2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

LOUPE: End-to-End Learning of Loop Unrolling Heuristics for Abstract Interpretation

Maykel Mattar
Université Paris-Saclay, CEA, List
Université Bretagne Sud, IRISA
Palaiseau, Vannes, France
maykel.mattar@{cea.fr, irisa.fr}

Michele Alberti
Université Paris-Saclay, CEA, List
Palaiseau, France
michele.alberti@cea.fr

Valentin Perrelle
Université Paris-Saclay, CEA, List
Palaiseau, France
valentin.perrelle@cea.fr

Salah Sadou
Université Bretagne Sud, IRISA
Vannes, France
salah.sadou@irisa.fr

***Abstract*—While static program analyzers based on abstract interpretation implement precision-improving techniques to reduce false alarms, such as loop unrolling, their computational cost requires carefully devised heuristics for selective application. Manually designing such heuristics is non-trivial and error-prone, possibly leading to state explosion.**

This paper presents LOUPE, a novel end-to-end approach for automatically learning loop unrolling heuristics for static program analysis. Unlike previous data-driven methods, LOUPE leverages Graph Neural Networks (GNNs) to learn directly from graph-based program representations. To enable supervised learning, we use the static analyzer itself to automatically label training data. We implement LOUPE on top of FRAMA-C/EVA, an open source C static analyzer, and demonstrate that the best performing heuristic (GINE) outperforms the FRAMA-C/EVA built-in heuristic on real-world programs, reducing false alarms by 1.5x while improving analysis performance by 56%. Remarkably, GINE accurately predicts loop unrolling decisions made by expert FRAMA-C/EVA engineers, while maintaining acceptable false-positive rates. Finally, we show that LOUPE can effectively learn heuristics for other static analyzers such as MOPSA.

***Index Terms*—automatic parametrization, static program analysis, machine learning, graph neural networks**

I. INTRODUCTION

Static program analysis by abstract interpretation has proven effective at approximating program behavior to verify properties and detect bugs. Nonetheless, the computed abstractions are often imprecise and may result in *false alarms*, indicating potential errors that, upon closer examination, turn out to be non-issues. As an excessive number of false alarms undermines user trust and hinders adoption, state-of-the-art static analyzers, such as ASTRÉE [1], FRAMA-C [2], GOBLINT [3], MOPSA [4] SPARROW [5] and POLYSPACE [6], provide users with abstractions and strategies to improve analysis precision and reduce false alarms, albeit at the expense of efficiency.

This work was supported by France Agence Nationale de la Recherche (ANR), program France 2030, reference ANR-22-PTCC-0001. This publication was made possible by the use of the FactoryIA supercomputer, financially supported by the Ile-De-France Regional Council.

Notable strategies are those based on trace partitioning [7], a collection of precision-enhancing techniques that require an analysis to maintain multiple abstract states per program control point instead of a single one. For instance, *loop unrolling* infers a separate abstract state for each loop iteration. Nevertheless, unconditional trace partitioning is prohibitively resource-intensive on real-world programs (potentially leading to non-termination due to infinite execution paths), requiring careful analysis parametrization by users.

Problem: Adequate parametrization is essential for static analysis to be effective, yet it remains a real challenge for users lacking specialized knowledge and expertise. Even for domain experts, it still demands considerable effort and time.

Literature on automating parametrization for static program analysis gravitates towards two opposing methodological paradigms in algorithm configuration [8]. The first class of approaches [9]–[11] provides per-program analysis parametrization by iteratively executing the analysis with varying parameter settings to optimize results *w.r.t.* a cost function (typically, the total number of alarms), within a time budget. Parameter exploration techniques span from local search algorithms [9], [10] to probability distribution refinement methods [11]. Conversely, the second class of approaches [12]–[16] leverages machine learning on large codebases to learn heuristics that selectively apply strategies only to *program components* (*e.g.* loop statements, functions, *etc.*) to optimize adaptive static analysis [17]. The learned heuristics are then used on new, unseen programs without further program-specific tuning.

Ultimately, the first class of approaches aims to tailor static analyzer configurations to individual programs, while the second class aims to apply analysis strategies to specific program components by learning to identify common characteristics.

Goal: This work aims to draw upon modern machine learning techniques to propose a general approach to automatically learn effective heuristics for automating strategy parametrization in abstract interpretation-based static analysis.

As a case study, we consider loop unrolling—arguably one of the most effective, widespread, and adaptable heuristics for improving precision in abstract interpretation. Loop unrolling provides a simple yet powerful form of trace partitioning that mitigates the precision loss inherent in loop analysis by strategically delaying or bypassing imprecise abstract operations. Its fundamental importance is demonstrated by its widespread adoption across modern static analyzers, including FRAMA-C/EVA [2], GOBLINT [3], MOPSA [4], and SPARROW [5]. These tools adapt loop unrolling to fit diverse analysis philosophies, ranging from proving program properties to prioritizing pragmatic bug-finding.

While existing methods have yielded cost-efficient strategies, such as determining flow and context-sensitivity [12], variable clustering in the octagon domain [13], and selecting widening thresholds [14], their effectiveness crucially depends on *feature engineering*, where domain experts manually identify meaningful program characteristics that serve as input for machine learning to build predictive models. Yet, feature engineering suffers from serious drawbacks. Designing meaningful features requires substantial domain expertise and demands considerable time and effort through an iterative trial-and-error process. As such, handcrafted features reflect the designer bias and rarely transfer between analyses and analyzers. Ultimately, current data-driven approaches shift the burden of engineering heuristics from coding to feature design.

Prior works have investigated automatic feature generation to overcome these limitations. However, these approaches are either designed for specific analysis strategies [18] or rely on methods that limit their broader applicability [19], revealing their fundamental inability to provide a general approach for automatically learning heuristics for static analysis.

Proposal: We introduce LOUPE, a novel *end-to-end* data-driven approach (and tool) to automatically learn heuristics for loop unrolling parametrization in static program analysis from existing codebases. The learned heuristic serves as an oracle within the static analyzer to determine loop unrolling decisions when analyzing new programs. Unlike existing ones, ours is an *end-to-end learning* approach that directly maps programs to loop unrolling decisions without requiring feature engineering. While we focus on loop unrolling, LOUPE is designed to be, in principle, general enough to support other analysis strategies.

Our approach is made possible by two key ideas. The first idea enables us to rely on the supervised learning paradigm while avoiding one of its fundamental bottlenecks, namely, manual data labeling. Drawing from differential testing [20], we develop *differential analysis* to automatically label loops by leveraging the loop unrolling strategy provided by the static analyzer. Specifically, for each loop in a program, we perform two static analyses, one with default settings and another with loop unrolling enabled. We then assign label 1 to loops where loop unrolling reduces the number of alarms, and 0 otherwise. The second idea is to leverage modern deep learning techniques to learn features directly from data, circumventing the limitations of feature engineering. We train off-the-shelf Graph Neural Networks (GNNs) to learn from graph-based program representations, which have proven effective across several studies, ranging from vulnerability detection to compiler optimizations [21]–[24].

We have implemented LOUPE as a modular pipeline, using FRAMA-C/EVA [25] as the primary static analyzer. The pipeline encompasses data preprocessing, model training and evaluation. Its extensible architecture reflects the generality of our framework, enabling future experimentation and adaptation to other analysis strategies beyond loop unrolling.

Evaluation: We use LOUPE to train and evaluate multiple program representations and machine learning architectures. Manually-crafted features [15] perform the worst among vector-based embeddings, while graph-based ones are the best. Across all models, the Graph Isomorphism Network with Edge features (GINE) [26] achieves the best overall performance.

On the Open Source Case Study (OSCS) benchmark [27], the GINE-parametrized FRAMA-C/EVA analysis outperforms the analysis using the built-in `-eva-auto-loop-unroll` heuristic, achieving a *1.5x higher false alarm reduction rate while also reducing analysis time by approximately 56%*. Remarkably, when evaluated on OSCS projects with expert-provided loop unrolling annotations by the FRAMA-C team, GINE correctly predicts over *81%* of these decisions, while maintaining acceptable false-positive rates.

To assess the generality of our approach, we first apply the same GINE model—originally trained on FRAMA-C/EVA data—to guide loop unrolling decisions in MOPSA [4], a second abstract interpretation-based static analyzer. The transferred model achieves promising results, demonstrating cross-analyzer applicability. We then fully integrate MOPSA into the LOUPE pipeline to train a new, specific GINE model, which significantly outperforms the transferred model.

