Deep Language Models for Software Testing and Optimisation

Foivos Tsimpourlas

Doctor of Philosophy

Laboratory for Foundations of Computer Science
CDT Pervasive Parallelism
School of Informatics
The University of Edinburgh

2023

Abstract

Developing software is difficult. A challenging part of production development is ensuring programs are correct and fast, two properties satisfied with software testing and optimisation. While both tasks still rely on manual effort and expertise, the recent surge in software applications has led them to become tedious and time-consuming. Under this fast-pace environment, manual testing and optimisation hinders productivity significantly and leads to error-prone or sub-optimal programs that waste energy and lead users to frustration. In this thesis, we propose three novel approaches to automate software testing and optimisation with modern language models based on deep learning. In contrast to our methods, existing few techniques in these two domains have limited scalability and struggle when they face real-world applications.

Our first contribution lies in the field of software testing and aims to automate the test oracle problem, which is the procedure of determining the correctness of test executions. The test oracle is still largely manual, relying on human experts. Automating the oracle is a non-trivial task that requires software specifications or derived information that are often too difficult to extract. We present the first application of deep language models over program execution traces to predict runtime correctness. Our technique classifies test executions of large-scale codebases used in production as "pass" or "fail". Our proposed approach reduces by 86% the amount of test inputs an expert has to label by training only on 14% and classifying the rest automatically.

Our next two contributions improve the effectiveness of compiler optimisation. Compilers optimise programs by applying heuristic-based transformations constructed by compiler engineers. Selecting the right transformations requires extensive knowledge of the compiler, the subject program and the target architecture. Predictive models have been successfully used to automate heuristics construction but their performance is hindered by a shortage of training benchmarks in quantity and feature diversity. Our next contributions address the scarcity of compiler benchmarks by generating human-likely synthetic programs to improve the performance of predictive models.

Our second contribution is BENCHPRESS, the first steerable deep learning synthesizer for executable compiler benchmarks. BENCHPRESS produces human-like programs that compile at a rate of 87%. It targets parts of the feature space previously unreachable by other synthesizers, addressing the scarcity of high-quality training data for compilers. BENCHPRESS improves the performance of a device mapping predictive model by 50% when it introduces synthetic benchmarks into its training data.

BENCHPRESS is restricted by a feature-agnostic synthesizer that requires thousands of random inferences to select a few that target the desired features. Our third contribution addresses this inefficiency. We develop BENCHDIRECT, a directed language model for compiler benchmark generation. BENCHDIRECT synthesizes programs by jointly observing the source code context and the compiler features that are targeted. This enables efficient steerable generation on large scale tasks. Compared to BENCHPRESS, BENCHDIRECT matches successfully 1.8 more Rodinia target benchmarks, while it is up to 36% more accurate and up to 72% faster in targeting three different feature spaces for compilers.

All three contributions demonstrate the exciting potential of deep learning and language models to simplify the testing of programs and the construction of better optimisation heuristics for compilers. The outcomes of this thesis provides developers with tools to keep up with the rapidly evolving landscape of software engineering.

Acknowledgements

First of all, I would like to express my deep gratitude to my three advisors Ajitha Rajan, Hugh Leather and Pavlos Petoumenos for providing me with the best life, research, professional advice that I could ever wish for. Their mentorship throughout this whole journey was crucial for its successful completion. They were always there for me and they influenced me, shaped me and improved me as a researcher, professional and person. I also want to thank my collaborator Min Xu for supporting me and providing me with great advice when I needed it.

I want to thank all the great friends that I made during my PhD which made the journey so much more fun. Andrey for the fun times we had in Edinburgh and London, Anuraag for being a great colleague at ARM, Stefanos for all the fun rides and discussions and Riyasat, Marco and Ansong for all the laughs we had at the office. Their friendship has been invaluable to me.

Next, I want to express my gratitude to all the special people in my life. My parents, Dimitris and Lalita, for supporting me unconditionally throughout my whole life and wanting the best for me. Dimitra, for being always there for me and supporting me unconditionally to all my steps for so many years. I owe to her and my parents a lot for all the support, advice and love that I have received.

Finally, I want to thank all my Greek life-long friends for all the memories and experiences that we have shared together. Each in their own way, they have influenced me to be a better person throughout all these years. I feel extremely lucky to spend quality time with them no matter how many years go by.

Declaration

I declare that this thesis was composed by myself, 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 quali cation except as speci ed.

(Foivos Tsimpourlas)

To my late friend.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software has become a crucial part of modern life. The exponential growth of software in technological applications has scaled the complexity of programs signi cantly [165, 1]. Taking GitHub as an example, the number of developers and repositories have been increasing exponentially since 2008. In just two years (2020 to 2022), the number of developers has risen from 40 to 139 million and the number of repositories has increased from 139 to 200 million (Table 1.1).

The success of software-based products relies on ful lling their promise to make our every day lives more productive, safer, more convenient. Whether that may be a self driving car, a robot vacuum cleaner or a streaming service on the world wide web, they all share the same success factors: Being safe and fast [5, 4]. Software correctness is a program's ability to perform the exact tasks as de ned by their speci cation and is required for it to be established and trusted by millions of users. In privacy or real-time applications (e.g., an autopilot) it is critical indeed; a minor bug can be the cause of a devastating tragedy. Execution ef ciency is equally important. An under-optimised application leads to user frustration and can cause failure on real time systems. In both these domains, compiler engineers and software testing experts have put an enormous effort in producing software that is fast and correct.

However, when the complexity of software increases, so does the effort of testing and optimising it. This is exacerbated by the recent advance of hardware with GPUs, FPGAs and heterogeneous platforms enabling the amount of applications for software to explode. Existing approaches to testing and optimising software require enormous effort and time, while also relying on compiler and low-level systems expertise. Such manual effort is no longer sustainable, leading to under-optimised and buggy software being pushed to production. To keep up with the pace of change in software appli-

	2008	2010	2012	2014	2016	2018	2020	2022
# Git Devs	100K	1M	3M	9M	14M	31M	40M	94M
# Git Repos	6.2K	1M	5M	10M	29M	100M	139M	200M

Table 1.1: The growth of GitHub since it was founded in 2008, depicted by the total number of registered developers and total number of repositories, including both public and private repositories. As of 2022, almost 30 million out of 200 million repositories were public.

cations development, researchers must develop novel techniques that enable the automation of software testing and optimisation processes. This thesis proposes methods that reduce the overhead and improve performance for software testing and compiler optimisation, two important parts of development cycle. In the next two sections, the relevant problems are stated and elaborated, existing techniques are discussed and our proposed approaches are motivated.

1.1 Software Testing and Machine Learning

Software testing is a critical part of software's development cycle as it ensures reliability, security and high performance which further results in time saving, cost effectiveness and customer satisfaction. Testing is divided in many different categories, such as unit, functional and regression testing. All these types heavily rely on human experts developing test cases, namely programs that will exercise a PUT's (Program Under Test) features.

As the scale of software increases, the number of tests needed for effective validation becomes extremely large, ultimately making software development challenging and costly [15]. To achieve cheaper and faster testing, as much of the process as possible needs to be automated. With respect to test input generation, researchers have made remarkable progress in generating effective test inputs [96, 24, 34]. Automated test input generation tools, however, generate substantially more tests than manual approaches. This becomes an issue when determining the correctness of test executions, a procedure referred to as the test oracle In Figure 1.1, we show how a test oracle compares the actual and the expected output of a test execution to determine the subject program's correctness.

The test oracle is still largely manual, relying on developer expertise. Recent surveys on the test oracle problem [20, 118, 96] show that automated oracles based on formal speci cations, metamorphic relations [105] and independent program versions are not widely applicable and are dif cult to use in practice. The recent success of arti- cial intelligence in many applied tasks in computer science has motivated researchers, including the author of this thesis, to explore machine learning as a tool to address the test oracle problem. For codebases with thousands of test inputs that are automatically generated, a machine learning classi er that can automatically predict their outcome when executed would be especially useful.

Figure 1.1: Test oracles compute the test inputs' expected outputs. They identify program correctness by comparing the actual output of a test execution with the expected output.

Previous works exploring the use of machine learning for test oracles have been in a restrictive context - applied to very small programs with primitive data types, and only considering their inputs and outputs [148, 87]. Information in execution traces has not been considered by existing ML-based approaches. Other bodies of work in program analysis have used neural networks to predict method or variable names and detect name-based bug patterns [12, 127] relying on static program information, namely, embeddings of the Abstract Syntax Tree (AST) or source code.

One of this thesis objectives is to propose machine learning based methods to identify the runtime behavior of test executions. Automating the test oracle reduces the cost of software construction and maintenance and improves overall functionality and user experience. The proposed methodology for a machine learning oracle is designed to be widely applicable to real, large-scale codebases and provide testing experts with intuitive abstractions to incorporate it into their testing pipeline.

Figure 1.2: Training pipeline of a predictive model.

1.2 Compiler optimisation and Machine Learning

Program optimisation is the second area of focus in this thesis. The most essential software tool in producing efficient programs is the compiler. Compilers optimise programs through a set of transformations, from an input code to machine instructions that best utilise the resources of a target architecture. The selection of these transformations, or their parameters, is based on heuristics applied by compiler engineers. Each architecture supported by a compiler requires extensive manual tuning by experts to achieve great performance. As knowledge of the whole compiler is required to optimise a single heuristic, this become increasingly difficult when software complexity increases, leading to high development cost, complexity [159] and slow adaptation to the rapidly changing hardware landscape [38, 114, 89]. When compilers cannot keep up with the pace of change, the result is sub-optimal executables that consume more time and energy.

To aid the labour-intensive process of constructing optimisation heuristics, machine learning has been successfully used in several approaches [111, 157, 108, 39, 153, 158, 154, 51, 37, 99, 120, 142] in the form of predictive modeling. Predictive models predict outcomes by analyzing patterns in a given set of features extracted from input data as shown in Figure 1.2. For example, instead of engineers expertly crafting the loop unrolling heuristic through intuition and experimentation, a predictive model can be trained on empirical data of the performance of loops under multiple configurations. It can then be used to predict the best loop unrolling decision for any program on any hardware. Unlike manual-driven techniques, a predictive model can be easily adapted to new architectures and compilers simply by repeating the data collection for re-training. Estimating compiler optimisation heuristics through predictive modeling has been shown to outperform human experts and reduce development time [40, 42].

However, designing effective predictive models requires extensive and diverse training data to help learn accurate optimisation heuristics. In the eld of compilers there is an acute shortage of benchmarks, both in quantity and diversity of features [155, 42, 40]. The average number of benchmarks used in performance tuning papers was 17 in 2017 [40, 155, 33, 69, 163, 18, 36, 55], while other areas of machine learning rely on orders of magnitude more data [45]. A shortage of benchmarks in training leads to poor feature space coverage that degrades the performance of predictive models [42, 59]. Because of this, their potential for success depends on data augmentation techniques.

The most common approach to generating programs is fuzzing [164, 103]. Fuzzers generate programs by inserting random statements and expressions that conform to a target language's standard. In the eld of compilers, fuzzers [164, 103] are commonly used to produce compiler benchmarks in C and OpenCL respectively. However, fuzzed programs differ signi cantly from code that has been written by humans so much that predictive models perform worse when trained on their extracted features [40]. Prior research [40] has shown generative models based on deep learning are an alternative solution by generating an unbounded number of benchmarks. Synthesized benchmarks resemble programs written by humans but a recent survey [59] shows they are short, repetitive and do not extend the diversity of features of existing training benchmarks, they are therefore ineffective. There are speci c areas of the feature space that are missing from our existing datasets, but no research work covers these missing features with new programs.

To improve compiler benchmarks for predictive models, we need a generative approach that identi es and targets those features that are missing from existing datasets. This thesis aims to bridge the gap between the achieved and potential performance of predictive modeling for compiler heuristics by proposing novel techniques to generate high quality compiler benchmarks at scale.

1.3 Contributions

Many software testing and compiler optimisation tasks that are vital for the software industry's success still remain largely manual. Inspired by the ever increasing performance of machine learning applications in many tasks, we claim they have the potential to be faster and better in performing tasks that have been long considered achievable only by experts. This thesis tackles this issue and proposes three novel contributions that reduce development cost and take away the need for domain expertise, human

Figure 1.3: Three contributions presented in applying ML to automate software testing and compiler optimisation: (a) We propose an ML oracle, which allows developer to label only 14% of test cases with the remaining being classified automatically. (b) We present BenchPress, a steerable program generator that uses active learning to improve the training data of predictive models for compilers. (c) We develop BenchDirect, the first directed language model for targeted compiler benchmark generation. This contribution targets benchmarks written by compiler experts up to 36% more accurately and up to 72% faster compared to BenchPress.

intuition and manual effort. One contribution lies in the field of software testing and two in compiler optimisation. The overview of this thesis's contributions are shown in Figure 1.3.

The key contributions of this thesis are the following:

- The first application of deep learning over program execution traces to predict runtime correctness. We develop the first NN-based test oracle that classify test executions as “pass” or “fail”. Our model achieves a near maximum classification accuracy on 15 real, large-scale codebases by training only on 14% of the initial labelled data. This addresses the manual effort of labelling manually the expected output of test executions for large scale production code.

- BenchPress, the first steerable deep learning program synthesizer to generate compilable, executable benchmarks for compilers. BenchPress produces human-like programs that compile at a rate of 87% in contrast to 2.33% achieved by the current state of the art generator. BenchPress synthesizes benchmarks that target parts of the feature space previously unreachable by human-written code from GitHub. This contribution addresses the scarcity of high-quality training data for compiler predictive models.

- BenchDirect, the first directed language model for compilers. We develop a language representation model that synthesizes compiler benchmarks by jointly conditioning on source code context and the desired compiler features in any feature space. BenchDirect outperforms BenchPress in the task of directing program generation towards the features of human-written benchmarks by targeting them up to 36% more accurately and up to 72% faster. This contribution addresses the inefficiency of existing directed program generation techniques, enabling large scale tasks.

1.4 Publications

This thesis consists of four publications describing our research ideas and results.

Chapter 4 elaborates on our approach to classify program executions, published in:

- “Supervised learning over test executions as a test oracle”
  F. Tsimpourlas, M. Allamanis, A. Rajan, SACSE 2021 [145]).

