data. We try to mimic practical scenarios by testing the trained model on a benchmark suite not used in the training data or code samples newer than the model training data and the Prom calibration dataset. Note that our primary goal is to detect whether Prom can successfully detect drifting samples, not to improve the design of the underlying model.

Prior practices. Prior work often assumes an ideal scenario by splitting training and test samples at the benchmark or method level, where both sets may share similar characteristics [244]. In contrast, our evaluation introduces data drift to reflect real-world scenarios where workload characteristics change during deployment. As a result, baseline ML models perform worse on test samples than reported in their original publications [229, 245].

6.5.1 Case Study 1: Thread Coarsening

This problem develops a model to determine the optimal OpenCL GPU thread coarsening factor for performance optimisation. Following [228], an ML model predicts a coarsening factor (ranging from 1 to 32) for a test OpenCL kernel, where 1 indicates no coarsening.

Underlying models. We consider three ML models designed for this problem: a Multilayer Perceptron (MLP) used in [228], an LSTM used in DeepTune [11], and a Gradient boosting classifier (GBC) used in IR2Vec [242]. Like these works, we train and test the models using the labelled dataset from [228], comprising 17 OpenCL kernels from three benchmark suites on four GPU platforms.

Methodology. As in [11, 145], we train the baseline model using leave-one-out cross-validation, which involves training the baseline model on 16 OpenCL kernels and testing on another one. We then repeat this process until all benchmark suites have been tested once. To introduce data drift, we train the ML models on OpenCL benchmarks from two suites and then test the trained model on another benchmark suite.

6.5.2 Case Study 2: Loop Vectorisation

This task constructs a predictive model to determine the optimal Vectorisation Factor (VF) and Interleaving Factor (IF) for individual vectorizable loops in C programs [135, 246]. Following [135], we explore 35 combinations of VF (1, 2, 4, 8, 16, 32, 64) and IF (1, 2, 4, 8, 16). We use LLVM version 17.0 as our compiler, configuring VF and IF individually for each loop using Clang vectorisation directives.

Underlying models. We replicate three ML approaches: K.Stock *et al.* [230] (using SVM), DeepTune [11], and Magni *et al.* [228], which use neural networks. We use the 6,000 synthetic loops from [135], created by changing the names of the parameters from 18 original benchmarks in the LLVM vectorisation test suite. We used the labelled data from [145], collected on a multi-core system with a 3.6 GHz Intel Core i7 CPU and 64GB of RAM.

Methodology. Following [145], we initially allocate 80% (4,800) of loop programs to train the model, reserving the remaining 20% (1,200) for testing its performance. To introduce data drift, we use programs generated from 14 benchmarks for training and evaluate the model on the programs from the remaining 4 benchmarks.

6.5.3 Case Study 3: Heterogeneous Mapping

This task develops a binary classifier to determine if the CPU or the GPU gives faster performance for an OpenCL kernel.

Underlying models. We replicated three deep neural networks (DNNs) proposed for this task: DeepTune [11],
ProgramL [8], and IR2Vec [242]. We use the DeepTune dataset, comprising 680 labelled instances collected by profiling 256 OpenCL kernels from 7 benchmark suites.

Methodology. Following [11], we train and evaluate the baseline model using 10-fold cross-validation. This involves training a model on programs from all but one of the subsets and then testing it on the programs from the remaining subset. To introduce data drift, we train the models using 6 benchmark suites and then test the trained models on the remaining suites. We repeat this process until all benchmark suites have been tested at least once.

6.5.4 Case Study 4: Vulnerability Detection

This task develops an ML classifier to predict if a given C function contains a potential code vulnerability. Following [9], we consider the top-8 types of bugs from the 2023 CWE [247].

Underlying models. We replicated four representative ML models designed for bug detection: CodeXGLUE [49], LineVul [147], both based on Transformer networks, and VulDeePecker [21] which is based on a Bi-LSTM network. We evaluate this task with a dataset comprising 4,000 vulnerable C program samples labelled with one of the eight vulnerability types, each with around 500 samples. The vulnerable code samples cover 2013 and 2023 and are collected from the National Vulnerability Database (NVD), CVE, and open datasets from GitHub.

Methodology. As with prior approaches, we initially train the model on 80% of the randomly selected samples and evaluate its performance on the remaining 20% samples. Then, we introduce data drift by training the model on data collected between 2013 and 2020 and testing the trained model on samples collected between 2021 and 2023.

6.5.5 Case Study 5: DNN Code Generation

This task builds a *regression-based* cost model to drive the schedule search process in TVM [44] for DNN code generation on multi-core CPUs. The cost model estimates the potential gain of a schedule (e.g., instruction orders and data placement) to guide the search.

Underlying model. We apply Prom to Tlp [146], a cost model-based tensor program tuning method integrated into the TVM compiler v0.8 [44]. We use $2,308 \times 4$ samples collected from 4 Transformer-based BERT models of different sizes in the TenSet dataset [248].

Methodology. For the baseline, we trained and tested the cost model on the BERT-base dataset, where the model is trained on 80% (400K) randomly selected samples and then tested on the remaining 20% (100K)samples. To introduce data drift, we tested the trained model on the other three variants of the BERT model. We ran the TVM search engine for around 8 hours (4,000 iterations) for each DNN on a 12-core 3.70GHz AMD Ryzen9 5900X CPU server.

6.5.6 Performance Metrics

Performance to the oracle. For code optimisation tasks (case studies 1 to 3), we compute the ratio of the predicted performance to the best performance obtained by exhaustively trying all options. A ratio of 1.0 means the prediction leads to the best performance given by an "oracle" method.

Misprediction threholds. For code optimisation, we consider a prediction to be a misprediction if runtime performance is 20% or more below the Oracle performance (case studies 1-3) or if predicted performance deviates by 20% or more from profiling results (case study 5). For bug detection (case study 4), a misprediction happens when the model misclassifies a test input.

Coverage deviation. This "*smaller-is-better*" metric measures the difference between the confidence level and Prom's true coverage rate on the model. A zero deviation means the coverage rate matches the predefined significance level.

Table 6.2: Summary of our main evaluation results

Perf. to the Oracle				Prom performence		
Training	**Deploy.**	**Prom on deploy.**	**Acc.**	**Pre.**	**Recall**	**F1**
0.836	0.544	0.807	86.8%	86.0%	96.2%	90.8%

Metrics for data drift detection. We consider the following "*higher-is-better*" metrics for detecting drifting samples:

Accuracy. The ratio of the number of correctly predicted samples to the total number of testing samples.

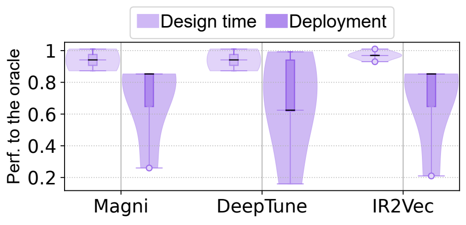
Precision. The ratio of mispredicted samples that were correctly rejected to the total number of mispredicted samples. This metric answers questions like "*Of all the rejected predictions, how many are actually mispredictions?*". High precision indicates a low *false-positive* rate, meaning PROM rarely incorrectly rejects predictions.

Recall (or Sensitivity). The ratio of mispredicted samples that were correctly rejected to the total number of mispredicted samples. This metric answers questions like "*Of all the mispredicted test samples, how many are rejected by PROM?*". High recall suggests a low *false-negative* rate, indicating PROM can identify most mispredictions.

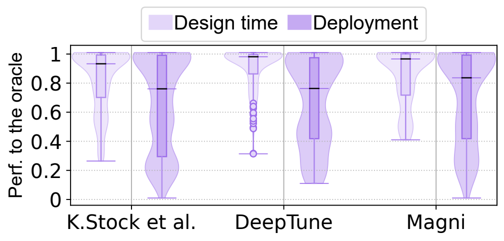
F1 score. The harmonic mean of Precision and Recall, calculated as $2 \times \frac{Recall \times Precision}{Recall + Precision}$. It is useful when the test data has an uneven distribution of drifting and normal samples. The highest possible F1 score is 1.0, indicating perfect precision and recall.

6.6 Experimental Results

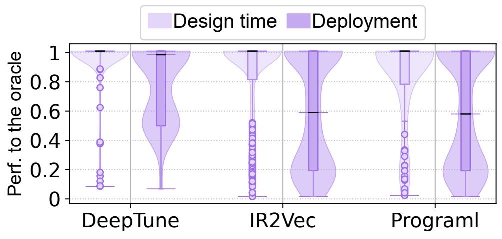
Highlights. Table 6.2 summarizes the main results of our evaluation. All the tested models were impacted by changes in the application workloads (Sec. 6.6.1), where the performance relative to the Oracle predictor drops significantly from training time to deployment time. PROM can detect 96.2% of the drifting samples on average (Sec. 6.6.2). When combined with incremental learning, PROM enhances the performance of deployed models, improving prediction performance by up to 6.6x (Sec. 6.6.3). PROM also outperforms existing CP-based methods and related work (Sec. 6.6.5).