Contributions: We claim the following contributions:

- We propose LOUPE, a novel *end-to-end* data-driven approach to automatically learn loop unrolling heuristics for abstract interpretation-based static analysis;
- We develop *differential analysis* to automatically label large codebases *w.r.t.* loop unrolling by leveraging the underlying static analyzer's loop unrolling technique;
- We implement LOUPE as a modular pipeline, encompassing data preprocessing, model training and evaluation. Its extensible architecture admits experimenting with multiple analyzers and strategies beyond loop unrolling;
- We experimentally demonstrate the effectiveness of the best GNN model (GINE) *w.r.t.* the FRAMA-C/EVA built-in heuristic `-eva-auto-loop-unroll` when analyzing real-world C programs;
- We evaluate model transferability, and extend LOUPE to learn heuristics for the static analyzers MOPSA.

Data Availability Statement: A result replication package is available at https://doi.org/10.5281/zenodo.15559653.

II. MOTIVATION

A. Loop Analysis by Abstract Interpretation

Among static analysis techniques, abstract interpretation [28] provides a theoretical framework for systematically deriving sound approximations of program semantics. This approach enables the static verification of program properties by reasoning about all possible program behaviors. Several industrial-strength static analyzers [1], [3]–[6], [25] implement the principles of this framework to detect, among others, undefined behaviors and prove the absence of runtime errors.

Abstract interpretation is based on three fundamental principles that address the challenge of computing an infinite set of reachable program states in finite time. First, the set of reachable states is over-approximated using a concise abstraction—for instance, representing each numerical variable by the interval of its possible values. Second, this abstraction is computed at each control point—typically by traversing the control flow forward or backward—without distinguishing the specific path taken to reach a given state. Third, analysis termination is guaranteed by using a widening operator that extrapolates from successive potential invariants until one is proved to be a sound inductive invariant.

However, these principles can sometimes introduce excessive imprecision. In such cases, although the inferred set of states captures all reachable states, it may also include unreachable ones, causing the analyzer to emit *false alarms*.

```c
#include <stdio.h>

int fib(int n) {
  int a = 1, b = 1;
  for (int i = 3; i <= n; i++) {
    int tmp = a;
    a += b;
    b = tmp;
  }
  return a;
}

void main() {
  for (int i = 1, n; i <= 10; i++) {
    printf("Enter␣a␣number␣<=␣30:␣");
    scanf("%d", &n);
    if (n > 0 && n <= 30) {
      printf("fib(%d)=%d\n", n, fib(n));
      break;
    }
  }
}
```

Fig. 1: Example of a Fibonacci C program.

To illustrate these concepts, consider the simple program shown in Fig. 1, which implements a Fibonacci function called within a loop in the `main` function. When analyzing this program with FRAMA-C/EVA without additional parameters, the analyzer determines that at the end of the `fib` function, `a` and `b` are in the range $[1, 2^{31} - 1]$, leading to a potential overflow alarm for the operation `a += b`. This imprecision can arise from two sources: (i) the interval abstraction of `a` and `b` fails to capture the relationship between these variables and `i`; (ii) the widening operator may not find a precise invariant before considering the entire positive range of 32-bit integers, which is a trivial invariant[1].

The analysis precision can be improved by enabling path-sensitive reasoning, locally. Loop unrolling, a commonly used trace partitioning technique [7], is particularly effective in this case since the loop has a bounded and small number of iterations. Moreover, by distinguishing between loop iterations, the analysis maintains precise values for the local variables at each iteration. Trace partitioning is implemented in several static analyzers including FRAMA-C/EVA and others [3], [4], [29]–[31]. In FRAMA-C/EVA, increasing analysis precision with the `-eva-precision 2` parameter enables a built-in heuristic that unrolls both loops in the program[2], producing precise intervals and eliminating the false alarm.

B. Loop Unrolling in FRAMA-C/EVA

Automatic loop unrolling in FRAMA-C/EVA is controlled by the parameter `-eva-auto-loop-unroll N`, which instructs the analyzer to unroll every loop that can be proven—by means of an undocumented reasoning—to have fewer than `N` iterations (requiring `N` ≥ 28 in our example). This behavior is automatically enabled with `N` set to 32 when using `-eva-precision 2`.

While loop unrolling can sometimes accelerate the analysis, it often increases the computational cost significantly, particularly for nested loops. In our example, setting `-eva-auto-loop-unroll` with unrolling factor `10` or higher will cause FRAMA-C/EVA to unroll the `main` function's loop, but this provides no precision benefit and merely increases analysis time. In real-world applications, unrolling too many loops can be computationally prohibitive, suggesting the use of lower values for `-eva-auto-loop-unroll`, though this may lead to missed opportunities.

Our experience with FRAMA-C/EVA on industrial case studies has shown the need for careful management of trace partitioning, often requiring loop-by-loop decisions about unrolling. Such fine-grained control can be achieved through code annotations of the form `//@ loop unroll N;`.These annotations instruct the analyzer to unroll the first `N` iterations of the loop. When such an annotation is present on a loop, it overrides the `-eva-auto-loop-unroll` setting for that specific loop. For instance, in our example, we can place a `//@ loop unroll 100;` annotation before the loop in `fib` to unroll it, while preventing the unrolling of the `main`'s one with `//@ loop unroll 0;`, independently of the `-eva-auto-loop-unroll` parameter.

Loop annotations do not require specifying the exact number of iterations: if a higher unrolling factor is provided than needed to reach a fixpoint, the analyzer will stop when no further states are reachable. For loops that must be unrolled, we can specify an arbitrarily high unrolling factor; the analyzer only restricts unbounded unrolling to ensure termination.

[1]As of FRAMA-C 30.0 (Zinc), the interval for `i` is inferred precisely without the need of any additional analysis parameter. In our example of Fig. 1, the imprecision arises solely from the relationship between the variables, not the widening operator.

[2]Using `-eva-precision` with higher values produces the same result.

Manual annotations are useful for unrolling loops that the `-eva-auto-loop-unroll` heuristic would ignore due to high iteration counts, and they prevent unnecessary unrolling when the computational cost outweighs the precision gains. However, this annotation-based approach has significant drawbacks: it requires deep understanding of both the analyzed source code and the analyzer, it is time-consuming, and annotations may become ineffective as the analyzer evolves.

These limitations highlight the importance of developing new automatic heuristics as the one we propose in this paper. Among existing trace-partitioning techniques, loop unrolling stands out as particularly promising: it is broadly supported by state-of-the-art analyzers, delivers one of the most significant improvements in the precision-cost trade-off, and is consequently widely applied—making it ideal for learning automatic parametrization heuristics via machine learning.

III. Background: Supervised Learning on Graphs

A. The Three Ingredients of Supervised Machine Learning

In supervised binary classification, we typically have a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, where each $x_i \in \mathbb{X}$ represents an input and $y_i \in \{0, 1\}$ its corresponding class, or label. Our goal is to learn a *model* $h : \mathbb{X} \to \mathbb{R}$ that maps inputs to class probability, which represent the likelihood that a given input belongs to the positive class (usually labelled as class 1). This function is learned by training a machine learning algorithm on the (x_i, y_i) to optimize its predictive performance.

Supervised machine learning is built on three ingredients:

1) *Hypothesis class.* A family h_Θ of functions, over the set of *(hyper)parameters* Θ, that describes how to map inputs (*e.g.* images, programs) to outputs (*e.g.* class probabilities, predictions). This class represents all possible models $h_\theta \in h_\Theta$ a machine learning algorithm can learn;
2) *Loss function.* A function that evaluates the performance of a specific hypothesis h_θ by measuring the difference between predicted and actual outputs. Given set of parameters $\theta \in \Theta$, it quantifies how well a model achieves its intended task, with lower values indicating better performance;
3) *Optimization method.* A procedure that systematically searches for the optimal parameters θ that minimize the sum of losses over the (training) dataset. This process *learns* the best model within the hypothesis class by iteratively adjusting parameters to improve performance.

While choosing a hypothesis class h_Θ depends on the problem at hand, a typical loss function for binary classification is the binary cross-entropy loss ℓ_{BCE}. Then, supervised binary classification can be formalized as the optimization problem that minimizes the average binary cross-entropy loss between predicted probabilities $h_\theta(x_i)$ and true class y_i across all N training examples:

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} \ell_{\text{BCE}}(h_\theta(x_i), y_i)$$

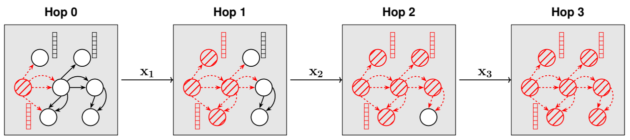

Fig. 2: Illustration of message passing iterations in a GNN.

The optimal parameters θ are found via iterative *gradient descent* optimization, which iteratively updates the parameters in the opposite direction of the gradient of the loss function $\nabla_\theta \ell_{\text{BCE}}(h_\theta, y)$. More formally, given θ_t the parameters at iteration t, $\theta_{t+1} = \theta_t - \eta \nabla_\theta \ell_{\text{BCE}}(h_{\theta_t}, y)$, where $\eta \in \mathbb{R}$, with $\eta > 0$, is called the learning rate.