- "Embedding and classifying test execution traces using neural networks"
  F. Tsimpourlas, G. Rooijackers, A. Rajan, M. Allamanis, IET Software 2022 [146].

Chapter 5 presents BENCHPRESS, the first guided program generator for compiler benchmarks, based on language modeling and active learning:

- "BenchPress: A Deep Active Benchmark Generator", F. Tsimpourlas, P. Petoumenos, M. Xu, C. Cummins, K. Hazelwood, A. Rajan, H. Leather, PACT 2022 [144].

Chapter 6 discusses BENCHDIRECT, a directed language model for efficient and accurate steerable compiler benchmark generation:

- "BenchDirect: A Directed Language Model for Compiler Benchmarks", F. Tsimpourlas, P. Petoumenos, M. Xu, C. Cummins, K. Hazelwood, A. Rajan, H. Leather, Submitted to TACO, March 2023 [143].

The source code and experimental data for all three contributions are publicly available on GitHub for other researchers to use on the following repositories:

1. https://github.com/fivosts/Learning-over-test-executions
2. https://github.com/fivosts/BenchPress

1.5 Structure

This thesis is organized as follows:

Chapter 2 provides background. The relevant terminology is defined and all techniques used in this work are described.

Chapter 3 offers a review of existing literature, divided into three categories: program testing, program optimisation and language modeling for program generation.

Chapter 4 presents a novel approach for solving the test oracle problem. A qualitative evaluation shows the effectiveness of machine learning identifying incorrect program executions with near maximum accuracy.

Chapter 5 introduces BENCHPRESS, a novel directed program synthesizer that finds important program features with active learning and generates programs with such features. BENCHPRESS outperforms the state of the art in multiple program generation tasks and improves the performance of predictive models for compilers.

Chapter 6 presents BenchDirect, an optimised steerable program synthesizer based on a novel, feature-conditioned language model. BenchDirect outperforms BenchPress in targeting compiler features in all three accuracy, speed and code quality measured as its human-likeness.

Chapter 7 summarizes the overall findings of the thesis, provides a review of current limitations and outlines potential future directions.

Chapter 2

Background

In this Chapter, we provide necessary background information on various aspects and concepts of software testing, compiler optimisation and machine learning that are relevant to the scope of this thesis.

Section 2.1 describes the software testing workflow and concepts involved in the testing process. Section 2.2 illustrates the fundamentals of compilers and software optimisation techniques. Section 2.3 illustrates modern machine learning methodologies that are used in all thesis contributions in software testing and compiler optimisation.

2.1 Software Testing

Software testing is the procedure that examines the behavior of subject programs, known as Programs Under Test (PUT). The primary purpose of software testing is to expose failures so they can be discovered and corrected before programs are released to users [92]. Testing's scope includes the examination as well as the execution of code in various conditions to determine correctness. To validate programs through execution, a set of test inputs are needed, also known as test cases. An overview of the software testing process is shown in Figure 2.1.

2.1.1 Terminology

First, in Table 2.1 all the definitions and terms used in this thesis are provided to avoid confusion and be consistent with the current literature.

Figure 2.1: An overview of the software testing process.

De nition	Description
System/Program Under Test (SUT/PUT)	Software or System that is tested.
Test Oracle	a mechanism that determines whether a test execution is correct or not.
Software Speci cation	A description of a PUT's intended behaviour and the foundation of the test oracle.
Test Input	The input used to execute a PUT.
Test Output	The real output data collected from a PUT's execution given a speci c test input.
Expected Output	The expected output data from the execution of a speci c test input.
Test Case	The test input with its expected output.
Test Suite	A collection of test cases.
Test Failure	A state where a PUT raises an exception or an unde ned behavior leading to not producing a test output.
Test Result	The status indicating whether a test has passed or failed.

Table 2.1: Software testing terms and de nitions used throughout this thesis.

2.1.2 Software Testing Workflow

We illustrate the process of software testing with an example. Listing 2.1 presents a binary search function implemented in Python. The function has four inputs: a sorted array of integers array, two numbers low and high indicating the bounds of the array, and a number target, for which to search in the array. It performs a binary search in the array and returns the first index at which the target is found.

```python
def bin_search(array, low, high target):
  if high>=low:
    mid=low+(high-low)//2
      if array[mid]==target:
        return mid
      elif array[mid]>target:
        return bin_search(array, low, mid-1, target)
      else:
        return bin_search(array, mid+1, high, target)
  else:
    return -1
```

Listing 2.1: An example Python program performing binary search on sorted integer array

The functional specification for this binary search is the following:

1. It accepts a sorted array of integer values, two integers to index the array and one integer to search the array for.
2. It returns the first index at which the target is present in the array. If the number is not present, it returns -1.
3. If the array is empty, it returns -1.

Table 2.2 shows a test suite for this program based on this specification which contains four tests. Listing 2.2 presents an implementation for these tests. In this case the PUT is the bin_search() function. The test inputs are declared and initialized, the PUT is executed with the test inputs and the test output is recorded. The test output is compared to the expected output using assertions.

Test ID	array	low	high	target	Expected Output
1	fg	0	0	2	-1
2	f 1, 2, 2, 4g	0	3	2	1
3	f 1, 6, 7g	0	2	7	2
4	f 1, 6, 7g	0	2	8	-1

Table 2.2: An simple test suite for the provided binary search example on Listing 2.1.

```
def test1():
  assert bin_search([], 0, 0, 2)==-1

def test2():
  assert bin_search([1,2,2,4], 0, 3, 2)==-1

def test3():
  assert bin_search([1,6,7], 0, 2, 7)==2

def test4():
  assert bin_search([1,6,7],0,2,8)==-1
```

Listing 2.2: A simple implementation for the example test suite given

2.1.3 Testing Practices

The purpose of software testing is to provide con dence that a system's behaviour conforms to speci cation. Ideally, test suites will exhaustively sample the entire input space of a system, but in practice this is infeasible because it would require billions of tests even for trivial programs. However, the goal of software testing is to uncover faults in the system which can be xed, therefore suf cient testing should eventually provide enough con dence that no critical faults remain.

Testing can be applied at any level of system granularity - individual functions, modules and entire systems. *Unit testing* checks the behaviour of the smallest functional units of a program. *Integration testing* checks the correctness of the interactions between units and modules. *System testing* focuses on the behaviour of the system as a whole. *Regression testing* aims to ensure that changes to the existing codebase do not introduce faults in the system. *Mutation testing* is a type of testing that injects errors by modifying the source code of the PUT with random changes. The PUT is executed with a test suite and it is checked whether the test cases are able to expose these injected alterations [85].

2.1.4 Program Instrumentation

Program instrumentation is used to measure performance, discover errors and collect runtime information in the form of execution traces by inserting extra code to a program under test [109]. The inserted code must not alter the original functionality of the program and must also be used to monitor certain metrics from a program's execution.

The most common use cases of program instrumentation include:

1. Pro ling [138] measures dynamic program behaviour through execution. Pro ling information helps developers analyse dynamic information that cannot be measured with static analysis.

2. Performance estimation[156] uses timers to code segments that are computationally extensive. Timing information is used to estimate overhead and reveal the time consuming sections of a program.

3. Logging execution information [80] involves recording events related to a program's runtime including crashes, code coverage or control ow. Execution information is used to measure the achieved test effectiveness of test executions.

There is a plethora of instrumentation tools available, written in different languages [141, 31, 81]. In this thesis, we use LibTooling [106] to apply static analysis on C, C++ and OpenCL programs for feature extraction. More details on this approach will be given in Chapter 5. For runtime instrumentation, we use the LLVM [97] framework to record the control and data ow of test executions and collect data on a program's runtime behaviour. The details of this technique is discussed in Chapter 4.

2.1.5 Test Oracle

Test oracles [82] belong in the family of black-box testing techniques. Black-box testing examines the functionality of an application without peering into its internal structures [57, 84] and can be applied to any type of testing, unit, functional or system. Test oracles determine the correctness of a program by comparing the test output of a PUT for a given test input with its expected output. Determining the correct output given a test input is known as the oracle problem, a difficult challenge that involves working with the controllability and observability of a system. This process still remains largely manual.

There exist several categories of test oracles found on the literature. The most common forms are the following:

1. Specified Oracle judges a program under test's correctness based on formal specifications.

2. Implicit Oracle relies on implied information and assumptions such as conclusions based on program crashes.

3. Derived Oracle uses code documentation or system executions when specified oracles are unavailable.

4. Human Oracle. When no other oracle can be used human experts estimate a program's expected behaviour.

5. ML Oracle. Statistical methods based on machine learning that estimate a program's test execution correctness.

This thesis focuses in machine learning oracles. The first contribution described is a machine learning-based technique to automate the test oracle by classifying test executions as “pass” or “fail”. This approach is presented in Chapter 4.

2.2 Compiler Infrastructure

The compiler is a programming tool that translates programs from a given source language to a lower-level target language. Throughout the compilation process, program semantics are preserved. Compilers are expected to produce a good quality representation of programs in the target language, being optimal with respect to objective functions. An important objective function is execution time, i.e., the optimisation goal is to produce a target language representation of the program that will execute as fast as possible.

Compilers are usually presented as three stage architecture, as shown in Figure 2.2. These stages are the front end, the middle end (or optimiser) and the back end. The front end parses and validates the source code, making sure it conforms to the

Figure 2.2: The three-phase pipeline of a compiler.

source language's grammar. This parsed code is translated into an *Intermediate Representation (IR)*. The compiler's optimiser performs a broad variety of transformations to the IR, called *optimisations*. The compiler's optimisations are usually independent of the source language or the target hardware and their goal is to improve the code's performance. The back end is the compiler's *code generator* and translates the IR to the target language. Hardware-related optimisations can also take place during code generation to exploit the architecture's supported features.

One of the most commonly used compilers is the LLVM [97]. LLVM's intermediate representation, *LLVM-IR*, is a human-readable, assembly-like representation that is used by the compiler's middle-end to apply optimisations. LLVM-IR's most notable feature is *Static Single Assignment* (SSA) form. In SSA form, each variable is assigned to only once, and every reference to a variable is a reference to its single assignment. SSA form has a number of useful properties that make it useful for optimising code. For example, it makes it easy to determine the live ranges of variables, which can be used for register allocation. It also makes it easy to detect when variables are used before they are defined, which can help expose errors in the code.

2.3 Machine Learning

Machine learning is a family of statistical methods and algorithms. They can be predictive to make predictions in the future, or descriptive to gain knowledge from data. ML algorithms are broadly classified into four major categories, depending on the nature of the learning response available to a learning system:

1. *Supervised Learning* models learn by labelled input/output examples.

2. *Unsupervised Learning* models learn by similarities on unlabelled/unstructured data.

3. *Reinforcement Learning* models learn by trial and error to maximize their accumulated reward based on a reward function.

4. *Semi-supervised Learning* models learn on partly-labelled or incomplete data.

Predictive models make predictions by correlating their input variables, known as *features*, with their outputs, or *labels*. The n-dimensional space described by n input features is called a *feature space*. A *feature vector* is the set of features describing a single point in the feature space. The deduction of features from raw data is called *feature selection* and it is an important process in machine learning architecture design, which significantly determines the model's performance. Instead of manually selecting features, ML architectures can also be used by learning a numerical vector representation of raw data in the latent space. This process is commonly referred to as *summarization*. The learnt features they extract from raw data are called *embeddings* [29]. Trained embeddings have been shown to significantly outperform manual feature selection in their quality to represent input data [112].

For classification tasks, there are multiple ways to measure a machine learner's performance in predicting labels. All metrics make use of the following scores:

- True Positives (TP): The number of positive class items correctly labelled.
- False Positives (FP): The number of negative class items labelled as belonging to the positive class.
- True Negatives (TN): The number of negative class items correctly labelled.
- False Negatives (FN): The number of positive class items labelled as belonging to the negative class.

To evaluate machine learning models in this thesis, we make use of *precision*, *recall* and *specificity*. These three metrics are defined as follows:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.2}$$

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{2.3}$$

2.3.1 Neural Networks

Arti cial Neural Networks (ANNs) are the most successful and commonly used type of machine learning architecture and are divided into many different categories. The simplest one is *feed-forward neural networks* (FNN) [21]. FNNs consist of a sequence of neuron layers, where every neuron of a previous layers forms a weighted connection with all neurons of the next layer. This way, the ow of information is unidirectional, mapping the input variables to the neural network's outputs, as shown in Figure 2.3.

Figure 2.3: A feed-forward neural network with four input features, two hidden layers and one output neuron.

ANNs consist of the *input layer* that holds as many neurons as the feature space and the *output layer* that consists of one neuron per predicted label. For regression, only one neuron is needed. For classi cation, there is one neuron per label in the label space. All intermediate layers for which there are no ground truth values are known as *hidden layers*. Feed-forward neural networks are powerful in approximating functions. In theory, they can learn any bounded continuous function to arbitrary precision given the right amount of neurons [115].

The prevailing method of training a neural network is back-propagation. A *batch* of B observations is propagated through the network in a *forward* pass, from the inputs to the outputs. The nal output vector of the network $\hat{y}$ are compared against the labels y and the distance between them is computed, $L(\hat{y}, y)$. The error metric, or *loss function*, depends on the task. A common loss function used for the task of classi cation is categorical cross entropy [116]. Neural networks are prone to *over- tting*, when the parameters of the model lose the ability to generalise to unseen data by becoming too specialized on the training data. Many *regularisation* techniques have been adopted to mitigate the risk of over-fitting. One of the most successful is *Dropout*, in which a parameter in the range $[0, 1]$ is used to determine a proportion of artificial neurons to be removed. This helps training by preventing complex co-adaptations on training values [137].

2.3.2 Language Modeling

Language models are a descriptive type of neural networks that gain knowledge from natural or programming languages used by humans to communicate. The contributions of this thesis heavily rely on the use of language models to represent, embed and generate human-readable programming languages including C, C++ and OpenCL.

Modeling languages is a challenging task that requires large amount of data and large models that are computationally expensive. The high-level steps for designing a NN-based language model architecture for source code are the following:

1. *Tokenize*: Separate source code into words.
2. *Encode*: Convert string-formatted words into numerical vectors.
3. *Train*: Apply language modeling task on encoded corpus.