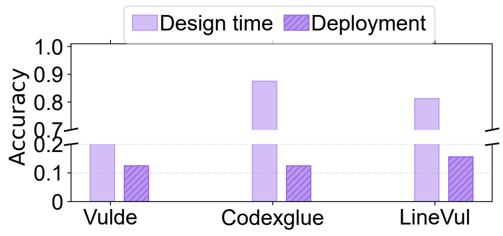
(a) C1: thread coarsening



(b) C2: loop vectorisation



(c) C3: heterogeneous device mapping



(d) C4: vulnerability detection

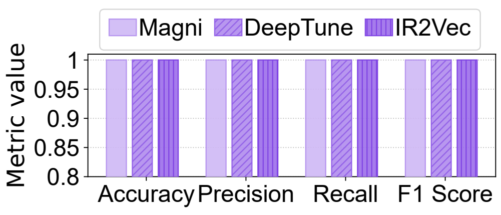
Figure 6.7: The resulting performance when using an ML model for decision making. The performance of all learning-based models can suffer during the deployment phase when the test samples significantly differ from the training data.

6.6.1 Impact of Drifting Data

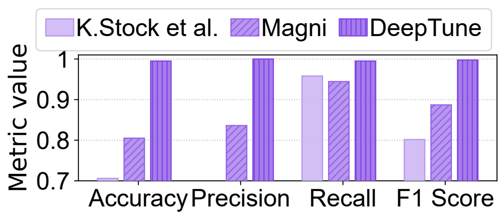
This experiment assesses the impact of data drift by applying a trained model to programs from an unseen benchmark suite or CVE dataset (Sec. 6.5). For code optimisation tasks (case studies 1-3 and 5), we report the performance ratio relative to the oracle (Sec. 6.5.6). In case study 4, we report the accuracy of bug prediction.

Figure 6.7 shows the classifiers' performance (case studies 1-4) during design and deployment, while Table 6.3 presents the regression model's performance (case study 5). The violin diagrams in Figure 6.7 show the distribution of test sample performance, with the violin's width representing the number of samples in each range. The line inside shows the median, and the top and bottom lines indicate the extremes. Outliers are marked as circles. Ideally, a model's violin would be wide at the top, reflecting good performance for most samples.

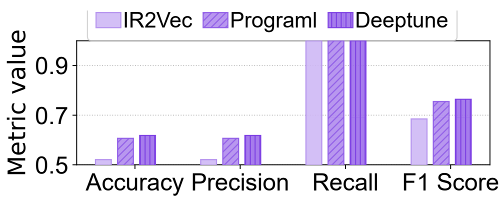
Design time performance. For case studies 1-4, we assess design-time performance by holding out 10% of the training samples as a validation set and applying the trained model to it. In case study 5, the training data covers all DNN models, but the model is tested on samples from unseen schedules. This process is repeated 10 times, and



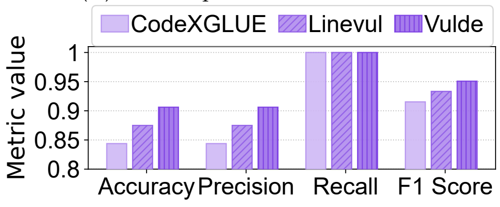
(a) C1: thread coarsening



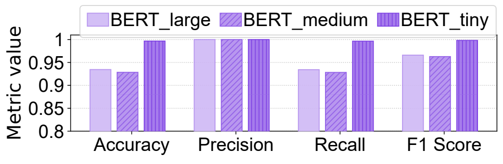
(b) C2: loop vectorisation



(c) C3: heterogeneous mapping



(d) C4: vulnerability detection

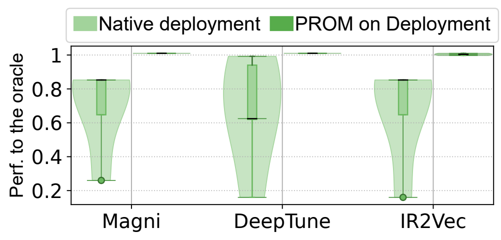


(e) C5: DNN code generation

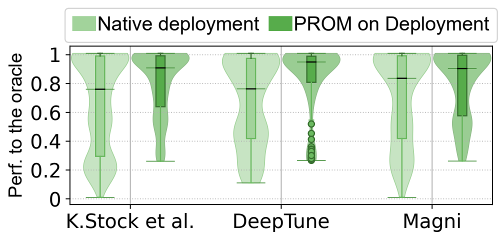
Figure 6.8: Prom's performance for detecting drifting samples across case studies and underlying models (higher is better).

the average performance is used as the design-time result. This *ideal setting* assumes that validation and training samples come from the same project or benchmark, with similar workload patterns, yielding comparable results to those reported in the original model publications.

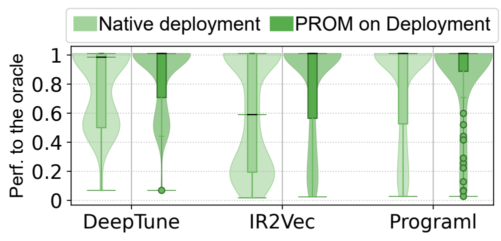
Deployment time performance. An ML model's robustness can suffer during deployment. From Figure 6.7, this can be observed from the bimodal distribution of the violin shape or a lower prediction accuracy at the deployment stage. From the violin diagrams, we observe a wider violin shape towards the bottom of the y-axis and a lower median value compared to the design-time result. From Table 6.3, this also can be seen from a lower deployment-time prediction accuracy than the design-time performance. The impact of drifting samples is clearly shown in vulnerability detection of Figure 6.7(d), where the prediction accuracy drops by an average of 62.5%. For DNN code generation (Table 6.3), the accuracy of performance estimation can also drop from 84.5% to as low as 22.4%. The results highlight the impact of data drift.



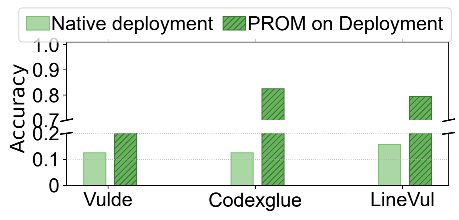
(a) C1: thread coarsening



(b) C2: loop vectorisation



(c) C3: heterog. mapping



(d) C4: vuln. detection

Figure 6.9: PROM enhances performance through incremental learning in different underlying models.

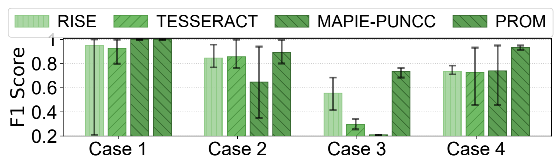


Figure 6.10: Geometric mean and variances across models of the F1 score in classification tasks.

6.6.2 Detecting Drifting Samples

Figure 6.8 reports PROM's performance in predicting drifting samples across case studies and underlying models. For all tasks, PROM achieves an average precision of 0.86 with an average accuracy of 0.87. This means it rarely filters out correct predictions. For the binary classification task of heterogeneous mapping (Figure 6.8(c)), PROM achieves an average F1 score of 0.74. In this case, PROM sometimes rejects correct predictions. This is because the probability distribution of binary classification is often less informative for CP than in multiclass cases [157]. For the regression task of case study 5, PROM can detect most of the drifting samples with a recall of 0.95 and an average precision of 1. Furthermore, the underlying model's quality also limits the performance of PROM. When the information given by the underlying model becomes noisy, PROM

Table 6.3: C5: DNN code generation (performance to the oracle ratio) - trained on BERT-base and tested on BERT variants.

Network	**BERT-base**	**BERT-tiny**	**BERT-medium**	**BERT-large**
Native deployment	0.845	0.224	0.668	0.642
Prom assisted deployment	/	0.794	0.871	0.843

can be less effective. Averaged across case studies, in detecting mispredictions, Prom achieves a recall of 0.96, a false-positive rate of 0.14 and a false-negative rate of 0.04, suggesting that Prom is highly accurate in detecting drifting samples.

6.6.3 Incremental Learning

In this experiment, we use Prom to identify drifting samples and update the underlying models by retraining with a small set of Prom-identified samples. Prom preserves the performance of the methods close to their original levels, as shown by improved accuracy (Figure 6.9(d)), the performance-to-oracle ratio (Table 6.3), and violin plots (Figures 6.9(a) to 6.9(c)), where test sample distributions shift towards higher performance with a better median than native deployment without Prom. Overall, Prom requires labelling at most 5% (sometimes just one) of drifting samples to update the model. Without it, one would need to label random test samples, leading to higher maintenance costs and unnecessary user confirmations for samples the model can predict correctly.

6.6.4 Individual Case Studies

We now examine case studies showing how Prom improves the underlying model with incremental learning.

Case study 1: thread coarsening. In this experiment, we tested the trained model on kernels from OpenCL benchmarks unseen at the model training stage. Figure 6.7(a) and 6.9(a) show that the performance of all ML models drops as the test dataset changes. By relabelling just one drifting sample using incremental learning, Prom improves the performance to the oracle ratio from an average of 77.6% to 99.0% (21.4% improvement) during deployment.