B. Learning on Graphs with Graph Neural Networks

Supervised learning on graphs is particularly relevant since each loop unrolling decision can be framed as a binary classification task on graph-based program representations (*cf.* Section V). It follows the binary classification principles, but for the Graph Neural Networks (GNNs) hypothesis class.

GNNs for graph-classification are a composition of functions (*layers*) that transform node-level representations (*node features*) into whole graph-level representations (*graph features*). What distinguish GNNs from traditional NNs is their ability to capture relational information between nodes through iterative *message passing* operations over the graph topology. During each iteration, node features are updated by aggregating information from neighbor nodes, allowing the network to learn both local and global structural patterns. Formally, a GNN architecture h_Θ is defined as a composition of functions:

$$h_\Theta = \text{CLASS} \circ \text{READOUT} \circ \text{MP}_K \circ \ldots \circ \text{MP}_1$$

where each MP_k represents a message passing operation, READOUT is a graph-level aggregation operation, and CLASS computes the positive class probability. Parameters Θ comprise all those in $(\text{MP}_k)_{k=1}^{K}$, READOUT and CLASS.

While CLASS is a learnable function implemented as a standard neural network, we now detail the operations performed by the MP and READOUT layers. Given a graph $G = (V, E)$ with initial node features $\{\mathbf{x}_0^v\}_{v \in V}$, each message passing layer MP_k, for $k = 1, \ldots, K$, updates all node features by aggregating information from their neighbors as follows:

- *Message computation.* For each node $v \in V$ and its neighbors $u \in \mathcal{N}(v)$, node features $\{\mathbf{x}_{k-1}^u\}_{u \in \mathcal{N}(v)}$ are transformed as $\mathbf{m}_k^{v,u} = \text{MSG}_k(\mathbf{x}_{k-1}^v, \mathbf{x}_{k-1}^u)$, where function MSG_k is learnable.
- *Message aggregation.* For each node $v \in V$, neighbor's messages $\{\mathbf{m}_k^{v,u}\}_{u \in \mathcal{N}(v)}$ are combined as $\mathbf{a}_k^v = \text{AGGREGATE}_k(\{\mathbf{m}_k^{v,u}\}_{u \in \mathcal{N}(v)})$, where function AGGREGATE_k is learnable and based on permutation-invariant operations such as *sum*, *mean*, *max* or *min*.
- *Node features update.* For each node $v \in V$, all node features are updated as $\mathbf{x}_k^v = \text{UPDATE}(\mathbf{x}_{k-1}^v, \mathbf{a}_k^v)$, where function UPDATE is learnable and based on aggregating operations such as *sum* or *concatenation*.

Fig. 2 illustrates how node features propagate through the graph structure of the network, with each iteration k allowing information to flow to k-hop neighbors.

The graph-level representation $\mathbf{x}_G$ for graph G is obtained by combining the final node features $\{\mathbf{x}_K^v\}_{v \in V}$ as $\mathbf{x}_G = \text{READOUT}(\{\mathbf{x}_K^v\})$, where function READOUT is learnable. Like AGGREGATE, it must be permutation-invariant to ensure independence from node ordering.

While message passing is the common principle behind most GNNs, different architectures arise from particular implementations of the MSG, AGGREGATE, UPDATE and READOUT functions (*cf.* Section V-C).

IV. Problem Formulation

Setting: Strategy parametrization is naturally captured by the notion of *adaptive static analysis* [17]. In this setting, the analysis of a program is guided by an explicit specification of which program components should be subject to particular analysis strategies, enabling fine-grained control over the trade-off between precision and performance inherent in static analysis. For instance, in the case of loop unrolling, the analysis receives a specific set of loop statements to unroll.

Formally, given a program $p \in \mathcal{P}$, let $\mathcal{L}_{S,p} \subseteq \mathcal{L}_S$ denote all program components of p to which S can be applied, and let $\ell_{S,p} \subseteq \mathcal{L}_{S,p}$ denote a specific parametrization of strategy S on p—that is, the specific components of p to which S is applied during analysis. Then, an adaptive analysis *w.r.t.* strategy S is a function $\mathcal{A}_S : \mathcal{P} \times \mathcal{L}_S \to \mathbb{N}$, such that $\mathcal{A}_S(p, \ell_{S,p})$ returns the number of alarms emitted when analyzing program p with strategy S, under strategy parametrization $\ell_{S,p}$.

We write $\bot_S = \mathcal{A}_S(p, \emptyset)$ and $\top_S = \mathcal{A}_S(p, \mathcal{L}_{S,p})$ the analyses that apply strategy S on no and all relevant components in p, respectively. Since we will focus on a single strategy, we omit the subscripts S and p when clear from context.

Static analyses are usually monotonic *w.r.t.* parametrization, that is, for any pair of strategy parametrization ℓ_1, ℓ_2, if $\ell_1 \subseteq \ell_2$ then $\mathcal{A}(p, \ell_2) \leq \mathcal{A}(p, \ell_1)$. In other words, applying a strategy to more program components generally results in fewer alarms, although at the cost of increased analysis time.

Optimization Problem: The objective is a cost-effective function $h : \mathcal{P} \to \mathcal{L}$ such that, for every $p \in \mathcal{P}$, the analysis $\mathcal{A}(p, h(p))$ yields results close to the best parametrization—usually $\top$—while incurring significantly lower cost—ideally, comparable to $\bot$. Such a function can be learned by solving the *adaptive static analysis optimization problem* over a training codebase $\{p_1, \ldots, p_n\}$: given a hypothesis class h_Θ, find $\theta \in \Theta$ that minimizes $\sum_{i=1}^{n} \mathcal{A}(p_i, h_\theta(p_i))$.

While seminal works [12]–[15] use various machine learning techniques to solve this problem, from Bayesian optimization [12] to gradient-based methods [13]–[15], they all rely on feature engineering to construct program embeddings for h_θ. Prior research on automatic feature generation [19] remains limited, relying on a code reducer to generate such program representations. We defer to Section VIII for more details.

Goal and Challenges: As a case study, we focus on the loop unrolling strategy [7] for C code static analysis.

Our goal is to develop an *end-to-end supervised learning framework* to solve the adaptive static analysis optimization problem for loop unrolling—that is, to automatically learn a parametrization heuristic $h : \mathcal{P} \to \mathcal{L}$ that maps programs to loops to unroll—from existing codebases.

Inspired by previous work [13]–[15], we reduce the adaptive static analysis optimization problem to learning a binary classifier ℓ^* that identifies the minimal loop unrolling parametrization that closely approximates $\top$ on a training codebase. Indeed, although $\top$ provides the fewest alarms, unrolling all loops is rarely necessary. Formally, given a hypothesis class h_Θ, this amounts to finding $\theta \in \Theta$ that minimizes the average binary cross-entropy loss between $h_\theta(p_i)$ and true class $\ell^*(p_i)$ across a training codebase $\{p_1, \ldots, p_n\}$.

Although learning loop unrolling parametrization in our setting reduces to a binary classification task, several challenges remain. First, supervised learning requires substantial labeled data, but no dataset exists for our task, and manually labeling publicly available C code is infeasible. The task also exhibits class imbalance, as loops requiring unrolling in static analysis are the minority. Moreover, we must consider a code representation capturing the syntactic and the semantic program characteristics that drive unrolling decisions, which in turn influences the choice of model architecture. Finally, training effective models requires careful hyperparameter tuning.

V. The Loupe Framework

A. Overview

Fig. 3 shows the phases of Loupe, our framework for end-to-end learning of loop unrolling heuristics from codebases.

While grounded in supervised binary classification, Loupe starts with unlabeled C codebases. To enable automatic and scalable labeling, Loupe employs a dedicated data preparation process (*cf.* Section V-B). First, *Slicing* extracts loops into self-contained, analyzable code slices. These are labeled for loop unrolling via *Differential Analysis* leveraging static analysis, and then transformed into suitable program representations by *Embeddings Construction*. Finally, to address class imbalance—as loops requiring unrolling are the minority—*Data Sampling* applies rebalancing techniques to produce a labeled dataset with balanced classes.

In the *Model Training*, Loupe trains various off-the-shelf machine learning models on the resulting dataset, without requiring manual feature engineering (*cf.* Section V-C).

Loupe is also used during static analysis when a loop unrolling decision is required (solid arrows in Fig. 3). Given a loop statement, it extracts the corresponding slice, generates its representation, and uses the trained model (*i.e.*, the learned heuristic) to predict whether unrolling would reduce the number of alarms reported by the static analysis.