For LM tasks, there are certain architectures that have been shown to capture more effectively than FNNs the relationship of words in a sentence. The most common of them is the *Recurrent Neural Network* (RNN) [133]. The RNN is a type of ANN where the connections between neurons form a feedback loop from successive layers back to predecessing ones. This enables processing arbitrarily large sequences of tokens while learning the correlations between neighbouring words in the sentence. The RNN cell's hidden state at each time step t is calculated based on the input and previous hidden state h_{t-1}, using the weight matrices W and bias vector b to combine the inputs and apply the activation function:

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \tag{2.4}$$

Where, h_t represents the hidden state at time step t, x_t represents the input at time step t, and f represents the activation function. W_{xh} and W_{hh} are the weight matrices that connect the input and hidden states, and b_h is the bias vector.

RNNs can be trained with ordinary back-propagation by unfolding the computation graph over time as shown on Figure 2.4. Back-propagating through time allows the

(a) (b)

Figure 2.4: The computational graph of a RNN, shown as a recurrence relation in a, and unfolded in b. $\mathbf{x}^{(t)}$ is the input, $\mathbf{h}^{(t)}$ is the hidden state, and $\hat{\mathbf{y}}^{(t)}$ is the output. The network comprises of three weight matrices: inputs-to-hidden weights $\mathbf{U}$, hidden-to-hidden weights $\mathbf{W}$, and hidden-to-output weights $\mathbf{V}$.

propagation of error in the temporal domain in the same manner as through layers. RNN's accuracy suffers when they are tasked to learn correlations over long sequences. This is caused by the exponential diminishing or increase of gradients as they are propagated through the activation functions by the recurrence relation. This issue is also known as the *vanishing gradients problem* [76].

A bolstered, RNN-based architecture addresses this issue, namely the *Long Short-Term Memory* (LSTM) [77]. The LSTM inherits the RNN design with the addition of a cell that stores information and three gates which control the flow of it into and out of the cell, shown in Figure 2.5. For a time-step t, an input at this time-step x_t, the hidden-state h_t, the memory's cell-state c_t and input, forget and output gates i_t, f_t, o_t respectively, the output prediction from the LSTM for the next time-step is as follows:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \tag{2.5}$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \tag{2.6}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{2.7}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \tag{2.8}$$

$$h_t = o_t \odot \tanh(c_t) \tag{2.9}$$

Where:

σ is the sigmoid activation function [66] and W, b are the weight matrices and bias vectors, respectively, that are learnt during training. This formula represents the

Figure 2.5: A Long Short-Term Memory cell. Input $\mathbf{x}^{(t)}$ is concatenated with prior hidden state $\mathbf{h}^{(t-1)}$ and used with prior cell state $\mathbf{c}^{(t-1)}$ to compute the next step's hidden state $\mathbf{h}^{(t)}$ and cell state $\mathbf{c}^{(t)}$.

computations performed by one LSTM unit at each time step. In an LSTM network, multiple LSTM units are typically stacked together, with the hidden state of one unit being used as the input to the next unit. The final hidden state summarizes the state of the whole time series and is used as an input to the downstream task, whether that be classification or regression.

The LSTM has been firmly established as the state of the art in sequence modeling and transduction problems. However, its approach to processing words within a sentence in a sequential way precludes parallelization within a training example, meaning words cannot be computed in parallel because of their temporal dependency. This constraint poses a severe performance bottleneck when it comes to designing large language models with billions of training data.

The Transformer[149] addresses this issue and enables the creation of language models with hundreds of billions of parameters. The Transformer is based on the attention mechanism [56, 136, 32]. Its input is first embedded into a continuous representation and then processed by multiple layers of self-attention. The self-attention layers allow the model to attend to different positions of the input sequence simultaneously and weigh their importance when computing the output:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.10}$$

Where, Q, K, and V are the query, key, and value matrices, respectively, and d_k is

Figure 2.6: The architecture of a Transformer Encoder-Decoder.

the dimension of the keys. The output of the self-attention function is a weighted sum of the values, where the weights are computed based on the dot product between the query and the keys. Then this output is processed in parallel by a feed-forward layer followed by a ReLU activation [2]. The Transformer's architecture is shown on Figure 2.6.

The Transformer is applied to numerous tasks in language modeling, generation and predictive modeling. It also has been used as a foundation for many derived architectures. The most notable is BERT (Bidirectional Encoder Representations from Transformers) [46]. BERT learns deep bidirectional representations by jointly conditioning on both left and right context of an input. One of BERT's key features is its ability to preserve the context of the words in a sentence, rather than just processing individual words in isolation. This allows capturing the relationship between words in a more accurate way compared to traditional NLP models, leading to excellent performance on a range of NLP tasks [46]. BERT has been widely adopted as the state-of-the-art in language translation, text summarization, and sentiment analysis tasks [95, 52, 91]. It has also been used to improve the performance of other machine learning models, such as those used for image classification and object detection [101]. The second and third contribution of this thesis extends BERT from a masked language model to a pre-trained generative model for programming languages.

2.3.3 Active Learning

This thesis applies active learning to search feature spaces applied to the field of compilers for areas with few or no representative samples. Active learning is a type of machine learning that involves learning from actively selected, informative examples rather than passively accepting a pre-determined dataset. In active learning, the learning algorithm can interactively query the user (or some other information source) to obtain the desired outputs, and then use these to improve the accuracy of the model.

Active learning is useful in situations where there is a limited amount of labeled data available, or where the cost of labeling data is high. By allowing the learning algorithm to select which examples to label, active learning can improve the efficiency of the learning process and result in better performance compared to using a fixed dataset. There are several strategies for selecting the most informative examples in active learning. In this thesis, we explore two:

- *Query by Committee*, an approach to selective sampling in which disagreement among a committee of oracles is used to select data for labeling [135].
- *Expected Error Reduction*, a technique that selects a datapoint based on its estimated impact on the classifier's future error over all other datapoints [131].

Query by Committee (QbC) is a simple yet powerful algorithm. During training, a committee of students is trained on the same set of data. The next selected datapoint for labeling (query) is chosen according to the principle of *maximal disagreement*. This disagreement is typically measured with *entropy*, a statistical metric which indicates the amount of uncertainty in a set of samples. Given an input data point x that is candidate for labelling, the entropy among N committee members' predictions is defined as:

$$H(x) = \sum_{i=1}^{N} p(o_l^i) \log_2 p(o_l^i); \quad o_l \in L := \{o_0; o_1; \ldots; o_L\} \tag{2.11}$$

Where l is one possible label that belongs in the label space L and $p(o_l^i)$ is the likelihood of the predicted label l by member i. Query by Committee has shown it is an effective active learning algorithm for relatively simple tasks where the label space has only a few dimensions. For more complicated tasks, QbC suffers from poor accuracy with noisy predictions (high entropy) among members' predictions. Also, members of the same architecture (e.g. NNs) tend to produce the same predictions regardless of their number of parameters, initialization of weights and the worthiness of a query. This also hurts the committee's overall accuracy.

Expected Error Reduction (EER) is a more advanced technique that tackles QbC's shortfall in complex tasks. In EER, a single predictive model is directly prompted to query a datapoint among a provided sample set. The point that is estimated to cause the lowest possible aggregated error on the training dataset is selected. This is measured by re-training the predictive model on one pair (x, y) at a time, where x is a datapoint in the sample set and y is a label in the label space, and then measuring the aggregated error over the training set with the categorical cross entropy error function. This process is repeated for all sample datapoints x combined with each possible label y in the label space. EER has shown to learn faster than QbC the classification task, requiring less labelled data. However, to query a single datapoint in the feature space, it is required to re-train the predictive model $N \times L$ times, where N is the number of datapoints in the sample set and L is the size of the label space. This is a significant overhead which is prohibitive for large predictive models or multi-dimensional feature spaces.

2.4 Summary

In this Chapter, all necessary background related to the problem and the contributions presented in this thesis is presented. The reader is introduced to the process of software testing and its categories in Section 2.1. Section 2.2 elaborates on modern compiler architectures and how they are used to optimize programs. Finally, Section 2.3 provides an overview of all machine learning tools and approaches that were used in this thesis. The following Chapter surveys the existing related literature.

Chapter 3

Related Work

3.1 Introduction

This Chapter surveys the existing literature relevant to the contributions of this thesis. Section 3.2 presents the existing literature in the field of software testing and runtime analysis, focused on the test oracle. Section 3.3 reviews current research in program embedding and representation relevant to our test oracle approach. In Section 3.4, a literature review of benchmark generation for compilers is presented, to provide intuition about our second and third contributions.

3.2 The Test Oracle

Test oracles are an important part of the software testing process as they are used as a baseline to compare the actual output of a test input to the expected output and determine whether the test execution has passed or failed.

The importance of oracles as an integral part of the testing process has been a key topic of research for over three decades. We distinguish three different kinds of test oracles, based on the survey by [20]. The most common form of test oracle is a specified oracle, one that judges behavioural aspects of the system under test with respect to formal specifications. Although formal specifications are effective in identifying failures, defining and maintaining such specifications is expensive and also relatively rare in practice. Implicit test oracles require no domain knowledge and are easy to obtain at no cost. However, they are limited in their scope as they are only able to reveal particular anomalies like buffer overflows, segmentation faults, deadlocks. Derived test oracles use documentations or system executions, to judge a system's behaviour, when speci ed test oracles are unavailable. However, derived test oracles, like metamorphic relations and inferring invariants, is either not automated or it are inaccurate and irrelevant making it challenging to use.

For many systems and much of testing as currently practised in industry, the tester does not have the luxury of formal speci cations or assertions or even automated partial oracles [70, 71]. Predicting the behavior of a large scale program is a tedious task; for this reason researchers have tried several methods to automate this process. Statistical analysis and machine learning techniques provide a useful alternative for understanding software behaviour using data gathered from a large set of test executions. Briand et al. [27] present a comprehensive overview of existing techniques that apply machine learning for addressing testing challenges. Among these, the closest related work is that of Bowring et al. [25]. They propose an active learning approach to build a classi er of program behaviours using a frequency pro le of single events in the execution trace. Evaluation of their approach was conducted over one small program whose speci c structure was well suited to their technique.

More recently, Rigger et al. [130] introduced Intramorphic Testing a white-box methodology for the test oracle that comes in contrast to common black-box approaches such as differential and metamorphic testing. Rigger's et al. approach includes changing one component of the SUT in a known way so that the SUT's original output can be related to the changed output. They illustrate how Intramorphic Testing can expose bugs with three example programs: (a) AST printing, (b) Monte Carlo simulations and (c) the Knapsack problem [93]. Intramorphic Testing is a novel approach to the test oracle problem, however it poses several limitations. First, a developer must not only know the whole codebase and how its components interact, but also how a small alteration in one component changes the program's output. This is very dif cult on industry-scale codebases, where the tester is often not the developer. Second, constructing relations between program versions for simple operations might be relatively easy but such relations in complex software is near-impossible. To make things worse, many mutations may have no effect to the SUT's output thus leading to no intramorphic relations. These two factors increase exponentially the amount of effort and time needed to construct high-quality relations in large-scale software and make use of this technique.

Machine learning techniques have also been used in fault detection. Brun and Ernst [28], explored the use of support vector machines and decision trees to rank program properties, provided by the user, that are likely to indicate errors in the program.

Podgurski et al. [126] use clustering over function call pro les to determine which failure reports are likely to be manifestations of an underlying error. A training step determines which features are of interest by evaluating those that enable a model to distinguish failures from non-failures. The technique does not consider passing runs. In their experiments, most clusters contain failures resulting from a single error.

Almaghairbe et al. [11] propose an unsupervised learning technique to classify unlabelled execution traces of simple programs. They gather two kinds of execution traces, one with only inputs and outputs, and another that includes the sequence of method entry and exit points, with only method names. Arguments and return values are not used. They use agglomerative hierarchical clustering algorithms to build an automated test oracle, assuming passing traces are grouped into large, dense clusters and failing traces into many small clusters. They evaluate their technique on 3 programs from the SIR repository [49]. The proposed approach has several limitations. They only support programs with strings as inputs. They do not consider correct classi cation of passing traces. The accuracy achieved by the technique is not high, classifying approximately 60% of the failures. Additionally, fraction of outputs that need to be examined by the developer is around 40% of the total tests.

Almaghairbe et al. [11] assumed that all the passing traces present the same behaviour, leading to them being organized in one, large cluster. On the other hand, according to them, failing traces tend to present non-uniform, wide-ranging patterns, which results in failing traces being spread among many small clusters. According to their methodology, the clusters that are sized below the average of the set, are considered to contain failing traces and clusters sized above the average are considered to contain passing traces. For their evaluation, they used different clustering techniques and experimented on multiple cluster sizes. Almaghairbe et al. used `Daikon' for instrumentation, an invariant detector that uses machine learning to observe program values and summarize them into a set of formulas.

Existing work using execution traces for bug detection has primarily been based on clustering techniques. Neural networks, especially with deep learning, have been successful for complex classi cation problems in other domains like natural language processing, speech recognition, computer vision. There is limited work exploring their bene ts for software testing problems.

NNs were rst used by Vanmali et al. [148] to simulate behaviour of simple programs using their previous version, and applied this model to regression testing of unchanged functionalities. Aggarwal et al. [3] and Jin et al. [87] apply the same approach to test a triangle classification program, that computes the relationship among three edge inputs to determine the type of triangle. Mytkowicz [48] et al. propose TOGA, a Transformer-based approach to infer exceptional and assertion bug finding test oracles. Based on the observation that oracles in developer-written unit tests typically follow a small number of common patterns, Mytkowicz et al. define a grammar that expresses a taxonomy of these patterns. Using this grammar, a two-step neural ranking procedure scores candidate oracles. They evaluate TOGA on test oracle inference, reporting an accuracy improvement of 11% and 33% over related work in two oracle inference datasets. Their tool is also evaluated in bug finding, exposing 57 real world bugs in Java benchmark, DEFECTS4J [90]. TOGA outperforms the comparing oracles in exposing more real bugs and developing a lower False Positive Rate of 25%.

The few existing approaches using NNs have been applied to simple programs having small I/O domains. The following challenges have not been addressed in existing work,

1. Training with test execution data and their vector representation – Existing work only considers program inputs and outputs that are of primitive data types (integers, doubles, characters). Test data for real programs often use complex data structures and data types defined in libraries. There is a need for techniques that encode such data. In addition, existing work has not attempted to use program execution information in NNs to classify tests. Achieving this will require novel techniques for encoding execution traces and designing a NN that can learn from them.