Case study 2: loop vectorisation. This experiment introduced changes by testing the underlying model on loops extracted from unseen benchmarks. Figure 6.7(b) and 6.9(b) show that drifting data led to a performance reduction for all methods, averaging 9%. Retraining the model using only 5% of the PROM-identified drifting samples helps restore the underlying model's initial performance, leaving only a 1.6% gap to the design-time performance.

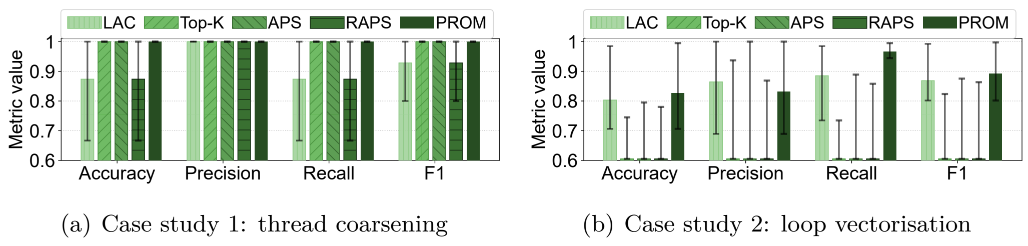
Case study 3: heterogeneous mapping. This experiment tests the underlying models on OpenCL kernels from *unseen* benchmarks. Figures 6.7(c) and 6.9(c) show that all ML models deliver low performance on *unseen* benchmarks. PROM also successfully detects 100% of the drifting samples (recall) on average with an accuracy rate of 58%. Further, by utilising incremental learning on 5% of the drifting samples, PROM improved the performance to oracle ratio of all systems from 63.1% to 78.9% (15.8% improvement) on average.

Case study 4: vulnerability detection. In this experiment, we tested a trained model on a vulnerability dataset from a time period not covered by the training data. Figures 6.7(d) and 6.9(d) show that all models initially had low prediction accuracy, ranging from 12.5% to 15.6% when facing new code patterns. PROM correctly identified all mispredictions with a recall of 1. By relabelling up to 5% of the PROM-identified drifting samples and updating the model, we improved the accuracy from an average of 13.5% to 72.5%, achieving 95.3% of the design-time performance.

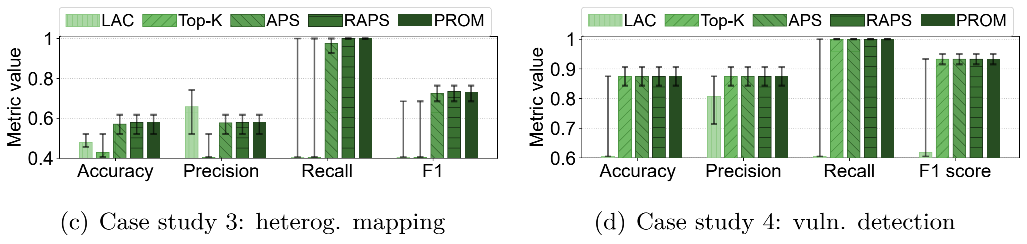
Case study 5: DNN code generation. In this experiment, we apply the cost model trained on the Bert-base dataset to three other variants of the BERT network. Once again, from Table 6.3, we see the performance of the trained model experiences a reduction from 84.5% to 53.2%. For BERT-tiny, the performance drops as much as 65.3%. PROM can detect 95.3% of drifting data and achieve a precision of 1. After profiling just 5% of the PROM-identified drifting data and using them to retrain the cost model online during the TVM code search process, the average performance of the cost model improves to 80.4%, resulting in a 2.0x enhancement.

6.6.5 Comparison to CP Libraries and Methods

We compare PROM with MAPIE [157] and PUNCC [158], two CP libraries used for outlier detection, as well as RISE [40] and TESSERACT [72], which use a single nonconformity function. RISE also trains an SVM for misprediction detection, but, like TESSERACT, supports only classical classifiers, so we evaluate them on cases 1 to 4. Figure 6.10 shows average results, with min-max bars indicating variation across



(a) Case study 1: thread coarsening
(b) Case study 2: loop vectorisation



(c) Case study 3: heterog. mapping
(d) Case study 4: vuln. detection

Figure 6.11: Performance of individual nonconformity functions. Min-max bar shows the performance across ML models.

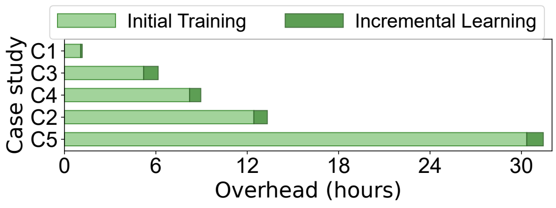
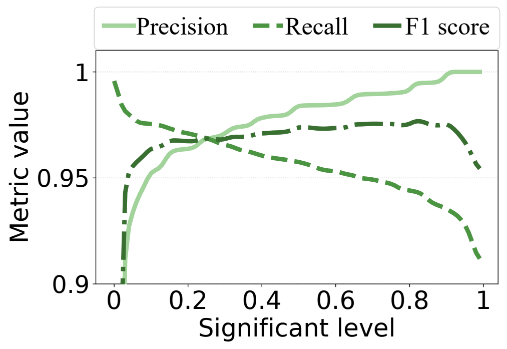


Figure 6.12: The average training and incremental learning overhead of individual case studies.

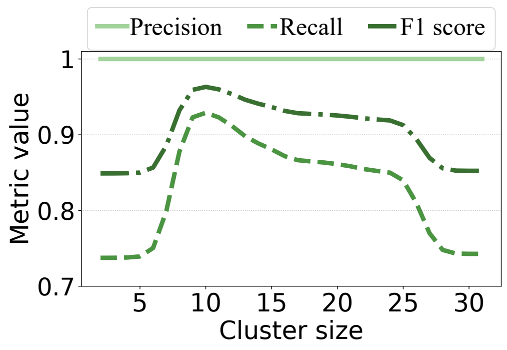
models. Prom outperforms TESSERACT by 17.6%, achieving a higher F1 score due to its improved nonconformity strategy. RISE struggles with uneven data or tasks with many labels, while Prom's model-free ensemble approach handles these cases better. Naive CP yields the lowest F1 score, showing its limitations in detecting drift in large-scale tasks.

6.6.6 Further Analysis

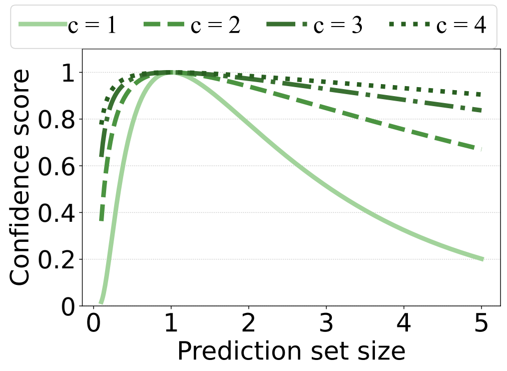
Nonconformity functions. Figure 6.11 shows the average performance of the four default Prom functions in detecting drifting samples across case studies, with min-max bars indicating variance across models. Prom's ensemble strategy outperforms individual functions in all metrics, demonstrating that no single function performs well across all case studies. By combining multiple functions, Prom enhances the generalisation of statistical assessments.



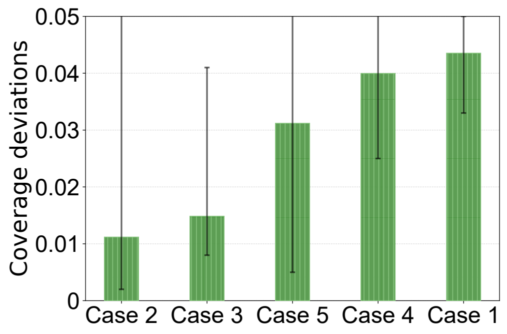
(a) Prom performance as the threshold increases in loop vectorisation



(b) Prom performance as cluster size increases in regression task



(c) Prom performance as Gaussian scale parameter changes



(d) Coverage deviations across 5 case studies

Figure 6.13: Sensitivity analysis of Prom hyperparameters.

Sensitivity analysis. Figures 6.13(a) to 6.13(c) show how the significance threshold and other parameters (e.g., cluster size and Gaussian scale) impact Prom's performance for data drift detection. A larger significance threshold reduces false positives, improving precision but potentially lowering recall. Prom automatically determines the optimal cluster size using the Gap statistic method [241]. Deviations from the optimal cluster size affect detection performance. For the Gaussian scale parameter, prediction set sizes smaller or larger than 1 lead to reduced confidence by suggesting too few or too many classes for a sample.

Coverage deviation. Figure 6.13(d) shows the coverage deviation across five cases. The min-max bar represents the variance across the underlying models. The geomean of deviation is 2.5%, which is a benign fluctuation and indicates a good fit for conformal prediction on the underlying models. The thread coarsening task shows a 4.4% deviation due to the small calibration dataset, which could be mitigated by adding more calibration samples.