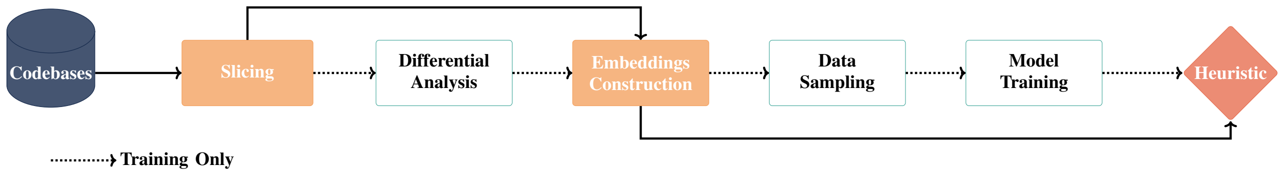

Fig. 3: LOUPE: framework for end-to-end learning of heuristics from C codebases.

B. Data Preparation

1) Codebases: Whole-program static analyzers [1], [2], [6] require complete source code access, including all libraries and dependencies, to successfully parse and process input programs. These strict requirements make it difficult to use preprocessed C datasets commonly found in literature.

To overcome this limitation, we adopt ANGHABENCH [32], a large-scale repository containing over one million self-contained, compilable C files. Because each file is self-contained, it can be process directly by static analyzers without requiring additional modifications. The repository covers a broad spectrum of domains—ranging from multimedia, networking, and operating systems to artificial intelligence, cryptography, and software development tools—providing a rich and diverse basis for training machine learning models.

2) Slicing: Large codebases often contain substantial portions of code irrelevant to specific analysis strategies like loop unrolling. Processing them in full introduces scalability challenges and noise, which can bias the learning process. To address this, we apply program slicing [33] to extract loops with their direct dependencies. This approach isolates the code that directly affect loop behavior, preserving essential control and data dependencies. The resulting program slices are self-contained, offer a more focused, noise-reduced, and computationally efficient representation of loops for training.

3) Differential Analysis: Supervised learning requires labeled training datasets, where each instance is paired with a ground truth label (*cf.* Section III-A). However, in cases like ours, manual labeling is prohibitively expensive: annotators must carefully inspect individual code segments and possess both domain expertise and proficiency in static analyzers. These demands make large-scale manual labeling impractical.

We address this labeling bottleneck by introducing *differential analysis*, a technique inspired by differential testing [20]. Our approach automatically labels loops by comparing static analysis outcomes under different loop unrolling parametrizations. Specifically, for each loop, we run the analyzer twice: once with default settings and once with loop unrolling enabled. If unrolling reduces the number of emitted alarms, the loop is labeled 1; otherwise, it is labeled 0 (*cf.* Algorithm 1). We call ANGHABENCH *dataset* the resulting labeled slices.

This procedure avoids manual labeling while ensuring that labels directly reflect the optimization objective—minimizing alarms in static analysis. Note that differential analysis is only required during training, as the learned heuristic (*i.e.*, trained model) can later predict unrolling decisions without comparative analysis, amortizing the initial computational cost.

Algorithm 1 Differential Analysis *w.r.t.* Loop Unrolling for Automatic Labeling

```
Input: Code slices $\mathcal{S} = \{s_1, ..., s_n\}$, each with a single loop,
    and static analyzer $\mathcal{A}$
Output: Labeled dataset $\mathcal{D} = \{(s_i, y_i) | s_i \in \mathcal{S}, y_i \in \{0, 1\}\}$
 1: $\mathcal{D} \leftarrow \emptyset$
 2: for all $s \in \mathcal{S}$ do
 3:     $\alpha \leftarrow \mathcal{A}(s, \text{LOOPUNROLL} \leftarrow$ false$)$   ▷ #alarms w/o loop unroll
 4:     $\beta \leftarrow \mathcal{A}(s, \text{LOOPUNROLL} \leftarrow$ true$)$   ▷ #alarms w/ loop unroll
 5:     if $\beta < \alpha$ then
 6:         $y \leftarrow 1$          ▷ Loop unrolling reduces number of alarms
 7:     else
 8:         $y \leftarrow 0$          ▷ No improvement observed
 9:     end if
10:     $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s, y)\}$
11: end for
```

While similar techniques have been proposed [13]–[15], they rely either on designing lightweight pre-analyses [13], [14] (*e.g.* impact pre-analysis [34]), or on datasets pre-annotated with buggy program points for labeling [15]. In contrast, by operating on single-loop slices, differential analysis leverages the actual analyzer's unrolling strategy to automatically generate ground truth labels.

4) Embeddings Construction: Applying machine learning to programs requires transforming code into numerical feature representations, or *embeddings*. We consider two alternative approaches: vector embeddings using IR2VEC, and graph embeddings leveraging Code Property Graph (CPG). While IR2VEC leverages fixed, pretrained seed embeddings suitable for feature extraction, graph-based representations support joint training with GNN models, enabling end-to-end learning.

Vector Embeddings: IR2VEC [35] generates dense vector representations of programs, by encoding LLVM intermediate representation (LLVM-IR) instructions into fixed-size embeddings that reflect both syntactic and semantic program characteristics, including control-flow and data-flow information. We fine-tuned the seed embeddings on our ANGHABENCH-derived LLVM-IR corpus, and use IR2VEC to encode the LLVM-IR corresponding to each program slice into a fixed-length vector. The resulting vectors are fed into downstream classifiers for training and inference.

Graph Embeddings: To capture richer code semantics, we combine multiple graph-based program representations—including Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Program Dependence Graphs (PDGs)—into a unified representation, *Code Property Graph* (CPG), by using JOERN [36]. Formally, a CPG is a tuple (V, E, λ, μ), where V are the nodes (from the AST), E are the edges (from AST, CFG, and PDG), λ provides metadata labels, and μ assigns comprehensive properties.

To enable learning, we encode CPG elements into vector representations. Node names and code snippets are embedded using a pretrained sentence transformer model. Categorical features—such as node types and edge labels—are transformed via one-hot encoding. This process yields graphs with the same topology but numerical features instead of raw text.

Formally, the CPG of a program slice is encoded as a triple $(X, \mathit{EdgeAttributes}, \mathit{EdgeIndex})$, where:

- $X \in \mathbb{R}^{n \times d}$ is the matrix of encoded node properties, with n nodes, each with embedding dimension d,
- $\mathit{EdgeAttributes} \in \mathbb{R}^{m \times p}$ is the matrix of edge features, with m edges, each with embedding dimension p,
- $\mathit{EdgeIndex} \in \mathbb{R}^{m \times 2}$ is the list of edge connections as node index pairs.

This representation retains structural and semantic information and ensures compatibility with GNN-based architectures. Importantly, it maintains the end-to-end nature of our learning pipeline: embeddings are learned or transferred, not handcrafted, ensuring full differentiability and scalability.

5) Data Sampling: Due to the infrequent necessity of loop unrolling, our ANGHABENCH dataset exhibits a substantial class imbalance, with a positive-to-negative ratio of approximately 1:11. We consider several strategies to mitigate this imbalance, including random undersampling and oversampling, α- undersampling and oversampling, and SMOTE [37] for vector embeddings only. Crucially, all sampling procedures are applied *only* to the train split. This ensures that validation and test splits remain untouched, preserving the validity of our performance metrics and ensuring that the experimental evaluation outcomes accurately reflect real-world conditions.

C. Model Training

1) Architectures: We investigate three model families: traditional vector-based models such as XGBOOST, pretrained code language models, and Graph Neural Networks.

Gradient-Boosted Decision Trees: XGBOOST [38] is a fast, scalable implementation of gradient boosting that builds tree ensembles to correct residual errors iteratively. It includes regularization to reduce overfitting and handles non-linear feature interactions effectively. In our setup, it serves as a strong baseline for vector-based classification.

Code Language Models (CodeLLM): CODEBERT [39] and GRAPHCODEBERT [40] are transformer models pretrained on large codebases via masked language modeling and bimodal source-code alignment. CODEBERT learns from code–natural language pairs, while GRAPHCODEBERT also integrates data-flow graphs to better capture program structure. We fine-tune both for classification, leveraging their pretrained representations to boost code-related task performance.

Graph Neural Networks (GNN): Programs can be naturally represented as graphs (e.g., control- and data-flow), making GNNs a strong choice. Prior work has demonstrated their effectiveness on program-analysis tasks [23], [36]. We consider several GNN architectures with varying properties in terms of expressiveness, scalability, and learning capabilities.

Dynamic Graph CNN (DGCNN) [41] introduces edge convolutions to capture local structural patterns in control- and data-flow graphs, though the memory-intensive dynamic graph construction limits its scalability on large CPGs.

Graph Attention Network (GAT) [42] employs an attention mechanism to focus on important elements. While effective on node tasks, GAT may underperform on graph-level tasks.