2. Test oracles for industrial case studies - Realistic programs with complex behaviours and input data structures has not been previously explored.

3. Effort for generating labelled training data - Training data in existing work has been over simple programs, like the triangle classification program, where labelling the tests was straightforward. Availability of labelled data that includes failing tests has not been previously discussed. Additionally, the proportion of labelled data needed for training and its effect on model prediction accuracy has not been systematically explored.

The performance of neural networks as classifiers was boosted with the birth of deep learning in 2006 [74]. Deep learning methods have not been explored extensively for software testing, and in particular for the test oracle problem. Recently, a few techniques have been proposed for automatic pattern-based bug detection. For example, Pradel et al. [127] propose DeepBugs, a deep learning-based static analysis tool for automatic name-based bug detection. Allamanis et al. [8] use graph-based neural static analysis to detect variable misuse bugs. In addition to these techniques, several other deep learning methods for statically representing code have been developed [13, 10]. We do not discuss these further since we are interested in execution trace classification and in NNs that use dynamic trace information rather than a static view of the code.

Our first contribution is a deep-learning based approach to embed and classify program executions. In the next section, we provide the reader with the current research in program embeddings.

3.3 Program Representation

Creating representations is a vital process in transforming input data into a format that is readable by a machine learning model. Good representations highlight the data's key qualities and optimize the machine learner's overall performance. Source code representation is an active research field with a large impact on language models' ability to understand and generate synthetic programs.

Natural language representation techniques are usually also relevant to source code representation. Natural and programming languages share some structure similarities, a property known as naturalness. The naturalness hypothesis [7, 73, 72] states that software is a form of human communication and software corpora have similar statistical properties to natural language corpora. Using these properties is critical to build better software engineering tools. The naturalness hypothesis, inspires the goal to apply machine learning approaches to learn how developers naturally write and use code. These models can be used to augment existing tools with statistical information and enable new machine learning-based software engineering tools, such as recommender systems and program analyzers.

Representation of words as continuous vectors is a field with a long research history [50, 75, 132]. The Neural Network Language Model (NNLM) [88] is a popular architecture for word representation, where a feed-forward NN with a linear projection layer and a non-linear hidden layer learn jointly word vector representations. Following the NNLM, neural probabilistic models were presented [113, 23] where word vectors are first learnt using neural network with a single hidden layer. They are then used to train the NNLM. The benefit of this approach compared to the NNLM, is that word vectors can be obtained even without fully constructing the NNLM.

The N-gram model [26] is a simple, yet effective language representation model. N-gram models are statistical models and sentences as sequences of atomic units, each statically representing a single word. N-gram models are easy to implement for simpler tasks but their performance is limited on more complex ones. In tasks such as machine translation where there is an abundance of training data, more advanced techniques are needed to achieve good performance.

Following the N-gram's limitations, a more advanced word embedding algorithm is Word2Vec [112]. This technique is based on neural networks that are trained to extract word embeddings from a corpora of natural language data efficiently. Word2Vec constructs vector representations of words that are dispersed on the feature space in such way so words that have a similar meaning or context in humans' language have vector representations that are also close to the feature space of the embedded vectors. Compared to non neural-based embedding algorithms, Word2Vec produces significantly more accurate word representations at a much lower computational cost.

These representation models have been widely used by language models to embed software. Such example is code2vec proposed by [14]. However, as source code has the feature of being represented as a graph, such representation models are naturally unequipped to capture this kind of structure which may be sub-optimal. This issue is addressed with graph representation models, most notably Graph2Vec [117]. Graph2Vec is a neural embedding framework that learns data-driven distributed representations of arbitrary sized graphs. Graph2Vec's embeddings are learnt in an unsupervised manner and can be used for a range of downstream tasks, including graph classification apart from programming language modeling.

Embedded representations for programs are not restricted to encoding text-level source code. They can also be used to represent dynamic information, such as program executions, or execution traces. An execution trace is a recorded sequence of instructions executed for a given program, as well as any other data that have been accessed or modified throughout the execution. Execution traces capture the state of a program at different runtime states and can provide critical insight to software engineers about the behavior of a program. Our first contribution focuses in embedding execution traces to automate the classification of program runtime behaviour.

Wang et al. [151] propose an approach to embed program runtime flow. They use execution traces captured as a sequence of variable values at different program points. A program point is when a variable gets updated. Their approach uses RNNs to summarise the information in the execution trace. The execution trace embeddings are given as an input to a program repair tool. This embedding technique has several limitations - 1. Capturing execution traces as sequences of updates to every variable in the program has an extremely high overhead and will not scale to large programs. The paper does not describe how the execution traces are captured, they simply assume they have them. 2. The approach does not discuss how variables of complex data types such as structs, arrays, pointers, objects are encoded. It is not clear if the traces only capture updates to user-defined variables, or if system variables are also taken into account. 3. The evaluation uses three simple, small programs (eg. counting parentheses in a string) from students in an introductory programming course. The complexity and scale of real programs is not assessed in their experiments. Their technique for capturing and directly embedding traces as sequences of updates to every variable is infeasible in real programs.

A novel blended approach to learn program representations with execution traces is presented by Wang et al. [152]. In this work, they collect a set of symbolic traces from programs, one for each execution path. They also obtain concrete traces from program executions, one for each test input. They create a blended trace by merging one symbolic trace with all concrete traces that exercise this corresponding execution path. They develop a NN architecture called LiGer, an attention-based RNN. Their model consists of a vocabulary embedding layer, a fusion layer and a programs embedding layer. The first encodes words to embedding vectors. In the fusion layer, one RNN embeds statements and a second RNN embeds all program states of that statement within the same time step. Attention vectors are calculated and concatenated using these embeddings as input. Finally, all attention vectors are fed into an RNN sequentially and all time outputs are pooled. LiGer's embedding quality is evaluated with COSET [150] benchmark. Wang et al. [150] also extend their model into an encoder-decoder architecture and evaluate their model for the purpose of method name prediction. LiGer outperforms three relevant code embedding approaches across a set of benchmarks. However, their execution trace processing technique implies significant complexity. They only evaluate it on small functions with simple contexts. It is unlikely whether this technique can be scaled across multiple of functions of a real codebase. On the other hand, we show that our approach scales effectively over real and complex programs from different domains.

Recent works in the field of program representation includes GNN-based approaches that take advantage of the intermediate representations' graph structure. Guo et al. [63] develop a GNN-based approach to embed program binaries and fuse them with the semantics of control flow, data flow and call graphs into one model. They abstract programs into multiple graphs for multi-layer analysis. Their approach aims to enable program analysis and compilation-related tasks and it achieves an accuracy of 83% in binary similarity detection and dead store prediction.

In GraphCode2Vec, Ma et al. [107] propose a generic approach for task-agnostic, generic embeddings that capture syntax and semantics. GraphCode2Vec's embeddings are evaluated on a range of downstream tasks, including method name prediction, mutation testing classification and is compared against other state of the art embedding techniques such as CodeBERT [52] and Code2Seq, developing high performance in both generic and task-specific baselines. CodeBERT is a relevant publication to our second and third contributions and is discussed further in the next section.

Xu et al. [161] present M3V, a multi-modal, multi-view program embedding for repair operator prediction. Their proposed approach attempts to capture the context of a faulty location in a program using two models, (a) a tree-LSTM receiving text that captures the fault's signature in natural language and (b) a Graph Neural Network that encapsulates its structure in two views, data and control dependencies. Xu et al. evaluate M3V against state of the art context embedding approaches in repairing two common types of bugs in Java, null pointer exceptions and index out of bounds. They improve the prediction of repair operators in repairing null pointer exceptions by 11% to 41% using their context embeddings and 9% to 30% in index out of bounds.

3.4 Language Modeling for Program Synthesis

In the previous two sections, we survey the current literature of test oracles and their intersection with machine learning in the form of language modeling and program representation. This section discusses the use of language modeling and program representation for compiler benchmark synthesis that aims in optimizing compiler heuristics.

In their 2017 survey, Allamanis et al. [7] describe the fast-moving field of deep language models for source code [7]. Wong et al. [160] develop AutoComment that mines StackOverflow to automatically generate code comments. Allamanis et al. [6] develop Naturalize which employs techniques developed in the natural language processing domain to model coding conventions. JSNice [129] leverages probabilistic graphical models to predict program properties such Javascript identifier names. Allamanis et al. [9] use attention-based neural networks to generate summaries of source code. Nero [43] uses an encoder-decoder architecture to predict method names in stripped binaries. This technique receives an input sequence of call sites from the execution of a binary as an input and produces a predicted method name.

A recent work that has unlocked numerous applications in code generation and classification is BERT, a natural language representation model by Devlin et al.[46]. Contrary to previous language modeling tools [125, 128], BERT is designed to learn on unlabeled text data by jointly conditioning on both left and right context in all layers. This is achieved by learning to predict single words hidden behind [MASK] tokens. BERT achieves state of the art results in 11 natural language tasks and enables multiple applications of this architecture to a wide variety of difficult machine learning tasks, such as machine translation, question answering etc.

BERT has found many applications in programming languages. In CuBERT [91], Kanade et al. apply BERT over Python programs and evaluate it on the identification of typical mutation faults such as variable misuse localization, swapped operands, function-docstring mismatch and exception type checking. In CodeBERT [52], Feng et al. fine-tune BERT to perform NL-PL and PL-NL transformations. First, they synthesize functions from a set of natural language specifications. They also attempt to generate natural language documentation, provided a source code function. In both settings, they measure BERT's prediction accuracy with respect to the ground truth, i.e. programs and documentation strings.

In the field of compiler benchmarks, there is limited work coming from generative modeling techniques. In 2017, Cummins et al. [40] develop CLGEN, a deep learning generator based on LSTM [78] for OpenCL programs. CLGEN learns source code representations with program fragments collected from GITHUB's open source repositories. CLGEN is developed to enhance existing benchmark suites with synthetic ones in order to tackle the compiler benchmarks shortage. Mitigating this shortage compiler predictive models are improved by training on more datapoints with new, unseen features. They generate synthetic benchmarks to enhance existing datasets and use the enhanced datasets to train the Grewe et al.[61] heuristic model that predicts whether an OpenCL kernel should execute on the CPU or the GPU for optimal performance. Training the predictive model on training data enhanced with synthetic benchmarks improved its performance by 1.27 .

However, CLGEN poses several limitations, illustrated by Goens et al. recent case study [59]. First, after reproducing CLGEN, they show the Grewe et al. [61] predictive model performs better when trained on any of standard benchmarks or GITHUB code, while its performance gets worse when synthetic benchmarks are included in the training set. They also analyze the AST depth distribution of CLGEN's samples and compare it to standard benchmarks and code from GitHub. They prove synthetic benchmarks are significantly smaller, 3× on average. They provide similar results on the feature space coverage comparison, showing CLGEN covers a narrow space of features compared to human-written code, which already exists. This is evidence that CLGEN's synthetic benchmarks do not improve the feature diversity of training data for compilers. CLGEN generates millions of different examples but the probability of generating a kernel with syntactic errors increases exponentially as new tokens are appended. This is due to its LSTM's sequential architecture which is unable to regress to earlier parts of the generated sequence and repair them. As a result, only a tiny fraction of synthetic benchmarks compile and those that do are always small in size.

More recent techniques include SketchAdapt by Nye et al. [119]. SketchAdapt is a synthesizer that generates code from specifications. Their model learns program sketches by training a generator-synthesizer [19, 47] on input-output pairs of functions. Then they generate new program sketches that match a given I/O specification. The RNN generator samples a range of possible generic sketches that match the specification of an input/output pair. Sketches contain <HOLE> tokens, which the symbolic synthesizer fills sequentially with statements. They evaluate SketchAdapt on 2 types of input specifications: 1) A list of integers as I/O pairs and 2) a natural language description of source code. Their model performs better compared to the generator-only [19] and synthesizer-only architectures [47] separately.

SketchAdapt samples a pre-defined pool of operations (in the form of lambda functions) that may match a program's behavior. This restricts the amount of different operations it can generate to the size of its operation pool. Also, the functions they infer are required to have inputs and outputs as input specifications, which further restricts its diversity. Finally, they train and evaluate on short sketches of up to 4 operations. We are inspired to use the <HOLE> token as a means to generate statements within the middle of an existing function. However, we do not face SketchAdapt's restrictions.

Bruen et al. [44], propose a Tree2Tree approach for source code generation using Variational Autoencoders. They use GloVe [124] to embed representations of programs' AST nodes. They encode and decode AST representations using Tree-LSTMs as defined by Tai et al. [140]. They learn latent code representations by minimizing the reconstruction loss of the variational autoencoder. Bruen et al. train their model on two million C++ compiling functions gathered from coding competitions. They evaluate their Tree2Tree model against a simple VAE with a simple LSTM encoder-decoder architecture (Seq2Seq) and measure BLEU [122] and compilation rate scores between these two architectures. They also experiment with using both architectures as generative models. In this case, they sample 1000 random latent vector representations and feed them to the decoder to collect a new sample. Their Seq2Seq model achieves a compilation rate of up to 67% with greedy search, however the authors argue this happens because the model greedily selects always the most probable labels, leading to repetitive samples that compile. When sampling with temperature, their Seq2Seq model achieves a compilation rate of 40%. Their Tree2Tree architecture is able to generate a wider variety of unique samples, but only achieves a compilation rate of up to 22%, which translates to 200 functions out of these 1000 random latent vectors. This shows bad generalization on the trained dataset which accounts for 2 million C++ functions.

Researchers have recently began to explore the field of infilling language models for more large scale tasks. Infilling is defined as the bi-directional synthesis process that produces valid tokens by observing the left and right context of an incomplete input sentence. Li et al. [102] develop AlphaCode, a coding problem-solving tool based on deep language modeling and reinforcement learning. AlphaCode uses deep natural language reasoning to translate a coding problem description into a valid solution. Among 5,000 participants, it achieves a ranking of top 54.3%. Fried et al. [54] develop InCoder, a Transformer-based language model that is trained on 28 programming languages to perform left-to-right generation or bi-directional generation (infilling) to provide executable functions. Guo et al. [62] develop GRAMMFORMER, which guides sketch generation by the programming language grammar. This model places “holes” where the model is uncertain during generation, which will be filled at a later edit step.