Training overhead. Figure 6.12 reports the overhead of initial model training and incremental learning. Initial training takes several hours to one day, while incremental learning with fine-tuning takes less than one hour on a multi-core CPU or a desktop GPU - a negligible overhead compared to the initial training time.

Runtime overhead. During deployment, the main runtime overhead in Prom comes from computing nonconformity scores and detecting drifting samples. This overhead is minimal: on a low-end laptop, computing confidence and credibility scores and performing drift detection take less than 10 and 2 milliseconds, respectively.

6.7 Summary

In this chapter, we introduced Prom, an open-source Python library designed to enhance the robustness of learning-based models during deployment for code-related tasks. Prom measures the credibility and confidence of predictions to detect likely mispredictions. We evaluated Prom on 13 ML models across five code optimisation and analysis tasks. Prom can effectively detect samples that an ML model can mispredict by successfully identifying an average of 97% of mispredictions. It also improves deployed model performance by updating the trained model with a few identified test samples.

There is growing interest in applying ML to systems research [4, 218], yet prior work has primarily focused on model optimisations during the design phase. Prom addresses post-deployment optimisation. We hope it will be a valuable tool for post-deployment optimisation.

Chapter 7

Conclusions

This thesis presents three novel techniques to address key challenges in practical ML for program modelling in code-related tasks. These challenges include precise program representation, the high barrier between programmers and the application of DRL, and data drift during deployment, as outlined in Section 1.1.

Chapter 4 introduces a new program representation technique by combining static and dynamic information to address the limitations of relying solely on static code information (Sec. 1.1.1).

Chapter 5 addresses the expertise requirements in DRL (Sec. 1.1.2) by providing a software framework to automate the search for the appropriate DRL architecture and the optimal parameters for optimisation tasks.

Chapter 6 detects when a trained model is likely to mispredict during the model deployment time (Sec. 1.1.3) by applying CP and statistical analysis to detect low-confidence prediction samples.

This chapter is structured as follows: Section 7.1 summarises the main contributions of this thesis, Section 7.2 presents a critical analysis of this work, Section 7.3 describes future research directions, and finally, Section 7.6 provides the summary.

7.1 Contributions

In Section 1.1, several significant challenges are identified that hinder the application of ML in code optimisation and analysis tasks. This section summarises the main contributions of the thesis, showing how to make ML practical for code-based tasks.

7.1.1 Combining Static and Dynamic Information for Precise Program Modelling

A key challenge in ML-based approaches for program representation is that they often affect the performance of ML models due to reliance solely on information from the static source code. Previous approaches [18] have attempted to incorporate deeper information, such as graph relationships in programs. However, they remain limited by their reliance on static data.

Chapter 4 introduces a new DL system for code-related tasks, using source code-level vulnerability detection as a case study. This is the first DL framework to effectively combine structured code information with symbolic execution for vulnerability prediction. The system utilises both structured static code features and dynamic symbolic execution traces to learn program representations, thereby enabling accurate bug prediction. The model is trained using a combination of unsupervised and supervised learning while minimising the overhead of symbolic execution through the use of a path selection network.

We applied this system to detect bugs and vulnerabilities in C programs from 20 open-source projects. In 200 hours of automated concurrent test runs, the system successfully detected vulnerabilities in all tested projects, discovering 54 unique vulnerabilities and resulting in 37 new, unique CVE IDs. Compared to 16 previous methods, this approach detects more vulnerabilities with higher accuracy and a lower false positive rate. This novel technique significantly improves ML performance by enabling models to learn dynamic information, such as input and execution traces, thereby addressing the challenge of precise ML program modelling as outlined in Section 1.1.1.

7.1.2 Automating the Design of Deep Reinforcement Learning Architectures

To integrate ML into downstream tasks, programmers often need extensive knowledge of ML, particularly in designing effective architectures. This is especially challenging for DRL, which requires a higher level of expertise.

Chapter 5 introduces a framework that lowers the barrier between programmers and DRL expertise, enabling them to integrate DRL into their fields with just a few dozen lines of code, regardless of their prior knowledge. This is the first generic framework that automatically selects and tunes suitable RL architectures for code optimisation tasks.

It automates the process of designing and tuning RL algorithm structures, simplifying the integration of DRL into various optimisation problems. We evaluate our framework by applying it to four different optimisation tasks.

Experimental results show that the RL architectures generated by our tool outperform alternative search techniques, including manually tuned RL strategies. The framework supports mainstream and emerging RL environments, such as RLlib and CompilerGym, and provides customizable interfaces for developers to introduce new algorithms and methods into the RL policy architecture. As the community contributes more models and methods, we will be able to explore a more extensive policy search space, reducing the need for domain-specific methods for each project. In the long term, we anticipate that our framework will work out-of-the-box for most performance developers once they have defined their optimisation tasks. By enabling automated architecture design and parameter tuning for various downstream optimisation tasks, this work addresses the challenge of the ML barrier identified in Section 1.1.2.

7.1.3 Enhancing Predictive Model Robustness at Deployment Time Using a Statistical Analysis Approach

A significant challenge during model deployment is data drift. Detecting drifting samples is critical for maintaining model robustness. Chapter 6 presents a framework designed to preserve robustness during deployment by leveraging conformal prediction to identify low-confidence predictions and alert users to potentially unreliable results. Importantly, this process integrates seamlessly with the original prediction workflow-users do not need to modify their model architecture. Instead, they simply provide intermediate outputs from their existing models through our API.

We evaluated the framework on 13 machine learning models across five code optimisation and analysis tasks, successfully detecting an average of 97% of mispredictions. Additionally, PROM enhances the performance of deployed models by updating them using a small subset of the identified test samples.

This method directly addresses the design-deployment gap discussed in Section 1.1.3 by enabling post-deployment optimisation. We believe it provides a practical and effective solution for improving model reliability in real-world applications.

7.2 Critique

This section provides a critical analysis of the techniques presented in this thesis.

7.2.1 Runtime Overhead

Runtime overhead is always one of the most important concerns for both programmers and researchers. For ML, the primary source of runtime overhead arises from the training phase, which can range from several hours to several days. However, since training is performed offline and typically occurs only once, this overhead does not impact real-time performance. During inference, predictions are usually generated within seconds, ensuring that the system remains responsive.

Another factor contributing to overhead is feature collection. For example, in the case of Concoction, the collection of symbolic execution traces increases computational cost. To mitigate this, we employ a path selection component that reduces the burden of symbolic execution. Additionally, techniques such as parallelising symbolic test case generation can further accelerate the process, although these are orthogonal to our core contribution.

7.2.2 Cross-Language Support and Integration

The primary focus of this thesis is on code-related tasks. We do not emphasise application to a specific programming language; rather, we aim to develop a general method that can be applied across multiple languages. For the backend ML interfaces used in this thesis, Python is chosen due to its extensive ecosystem and strong support for ML libraries. However, the proposed methods are designed to facilitate seamless integration with non-Python environments. For example, in C/C++ settings, we use the pybind11 API to bridge probabilistic model predictions with the core system, enabling binary output for prediction acceptance.

Although our evaluations may focus on the C and Python languages, the underlying methods are extensible to other programming languages. This requires adapting source code rewriting tools and integrating language-specific symbolic execution engines, such as JDart for Java or PyExZ3 for Python. We provide a method supported by the Language Server Protocol (LSP) to enable cross-language compatibility. However, this extension process is still ongoing and requires further refinement.

7.2.3 Overfitting

Overfitting remains a significant challenge for ML models, particularly during deployment. Expanding the training dataset to cover a wide range of scenarios improves generalisation and reduces the risk of overfitting. For example, in PROM, we retrain the models using both the original dataset and mispredictions identified during deployment, thereby enhancing the model's adaptability to real-world data.

7.2.4 Reliability

ML solutions can become fragile over long-term deployment. To address this issue, we apply a model-free approach in our drifting data detection framework to maintain robustness during operation. Also, the calibration dataset is updated incrementally, allowing the detection framework to evolve as the model undergoes retraining. This continuous adaptation enables our approach to handle shifts in data distribution post-deployment, ensuring sustained reliability and accuracy across diverse environments.

7.2.5 Scalability

Scalability is important for industry, particularly when dealing with large and complex codebases. ML offers the advantage of transferring knowledge from one program to another. For instance, it can learn that applying specific optimisation passes consistently reduces code size across different programs. This transferability is a notable strength of ML compared to evolutionary techniques.

7.2.6 Practical Applications

For CONCOCTION, the system has been applied to 20 real-world projects. However, it currently lacks support for complex code patterns such as recursive functions and dynamic dispatch via function pointers. Addressing these limitations will require future research into advanced code analysis techniques, including interprocedural data flow tracking and pointer alias analysis, to enable the framework to scale to larger and more intricate code regions.

For SUPERSONIC, the system is capable of designing DRL architectures, incorporating 23 RL algorithms, and can easily be extended to design DL architectures and incorporate user-defined algorithms. However, the performance of SUPERSONIC in DL and traditional machine learning classifiers requires further evaluation to fully assess its applicability and efficacy.