Graph Isomorphism Network with Edge features (GINE) [26] extends the GIN [43] architecture by incorporating edge features in its message passing definition:

$$\mathbf{x}_k^v = \text{UPDATE}\big((1+\epsilon)\cdot\mathbf{x}_{k-1}^v + \sum_{u\in\mathcal{N}(v)} \text{ReLU}(\mathbf{x}_{k-1}^u + \mathbf{e}_{v,u})\big)$$

where UPDATE is a learnable function, $\mathbf{e}_{v,u}$ encodes relationships between connected nodes, and ϵ is a learnable parameter that balances the importance of node features $\mathbf{x}_{k-1}^v$ relative to neighbors' features $\mathbf{x}_{k-1}^u$. This flexibility allows balancing local and aggregate information based on task requirements. In principle, GINE is the most expressive architecture in terms of graph classification, matching the Weisfeiler-Lehman graph isomorphism test [44].

2) Training and Fine-Tuning: We train models on an 80% class-balanced train split of our ANGHABENCH dataset. We fine-tune models on the validation split (10%) and select the best-performing configurations based on their performance on the test split (10%). To reduce variance and avoid split-specific bias, we repeat this process on five different train/validation/test splits. Fine-tuning is automated through SLURM jobs, each exploring distinct hyperparameter combinations tailored to the respective model type. This search space includes standard parameters (*e.g.* layer count and dimensionality, dropout rates, *etc.*), enabling the systematic identification of the optimal configuration for each model.

D. Pipeline Implementation

The artifact accompanying this paper provides a complete implementation of our LOUPE framework (*cf.* Section V). It includes detailed technical specifications, library dependencies, and a modular pipeline architecture to facilitate replication. The pipeline builds on state-of-the-art tools and libraries, including PYTORCH GEOMETRIC [45] and SCIKIT-LEARN [46] for machine learning, as well as IR2VEC [35] and JOERN [36] for vector and CPG extraction, respectively.

LOUPE supports both FRAMA-C/EVA and MOPSA analyzers. The full pipeline is implemented in PYTHON. Slicing and FRAMA-C/EVA-powered differential analysis are FRAMA-C plug-ins, whereas the MOPSA support reuses the same slicing infrastructure but employs a custom differential analysis prototype implemented in PYTHON.

The learned heuristics serve to automatically parametrize loop unrolling strategies in both analyzers. In FRAMA-C/EVA, loop unrolling decisions are applied via code annotations, whereas in MOPSA, they are specified using the `-loop-unrolling-at` command-line option.

VI. EXPERIMENTAL EVALUATION

A. General Overview

We aim at answering the following research questions:

RQ1 Efficacy: Can LOUPE produce effective models, and do they generalize to real-world codebases? How does the best model compare against one trained on handcrafted features?

RQ2 Performance: Does the best learned heuristic improve FRAMA-C/EVA performance, and how does it compare to the built-in `-eva-auto-loop-unroll` heuristic? To what extent do its predictions align with loop unrolling annotations in OSCS provided by the FRAMA-C team?

RQ3 Generality: How well does the best heuristic learned for FRAMA-C/EVA perform when applied to other static analyzers? Can LOUPE be generalized to other static analyzers?

To answer these research questions, we must evaluate LOUPE across different settings, analyzers, and benchmarks.

RQ1 asks whether LOUPE can produce effective models that generalize to real-world code. To answer this, we need to benchmark our approach against multiple baselines, including models trained on handcrafted features and alternative architectures. We therefore use two datasets: the ANGHABENCH dataset test split, which offers realistic class imbalance in standalone C files, and OSCS, a set of real-world C projects with loop unrolling annotations by the FRAMA-C team [27].

RQ2 investigates whether the best learned heuristic provided by LOUPE translates into meaningful performance improvements in the FRAMA-C/EVA analyzer. This requires comparing it against various loop unrolling strategies—ranging from no unrolling ($\perp$) to full unrolling ($\top$), as well as the FRAMA-C/EVA built-in heuristic `-eva-auto-loop-unroll` (`Alu`)—under multiple unrolling budgets and abstract domains. We also assess its prediction alignment with expert-provided annotations in OSCS.

RQ3 focuses on the generality of both the best learned heuristic and the overall LOUPE pipeline. Addressing this requires adapting the differential analysis component to support MOPSA as the underlying static analyzer. We then evaluate whether the best learned model, originally trained on FRAMA-C/EVA data, maintains its effectiveness when applied to MOPSA, and whether retraining on MOPSA data yields further improvements.

B. RQ1 Efficacy: End-to-End Learning Effectiveness

To ensure conceptual rigor and practical relevance, the benchmark reflects the inherent class imbalance of real-world scenarios. Only the best-performing sampling strategy per model is reported: undersampling is applied to all models except XGBOOST [38], which uses SMOTE [37].

As baselines, we consider a random classifier that assigns a 50% probability of unrolling, an SVM leveraging handcrafted features of SPARROW [15], as well as XGBOOST combined with IR2VEC [35] and SMOTE. We also fine-tune CodeLLM models such as GRAPHCODEBERT [40] via transfer learning with encoder-based vectorization for classification. Subsequently, we train multiple Graph Neural Network (GNN) architectures with LOUPE, namely DGCNN [41], GAT [42], GIN [43], and GINE [26].

TABLE I: Performance comparison of loop unrolling heuristic learning on preprocessed and real-world imbalanced datasets.

Model	**ANGHABENCH (Test Split)**				**OSCS**			
	P / R	F1	F2	B-ACC	P / R	F1	F2	B-ACC
Random	8.4 / 50.3	14.3	25.1	50.1	11.5 / 48.2	18.6	29.5	49.2
XGBOOST- IR2VEC	28.3 / 33.5	30.7	32.3	62.7	54.9 / 14.7	23.2	17.2	56.3
SVM- SPARROW	8.2 / 88.6	15.0	29.9	49.2	7.2 / 41.3	12.3	21.3	35.1
GRAPHCODEBERT	20.4 / 81.8	32.7	51.1	76.6	23.3 / 80.7	36.1	54.0	66.2
DGCNN	24.3 / 79.3	37.4	54.8	78.7	21.7 / 50.8	26.6	34.70	58.3
GAT	22.8 / 73.6	35.6	52.2	76.7	21.8 / 68.7	31.1	51.27	66.7
GIN	11.1 / 77.1	34.8	52.1	76.9	23.1 / 49.1	33.4	48.95	67.0
GINE	26.8 / 81.6	**40.4**	**57.98**	**80.85**	26.9 / 79.1	**40.1**	**57.01**	**70.06**

Detailed results are presented in TABLE I. We perform model selection on a 10% validation split, primarily optimizing the F2 score due to the critical impact of missed unrolling. We also consider the F1 score for balancing precision and recall, while balanced accuracy (B-ACC) addresses class imbalance. We report precision and recall (P/R) scores as well. All results are based on the 10% held-out test set.

GNN models consistently outperform all baselines on ANGHABENCH and demonstrate scalability on OSCS, where all models trained with LOUPE outscore the SVM trained on handcrafted features provided by SPARROW. Notably, GINE outperformes the best-performing baseline, a transformer-based CodeLLM (*i.e.*, GRAPHCODEBERT) on both datasets. For reference, on balanced data, GINE achieved F2 and F1 scores exceeding 80%. These improvements can be attributed to the end-to-end nature of the LOUPE approach and the enhanced ability of graph-based models to capture complex code semantics and logical structures. In contrast to handcrafted feature-based methods, which often struggle with semantic ambiguity and representation collisions, LOUPE-trained models—including XGBOOST with IR2VEC—showed less performance degradation on the real-world OSCS benchmark. This robustness likely stems from the larger vector space and end-to-end training, which allow vectorization to adapt dynamically without hardcoded constraints.

Answer to RQ1:
LOUPE demonstrates its effectiveness in producing efficient heuristics, outperforming feature engineering-based methods by over 25% in F2 score and scaling to real-world datasets. GINE consistently outperforms all baseline models, across both the ANGHABENCH dataset test split and OSCS, validating the effectiveness of our end-to-end approach.