There have also been code generating approaches coming from the field of program repair / program reconstruction from input/output specifications. Gupta et al. [64] develop a program generator - program repair framework named SED. SED is a two-stage code generator. First, a synthesizer receives the input/output specifications that must be met and generates a set of program candidates that are likely to satisfy them. Second, a neural debugger evaluates each candidate program and performs program repair to reform generated candidates into a function that will match the ground truth. Their model is trained and tested on Karel, an educational programming language. Gupta et al. evaluate the performance of three different synthesizer architectures with multiple configurations and measure (a) how well do generated programs generalize across the test cases they are expected to pass and (b) the accuracy of their debugger to repair synthesized programs across a different count of mutations, 1 to 5. SED is a meaningful research work in the intersection between program generation and program repair. However, Karel is an unrealistic language and SED's generative performance on a large scale, complex programming is not evaluated.

Faustino et al. develop Anghabench [42], a collection of real world C programs mined from GITHUB. They argue generative models cannot be easily employed in general purpose compilers because benchmarks usually target very speci c aspects of target hardware or programming language. In their approach, they mine C code from open-source repositories to tackle the benchmark shortage [40, 155]. The main challenge of their work is to automatically compile collected code. To do so, they use Psyche-C [110] type inference engine for C to apply type reconstruction and resolve missing dependencies. Structs, unions and other custom data types are all omitted or re-declared with primitive types, if possible. They collect around 1.5 million C compiling functions. Anghabench does not support custom data types and user-de ned functions and this is a serious limitation which alters original programs' semantics and leads to more simplistic data type relations. Furthermore, their benchmarks are compiling, but cannot be executed.

Chapter 4

Supervised learning over test executions as a test oracle

4.1 Introduction

The research contributions presented in this Chapter are the design and implementation of a NN-based test oracle for automatic program runtime correctness classi cation. This Chapter describes the underlying approach, the implementation and the evaluation of this technique using fteen industry-standard codebases.

We explore supervised machine learning to infer a test oracle from labelled execution traces of a given system. In particular, we use neural networks (NNs), well suited to learning complex functions and classifying patterns, to design the test oracles. Our technique is widely applicable and easy to use, as it only requires execution traces gathered from running tests through the program under test (PUT) to design the oracle. This is shown in Figure 4.1 where a small fraction of the gathered execution traces labelled with pass/fail (shown in light gray) is used to train the NN model which is then used to automatically classify the remaining unseen execution traces (colored dark gray).

Previous work exploring the use of NNs for test oracles has been in a restricted context – applied to very small programs with primitive data types, and only considering their inputs and outputs [148, 87]. Information in execution traces which we believe is useful for test oracles has not been considered by existing NN-based approaches. Other bodies of work in program analysis have used NNs to predict method or variable names and detecting name-based bug patterns [12, 127] relying on static program information, namely, embeddings of the Abstract Syntax Tree (AST) or source code. Our proposed approach is the rst attempt at using dynamic execution trace information in NN models for classifying test executions

Figure 4.1: Key idea in our approach.

Our approach for inferring a test oracle has the following steps,

1. Instrument a program to gather execution traces as sequences of method invocations.

2. Label a small fraction of the traces with their classi cation decision.

3. Design a NN component that embeds the execution traces to xed length vectors.

4. Design a NN component that uses the line-by-line trace information to classify traces as pass or fail.

5. Train a NN model that combines the above components and evaluate it on unseen execution traces for that program.

The novel contributions in this technique are in Steps 3, 4 and 5. Execution traces from a program vary widely in their length and information. We propose a technique to encode and summarise the information in a trace to a xed length vector that can be handled by a NN. We then design and train a NN to serve as a test oracle.

Labelled execution traces. Given a PUT and a test suite, we gather execution traces corresponding to each of the test inputs in the test suite with our instrumentation framework. Effectively learning a NN classi er for a PUT that distinguishes correct from incorrect executions requires labelled data with both passing and failing examples of traces. We require a small fraction of the overall execution traces to be labelled, which is likely to be a manual process. As a result, our proposed approach for test oracle is not fully automated. We hypothesize that the time invested in labelling a small proportion of the traces is justi ed with respect to the bene t gained in automatically classifying the remaining majority of traces. In contrast, with no classi er, the developer would have had to specify expected output for all the tests, which is clearly more time consuming than the small proportion of tests we need labelled.

NN Architecture. An execution trace in our approach comprises multiple lines, with each line containing information on a method invocation. Our architecture for encoding and classifying an execution trace uses multiple components: (1) Value encoder for encoding values within the trace line to a distributed vector representation, (2) Trace encoder encoding trace lines within a variable-length trace to a single vector, and (3) Trace Classi er that accepts the trace representation and classi es the trace. The components in our architecture are made up of LSTMs, one-hot encoders, and a multi-layer perceptron. We select LSTMs to represent execution traces because they are fast to train, require signi cantly less training data than other architectures (e.g. Transformers) and show high accuracy for the task of predicting failing execution traces.

Case Studies.We evaluate our approach using 4 subject programs and tests from different application domains - a single module from Ethereum project [30], a module from Pytorch [123], one component within Microsoft SEAL encryption library [134] and a Linux stream editor [104]. One of the 4 subject programs were accompanied by both passing and failing tests that we could directly use in our experiment. The remaining three programs were only accompanied by passing tests. We treated these programs as reference programs. We then generated PUTs by arti cially seeding faults into them. We generated traces through the PUTs using the existing tests, labelling the traces as passing or failing based on comparisons with traces from the reference program. We trained a NN model for each PUT using a fraction of the labelled traces. We found our approach for designing a NN classi cation model was effective for programs from different domains. We achieved high accuracies in detecting both failing and passing traces, with an average precision of 89% and recall of 88%. Only a small fraction of the overall traces (average 15%) needed to be labelled for training the classi cation models.

In summary, the Chapter makes the following contributions:

- Given a PUT and its test inputs, we provide a framework that instruments the PUT and gathers test execution traces as sequences of method invocations.
- A NN component for encoding variable-sized execution traces into xed length vectors.
- A NN for classifying the execution traces as pass or fail.
- We provide empirical evidence that this approach yields effective test oracles for programs and tests from different application domains.

4.1.1 Extended Contributions

In addition to our initial publication [145] of the contributions presented in the previous Subsection, we published extended contributions in [146]. The extended contributions are summarised as follows,

1. Support for Java programs. Our work in [145] provided tool support in the LLVM [97] framework to instrument the intermediate representation (LLVM-IR) of programs to gather execution traces. LLVM, however, does not provide front-end support for Java programs. In this Chapter, we provide tool support to gather execution traces for Java programs using the Soot framework [147].

2. Extensive empirical evaluation.We augment the experiments in [145] with 10 additional subject programs – 9 network protocols from L7- lter [35] and 1 Java utilities library from Defects4J [90], a database of real faults for open-source Java programs. For these subject programs, we evaluate precision, recall and speci city of our approach in classifying execution traces. We also assess the size of training set needed and compare accuracies against a hierarchical clustering technique for classifying execution traces proposed by Almaghairbe et al. [11].

3. Generalisation.We conduct an initial exploration into the ambitious possibility of using a model, trained using traces from one subject program, to classify traces from other programs in the same application domain. We use FSMs (Finite State Machines) from the network protocol domain to evaluate this possibility.

We found our approach for designing a NN classi cation model was effective for all subject programs. We achieved high accuracies in detecting both failing and passing traces, with an average precision of 93% and recall of 94%. Only a small fraction of the overall traces (average 14%) needed to be labelled for training the classi cation models. We found generalisation of a classi cation model from one network protocol to others in the domain was feasible. Generalisation accuracies were not as high as accuracy achieved using separate classi cation models, around 70%, but we believe there is scope for improvement using ne tuning in the future.

4.2 Approach

Our approach for building an automated test oracle for classifying execution traces has the following steps,

Step 1: Instrument the PUT to gather traces when executing the test inputs.

Step 2: Preprocess the traces to prune unnecessary information.

Step 3: Encode the preprocessed traces into vectors that can be accepted by the neural network.

Step 4: Design a NN model that takes as input an encoded trace, and outputs a verdict of pass or fail for that trace.

Figure 4.2 illustrates the steps in our approach, with the bottom half of the gure depicting steps 3 and 4 for any given preprocessed trace from step 2. We discuss each of the steps in the rest of this Section.

Figure 4.2: Gathering traces, encoding them, and using NNs to classify them. ENCODER 1 constructs a xed vector representation per trace line. A second LSTM Encoder receives all trace line representations as a sequence and outputs a vector that summarises the execution trace. Both models are jointly trained with the MLP such that a low error is achieved in predicting the trace's label.

4.2.1 Instrument and Gather Traces

For every test input executed through the PUT, we aim to collect an execution trace as a sequence of method invocations, where we capture the name of the method being called, values and data types of parameters, return values and their types, and, nally, the name of the parent method in the call graph. We nd gathering further information, eg. updates to local variables within each method, incurs a signi cant overhead and is dif cult to scale to large programs. To gather this information we develop two different tools with support for different programming languages to make our framework widely applicable. Our rst instrumentation tool is primarily aimed at C/C++ programs and uses the middleware of LLVM [97] and instruments the intermediate representation (LLVM-IR) of programs. This allows our implementation to be language-agnostic. LLVM provides front-end support for multiple programming languages in addition to C/C++ like CUDA, Haskell, Swift, Rust among others, along with numerous libraries for optimisation and code generation.

Our second tool is aimed at Java programs and uses Soot [147] to collect execution traces. Soot is a Java optimization framework and provides libraries for users to analyze, instrument and optimize applications. We develop a pass with Soot to compile Java into bytecode and a second pass to convert bytecode to Jimple-IR. Jimple is a typed 3-address intermediate representation suitable for code transformations. Using Soot's API, we develop a second pass to instrument Jimple and finally compile into an executable. Our Soot framework is also compatible with any other programming language that can be compiled into Java bytecode, e.g. Scala.

To perform the instrumentation, we traverse the PUT, visiting each method. Every time a method invocation is identified, code is injected to trace the caller-callee pair, the arguments and the return values. At the end of the program, code is inserted to write the trace information to the output.

Each trace contains a sequence of method invocations. This sequence comprises multiple lines, each line being a tuple $(n_p; n_c; r; a)$ that represents a single method invocation within it having:

- The names of the caller (parent) n_p and callee n_c functions.
- Return values r of the call, if any.
- Arguments passed a, if any.

The order of trace lines or method invocations is the order in which the methods complete and return to the calling point.

Our LLVM instrumentation supports all variable types including primitive types (such as int, float, char, bool), composite data types (such as structs, classes, arrays) defined by a user or library, and pointers for return and argument values. Structs and classes are associated with a sequence of values for their internal fields. We instrument these data structures in a depth first fashion, until all primitive types are traced. For pointers, we monitor the values they refer to.

Our instrumentation within Soot collects all Java primitive types, strings, primitive wrapper classes, atomic wrapper classes and arrays. We also support custom classes that are defined within the scope of a subject Java translation unit. Soot allows the instrumentation of public class members only; private methods and variables are not accessed.

4.2.2 Training Set

We execute the instrumented program with each test input in the test suite to gather a set of traces. A subset of the traces is labelled and used in training the classification model. To label the traces as pass or fail, we compare actual outputs through the PUT with expected outputs provided by a reference program or the specifications. Section 4.3.1 discusses how we label traces for the subject programs in our experiment. It is worth noting that in our approach, the developer will only need to provide expected outputs for a small proportion of tests rather than the whole test suite. In the absence of expected output in tests, how will tests be labelled is a common question. Answering this question will depend on what is currently being done by the developer or organisation for classifying tests as pass or fail. Our approach will entail applying the same practice to labelling, albeit to a significantly smaller proportion of tests.

To avoid data leakage, i.e. information about the expected outcome of the program existing in the trace, in our experiment in Section 4.3, we ensure that expected output is removed from the traces. We also remove exceptions, assertions and any other information in the program or test code that may act as a test oracle. This is further discussed in Section 4.3.2.

4.2.3 Preprocessing

The execution traces gathered with our approach include information on methods declared in external libraries, called during the linking phase. To keep the length of the traces tractable and relevant, we preprocess the traces to only keep trace lines for methods that are defined within the module, and remove trace lines for declared functions that are not defined, but simply linked to later.

For method invocations within loops, a new trace line is created for each invocation of the same method within the loop. For loops with large numbers of iterations, this can lead to redundancy when the method is invoked with similar arguments and return values. We address this potential redundancy issue by applying average pooling to trace lines with identical caller-callee methods within loops. This helps summarize huge sequences of function calls with identical caller-callee methods and similar features into compressed representations to save memory and time.

4.2.4 Neural Network Model

In this step, we perform the crucial task of designing a neural network that learns to classify the pre-processed traces as passing or failing. Shape and size of the input traces vary widely, and this presents a challenge when designing a NN that accepts fixed length vectors summarizing the traces. To address this, our network comprises three components that are trained jointly and end-to-end: 1. a VALENC that encodes values (such as the values of arguments and return values) into D_V-dimensional distributed vector representations, shown within ENCODER 1 in Figure 4.3, 2. a TRENC that encodes variable-sized traces into a single D_T-dimensional vector, shown as LSTM in Figure 4.4, and finally, 3. a TRACECLASSIFIER that accepts the trace representation for state and predicts whether the trace is passing or failing. The MULTILAYER PERCEPTRON in Figure 4.2 represents the TRACECLASSIFIER. We describe each component in detail in the rest of this Section.

Figure 4.3: ENCODER 1 representing a single line in a trace as a vector containing function caller, callee names, arguments and return values.

Figure 4.4: ENCODER 2 representing a sequence of trace lines as a single vector.

Encoding Values. Values within the trace provide useful indications about classifying a trace. However, values — such as ints, structs, and floats — vary widely in shape and size. We, therefore, design models that can summarize variable-sized sequences into fixed-length representations. In the machine learning literature, we predominantly find three kinds of models that can achieve this: recurrent neural networks (RNNs), 1D convolutional neural networks (CNN) and transformers. In this work, we employ LSTMs [79] — a commonly used flavour of RNNs. Testing other models is left as future work. At a high-level RNNs are recurrent functions that accept a vector h_t of the current state and an input vector x_t and compute a new state vector $h_{t+1} = \mathrm{RNN}(x_t, h_t)$ which “summarizes” the sequence of inputs up to time t. A special initial state h_0 is used at $t = 0$.