For Prom, handling rejected predictions requires balancing the trade-offs between automatic adjustments and manual interventions. The system allows users to adjust the significance level to manage these trade-offs effectively, detecting data drift and model degradation during post-deployment. Although human verification may still be necessary, Prom minimises this need by only requesting intervention when significant drift is detected. Prom are primarily targeted at probability-based classifiers and are broadly applicable across various ML algorithms. These techniques also optimise data labelling efforts by evaluating the potential benefits of incorporating new data into training, making them particularly valuable in reducing costs for code optimisation tasks.

7.3 Future Work

This section outlines two promising avenues for future research enabled by this thesis.

7.3.1 Explainable Machine Learning for Program Modelling

DL has gained prominence across various domains, such as text, images, videos, and graphs. However, despite its successes, it often functions as a black box, particularly in areas like compiler optimisation, where model robustness is crucial. The black-box nature of ML remains a significant barrier to its broader adoption in compilers. Mistrust often arises due to the lack of transparency in model-based results. Developers frequently question how deep learning models make predictions, why certain features are prioritised, and what adjustments are needed to enhance model performance. Unfortunately, efforts to answer these questions have only seen modest success. Without detailed explanations, developers are left uncertain about where and how to further investigate or improve the model.

The need for interpretability is paramount to uncover model vulnerabilities, enhance robustness, and establish trust by providing clear explanations for model decisions. While post-hoc interpretability techniques, applied after a model is trained, offer high predictive performance without compromising interpretation, they still fall short in terms of faithfulness to the original model. The potential inaccuracies inherent in these methods make it difficult to fully trust their outputs.

Current interpretability methods primarily focus on feature saliency analysis [249], which identifies the most relevant input data attributes influencing a model's predictions or latent representations. However, a comprehensive framework for explainability in program modelling remains unexplored.

To address this challenge, we propose the development of an open-source framework specifically designed to enhance the interpretability of learning-based models in program modelling. This framework would improve model accuracy and reliability by providing detailed insights into the inner workings of these models.

We aim to design a model that leverages feature visualisation to allow programmers to interact directly with machine learning models. In this context, features represent the key data elements, such as code tokens, statements, binary code blocks, or abstract syntax trees, that influence predictions. By understanding the importance of these features, developers can better comprehend how the model arrives at its predictions. This process will enable programmers to identify and address potential vulnerabilities, thus enhancing the model's accuracy and reliability during the program modelling phase.

In the long term, we envision our framework fostering greater trust in model decisions by generating localised explanations that clarify model outcomes. This will help users assess the trustworthiness of the AI system by understanding which features are most important in generating predictions, ultimately leading to more reliable and transparent machine learning models in program modelling.

7.3.2 Extending Scalability to Novel Machine Learning Approaches

As new ML techniques, such as federated learning and others, continue to evolve, they still face the challenges discussed earlier in this thesis (Sec. 1.1). Evaluating the scalability of our approach within these emerging deep learning architectures and addressing the unique challenges they present is a promising direction for future research.

Furthermore, while this thesis has focused primarily on code analysis and optimisation tasks, there is significant potential to extend our methods to other domains. Exploring the applicability of our framework across various fields will further demonstrate its versatility and scalability, broadening the impact of our research beyond its current scope.

7.4 The Impact of Large Language Models

In recent years, the rise of LLMs has brought transformative changes to many areas of computer science, including the downstream tasks we focus on: program analysis and optimization. LLMs have shown remarkable capabilities in software engineering tasks such as natural language processing, and their application is now expanding into more specialized fields [250]. This section explores the potential impact of LLMs on program analysis and optimization and discusses how the research contributions of this thesis complement these emerging trends.

7.4.1 Applications of LLMs in Software Engineering

LLMs are demonstrating immense potential across the fields of program analysis and optimization, promising to reshape traditional methodologies.

In the domain of optimization, which has historically required extensive domain expertise and computational resources, LLMs are emerging as a revolutionary force. Researchers have begun exploring their use to predict compiler options that maximize code performance directly. For instance, specialized LLMs for compilers, such as LLM-Compiler [251], can learn the complex relationships between code structures and optimization strategies by training on vast amounts of compiler IR and assembly code.

In parallel to their role in optimization, LLMs have also shown great potential in program analysis [252], particularly in software vulnerability detection. While traditional static analysis tools rely on predefined rules and patterns, LLMs can learn broader and more abstract patterns from massive code datasets. Research indicates that, in some cases, the performance of LLMs in this area can surpass that of traditional static analyzers, enabling them to identify not only known vulnerability patterns but also potentially discover new, unknown *zero-day* vulnerabilities.

7.4.2 The Value of This Thesis in the Age of LLMs

Although LLMs represent a novel technological wave, the core challenges addressed in this thesis, precision, automation, and robustness, have not become obsolete. On the contrary, they are more critical than ever in the era of LLMs.

Regarding precision, LLMs are inherently probabilistic, and their outputs may contain factual errors or hallucinations. The methods proposed in this thesis, which combine static and dynamic analysis, can provide a factual grounding for LLM-based analysis, thereby improving its precision and reliability.

Concerning automation, LLMs are a major driver of automation in software engineering. The automated ML framework developed in Chapter 5 can be viewed as a “meta-tool” for designing and optimizing specialized, efficient machine learning models (potentially including smaller, task-specific LLMs) tailored to different program optimization tasks.

Finally, the challenge of robustness is a key practical concern, as discussed in Chapter 6. The problem of data drift after a model is deployed is particularly significant. Ensuring that LLM-based tools remain reliable in the face of evolving software projects is a central issue, and the deployment-time robustness framework proposed in this thesis offers an important solution.

In summary, LLMs are opening new frontiers for research in program analysis and optimization. They do not replace the fundamental problems investigated in this thesis but rather highlight their importance in a new light and provide powerful new tools for addressing them. The contributions of this thesis, in enhancing the precision, automation, and robustness of machine learning models, lay a solid foundation for the future integration of LLMs into program analysis and optimization tools in a safer and more effective manner.

7.5 Vision

This thesis addresses three interrelated challenges in the application of ML to software engineering and compiler optimisation. While each contribution provides value independently, their collective integration points towards a broader vision: building reliable, precise, and developer-friendly ML systems that can be deployed in real-world software environments.

First, the development of an ML-based bug detection tool in Chapter 4 shows how hybrid feature-driven methods can capture subtle program behaviours that escape traditional rule-based analysis. The proposed method illustrates that combining static and dynamic information enables the detection of more bugs and vulnerabilities in real-world projects.

Second, the work on assisting developers in constructing DRL systems for compiler optimisation in Chapter 5 highlights the practical importance of bridging the gap between ML research and compiler engineering. The architecture proposed in this chapter provides a framework in which DRL agents can explore optimisation strategies while lowering the barrier for developers to design and train such agents effectively. This contribution shows that compiler optimisation can benefit from ML approaches, provided that developers are equipped with the appropriate methodological and architectural meta-optimiser tools.

Third, identifying when an ML model is likely to make incorrect predictions, as discussed in Chapter 6, addresses the critical challenge of reliability. In data-scarce or performance-sensitive domains, awareness of prediction uncertainty is essential. The proposed architecture in this thesis advances the state of the art by combining predictive modelling with statistical analysis to estimate prediction confidence and detect potential mispredictions, thereby enhancing robustness during deployment.

In summary, these three lines of research converge toward a unified vision: the design of machine learning systems for software engineering that are not only accurate, but also robust and developer-oriented. By connecting automated bug detection, compiler optimisation, and other program-modelling tasks, this thesis lays the groundwork for an ecosystem of ML tools that can adapt to diverse developer needs while providing performance and reliability guarantees. The architectures introduced in this thesis can be viewed as complementary components in this larger vision, each addressing a fundamental dimension of the challenge: precision, automation, and reliability.

In the long term, we envision extending this integrated framework to support a comprehensive software development environment, where ML models continuously interact with compilers, debuggers, and developers. Such an environment would enable not only the detection of bugs and performance bottlenecks, but also the dynamic adaptation of optimisation strategies and the reliable signalling of uncertainty, ultimately allowing developers to focus on higher-level design tasks while relying on machine learning systems for dependable automation.

7.6 Summary

This thesis explores how to make ML program modelling practical when integrating it into code-related tasks. While ML has proven to be a promising and successful technique in many fields, its practical application to code-related tasks still faces significant challenges, both in ML model design and at deployment time.

This thesis first presents a novel program modelling approach that combines structured static code information with dynamic symbolic execution, demonstrating strong performance in a vulnerability prediction case study. Then, it introduces a novel automated system that streamlines the selection and tuning of DRL architectures, specifically focusing on RL for code optimisation tasks. This system lowers the barrier between programmers and DRL, facilitating broader adoption of DRL in code optimisation. Lastly, the research extends into the robustness of ML models during deployment by proposing a novel framework that utilises CP to ensure and assess the reliability of ML models across five code-related deployment tasks. Results show its effectiveness in enhancing model robustness during deployment time.

These contributions address three key barriers to integrating ML into specific domains, offering more precise, automated, and robust ML for program modelling. The outcomes demonstrated in this thesis provide a strong foundation for future research and practical applications of ML in code-related tasks.