C. RQ2 Performance: Precision vs. Analysis Time

To evaluate the impact on FRAMA-C/EVA, we conduct a performance study using the OSCS benchmark. The evaluation compares the best learned heuristic (GINE) against three

TABLE II: Experimental results in terms of RQ2 and RQ3

Project			FRAMA-C (RQ2)									MOPSA (RQ3)							
Name	**LOC**	**#Loops**	⊥		⊤		Alu-100		**LOUPE (GINE)**			⊥		⊤		**LOUPE (GINE)**			
			#	Time	#	Time	#	Time	#	Time	Alignment	#	Time	#	Time	FRAMA-C #	Time	MOPSA #	Time
ioccc	5241	236	81	114.25	77	675.94	80	110.20	78*	127.56		398	5.75	337	1877.12	386	338.83	383	489.98
mini-gmp	11776	257	64	1.88	64	**TO**	64	2.02	64*	1.45									
genann	1186	82	235	3.38	136	58.76	232	4.18	169*	4.80									
papabench	12250	28	40	5.15	40	54.92	40	5.15	40*	3.68	1/1								
chrony	39533	200		**TO**		**TO**		**TO**		**TO**	27/38	15	1.60	15	1.82	15	7.90	15	1.67
kgflags	1478	34	4	4.69	4	976.12	4	4.80	4*	102.05		5700	20.59	5700	58.65	5700	47.02	5700	43.74
libspng	4478	1	237	11.17	233	44.59	237	11.58	237*	8.43									
gnugo	3529	148	114	21.84	114	**TO**	114	**TO**	**100***	27.22	4/4								
debie1	14210	4	31	39.54	9	1158.30	20	81.53	11*	110.77	5/6	3	0.04	3	0.14	3	0.03	3	0.04
basic-cwe-examples	417	9	1	1.62	1	1.17	1	1.69	1*	0.70		42	1.18	40	2.22	**40**	1.00	**40**	1.00
jsmn	1016	15	68	20.83	2	12.54	68	21.14	**2***	66.35		64	0.36	61	1.12	64	0.25	64	0.24
microstrain	42549	201	1287	36.51	1287	**TO**	699	25.61	**700**	19.35									
bench-moerman2018	22292	134	3	39.29	3	28.88	3	41.49	3*	34.74		392	21.32	213	18.04	249	16.21	245	15.98
khash	650	6	1	0.13	0	0.05	1	0.12	1*	0.03	1/1	19	0.34	19	**TO**	19	0.26	19	0.25
c-testsuite	6149	65	0	34.32	0	**TO**	0	**TO**	0*	23.49		156	2.58	65	432.85	117	3.22	112	3.26
icpc	1302	2	1	3.17	1	7.72	1	3.10	1*	2.30		1	0.25	1	0.51	1	0.18	1	0.51
solitaire	338	24	182	2.31	182	**TO**	173	4.17	**173***	18.46	6/6	729	1.03	729	**TO**	**692**	194.98	**692**	567.24
line-following-robot	6793	16	2	0.94	2	4.22	2	0.94	2*	0.64									
safestringlib	29271	479	987	39.94	987	**TO**	845	69.19	**453***	267.48									
stmr	781	6	67	5.21	67	602.19	67	5.38	67*	4.94		55	0.42	55	46.74	55	0.32	55	0.35
powerwindow	5438	4	1	18.18	1	67.33	1	18.17	1*	14.27									
tweetnacl-usable	1204	98	99	3.26	3	26.32	4	25.54	56	4.96	2/2	11	0.28	11	0.62	11	0.26	11	0.21
qlz	1191	11	26	11.13	26	**TO**	26	39.02	26*	130.15									
2048	440	26	30	0.99	10	55.19	13	4.54	12*	1.18	5/5	142	3.68	68	9.16	106	3.78	**68**	8.76
hiredis	9682	31	248	110.69	210	244.97	224	101.75	241	92.39									
verisec	22090	485										1113	11.52	793	68.55	974	11.18	853	39.25
c-utils	9371	202										380	2.87	375	33.26	377	17.79	**375**	22.00
Total**			3809	3230.488	3459	28319,22	2919	8681,39	**2442***	3766,84	51/63 (81%)	9220	73.88	8485	7950.86	8809	636.93	8636	1194.5

Bold values: Best performing result *w.r.t.* metric, **Underlined values**: Better than ⊥, but worse than ⊤, **TO**: Timeout , **#**: Number of alarms.
*: LOUPE outperforms or matches Alu-100. **Alignment**: GINE predictions *vs.* expert annotations (missing values means expert annotations unavailable).
**:In aggregate results, timeouts are treated as follows: alarm counts considered as ⊥, and analysis time is considered of 2700s.
Empty sections: Technical failure occurred when applying the analyzer to these projects.

loop unrolling strategies: unrolling no loops (⊥), unrolling all loops (⊤), and Alu with its default unrolling factor of 100.

As detailed in Table II, LOUPE (GINE) matches approximately 67% of the alarms reduced by ⊤, while achieving substantial speedup on projects where full unrolling did not exceed the 2700-seconds timeout threshold. Moreover, it significantly reduces alarms on projects where ⊤ exceeds the timeout, highlighting the importance of balancing analysis time and precision, and demonstrating its effectiveness. For projects with numerous loops, where unrolling all loops results in unacceptable analysis times, LOUPE (GINE) efficiently enables selective unrolling by leveraging the code's characteristics to guide its decisions. In total, when applied, LOUPE (GINE) succeeds in reducing around 35% of alarms compared to ⊥ while maintaining acceptable analysis time.

When compared to expert annotations in OSCS, LOUPE (GINE) aligns with 51 out of 63 manually labeled unrolling decisions (81%), validating its accuracy and practicality.

We also compare LOUPE (GINE) and Alu under multiple loop unrolling factors, revealing distinct behaviors as shown in Fig.4. For Alu, smaller factors (*i.e.*, 10, 20, 50) produce similar alarm counts, indicating limited unrolling activity.

The moderate factor (*i.e.*, 100) achieves some alarm reduction, while larger factors (*i.e.*, 500, 1000, 5000) reduce alarms further but cause dramatically increased analysis times, as the unrolling decisions depend on the factor value. In contrast, LOUPE (GINE) demonstrates consistent efficiency across all factor values, maintaining low alarm counts and stable, efficient analysis times. The learned model effectively selects loops whose unrolling significantly benefits verification while bypassing those with minimal impact but high computational cost. Fig. 4 shows that LOUPE (GINE) with a factor of 100 (the FRAMA-C/EVA default) achieves superior alarm

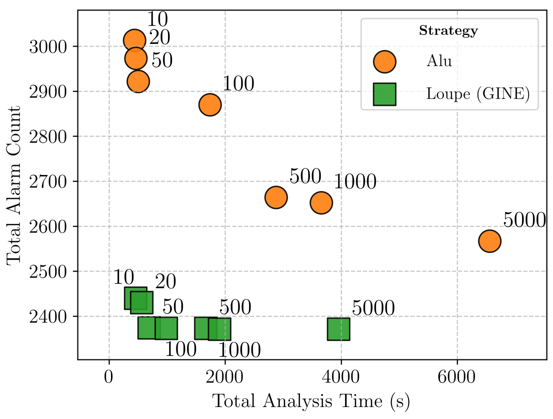

Fig. 4: Trade-off between analysis time and alarm count when using LOUPE (GINE) and Alu with varying unrolling factors.

reduction and acceptable analysis time, outperforming built-in Alu heuristic, even across larger factors. Results include only projects where all configurations complete analysis, as large factors frequently cause Alu to timeout.

To evaluate the consistency of the proposed approach across FRAMA-C/EVA's abstract domains, we consider five projects from OSCS where experts have enabled one or more abstract domains. We conduct such analyses using ⊥, Alu, and LOUPE (GINE)—both with loop unrolling factor of 100—across the four most impactful abstract domains (cvalue [47], equality [28], multidim [48], octagon [49]) and their combinations. As shown in Fig. 5, alarm reduction trends are consistent across different domain configurations, with LOUPE (GINE) outperforming Alu consistently. The absence

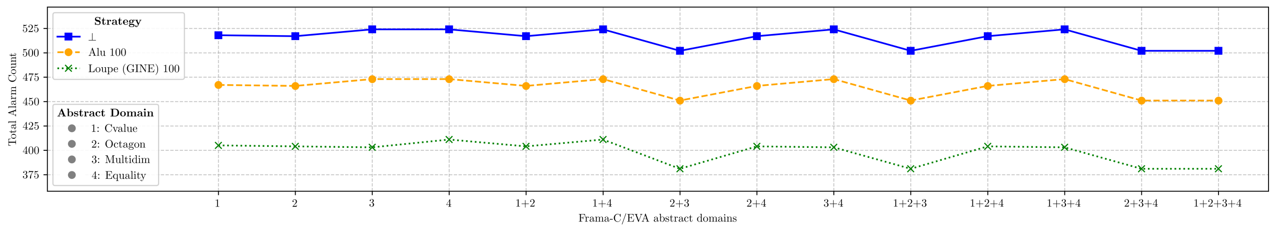

Fig. 5: Total alarm count across FRAMA-C/EVA's abstract domains (and their combinations).

of anomalies confirms the robustness and scalability of our approach across varied abstract domain settings.