To encode a value v, we decompose it into a sequence of primitives $v = [p_0, p_1, \ldots]$ (integers, floats, characters, etc.). Each primitive p_i is then represented as a binary vector $b_i = e(p_i)$ containing its bit representation padded to the largest primitive data type of the task. For example, if int64 is the largest primitive then all b_is have dimensionality of 64. This allows us to represent all values (integers, floats, strings, structs, pointers, etc.) as a unified sequence of binary vectors. We encode v into a D_V-dimensional vector by computing

$$\mathrm{ValEnc}(v) = \mathrm{LSTM}_V(e(p_L)_L, \mathrm{ValEnc}([p_0, p_1, \ldots, p_{L-1}])),$$

where LSTM_V is the LSTM that sequentially encodes the b_is. Note that we use the same ValEnc for encoding arguments and return values, as seen in Figure 4.3. The intuition behind this approach is that the bits of each primitive can contain valuable information. For example, the bits corresponding to the exponent range of a float can provide information about the order of magnitude of the represented number, which in turn may be able to discriminate between passing and failing traces.

Representing a Single Trace Line. Armed with a neural network component that encodes values, we can now represent a single line (n_p, n_c, r, a) of the trace. To do this, we use ValEnc to encode the arguments a and the return value r. We concatenate these representations along with one-hot representations of the caller and callee identities, as shown in Figure 4.3. Specifically, the vector encoding t_i of the ith trace line is the concatenation

$$t_i = [\mathrm{ValEnc}(a), \mathrm{ValEnc}(r), \mathrm{1Hot}(n_p), \mathrm{1Hot}(n_c)],$$

where 1Hot is a function that takes as input the names of the parent or called methods and returns a one-hot vector that uniquely encodes that method name. For methods that are rare (appear fewer than k_{min} times) in our data, 1Hot collapses them to a single special Unknown (UNK) name. This is similar to other machine learning and natural language processing models and reduces sparsity often improving generalization. The resulting vector t_i has size $2D_V + 2k$ where k is the size of each one-hot vector.

Encoding Traces. Now that we have built a neural network component that encodes single lines within a trace, we design TRENC that accepts a sequence of trace line representations $t_0 ::: t_N$ and summarizes them into a single D_T-dimensional vector as shown in Figure 4.4. We use an LSTM with a hidden size D_T, and thus

$$\text{TRENC}(t_0 ::: t_N) = LSTM_{tr}(t_N; \text{TRENC}(t_0 ::: t_{N-1})) ;$$

where $LSTM_{tr}()$ is an LSTM network that summarizes the trace line representations.

Classifying Traces. With the neural network components described so far we have managed to encode traces into fixed length vector representations. The final step is to use those computed representations to make a classification decision. We treat failing traces as the positive class and passing traces as the negative class since detecting failing runs is of more interest in testing. We compute the probability that a trace is failing as

$$P(\text{fail}) = \text{TRACECLASSIFIER}([\text{TRENC}(t_0 ::: t_N)]) ;$$

where the input of TRACECLASSIFIER is the output vector of TRENC. Our implementation of TRACECLASSIFIER is a multilayer perceptron (MLP) with sigmoid nonlinearities and a single output. The sigmoid enables the model's raw output to be viewed as a probability of the trace to represent a failing execution. It therefore helps developers measure the model's prediction certainty. It follows that $P(\text{pass}) = 1 - P(\text{fail})$.

Training and Implementation Details. We train our network end-to-end in a supervised fashion, minimizing the binary cross entropy loss. All network parameters (parameters of $LSTM_v$ and $LSTM_{tr}$ and parameters of the MLP) are initialized with random noise. For all the runs on our network we use $D_V = 128$, $D_T = 256$. The TRACECLASSIFIER is an MLP with 3 hidden layers of size 256, 128 and 64. We use the Adam optimizer [94] with a learning rate of $1e^{-5}$.

For our subject programs, we find the aforementioned feature values to be optimal for performance and training time, after having experimented with other NN architectures, varying the D_V, D_T sizes, and the hidden layers in the MLP. We explored increasing D_V to 256, 512, D_T to 512, 1024 and size of hidden layers to 512 and 1024.

To handle class imbalance in datasets, we explicitly counteract the imbalance in the loss function by down-weighting the samples within the most popular class such that samples of both class participate equally within this function.

Our implementation of the proposed approach is available at https://github.com/fivosts/Learning-over-test-executions.

4.3 Experiment

In our experiment, we evaluate the feasibility and accuracy of the NN architecture proposed in Section 4.2 to classify execution traces for 15 subject programs and their associated test suites. The selection of our subject programs is based on the build system and compiler they use. Our instrumentation tool depends on CMake and LLVM-7, therefore all case studies are required to satisfy this restriction. We also select programs with at least a few hundreds of test cases to enable the NN's training process. We investigate the following questions regarding feasibility and effectiveness:

Q1. Precision, Recall and Specificity: What is the precision, recall and specificity achieved over the subject programs?

To answer this question, we use our tool to instrument the source code to record execution traces as sequences of method invocations, arguments and return values. A small fraction of the execution traces are labelled (training set) and fed to our framework to infer a classification model. We then evaluate precision, recall and specificity achieved by the model over unseen execution traces (test set) for that program. The test set includes both passing and failing test executions. We use Monte Carlo cross-validation, creating random splits of the dataset into training and test data. We created 15 such random splits and averaged precision, recall and specificity computed over them.

Q2. Size of training set: How does size of the training set affect precision and recall of the classification model?

For each program, we vary the size of training set from 5% to 30% of the overall execution traces and observe its effect on precision and recall achieved.

Q3. Comparison against state of art: How does the precision, recall and specificity achieved by our technique compare against agglomerative hierarchical clustering, proposed by Almaghairbe et al. [11] in 2017?

We choose to compare against the hierarchical clustering work as it is the most relevant and recent in classifying execution traces, even though it is not NN-based like our approach. Traces used in their work are sequences of method invocations, similar to our approach. Other test oracle work that use NNs is not used in the comparison as they do not work over execution traces, and are limited in their applicability to programs with numerical input and output which is not the case for programs in our experiment.

Q4. Generalisation of classification model: Can a classification model inferred from execution traces of one program be used to classify test executions over other programs in the same domain?

For the network protocol domain, we evaluate the accuracy of using a classification model inferred using traces from a single protocol detection finite state machine (FSM) to classify test executions from other protocol FSMs.

4.3.1 Labelling Traces

All our subject programs are open source, and most of them were only accompanied by passing tests. This is not uncommmon as most released versions of programs are correct for the given tests. We take these correct programs to be reference implementations. To enable evaluation of our approach that distinguishes correct versus incorrect executions, we need subject programs with bugs. We, therefore, generate PUTs by automatically mutating the reference implementation using common mutation operators [86] listed below,

1. Arithmetic operator replacement applied to {+, , , =, , ++ }.
2. Logical connector replacement applied to {&&, ||, !}.
3. Bitwise operator replacement applied to {&, |, ^, , <<, >> }.
4. Assignment operator replacement applied to {+ = , = , = , = = , % = , << = , >> = , & = , | = , ^ = }.

A PUT is generated by seeding a single fault into the reference implementation at a random location using one of the above mutation operators. We used an independent open source mutation tool[1] to generate PUTs from a given reference program. Figure 4.5 shows a PUT generated by seeding a single fault into a reference program. As seen in Figure 4.5, we run each test, T_i, in the test suite, through both the reference program and PUT, and label the trace *passing* if the expected output, EO_i, from

[1] https://github.com/chao-peng/mutec

Figure 4.5: Labelling test executions by matching actual and expected behavior.

the reference matches the actual output, AO_i, from the PUT. If they do not match, the trace is labelled as *failing*. We rejected PUTs from mutations that did not result in any failing traces (outputs always match with the reference). This avoids the problem of equivalent mutants. All the PUTs in our experiment had both passing and failing traces.

4.3.2 Subject Programs

We chose subject programs from different domains to assess applicability of our approach, namely from the blockchain, deep learning, encryption and text editing domains. A description of the programs and associated tests is as follows.

1. Ethereum [30] is an open-source platform based on blockchain technology, which supports smart contracts. Within it, we evaluate our approach over the DIFFICULTY module that calculates the mining difficulty of a block, in relation to different versions (eras) of the cryptocurrency (Byzantium, Homestead, Constantinople etc.). The calculation is based on five fields of an ETHEREUM block, specified in the test input.

Tests. We use the default test inputs provided by Ethereum's master test suite for the DIFFICULTY module. We test this module for the Byzantium era of the cryptocurrency (version 3.0). The test suite contains 2254 automatically generated test inputs, using fuzzing. Each test input contains one hex field for the test input of the difficulty formula and another hex field for the expected output of the program. All the test inputs provided with the module are passing tests with the actual output equal to the expected output. As a result, we use the provided module as a reference implementation. As described in Section 4.3.1, we seed faults into the reference implementation to generate PUTs, each containing a single mutation. For the difficulty module, we generate 2 PUTs – 1. Ethereum-SE with a seeded fault in the core functionality of the difficulty module, and 2. Ethereum-CD with a fault seeded in one of the functions that is external to the core function but appears in the call graph of the module. The balance between passing and failing tests varies between the two PUTs, Ethereum-CD being perfectly balanced and Ethereum-SE being slightly imbalanced (828 failing and 1426 passing tests).

2. Pytorch [123] is an optimized tensor library for deep learning, widely used in research. In our experiment, we evaluate our model over the intrusive_ptr class, which implements a pointer type with an embedded reference count. We chose this class because it had a sizeable number of tests (other modules had < 20 published tests).

Tests. Implementation of the class is accompanied by 638 tests, all of which are passing. We, thus, use this as the reference implementation. As with Ethereum, we apply mutations to the intrusive_ptr implementation to generate a single PUT. Upon comparison with the reference, 318 of the existing tests are labelled passing through the PUT and 320 as failing.

3. Microsoft SEAL [134] is an open-source encryption library. In our experiment, we study one component within Microsoft SEAL, the Encryptor module, which is accompanied by tests. This component is responsible for performing data encryption.

Tests. The Encryptor component is accompanied by 133 tests. The provided tests were all passing tests, with matching expected and actual output. As with previous programs, we generate a PUT by mutating the original implementation. On the PUT, 11 tests fail and 122 pass.

4. Sed [104] is a Linux stream editor that performs text transformations on an input stream.

Tests. We use the fifth version of Sed available in the SIR repository [49]. This version is accompanied by 370 tests, of which 352 are passing and 18 are failing. The failing tests point to real faults in this version. Since the implementation was accompanied by both passing and failing, we used it as the PUT. We did not seed faults to generate the PUT.

5. L7-Filter [35] is a packet identifier for Linux. It uses regular expression match-

ing on the application layer data to determine what protocols are used. It works with unpredictable, non-standard and shared ports. We study the following 9 protocols, implemented as FSMs, separately in our evaluation

1. ARES - P2P lesharing
2. BGP - Border Gateway Protocol
3. BIFF - new mail noti cation
4. FINGER - User information server
5. FTP - File Transfer Protocol
6. RLOGIN - remote login
7. TEAMSPEAK - VoIP application
8. TELNET - Insecure remote login
9. WHOIS - query/response system (eg. for domain name)

Tests. For each of the network protocol FSMs, we use test suites generated by Yaneva et al.[162] that provide all-transition pair coverage. The test suites for the FSMs have both passing and failing tests.

6. commons-lang[17] is a java library from Apache Commons with utility classes for the java.lang API. This is a large codebase and contains Java classes such as OBJECT and CLASS. We gather this subject program from the Defects4J database that provides several versions of this library and a labelled set of passing and failing test cases for each version.

Tests. Defects4J contains different versions of COMMONS-LANG. Most of them have very few, or even no failing tests. These versions do not provide our model with failing examples to learn and predict, therefore we discard them. We use the 34th version of this program, which contains 559 passing and 27 failing tests. These 27 tests are caused by real bugs found in this version of the subject program, therefore we do not seed faults to generate the PUT.

Checks to avoid data leakage. We ensure no test oracle data is leaked into traces. We remove expected outputs, assertions, exceptions, test names and any other information that may act directly or indirectly as a test oracle. For example, Ethereum uses BOOST testing framework to deploy its unit tests. We remove expected outputs and assertions in the test code that compare actual with the expected output e.g. BOOST_CHECK_EQUAL.

For PUTs generated by seeding faults into the reference implementation, we only use one PUT for each reference implementation except in the case of Ethereum where we generated two PUTs, since faults were seeded in different files. Generating more PUTs for each reference implementation would be easy to do. However, we found our results across PUTs for a given reference program only varied slightly. As a result, we only report results over one to two PUTs for each reference implementation.

4.3.3 Performance Measurement

For each PUT, we evaluate performance of the classification model over unseen execution traces. As mentioned in Section 4.2.4, we use positive labels for failing traces and negative labels for passing. We measure

1. *Precision* as the ratio of number of traces correctly classified as "fail" (TP) to the total number of traces labelled as "fail" by the model (TP/(TP + FP)).
2. *Recall* as the ratio of failing traces that were correctly identified (TP/(TP + FN)).
3. *Specificity* or true negative rate (TNR) as the ratio of passing traces that were correctly identified (TN/(TN + FP)).

TP, FP, TN, FN represent true positive, false positive, true negative and false negative, respectively.

4.3.4 Hierarchical Clustering

In research question 3 in our experiment, we compare the classification accuracy of our approach against agglomerative hierarchical clustering proposed by Almaghairbe et al. [11]. Their technique also considers execution traces as sequences of method calls, but only encoding callee names. Caller names, return values and arguments are discarded. We attempted to add the discarded information, but found the technique was unable to scale to large number of traces due to both memory limitations and a time complexity of $O(n^3)$ where n is the number of traces. For setting clustering parameters for each subject program, we evaluate different types of linkage (SINGLE, AVERAGE, COMPLETE) and a range of different cluster counts (as a percentage of the total number of tests): 1, 5, 10, 20 and 25%. We use Euclidean distance as the distance measure for clustering. For each program, we report the best clustering results achieved over all parameter settings.