References

[1] Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639212. URL https://doi.org/10.1145/3597503.3639212.

[2] Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 129–143, 2022.

[3] Huanting Wang, Patrick Lenihan, and Zheng Wang. Enhancing deployment-time predictive model robustness for code analysis and optimization. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, page 31â€"46, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712753. doi: 10.1145/3696443.3708959. URL https://doi.org/10.1145/3696443.3708959.

[4] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[5] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2020.

[6] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *CGO*. IEEE, 2017.

[7] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99. IEEE, 2017.

[8] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning*, pages 2244–2253. PMLR, 2021.

[9] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2021. doi: 10.1109/TIFS.2020.3044773.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[11] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh 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, 2017.

[12] Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, pages 129–143, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391832. doi: 10.1145/3497776.3517769. URL `https://doi.org/10.1145/3497776.3517769`.

[13] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. doi: 10.1109/MS.2008.130.

[14] Iqbal H Sarker. Machine learning: Algorithms, real-world applications and research directions. *SN computer science*, 2(3):160, 2021.

[15] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. Compilergym: Robust, performant compiler optimization environments for ai research, 2021.

[16] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[17] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.

[18] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 10197–10207. 2019.

[19] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 201–213, 2016.

[20] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290353. URL https://doi.org/10.1145/3290353.

[21] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *Proceedings of the NDSS*, 2018.

[22] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μvuldeepecker:

A deep learning-based system for multiclass vulnerability detection. *Proceedings of the IEEE Transactions on Dependable and Secure Computing*, 2019.

[23] Frank Busse, Pritam Gharat, Cristian Cadar, and Alastair F Donaldson. Combining static analysis error traces with dynamic symbolic execution (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 568–579, 2022.

[24] Ke Wang and Zhendong Su. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 121–134, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385999. URL https://doi.org/10.1145/3385412.3385999.

[25] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[26] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3588–3600. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics.pdf.

[27] Guixin Ye, Zhanyong Tang, Huanting Wang, Dingyi Fang, Jianbin Fang, Songfang Huang, and Zheng Wang. Deep program structure modeling through multi-relational graph-based learning. In *Proceedings of the ACM International conference on parallel architectures and compilation techniques*, pages 111–123, 2020.

[28] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quinão Pereira. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390. IEEE, 2021.

[29] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013. URL http://arxiv.org/abs/1301.3781.

[30] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[33] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

[34] Shuangjiao Zhai, Zhanyong Tang, Petteri Nurmi, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. Rise: Robust wireless sensing using probabilistic and statistical assessments. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 309–322, 2021.

[35] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. Transcending transcend: Revisiting malware classification in the presence of concept drift, 2024.

[36] Murali Krishna Emani and Michael F. P. O'Boyle. Celebrating diversity: a mixture of experts approach for runtime mapping in dynamic environments.

In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 499–508. ACM, 2015. doi: 10.1145/2737924.2737999. URL https://doi.org/10.1145/2737924.2737999.

[37] Roberto Jordaney, Kumar Sharad, Kumar Dash Santanu, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *USENIX Security*, 2017.

[38] Wenlong Tang and Edward S Sazonov. Highly accurate recognition of human postures and activities through classification with rejection. *IEEE journal of biomedical and health informatics*, 2014.

[39] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. Transcending transcend: Revisiting malware classification with conformal evaluation. *arXiv*, 2020.

[40] Shuangjiao Zhai, Zhanyong Tang, Petteri Nurmi, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. Rise: Robust wireless sensing using probabilistic and statistical assessments. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, pages 309–322, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383424. doi: 10.1145/3447993.3483253. URL https://doi.org/10.1145/3447993.3483253.

[41] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Edmonton, Canada, Aug 2014. URL http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf.

[42] Masayo Haneda, Peter MW Knijnenburg, and Harry AG Wijshoff. On the impact of data input sets on statistical compiler tuning. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.

[43] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo

Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[44] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {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, 2018.

[45] JW Davidson, Gary S Tyson, DB Whalley, and PA Kulkarni. Evaluating heuristic optimization phase order search algorithms. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 157–169. IEEE, 2007.

[46] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation*, pages 304–314, 2002.

[47] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *Proceedings of the NDSS*, 2018.

[48] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. Software vulnerability discovery via learning multi-domain knowledge bases. *Proceedings of the IEEE Transactions on Dependable and Secure Computing*, 2019.

[49] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

[50] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, 2021.

[51] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the 44th International Conference on Software Engineering Companion (ICSE '22 Companion)*, 2022.

[52] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[53] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[54] Frédérick Garcia and Emmanuel Rachelson. Markov decision processes. *Markov Decision Processes in Artificial Intelligence*, pages 1–38, 2013.

[55] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[56] Alvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michele Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 60(1):25–64, 2010.

[57] Gongde Guo, Hui Wang, David Bell, Yaxin Bi, and Kieran Greer. Knn model-based approach in classification. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 986–996, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39964-3.

[58] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979. ISSN 00359254, 14679876. URL http://www.jstor.org/stable/2346830.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[60] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 33(12):6999–7019, 2021.

[61] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[62] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

[63] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018. URL http://arxiv.org/abs/1803.08375.

[64] Anqi Mao, Mehryar Mohri, and Yutao Zhong. Cross-entropy loss functions: Theoretical analysis and applications. In *International conference on Machine learning*, pages 23803–23828. PMLR, 2023.

[65] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[66] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

[67] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

[68] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[69] Vineeth Balasubramanian, Shen-Shyang Ho, and Vladimir Vovk. *Conformal prediction for reliable machine learning: theory, adaptations and applications*. Newnes, 2014.

[70] Janette Vazquez and Julio C Facelli. Conformal prediction in clinical medical sciences. *Journal of Healthcare Informatics Research*, 6(3):241–252, 2022.

[71] Wojciech Wisniewski, David Lindsay, and Sian Lindsay. Application of conformal prediction interval estimations to market makers'net positions. In *Conformal and probabilistic prediction and applications*, pages 285–301. PMLR, 2020.

[72] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. Tesseract: Eliminating experimental bias in malware classification across space and time. In *USENIX Security*, 2019.

[73] Anastasios N Angelopoulos and Stephen Bates. A gentle introduction to conformal prediction and distribution-free uncertainty quantification. *arXiv preprint arXiv:2107.07511*, 2021.

[74] Ronald A Thisted. What is a p-value. *Departments of Statistics and Health Studies*, 1998.

[75] Michael W Browne. Cross-validation methods. *Journal of mathematical psychology*, 2000.

[76] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[77] Ke Shi, Yang Lu, Jingfei Chang, and Zhen Wei. Pathpair2vec: An ast path pair-based code representation method for defect prediction. *Journal of Computer Languages*, 59:100979, 2020. ISSN 2590-1184. doi: https://doi.org/10.1016/j.cola.2020.100979. URL https://www.sciencedirect.com/science/article/pii/S2590118420300393.

[78] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pages 519–531, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534371. URL https://doi.org/10.1145/3533767.3534371.

[79] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated conformance

testing for javascript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 435–450, 2021.

[80] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, USA, 2020. USENIX Association.

[81] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017.

[82] Trong-Nghia Nguyen, Ngoc-Sang Vo, Dinh-Thuan Le, and Khuong Nguyen-An. A study onÂ deep graph neural networks forÂ security vulnerabilities detection inÂ web applications. In Tran Khanh Dang, Josef Küng, and Tai M. Chung, editors, *Future Data and Security Engineering. Big Data, Security and Privacy, Smart City and Industry 4.0 Applications*, pages 123–137, Singapore, 2024. Springer Nature Singapore. ISBN 978-981-96-0434-0.

[83] Ruitong Liu, Yanbin Wang, Haitao Xu, Bin Liu, Jianguo Sun, Zhenhao Guo, and Wenrui Ma. Source code vulnerability detection: Combining code language models and code property graphs, 2024. URL https://arxiv.org/abs/2404.14719.

[84] Wei Li, Xiang Li, Wanzheng Feng, Guanglu Jin, Zhihan Liu, and Jing Jia. Vulnerability detection based on unified code property graph. In Long Yuan, Shiyu Yang, Ruixuan Li, Evangelos Kanoulas, and Xiang Zhao, editors, *Web Information Systems and Applications*, pages 359–370, Singapore, 2023. Springer Nature Singapore. ISBN 978-981-99-6222-8.

[85] Jiacheng Liu, Sewon Min, Luke Zettlemoyer, Yejin Choi, and Hannaneh Hajishirzi. Infini-gram: Scaling unbounded n-gram language models to a trillion tokens, 2024. URL https://arxiv.org/abs/2401.17377.

[86] Jung Hyun An, Zhan Wang, and Inwhee Joe. A cnn-based automatic vulnerability detection. *EURASIP J. Wirel. Commun. Netw.*, 2023(1), May 2023. ISSN 1687-1472. doi: 10.1186/s13638-023-02255-2. URL https://doi.org/10.1186/s13638-023-02255-2.