Answer to RQ2:

LOUPE (GINE) reduces alarms by up to 35% *w.r.t.* ⊥, it is faster than ⊤, and more reliable under timeouts. LOUPE (GINE) outperformes the FRAMA-C/EVA built-in heuristic `-eva-auto-loop-unroll` across unrolling factors, balancing precision and analysis time. Additionally, LOUPE (GINE) predictions match over 81% of the FRAMA-C expert annotations in OSCS and showed consistent alarm reduction across abstract domain combinations.

D. RQ3 Generality: Scalability and Transferability

A potential threat to the validity of our approach is its reliance on FRAMA-C/EVA, which may limit generalizability to other static analysis frameworks. To mitigate this, we extended our evaluation to include MOPSA [4] (*cf.* TABLE II).

We perform two complementary evaluations. First, we test the transferability of the learned heuristic by applying the GINE model trained on FRAMA-C/EVA data to guide loop unrolling decisions in MOPSA. Second, we fully integrate MOPSA into LOUPE by adapting the differential analysis phase to support MOPSA as the underlying analyzer. This adaptation leverages the MOPSA command-line option `-loop-unrolling-at`, which allows specifying loop unrolling at precise source lines—mirroring the mechanism used in FRAMA-C/EVA via code annotations.

In our experiments, the transferred GINE model already yields performance improvements when used with MOPSA, reducing ⊥ alarms on OSCS. However, the best results are obtained when retraining a GINE model using a ANGHABENCH dataset labeled through MOPSA-based differential analysis. This MOPSA-specific model matches 75% of the alarms reduced by ⊤, while achieving over 6.5x speedup.

Answer to RQ3:

The GINE model trained on FRAMA-C/EVA data exhibits good transferability to MOPSA, improving over ⊥ on OSCS, even without retraining. LOUPE generalizes to MOPSA with minimal modifications—specifically, adapting the differential analysis phase via the `-loop-unrolling-at` option. When retrained on MOPSA-labeled data, the resulting GINE model achieves superior performance, matching 75% of the alarms reduced by ⊤ while delivering over 6.5x speedup.

VII. DISCUSSION

Data Imbalance: While conventional methodologies typically employ balanced datasets for training and evaluation, their performance significantly declines (approximately 73% reduction) in real-world imbalanced scenarios [23], [50]. LOUPE models trained on naturally imbalanced distributions that mirror real-world conditions maintain robust performance, with P/R, F1 and F2 over 80% on balanced datasets.

Data Extraction Overhead: Since our approach requires parsing and extracting CPG graphs, a natural question arises regarding whether their computational overhead reduces its overall efficiency. For the entire OSCS benchmark, slicing requires 287s while graph embeddings construction takes 2,322s, yielding a total runtime (including analysis) of 6,375.84s—still outperforming ⊤ by 4.44x and `Alu`-100 by 1.36x. Note that these preprocessing steps may be further optimized—*e.g.*, by parallelizing per-loop slicing and embeddings construction, or using coarser but cheaper slicing. To evaluate this further, we conduct a comparative study measuring the setup time of LOUPE (GINE) to an iterative process that unrolls loops one-by-one to assess their individual impact on alarms. LOUPE (GINE) is 137.81x faster than the iterative approach—even on a simple project like `solitaire` with only 24 loops.

Loop Unrolling Factor: Loop unrolling factors play a critical role in compilers. Accurately predicting these factors is essential due to the inherent difficulty of estimating them and their significant impact on performance [51]. While static analyzers have achieved notable success in approximating loop bounds [52], and as discussed in Section II-B, loop annotations do not require specifying the exact number of iterations. Consequently, the primary challenge shifts to predicting whether unrolling is necessary. Excessive unrolling can degrade performance; therefore, correctly determining the need for unrolling remains crucial.

To further investigate this, we use FRAMA-C/EVA statistics to extract the actual distribution of loop unrolling factors needed in the ANGHABENCH dataset, setting `Alu` with an unrolling factor of 10,000. Increasing the factor did not yield better outcomes past this point. Unlike other domains where unrolling factors typically range from 1 to 8 [51], our findings reveal that, in this setting, unrolling factors can assume a wide range of values as shown in Fig. 6. Given the efficiency of algorithmically handling unrolling factors and the challenge of predicting them via learned models due to data imbalance,

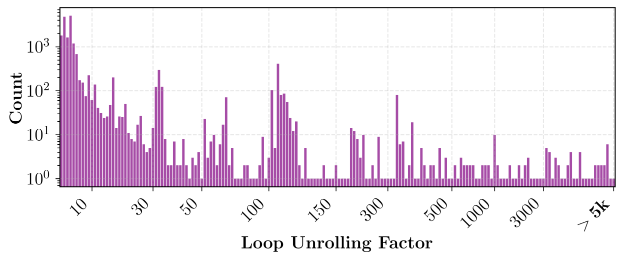

Fig. 6: Distribution of loop unrolling factors in AnghaBench.

this work focuses on binary classification. As shown in Fig. 4, the chosen factor does not significantly impact performance.

Multi-loop Slices: Challenges such as parsing complexity and reliance on labeled data have been mitigated through techniques like slicing and differential analysis. These strategies inadvertently reduce context-sensitivity. A key limitation arises with nested loops, where interdependencies can lead to identical slices—problematic when one loop requires unrolling and the other does not. We leave this issue for future work.

VIII. Related Work

Data-driven Static Analysis: Our work contributes to data-driven static analysis, where machine learning is used to automatically derive effective heuristics for program analysis parametrization [12]–[16]. These works optimize adaptive static analysis [17] via Bayesian optimization [12], decision-tree inference [13], logistic regression [14], SVMs [15], and boolean formula learning [16]. They all rely on handcrafted features, whereas we adopt end-to-end learning.

Efforts toward automatic feature generation remain limited in scope. Some focus exclusively on Java context-sensitivity [18] or rely on third-party tools such as C-reduce to extract features as program components [19]. The latter approach is not applicable to non-C languages and requires manually-crafted oracles to guide the reduction process. Additionally, it uses graph-matching algorithms for feature comparison, which are prone to spurious matches. Our approach does not suffer from such technical limitations and, in principle, could be applied to different languages and analysis strategies.

Automatic Analysis Configuration: Recent research has focused on finding the best analysis parametrization or abstractions automatically [9]–[11], [17], [53], [54], using a range of refinement techniques, such as local search [9], [10], probabilistic learning [11], [53], and counterexample-guided methods [54]. Our approach does not perform per-program parametrization tuning, rather it learns a loop unrolling heuristic from codebases to apply on relevant program components.

Program Embeddings: Embeddings are dense vectors that encode semantic structure, allowing models to detect patterns via similarity. Literature on program embeddings is extensive [35], [55]–[57]. Our baseline relies on the ir2vec embeddings [35], which outperforms syntax tree-based methods [55] by incorporating control- and data-flow information, and avoids the runtime overhead of execution-trace-based approaches [56], [57]. Recent work employs LLMs, treating code as token sequences [39], [40], and GNNs, which generate embeddings from program graphs to capture structural and semantic dependencies [22]–[24]. Our work empirically compares these techniques for learning static analysis heuristics.

ML in Compiler Optimizations: Loop unrolling is a classic compiler optimization that reduces branching and enables further optimizations like instruction-level parallelism. Machine learning has been used to predict unroll factors [51], [58] and identify parallelizable loops [59], [60]. These techniques are not directly applicable to our setting. Unlike compilers, for which a small set of unroll factors suffices (*e.g.* 1–15), abstract interpretation admits a much wider and more scattered value range (*cf.* Fig. 6), making a multiclass classification task substantially more complex. More broadly, there is a growing interest in augmenting traditional compiler optimizations with machine-learned heuristics [22], [61]–[63].

IX. Conclusion

We have presented Loupe, a novel end-to-end approach to automatically learn heuristics for loop unrolling parametrization in abstract interpretation-based static analysis. The learned heuristic outperforms the Frama-C/Eva built-in loop unrolling strategy, reducing false alarms by *1.5x* while improving analysis performance by more than *56%*, and accurately predicting expert annotations in OSCS projects.

Future work will extend the proposed methodology to different analysis strategies while investigating the design of tailored GNN architectures for code analysis.

Acknowledgment

We thank the anonymous reviewers for their constructive feedback, which helped improve the quality of this publication. We thank Sébastien Bardin for fruitful discussions, and former interns Albéric Despres and Florian Biboud for their valuable contributions during the early development of this work.