4.3.5 Results

In this Section, we present and discuss our results in the context of the research questions presented in Section 4.3.

4.3.6 Q1. Precision, Recall and Speci city

Table 4.1 shows the precision, recall and speci city achieved by the classi cation models in our approach for the different PUTs. Results with the hierarchical clustering approach by Almaghairbe et al. [11] are also presented in Table 4.1 for comparison, but this is discussed in Q3 in Section 4.3.8. The column showing % of traces used in training varies across programs, we show the lowest percentage that is needed to achieve near maximal precision and recall.

The classi cation models for all 15 PUTs achieve more than 71% precision and 86% recall, with an average of 93% and 94%, respectively. Our technique works particularly well for Pytorch, Sed and all networking protocols, achieving >= 94%. This implies that the number of false positives in the classi cation is very low and a large majority of the failing traces are correctly identi ed.

The classi cation models for all PUTs also achieve high speci city (>= 79%, average 96%). This implies that the NN models are able to learn runtime patterns that distinguish not only failing executions, but also passing executions with a high degree of accuracy. These results are unprecedented as we are not aware of any technique in the literature that can classify both passing and failing executions at this level of accuracy.

Analysis. To understand the results in Table 4.1, for each of the PUTs, we inspected and compared passing and failing traces using a combination of longest common subsequence, syntactic diffs, and manual inspection. We also perform ablation - systematically removing information (one parameter at a time) from the traces, training new classi cation models with the modi ed traces and observing their effect on precision, recall and speci city (TNR). In our experiments, we systematically remove the following parameters from the original traces – function call names, arguments,

PUT	Lines of Code	% Traces for training	Total # Traces	Our Approach			Hierarchical Clustering [11]		
				Precision	Recall	TNR	Precision	Recall	TNR
Ethereum-CD	55927	15	2254	0.80	0.82	0.79	1.0	0.49	1.0
Ethereum-SE	55927	15	2254	0.99	0.82	0.86	1.0	0.25	1.0
Pytorch	21090	10	638	0.99	0.98	0.99	0.48	1.0	0.16
SEAL Encryptor	25967	30	132	0.75	0.86	0.98	0.16	0.36	0.83
Sed	4492	10	370	0.94	0.94	0.99	0.35	0.63	0.86
commons-lang	49028	40	586	0.71	0.94	0.98	0.07	0.96	0.4
Ares protocol	1261	3	16066	0.97	0.98	0.97	0.94	0.24	0.0
BGP protocol	1025	5	16009	0.99	0.99	0.99	0.18	0.01	0.98
Biff protocol	627	15	1958	0.97	0.99	0.99	0.43	0.22	0.72
Finger protocol	791	10	2775	0.99	0.99	0.99	0.53	0.13	0.92
FTP protocol	995	10	9677	0.99	0.99	0.98	0.07	0.001	0.98
Rlogin protocol	955	10	4121	0.97	0.96	0.99	1.0	0.04	1.0
Teamspeak protocol	3284	10	1945	0.95	0.99	0.96	1.0	0.11	1.0
Telnet protocol	1019	10	319	0.98	0.96	0.95	0.29	0.02	0.87
Whois protocol	784	9	4412	0.98	0.99	0.99	0.49	0.03	0.98

Table 4.1: Precision, Recall and True Negative rate (TNR) using our approach and hierarchical clustering.

and return values. Tables 4.2 and 4.3 show the results from our ablation study. We discuss results for each of the programs in the following paragraphs.

Over SEAL Encryptor, our approach achieves 75% precision, 86% recall and 98% speci city when trained with 30% of the traces. Encryptor requires a higher fraction of traces for training when compared to other PUTs, as the number of failing traces is very small (= 11), unlike other programs. Although we handle imbalance in datasets by weighting samples in the loss function, the NN still needs some representatives of the failing class during training. Using 10% of the traces in training, will only provide one example of failing trace (10% of 11) which is not enough for the NN model to learn to distinguish failing versus passing behaviour. Training using 30% of the traces includes 3 failing traces which allows the NN to achieve 75% precision. High precision with only 3 failing traces is because all the failing traces for this program have the same call sequence, which is suf ciently different from passing traces. Passing traces do not all have the same sequence. However, due to the availability of a larger set of passing traces (training with 30% is 40 passing traces), the NN is able to identify the different method call patterns in passing traces accurately (98% speci city). The ablation study in Tables 4.2 and 4.3 shows that all the parameters contribute to model performance as removing them has a detrimental effect.

PUT	Omitted Info.	P	R	TNR
Ethereum-CD	Function names	0.63	0.64	0.62
	Return values	0.68	0.87	0.60
	Arguments	0.54	0.78	0.35
Ethereum-SE	Function names	0.96	0.84	0.35
	Return values	0.99	0.97	0.93
	Arguments	0.96	0.84	0.33
Pytorch	Function names	0.99	1.0	1.0
	Return values	0.99	0.99	0.99
	Arguments	0.51	0.99	0.04
Seal Encryptor	Function names	0.53	0.87	0.92
	Return values	0.46	0.99	0.90
	Arguments	0.28	0.88	0.76
Sed	Function names	0.19	0.72	0.24
	Return values	0.48	0.52	0.85
	Arguments	0.30	0.40	0.73
commons-lang	Function names	0.58	1.00	0.97
	Return values	0.74	0.88	0.99
	Arguments	0.78	0.95	0.99

Table 4.2: Precision (P), Recall (R) and Speci city (TNR) for each PUT omitting certain trace information.

PUT	Omitted Info.	P	R	TNR
Finger	Function names	0.99	0.95	0.99
	Return values	0.98	0.97	0.99
	Arguments	0.52	0.19	0.88
Telnet	Function names	0.93	1.0	0.76
	Return values	0.82	1.0	0.25
	Arguments	0.76	1.0	0.00
Ares	Function names	0.96	0.98	0.95
	Return values	0.95	0.99	0.75
	Arguments	0.93	0.96	0.68
BGP	Function names	0.99	0.98	0.98
	Return values	0.98	0.99	0.98
	Arguments	0.97	0.97	0.97
Biff	Function names	0.58	0.84	0.41
	Return values	0.56	0.92	0.35
	Arguments	0.51	0.64	0.40
FTP	Function names	0.99	0.99	0.98
	Return values	0.97	0.97	0.98
	Arguments	0.88	0.93	0.84
Rlogin	Function names	0.95	0.96	0.96
	Return values	1.0	0.92	1.0
	Arguments	0.85	0.91	0.94
Teamspeak	Function names	0.91	0.97	0.91
	Return values	0.94	0.98	0.94
	Arguments	0.77	0.86	0.77
Whois	Function names	0.96	0.96	0.96
	Return values	0.96	0.96	0.96
	Arguments	0.72	0.75	0.73

Table 4.3: Precision (P), Recall (R) and Speci city (TNR) for each PUT omitting certain trace information.

For PyTorch, we achieve 99% precision, 98% recall and 99% specificity when trained with 10% of the traces. The dataset for PyTorch PUT is balanced (318 passing and 320 failing). 10% of the traces during training provides sufficient examples from both passing and failing classes for the NN to learn to distinguish them. We find the reason for the superior performance of our model over PyTorch is because all failing traces have significantly fewer trace lines than passing traces. The consistent difference in length of traces between the two classes allows the NN to easily distinguish them. The ablation study in Tables 4.2 and 4.3 show arguments in traces matter for model performance, while method names and return values are irrelevant.

With Sed, our model achieves 94% precision and recall, and 99% specificity using 10% of the traces in training. The dataset for Sed is unbalanced, with only 18 failing and 352 passing. 10% of the traces in training uses 2 failing tests and 35 passing tests. Given the extremely small sample of failing tests, it is surprising that the model classifies and identifies failing traces with such high precision and recall. To understand this, we examined both the passing and failing trace lines. We find the length of passing and failing traces is similar. All failing traces, however, have a call to a function, getChar, towards the end of the trace. This function call is absent in passing traces. We believe associating this function call to failing traces may have helped the performance of the NN. The ablation study in Tables 4.2 and 4.3 show all the parameters considered in our traces are important for model performance.

For Ethereum-CD, our model achieves 80% precision, 82% recall and 79% specificity when trained with 15% of the traces - 169 passing and 169 failing. Ethereum-CD was generated from the reference implementation using an arithmetic operator mutation in a function deeply embedded in the call graph for the difficulty module. Differences between failing and passing traces are not apparent, and analysing longest common subsequence, syntactic diff and manual inspections did not reveal any characteristic feature for failing or passing traces. We believe the model performance of around 80% precision, recall and specificity is due to the similarity between passing and failing traces and the esoteric nature of the mutation. Ablation study for this program reveals that all features in the traces slightly impact model performance.

For COMMONS-LANG, our model achieves 71% precision, 94% recall and 98% specificity using 40% of the traces. This subject program only contains 27 failing executions versus 559 passing executions. There is a stark imbalance between passing and failing traces for this program which impacts the precision achieved. We also observe failing execution traces consist of multiple calls to a string conversion function, toString towards the final parts of the sequence. We find this is can serve as a distinguishing feature between passing and failing executions. It is worth noting that our classifier's performance significantly drops when removing function names in the ablation study and it may be because the toString function is no longer visible. In contrast, removing arguments or return values does not affect performance visibly.

For Ethereum-SE, our model achieves 99% precision, 82% recall and 86% specificity with 15% traces in training - 214 failing and 124 passing. Unlike Ethereum-CD, mutation to generate Ethereum-SE was in the core functionality. Failing traces when compared to passing traces had differences towards the end of the trace which is easily distinguished by the NN. Curiously, removing return values in the ablation study, increases recall and specificity. This may be because the model was previously overfitting to return values in traces which may not have been relevant to the classification.

For L7-Filter networking protocols, all programs have enough test inputs to help our model learn program features with a small percentage of execution traces. Especially for ARES protocol with 16066 test inputs, our model can achieve 97% precision, 98% recall and 97% specificity labelling only 3% of the total traces for training. For BGP protocol, we train on 5% of the total traces and achieve 99% precision, 99% recall and 99% specificity. In all networking protocols, failing traces correspond to executions that lead to non-accepted states of the protocol's finite state machine. We observe that the sequence of function invocations is similar in both passing and failing executions. However, state information in return values and arguments is critical in order to determine correctness. The ablation study supports this argument, as removing function names in any networking protocol has no effect in the classifier's performance. On the other hand, in all protocols except for BGP, removing arguments dramatically decreases the model's precision, recall and specificity. Removing return values leads to a slight performance reduction. In RFB and TELNET, return values seem to be as important as arguments for our model's accuracy.

Summary. Overall, we find NN models for all our PUTs perform well as a test oracle, achieving an average of 93% precision, 94% recall and 96% specificity. The NN models perform exceptionally well for programs whose traces have characteristic distinguishing features between passing and failing executions, such as differences in trace lengths or presence of certain function call patterns. In the absence of such features, NNs can still do well if it has enough training samples, as in Ethereum-CD. We also find our approach can cope effectively with unbalanced datasets – four of the fifteen programs in our experiment have unbalanced passing and failing traces. Even though we attempt to explain what patterns lead to our model to succeed or fail, it is not always possible to explain the relationship between the input features and the predicted outcome. Explainability in neural networks is still an open problem.

4.3.7 Q2. Size of training set

Figures 4.6 and 4.7 shows precision and recall achieved by our approach with different training set sizes. The fraction of traces needed in training to achieve near maximal performance was 3% to 40% across the PUTs. Excluding SEAL Encryptor and commons-lang, all the other programs only needed to be trained over 15% of the traces to achieve near maximal performance. Both SEAL encryptor and commons-lang had very few failing traces, requiring a larger fraction of traces to get suf cient representation of failing classes during training. As seen in the plots in Figures 4.6 and 4.7, increasing the % of traces used in training does not increase precision and recall for all PUTs. For instance, Pytorch and Sed observe a dramatic increase in precision and recall when going from 5 to 10% traces in training. Performance, however, stagnates after that point with increasing traces. With Ethereum-CD and Ethereum-SE, precision or recall becomes worse after 20% traces. This maybe because the model is over tting to the training traces.

It is also worth noting that the absolute size of our training set varies across subject programs. We nd our approach works with training sets with as few as 3 failing traces to as many as 214. The range of passing tests in training was between 31 and 169.

Figure 4.6: Precision and recall achieved by classi cation model over each PUT.

Figure 4.7: Precision and recall achieved by classi cation model over each PUT.

4.3.8 Q3. Comparison against state of art

Table 4.1 presents precision, recall, and speci city (TNR) achieved by the agglomerative hierarchical clustering proposed by Almaghairbe et al. [11] on each of the PUTs. Comparing the precision, recall and TNR of our approach versus hierarchical clustering, we nd our approach clearly outperforms the clustering approach on all but the Ethereum-CD PUT. This is because the hierarchical clustering assumption does not hold for these programs. According to this assumption, passing traces tend to be grouped in a few big clusters and failing traces are grouped into many small clusters. However, for these programs, passing traces tend to be grouped in many small clusters based on their call sequence pattern, making it hard to distinguish them from failing traces by simply comparing cluster sizes.

With Ethereum-CD, the hierarchical clustering approach achieves precision and speci city of 100% and a recall of 49%. This is achieved with complete-linkage clustering, Euclidean distance and a cluster count equal to 10% of total traces. In contrast our approach achieves a precision of 80%, recall of 82% and speci city of 79%. To enable better comparison, we plot the precision-recall curve of the NN model in Figure 4.8 for Ethereum-CD, using 15% of the traces in training.

This curve shows the precision and recall of our trained model with respect to different values of the classi cation threshold. It is clear from the plot that for the same value of recall (49%), hierarchical clutering performs marginally better than our approach - 100% versus 99%. Hierarchical clustering works well over the Ethereum-CD PUT because the traces are clustered into just one big passing cluster and one failing cluster. Lack of cluster fragmentation improved accuracy of the hierarchical clustering approach. Nevertheless, our model achieves comparable performance for such traces. In addition, our model allows trade off between precision and recall by changing the classi cation threshold which may be driven by requirements or priorities of the use case. This tradeoff is not possible with the clustering approach.