[87] Yongjun Lee, Hyun Kwon, Sang-Hoon Choi, Seung-Ho Lim, Sung Hoon Baek, and Ki-Woong Park. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn. *Applied Sciences*, 9(19), 2019. ISSN 2076-3417. doi: 10.3390/app9194086. URL https://www.mdpi.com/2076-3417/9/19/4086.

[88] Amin Parvaneh, Ehsan Abbasnejad, Damien Teney, Gholamreza Reza Haffari, Anton Van Den Hengel, and Javen Qinfeng Shi. Active learning by feature mixing. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12237–12246, 2022.

[89] Miles Q Li, Benjamin CM Fung, and Ashita Diwan. A novel deep multi-head attentive vulnerable line detector. *Procedia Computer Science*, 222:35–44, 2023.

[90] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[91] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. Learning to explore paths for symbolic execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2526–2540, 2021.

[92] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[93] Saparya Krishnamoorthy, Michael S Hsiao, and Loganathan Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *2010 19th IEEE Asian Test Symposium*, pages 59–64. IEEE, 2010.

[94] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[95] Zhangwei Xie, Zhanqi Cui, Jiaming Zhang, Xiulei Liu, and Liwei Zheng. Csefuzz: fuzz testing based on symbolic execution. *IEEE Access*, 8:187564–187574, 2020.

[96] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749, 2020.

[97] Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bračevac, and Tiark Rompf. Compiling parallel symbolic execution with continuations. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1316–1328. IEEE, 2023.

[98] Ashish Jaiswal, Ashwin Ramesh Babu, Mohammad Zaki Zadeh, Debapriya Banerjee, and Fillia Makedon. A survey on contrastive self-supervised learning. *Technologies*, 9(1):2, 2020.

[99] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. Unsupervised learning of visual features by contrasting cluster assignments, 2020. URL https://arxiv.org/abs/2006.09882.

[100] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[101] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[102] John Giorgi, Osvald Nitski, Bo Wang, and Gary Bader. Declutr: Deep contrastive learning for unsupervised textual representations. *arXiv preprint arXiv:2006.03659*, 2020.

[103] Sanjeev Arora, Hrishikesh Khandeparkar, Mikhail Khodak, Orestis Plevrakis, and Nikunj Saunshi. A theoretical analysis of contrastive unsupervised representation learning. *arXiv preprint arXiv:1902.09229*, 2019.

[104] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *Advances in Neural Information Processing Systems*, 33:18661–18673, 2020.

[105] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 516–527, 2020.

[106] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=jLoC4ez43PZ.

[107] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Mirchecker: detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2183–2196, 2021.

[108] David A Tomassi and Cindy Rubio-González. On the real-world effectiveness of static bug detectors at finding null pointer exceptions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 292–303. IEEE, 2021.

[109] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Concolic program repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 390–405, 2021.

[110] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. Deep just-in-time defect prediction: how far are we? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 427–438, 2021.

[111] Ning Li, Martin Shepperd, and Yuchen Guo. A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology*, 122:106287, 2020.

[112] Dan Zhang, Qing-Guo Wang, Gang Feng, Yang Shi, and Athanasios V Vasilakos. A survey on attack detection, estimation and control of industrial cyber–physical systems. *ISA transactions*, 116:1–16, 2021.

[113] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 128–139. IEEE, 2019.

[114] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 957–969. IEEE, 2021.

[115] R Clinton Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 38–38. IEEE, 1998.

[116] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[117] Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. *ACM SIGPLAN Notices*, 39(7):231–239, 2004.

[118] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, 2009.

[119] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim.*, 13(2), jun 2016. ISSN 1544-3566. doi: 10.1145/2928270. URL https://doi.org/10.1145/2928270.

[120] Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. Auto-tuning parameter choices in hpc applications using bayesian optimization. In *2020 IEEE Interna-*

tional Parallel and Distributed Processing Symposium (IPDPS), pages 831–840. IEEE, 2020.

[121] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209. IEEE, 2021.

[122] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1280–1295, 2021.

[123] Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, EunJung Park, and Johannes Doerfert. *Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning*. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450384414. URL https://doi.org/10.1145/3458744.3473355.

[124] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 11–pp. IEEE, 2006.

[125] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197. IEEE, 2007.

[126] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 147–162, 2012.

[127] Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P Sadayappan. Predictive modeling in a polyhedral optimization space. *International journal of parallel programming*, 41(5):704–750, 2013.

[128] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Trans. Archit. Code Optim.*, 14(3), 2017. ISSN 1544-3566.

[129] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

[130] Hui Guan, Xipeng Shen, and Seung-Hwan Lim. Wootz: A compiler-based framework for fast cnn pruning via composability. PLDI 2019, pages 717–730, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314652. URL https://doi.org/10.1145/3314221.3314652.

[131] Rik Mulder, Valentin Radu, and Christophe Dubach. Fast optimisation of convolutional neural network inference using system performance models. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, EuroMLSys '21, pages 104–110, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382984. doi: 10.1145/3437984.3458840. URL https://doi.org/10.1145/3437984.3458840.

[132] Concertio. Concertio: Autonomous optimization platform.

[133] Ameer Haj-Ali, Qijing (Jenny) Huang, William S. Moses, John Xiang, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling HLS phase orderings in random forests with deep reinforcement learning. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys*, 2020.

[134] Alexander Brauckmann, Andrés Goens, and Jerónimo Castrillón. A reinforcement learning environment for polyhedral optimizations. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021.

[135] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: end-to-end vectorization with deep re-

inforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 242–255, 2020.

[136] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 270–288, 2019.

[137] Shauharda Khadka, Estelle Aflalo, Mattias Marder, Avrech Ben-David, Santiago Miret, Shie Mannor, Tamir Hazan, Hanlin Tang, and Somdeb Majumdar. Optimizing memory placement using evolutionary graph reinforcement learning. 2021.

[138] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[139] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[140] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[141] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[142] Jakub Pachocki, Greg Brockman, Jonathan Raiman, Susan Zhang, Henrique Pondé, Jie Tang, Filip Wolski, Christy Dennison, Rafal Jozefowicz, Przemyslaw Debiak, et al. Openai five, 2018. *URL https://blog. openai. com/openai-five*, 2018.

[143] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael

Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[144] OpenAI. Gym: a toolkit for developing and comparing reinforcement learning algorithms. https://gym.openai.com/, 2020.

[145] Guixin Ye, Zhanyong Tang, Huanting Wang, Dingyi Fang, Jianbin Fang, Songfang Huang, and Zheng Wang. Deep program structure modeling through multi-relational graph-based learning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, pages 111–123, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380751. doi: 10.1145/3410463.3414670. URL https://doi.org/10.1145/3410463.3414670.

[146] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 833–845, 2023.

[147] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

[148] Mauro Ribeiro, Katarina Grolinger, and Miriam AM Capretz. Mlaas: Machine learning as a service. In *2015 IEEE 14th international conference on machine learning and applications (ICMLA)*, pages 896–902. IEEE, 2015.

[149] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *Proceedings of Machine Learning and Systems*, 4:77–94, 2022.

[150] Siddharth Singhal, Utkarsh Chawla, and Rajeev Shorey. Machine learning & concept drift based approach for malicious website detection. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 582–585. IEEE, 2020.

[151] Samuel Ackerman, Orna Raz, Marcel Zalmanovici, and Aviad Zlotnick. Automatically detecting data drift in machine learning classifiers. *arXiv preprint arXiv:2111.05672*, 2021.

[152] Ferhat Ozgur Catak, Javed Ahmed, Kevser Sahinbas, and Zahid Hussain Khand. Data augmentation based malware detection using convolutional neural networks. *Peerj computer science*, 7:e346, 2021.

[153] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, et al. A survey of uncertainty in deep neural networks. *Artificial Intelligence Review*, 56(Suppl 1):1513–1589, 2023.

[154] Daily Milanés-Hermosilla, Rafael Trujillo Codorniú, René López-Baracaldo, Roberto Sagaró-Zamora, Denis Delisle-Rodriguez, John Jairo Villarejo-Mayor, and José Ricardo Núñez-Álvarez. Monte carlo dropout for uncertainty estimation and motor imagery classification. *Sensors*, 21(21):7241, 2021.

[155] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles, 2017. URL https://arxiv.org/abs/1612.01474.

[156] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. Misbehaviour prediction for autonomous driving systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 359–371, 2020.

[157] Vianney Taquet, Vincent Blot, Thomas Morzadec, Louis Lacombe, and Nicolas Brunel. Mapie: an open-source library for distribution-free uncertainty quantification. *arXiv preprint arXiv:2207.12274*, 2022.

[158] Mouhcine Mendil, Luca Mossina, and David Vigouroux. Puncc: a python library for predictive uncertainty calibration and conformalization. In *Conformal and Probabilistic Prediction with Applications*, pages 582–601. PMLR, 2023.

[159] Antonio Guerriero, Roberto Pietrantuono, and Stefano Russo. Operation is the hardest teacher: estimating dnn accuracy looking for mispredictions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 348–358. IEEE, 2021.