References

[1] D. Kästner, R. Wilhelm, and C. Ferdinand, "Abstract Interpretation in Industry – Experience and Lessons Learned," in *Static Analysis: 30th International Symposium, SAS 2023, Cascais, Portugal*, 2023.
[2] N. Kosmatov, V. Prevosto, and J. Signoles, *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*, ser. Computer Science Foundations and Applied Logic. Springer International Publishing, 2024.
[3] S. Saan, M. Schwarz, K. Apinis, J. Erhard, H. Seidl, R. Vogler, and V. Vojdani, "Goblint: Thread-Modular Abstract Interpretation Using Side-Effecting Constraints," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. F. Groote and K. G. Larsen, Eds. Cham: Springer International Publishing, 2021, pp. 438–442.
[4] M. Journault, A. Miné, R. Monat, and A. Ouadjaout, "Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer," in *Verified Software. Theories, Tools, and Experiments*, S. Chakraborty and J. A. Navas, Eds., 2020.
[5] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, "Design and Implementation of Sparse Global Analyses for C-like Languages," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 229–238.
[6] MathWorks, "Polyspace," accessed: 2024-12-20. [Online]. Available: https://www.mathworks.com/products/polyspace.html

[7] X. Rival and L. Mauborgne, "The Trace Partitioning Abstract Domain," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 5, p. 26–es, Aug. 2007.

[8] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An Automatic Algorithm Configuration Framework," *J. Artif. Int. Res.*, vol. 36, no. 1, p. 267–306, Sep. 2009.

[9] M. N. Mansur, B. Mariano, M. Christakis, J. A. Navas, and V. Wüstholz, "Automatically Tailoring Abstract Interpretation to Custom Usage Scenarios," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*, 2021.

[10] C. Babu M, M. Lemerre, S. Bardin, and J.-Y. Marion, "Trace Partitioning as an Optimization Problem," in *Static Analysis: 31th International Symposium, SAS 2024, Pasadena, California, USA*, 2024.

[11] Z. Wang, L. Yang, M. Chen, Y. Bu, Z. Li, Q. Wang, S. Qin, X. Yi, and J. Yin, "Parf: Adaptive Parameter Refining for Abstract Interpretation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1082–1093.

[12] H. Oh, H. Yang, and K. Yi, "Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation," *SIGPLAN Not.*, vol. 50, no. 10, p. 572–588, Oct. 2015.

[13] K. Heo, H. Oh, and H. Yang, "Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis," in *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK*, ser. Lecture Notes in Computer Science, X. Rival, Ed., 2016.

[14] S. Cha, S. Jeong, and H. Oh, "Learning a Strategy for Choosing Widening Thresholds from a Large Codebase," in *Programming Languages and Systems*, A. Igarashi, Ed. Cham: Springer International Publishing, 2016, pp. 25–41.

[15] K. Heo, H. Oh, and K. Yi, "Machine-Learning-Guided Selectively Unsound Static Analysis," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 519–529.

[16] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-Driven Context-Sensitivity for Points-to Analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[17] P. Liang, O. Tripp, and M. Naik, "Learning Minimal Abstractions," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 31–42.

[18] M. Jeon, M. Lee, and H. Oh, "Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[19] K. Chae, H. Oh, K. Heo, and H. Yang, "Automatically Generating Features for Learning Program Analysis Heuristics for C-Like Languages," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[20] R. B. Evans and A. Savoia, "Differential Testing: A New Approach to Change Detection," in *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ser. ESEC-FSE companion '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 549–552.

[21] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 596–607.

[22] C. Cummins, Z. Fisches, T. Ben-Nun, T. Hoefler, M. O'Boyle, and H. Leather, "ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations," in *Thirty-eighth International Conference on Machine Learning (ICML)*, 2021.

[23] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[24] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to Represent Programs with Graphs," in *International Conference on Learning Representations (ICLR)*, 2018.

[25] D. Bühler, A. Maroneze, and V. Perrelle, *Abstract Interpretation with the Eva Plug-in*. Cham: Springer International Publishing, 2024, pp. 131–186.

[26] W. Hu*, B. Liu*, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec, "Strategies for Pre-training Graph Neural Networks," in *International Conference on Learning Representations*, 2020.

[27] Frama-C, "Open Source Case Studies: Collection of open source C codes to be used with Frama-C, in particular with the Eva plugin," April 10, 2020, latest commit: 658dae4c. [Online]. Available: https://git.frama-c.com/pub/open-source-case-studies/

[28] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 238–252.

[29] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software*. Berlin, Heidelberg: Springer-Verlag, 2002, p. 85–108.

[30] D. Beyer and M. E. Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 184–190.

[31] C. Calcagno and D. Distefano, "Infer: An Automatic Program Verifier for Memory Safety of C Programs," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465.

[32] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. Ferreira Guimarães, and F. M. 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)*, 2021, pp. 378–390.

[33] B. Monate and J. Signoles, "Slicing for Security of Code," in *Trusted Computing - Challenges and Applications*, P. Lipp, A.-R. Sadeghi, and K.-M. Koch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 133–142.

[34] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective Context-Sensitivity Guided by Impact Pre-Analysis," *SIGPLAN Not.*, vol. 49, no. 6, Jun. 2014.

[35] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant, "IR2Vec: LLVM IR Based Scalable Program Embeddings," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, Dec. 2020.

[36] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[37] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-Sampling Technique," *J. Artif. Int. Res.*, vol. 16, no. 1, p. 321–357, Jun. 2002.

[38] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794.

[39] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[40] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," 2021. [Online]. Available: https://arxiv.org/abs/2009.08366

[41] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic Graph CNN for Learning on Point Clouds," *ACM Trans. Graph.*, vol. 38, no. 5, Oct. 2019.

[42] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," 2018. [Online]. Available: https://arxiv.org/abs/1710.10903

[43] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful are Graph Neural Networks?" 2019.

[44] B. Weisfeiler and A. A. Lehman, "A Reduction of a Graph to a Canonical Form and an Algebra Arising During This Reduction," *Nauchno-Technicheskaya Informatsia*, vol. Ser. 2, no. N9, pp. 12–16, 1968.

[45] M. Fey and J. E. Lenssen, "Fast Graph Representation Learning with PyTorch Geometric," 2019. [Online]. Available: https://arxiv.org/abs/1903.02428

[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duch-

esnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[47] P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '79. New York, NY, USA: Association for Computing Machinery, 1979, p. 269–282. [Online]. Available: https://doi.org/10.1145/567752.567778

[48] P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints Among Variables of a Program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '78. New York, NY, USA: Association for Computing Machinery, 1978, p. 84–96.

[49] A. Miné, "The Octagon Abstract Domain," *CoRR*, vol. abs/cs/0703084, 2007. [Online]. Available: http://arxiv.org/abs/cs/0703084

[50] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep Learning Based Vulnerability Detection: Are We There Yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[51] M. Stephenson and S. Amarasinghe, "Predicting Unroll Factors Using Supervised Classification," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '05. USA: IEEE Computer Society, 2005.

[52] A. J. Venet, "The Gauge Domain: Scalable Analysis of Linear Inequality Invariants," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 139–154.

[53] R. Grigore and H. Yang, "Abstraction Refinement Guided by a Learnt Probabilistic Model," *SIGPLAN Not.*, vol. 51, no. 1, Jan. 2016.

[54] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, "On Abstraction Refinement for Program Analyses in Datalog," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014.

[55] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning Distributed Representations of Code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[56] K. Wang, "Learning Scalable and Precise Representation of Program Semantics," *arXiv preprint arXiv:1905.05251*, 2019.

[57] K. Wang and Z. Su, "Blended, Precise Semantic Program Embeddings," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, 2020.

[58] H. Leather, E. Bonilla, and M. O'boyle, "Automatic Feature Generation for Machine Learning–Based Optimising Compilation," *ACM Trans. Archit. Code Optim.*, 2014.

[59] D. Fried, Z. Li, A. Jannesari, and F. Wolf, "Predicting Parallelization of Sequential Programs Using Supervised Learning," in *2013 12th International Conference on Machine Learning and Applications*, 2013.

[60] A. Maramzin, C. Vasiladiotis, R. C. n. Lozano, M. Cole, and B. Franke, ""It looks like you're writing a parallel loop": A Machine Learning Based Parallelization Assistant," in *Proceedings of the 6th ACM SIGPLAN International Workshop on AI-Inspired and Empirical Methods for Software Engineering on Parallel Computing Systems*, ser. AI-SEPS 2019. New York, NY, USA: Association for Computing Machinery, 2019.

[61] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-End Deep Learning of Optimization Heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[62] S. VenkataKeerthy, S. Jain, U. Kalvakuntla, P. S. Gorantla, R. S. Chitale, E. Brevdo, A. Cohen, M. Trofin, and R. Upadrasta, "The Next 700 ML-Enabled Compiler Optimizations," in *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2024. New York, NY, USA: Association for Computing Machinery, 2024.

[63] V. Seeker, C. Cummins, M. Cole, B. Franke, K. Hazelwood, and H. Leather, "Revealing Compiler Heuristics through Automated Discovery and Optimization," in *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '24. IEEE Press, 2024.