[160] Marco A.F. Pimentel, David A. Clifton, Lei Clifton, and Lionel Tarassenko. A review of novelty detection. *Signal Processing*, 99:215–249, 2014. ISSN 0165-1684. doi: https://doi.org/10.1016/j.sigpro.2013.12.026. URL https://www.sciencedirect.com/science/article/pii/S016516841300515X.

[161] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *International Conference on Learning Representations*, 2018.

[162] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[163] Wilson L Taylor. Cloze procedure: A new tool for measuring readability. *Journalism quarterly*, 30(4):415–433, 1953.

[164] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.552. URL https://aclanthology.org/2021.emnlp-main.552.

[165] NIST. Software Assurance Reference Dataset Project. https://samate.nist.gov/SRD/.

[166] Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.

[167] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[168] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 596–607, 2022.

[169] Frank Busse, Martin Nowack, and Cristian Cadar. Running symbolic execution forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 63–74, 2020.

[170] Codeql, discover vulnerabilities with semantic code analysis engine. URL https://codeql.github.com/.

[171] Infer, a static program analyzer. URL https://fbinfer.com/docs/about-Infer.

[172] Gcc, the gnu compiler collection. https://gcc.gnu.org/.

[173] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, pages 1–20, 2008.

[174] Common Weakness Enumeration. https://cwe.mitre.org/.

[175] Jenkins, open source automation server. URL https://www.jenkins.io/.

[176] Weiwei Gu, Aditya Tandon, Yong-Yeol Ahn, and Filippo Radicchi. Principled approach to the selection of the embedding dimension of networks. *Nature Communications*, 12(1):3772, 2021.

[177] Nadeeshaan Gunasinghe and Nipuna Marcus. *Language Server Protocol and Implementation*. Springer, 2021.

[178] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *Proceedings of the ICLR*, 2018.

[179] Changsheng Li, Handong Ma, Zhao Kang, Ye Yuan, Xiao-Yu Zhang, and Guoren Wang. On deep unsupervised active learning. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 2626–2632, 2021.

[180] Andrew A Neath and Joseph E Cavanaugh. The bayesian information criterion: background, derivation, and applications. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(2):199–203, 2012.

[181] Joern(Open-Source Code Querying Engine for C/C++.). https://joern.io/.

[182] OSS-Fuzz. https://github.com/google/oss-fuzz.

[183] Davide Castelvecchi. Can we open the black box of ai? *Nature News*, 538(7623): 20, 2016.

[184] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *OSDI*, volume 20, pages 667–682, 2020.

[185] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[186] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[187] Leslie Rice, Eric Wong, and Zico Kolter. Overfitting in adversarially robust deep learning. In Hal DaumÃ© III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8093–8104. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/rice20a.html.

[188] Benoît Frénay and Michel Verleysen. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, 25(5): 845–869, 2013.

[189] Farzaneh S. Fard, Paul Hollensen, Stuart Mcilory, and Thomas Trappenberg. Impact of biased mislabeling on learning with deep networks. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2652–2657, 2017. doi: 10.1109/IJCNN.2017.7966180.

[190] Xiaojiang Peng, Kai Wang, Zhaoyang Zeng, Qing Li, Jianfei Yang, and Yu Qiao. Suppressing mislabeled data via grouping and self-attention. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*, pages 786–802. Springer, 2020.

[191] Carla E Brodley and Mark A Friedl. Identifying mislabeled training data. *Journal of artificial intelligence research*, 11:131–167, 1999.

[192] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski,

and David Li. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808*, 2021.

[193] Rahim Mammadli, Ali Jannesari, and Felix 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)*, pages 1–11. IEEE, 2020.

[194] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9):5149–5169, 2021.

[195] Ke Li and Jitendra Malik. Learning to optimize, 2016. URL https://arxiv.org/abs/1606.01885.

[196] Dehao Chen, Tipp Moseley, and David Xinliang Li. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 12–23. IEEE, 2016.

[197] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 305–316, 2013.

[198] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):1–11, 2016.

[199] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[200] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 3393–3404, 2018.

[201] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[202] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *International Conference on Learning Representations*, 2019.

[203] Henry Massalin. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126, 1987.

[204] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[205] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3796–3803, 2019.

[206] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[207] Hado P van Hasselt, Arthur Guez, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29, 2016.

[208] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[209] Marcelo Pecenin, André Murbach Maidl, and Daniel Weingaertner. Optimization of halide image processing schedules with reinforcement learning. In *Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 37–48. SBC, 2019.

[210] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[211] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1188–1196, Bejing, China, 22–24 Jun 2014. PMLR.

[212] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning, 2020.

[213] Collective benchmark. https://ctuning.org/wiki/index.php/CTools:CBench.

[214] llvm-test-suite. https://github.com/llvm/llvm-test-suite.

[215] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Function merging by sequence alignment. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 149–163. IEEE, 2019.

[216] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Effective function merging in the ssa form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 854–868, 2020.

[217] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. Hyfm: function merging for free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 110–121, 2021.

[218] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. 2014.

[219] Joaquin Quinonero-Candela, Masashi Sugiyama, Anton Schwaighofer, and Neil D Lawrence. *Dataset shift in machine learning*. Mit Press, 2008.

[220] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2):58, 2004.

[221] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 245–256. IEEE, 2017.

[222] Foivos Tsimpourlas, Pavlos Petoumenos, Min Xu, Chris Cummins, Kim Hazelwood, Ajitha Rajan, and Hugh Leather. Benchdirect: A directed language model for compiler benchmarks. In *The 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2023.

[223] Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[224] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. Improving spark application throughput via memory aware task co-location: A mixture of experts approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 95–108, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450347204. doi: 10.1145/3135974.3135984. URL https://doi.org/10.1145/3135974.3135984.

[225] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. Adaptive deep learning model selection on embedded systems. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 31–43, 2018.

[226] Alexander Gepperth and Barbara Hammer. Incremental learning algorithms and applications. In *ESANN*, 2016.

[227] RR Ade and PR Deshmukh. Methods for incremental learning: a survey. *International Journal of Data Mining & Knowledge Management Process*, 2013.

[228] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation techniques*, pages 455–466, 2014.

[229] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 201–211, 2020.

[230] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Trans. Archit. Code Optim.*, 8(4), jan 2012. ISSN 1544-3566. doi: 10.1145/2086696.2086729. URL `https://doi.org/10.1145/2086696.2086729`.

[231] Vladimir Vovk. Conditional validity of inductive conformal predictors. In *Asian conference on machine learning*, pages 475–490. PMLR, 2012.

[232] Yaniv Romano, Evan Patterson, and Emmanuel Candes. Conformalized quantile regression. *Advances in neural information processing systems*, 32, 2019.

[233] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[234] Mauricio Sadinle, Jing Lei, and Larry Wasserman. Least ambiguous set-valued classifiers with bounded error levels. *Journal of the American Statistical Association*, 114(525):223–234, 2019.

[235] Anastasios Nikolas Angelopoulos, Stephen Bates, Michael Jordan, and Jitendra Malik. Uncertainty sets for image classifiers using conformal prediction. In *International Conference on Learning Representations*, 2020.

[236] Yaniv Romano, Matteo Sesia, and Emmanuel Candes. Classification with valid and adaptive coverage. *Advances in Neural Information Processing Systems*, 33:3581–3591, 2020.

[237] Vladislav Ishimtsev, Alexander Bernstein, Evgeny Burnaev, and Ivan Nazarov. Conformal k-nn anomaly detector for univariate data streams. In *Conformal and Probabilistic Prediction and Applications*, pages 213–227. PMLR, 2017.

[238] Giovanni Cherubin, Konstantinos Chatzikokolakis, and Martin Jaggi. Exact optimization of conformal predictors via incremental and decremental learning. In *International Conference on Machine Learning*, pages 1836–1845. PMLR, 2021.

[239] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and image processing*, 14(3):227–248, 1980.

[240] David Arthur and Sergei Vassilvitskii. K-means++ the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, 2007.

[241] Robert Tibshirani, Guenther Walther, and Trevor Hastie. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2):411–423, 2001.

[242] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna 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.

[243] Alexey Natekin and Alois Knoll. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, 7:21, 2013.

[244] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3971–3988, 2022.

[245] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jerónimo Castrillón. A case study on machine learning for synthesizing benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, pages 38–46. ACM, 2019.

[246] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. *ACM SIGPLAN Notices*, 41(6):132–143, 2006.

[247] Common Weakness Enumeration. 2023 cwe top 25 most dangerous software errors. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023.

[248] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[249] Feng-Lei Fan, Jinjun Xiong, Mengzhou Li, and Ge Wang. On interpretability of artificial neural networks: A survey. *IEEE Transactions on Radiation and Plasma Medical Sciences*, 5(6):741–760, 2021.

[250] Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025.

[251] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Llm compiler: Foundation language models for compiler optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, CC '25, page 141â€“153, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714078. doi: 10.1145/3708493.3712691. URL https://doi.org/10.1145/3708493.3712691.

[252] Huanting Wang, Dejice Jacob, David Kelly, Yehia Elkhatib, Jeremy Singer, and Zheng Wang. Securemind: A framework for benchmarking large language models in memory bug detection and repair. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management*, pages 27–40, 2025.