4.3.9 Q4. Generalisation

In this research question, we conduct an initial exploration into the ambitious possibility of using a model, trained using traces from one subject program, to classify traces from other programs in the same application domain. Figures 4.9 and 4.10 represent precision and recall achieved by models trained using traces from BGP protocol and WHOIS protocol, respectively, to classify traces produced by other FSMs. We nd

Figure 4.8: Precision-Recall curve for ETHEREUM-CD.

that the model trained using traces from BIFF achieves high accuracy over the ARES protocol with precision and recall close to 1.0, and reasonable precision (> 0.8) for BGP, FTP, RLOGIN, TEAMSPEAK, WHOIS protocols. Lowest precision (0.7) was observed with TELNET. Average precision achieved in classifying traces from unseen FSMs was 0.79. Recall achieved by the model is lower than precision indicating that the model missed identifying failing traces in each of these protocols. Overall, the model trained with BIFF traces was successful in identifying failing traces in other FSMs that have similar patterns to BIFF. Failing traces with differing patterns were missed. We confirmed this observation by checking results from the WHOIS model. Although precision and recall numbers are different from the BIFF model, the reasoning for classification success was the same - extent of similarity in execution patterns between FSMs. With the current approach, we find there is scope to generalise a classification model from a single FSM to multiple FSMs in the networking domain. However, achieving high accuracies with generalisation is a difficult problem and we plan to take small, definitive steps towards addressing this challenge in the future. As a next step, we will explore tuning the classification model from an individual FSM with sample traces from other FSMs to improve generalisation performance.

Figure 4.9: Biff trained model - Precision and recall for unseen fsms.

Figure 4.10: WHOIS trained model - Precision and recall for unseen fsms.

4.3.10 Threats to Validity

We see three threats to validity of our experiment based on the selection of subject programs and associated tests.

First, PUTs for 4 out of the 15 subject programs in our experiment were generated by seeding faults into a reference implementation. A reference implementation with only passing tests is not suitable for evaluating our approach. To address this, we generated a faulty implementation and ran the original tests through the PUT to gather both passing and failing traces. It is possible using real faults in place of seeded faults may lead to different results, or introduce faults that are different than those humans make. However, Andrews et al. have shown the use of seeded faults leads to conclusions similar to those obtained using real faults [16, 83]. For one of the subject programs, Sed, we did not arti cially seed faults, but instead used the existing implementation as it was accompanied by both passing and failing tests. On a real-life scenario where no real bugs are known, the developer needs to introduce seeded faults into a tested codebase to help the neural network learn the distinctive features between the passing and failing class. According to [16, 83]such mutations would lead to similar behaviour with real bugs.

Second, the number of tests that accompanied our subject programs was not very large, ranging from 132 to 16066 tests. The NN models in our experiments produced good performance with small to medium sized test suites that may be automatically or manually generated. Our approach is constrained by the amount of training data and not by the size of the test suite. As a result for programs accompanied by large test suites, the NN model will need a larger training set (fraction of traces to be used in training might still be 14%). Depending on the total size of the test suite, the percentage needed to be labelled by an expert could still be large. Nevertheless, the labelling effort (i.e. computing manually the expected value for each test input by understanding a codebase's functionality) for a fraction of the tests in our approach is still less than the current practice of labelling all the tests.

Finally, we conducted our study on subject programs from 5 different application domains which is not representative of all application domains. Different application domains may have ranging programming paradigms, therefore different patterns in errors within execution traces that may be easier or more dif cult to predict. For example, a parser program is expected to have many recursive function calls whereas a video game a complex interaction between different classes. It is not guaranteed that a neural network will learn to represent each of them with the same accuracy. Given that our approach has no domain speci c constraints, we believe it will be widely applicable.

4.4 Summary

In this Chapter, we describe a novel approach for designing a test oracle as a NN model, learning from execution traces of a given program. We have implemented an end to end framework for automating the steps in our approach

1. Gathering execution traces as sequences of method invocations.

2. Encoding variable length execution traces into a xed length vector.

3. Designing a NN model that uses the trace information to classify the trace as pass or fail.

We support Java programs in addition to C/C++. In addition, we conduct an extensive evaluation using 15 realistic PUTs and tests. We found the classification model for each PUT was effective in classifying passing and failing executions, achieving an average of 93% precision, 94% recall and 96% specificity while only training with an average 14% of the total traces. We outperform the hierarchical clustering technique proposed in recent literature by a large margin of accuracy for 14 out of the 15 PUTs, and achieved comparable performance for the other PUT. We did an initial experiment with generalising a classification model learned over one protocol FSM to classify executions over other network protocol FSMs. The results for precision and recall over other unseen FSMs was not as high as the individual FSM classification models. In the future, we plan to explore techniques that will improve the generalisation performance of the NN models.

Practical use. Our approach can be applied out of the box for classifying tests for any software that can be compiled to LLVM IR or Jimple IR. We gather execution traces for test inputs automatically, and require a small fraction of the traces to be labelled with their pass or fail outcomes (average 15% in our experiments). The remaining traces will then be classified automatically. Our approach is promising with high accuracy and has clear benefits over current industry practices where developers label all the tests. Our future work will focus on methods to improve the classification accuracy while reducing the training data requirement using techniques like transfer learning.

Replacing a PUT with the Test Oracle. Our approach learns to classify program executions as “passing” or “failing” by extracting representations from execution traces. Implicitly, this requires the test oracle to reason about a program's reference result. Given the test oracle has this information internally it could replace the PUT and act as the actual output provider, given a test input. This could happen with a different NN design, training and labelling process (e.g. regression instead of binary classification) and is out of scope for this work. Our tool does not aim to replace a reference program. Instead we aim to provide the testing expert with a tool that will assist them in labeling test executions. A NN to provide reference outputs for a PUT would be helpful in cases where compiling and executing programs would be slow and acquiring quickly an output with a slight loss of accuracy is acceptable.

Chapter 5

BenchPress: A Deep Active Benchmark Generator

5.1 Introduction

The research contributions presented in this Chapter are the design and implementation of a deep learning, unsupervised generative model for compiler benchmarks that uses active learning to search compiler feature spaces for important features. In the following Sections, the approach, implementation and evaluation of this technique are described.

We develop BenchPress [53], a BERT-based OpenCL benchmark generator [46, 139] that targets and synthesizes benchmarks in desired parts of the feature space. We use active learning to choose parts of the feature space and beam search to steer BenchPress's generated samples towards the requested features. We train BenchPress with OpenCL code samples that we collect by mining BigQuery [60] and GitHub directly using its API [58]. We support composite data types and calls to user-defined functions in our dataset and benchmark generation. BenchPress is a bidirectional generative model and learns to generate code in any part of a sequence by jointly considering left and right context. We achieve this with a new learnt token, the [HOLE], which hides a sequence from the input, whose length is unknown to BenchPress during training. BenchPress learns to fill [HOLE] by iteratively predicting an arbitrary number tokens that are likely to lead to a compiling function.

BenchPress outperforms CLgen in the task of undirected program generation from a fixed input feed, generating 10× more unique OpenCL kernels that are 5.7× longer on average, with a compilation rate of 86% compared to CLgen's 2.33%.

BenchPress strongly outperforms benchmark synthesizers CLgen, CLSmith [164, 103], and human written code from GitHub in reaching close to the features of Rodinia benchmarks, developed by compiler experts. Finally, BenchPress uses active learning, specifically query by committee [135], to search the feature space and find missing features to improve Grewe's et al. [61] CPU vs GPU heuristic. Enhancing the heuristic's dataset with BenchPress's benchmarks improves the heuristic's speedup relative to the optimal static decision by 50%, increasing it from 4% to 6%, when the maximum possible speedup for this task is 12%.

In this Chapter, we present the following contributions:

1. We are the first to develop a feature-space agnostic, steerable code generator towards desired program features.

2. We develop an automated approach to rank the feature space of downstream tasks with active learning.

3. We enable bidirectional source code generation by inserting [HOLE] tokens in any part of a sequence.

5.2 Motivation

Figure 5.1 shows a two-dimensional slice of the Grewe's et al. [61] feature space: number of computational instructions vs number of memory instructions. Figure 5.1 also shows how the OpenCL benchmarks found in the Rodinia suite map into this plane, represented as purple diamonds. We select to plot the benchmarks' distribution with respect to these two features because they are accurate in characterizing the type of a subject OpenCL workload. Having enough benchmarks to cover lots of different ratios of computational to memory instructions is desirable for predictive models that generalize well. What we find is that much of this two dimensional space is uncovered. 54 of the 58 Rodinia examples cluster in the lower left corner, the rest of the space having only four examples. Any optimization decision for programs in this area of the space would not be accurate due to lack of representative examples.

CLgen attempted to address this problem by automatically generating more training examples. However, the generated kernels lacked feature diversity and provided even poorer coverage of the feature space. Figure 5.1 represents their position in the 2D space as red dots. Almost all of them are concentrated in a corner covering a small

Figure 5.1: # Memory operations and # computational instructions for (a) Rodinia benchmarks in purple diamonds and (b) CLgen's samples in red dots. Generating samples with missing features is vital for predictive modeling's performance.

percentage of the feature space. While CLgen can generate hundreds of millions of unique kernels, almost all of them will fail to compile. As the probability of having at least one illegal token in the kernel body increases with the number of tokens, only tiny kernels are valid. In our experiments in Section 4.3.5, the longest compiling CLgen kernel had 8 lines and 102 tokens. Given the small number of tokens in valid kernels, there is a high degree of repetitiveness in the generated corpus, not only in terms of features but also in terms of structure and functionality. As a result, this approach is not well suited to augmenting the training set with diverse feature benchmarks. There is a compelling need to generate training points for uncovered regions of the feature space and we attempt to address this need with BenchPress. In the following Sections, we discuss our approach and evaluation of BenchPress, comparing it to the existing state-of-the art for feature space coverage.

5.3 Approach

We present BenchPress, a deep learning model for directed compiler benchmark generation. BenchPress is the first steerable synthesizer that can synthesize compiling functions with target features. BenchPress steers its generation with a feature space-agnostic beam search algorithm to search the space and steer BenchPress's generation towards the target features. Given a downstream task, BenchPress learns what features to target in order to improve its performance by searching the space with active learning. BenchPress's language model is based on BERT [46], which we transform into a generative model.

The key feature in BenchPress that enables bidirectional code generation is a new token, namely, the [HOLE] token. Compared to traditional sequence to sequence generative models that can only add tokens at the end of a sequence, bidirectionality is an important feature that allows BenchPress to apply edits on previously generated code and also insert tokens that fit into the general context of an existing function. We train BenchPress to learn and understand how to iteratively fill holes of unknown length that can be found in any part of an input sequence by conditioning it on the left and right context of the [HOLE]. Later, this approach enables us to use beam search and steer benchmark generation into the feature space iteratively as it can regress to previously generated benchmarks with new holes and produce newer samples with better features.

Figure 5.2 illustrates an overview of our approach. BenchPress consists of three main components:

1. Learning corpus collection and processing.
2. Source code language modeling.
3. Feature space search and benchmark generation.

We discuss each step in the following Subsections.

5.3.1 Learning Corpus

Modeling source code accurately requires large amounts of data [100] similarly to other deep learning tasks. We develop a tool to collect data from BigQuery's GitHub dataset [60]. We also use GitHub's API [58] and mine directly extra repositories that are not included in BigQuery. We choose OpenCL to train and evaluate BenchPress for several reasons. First, many performance critical workloads are still being developed in OpenCL and ensuring optimal performance is important. Second, OpenCL is relatively simple to compile and execute compared to more generic languages (e.g. C/C++) because it has limited dependencies to third-party libraries and header files.

Figure 5.2: BenchPress's high-level approach. We highlight the corpus collection and processing in green, the language modeling for source code in red and the feature space search for benchmark generation in orange.

Finally, two easily accessible baselines that we can compare against are trained for OpenCL generation. Even though we use OpenCL in this contribution, BenchPress can work without any modifications for any programming language. However, a more appropriate tokenizer for the desired language would make the process more efficient. Also, a driver that satisfies the language's library dependencies, compiles and executes the generated code has to be developed.

There are a few innovations in how we pre-process the code compared to previous works. First, we inline included header files recursively into source files to resolve type dependencies. Additionally, we automatically extract custom data types (e.g. struct, typedef) and utility functions found in the unprocessed corpus and place them into header files that are accessible throughout BenchPress's pipeline. This way, we resolve most type dependencies by retaining the functionality and semantics of the original, human-written programs. These two steps enable us to increase significantly the amount of compiling kernels we end up with in our training dataset. Second, we isolate kernels into single instances because BenchPress is trained on complete functions. From the previous steps, the type dependencies of each kernel are known and we automatically provide them to the compiler, retaining their compilability. Finally, we compile all kernels with Clang and reject those that do not compile.

Next, we re-write identifiers by randomly sampling the alphabet, eliminating spurious naming patterns in the corpus. All kernels are padded to BenchPress's sequence length and kernels that are longer than this are truncated to fit. This helps BenchPress train its later indices' positional embeddings more effectively, for which we have less training information compared to earlier indices. Finally, we derive a tokenizer by parsing the AST of all source code. We reserve tokens for all OpenCL keywords and all intrinsic OpenCL function name identifiers found in the official OpenCL specifications [121]. We analyze the dataset and tokenize by word the most common function names and custom data type identifiers that we have collected. We encode all literals and infrequently used custom types and functions character by character to avoid exploding the size of the vocabulary. We define 5 meta tokens: [START], [END], [PAD], [HOLE], [ENDHOLE]. The derived tokenizer holds in total 2,201 unique tokens.

5.3.2 Language Modeling

BenchPress is based on BERT [46], a Transformer-based model originally designed for natural language modeling. BERT is trained to predict words that have been randomly hidden by [MASK] tokens. This way BERT learns fitting words with respect to their position in a sequence and also the left and right context, i.e., the text sequence before and after the masked token to be predicted. This type of training helps BERT learn what words mean within a given context, improving downstream tasks that rely on that knowledge.

While this is a useful property, it is not enough to turn BERT into a generative model. We also want to be able to extend a kernel by inserting an arbitrary number of tokens in arbitrary positions. We could iteratively add a [MASK] token to get one extra token at a time, until we have a full statement. This would be limiting. Each time the new token would be selected based on its probability of completing forming a compilable kernel. Every intermediate kernel in the iterative process would have to be compilable or similar to a compilable, which is not a general way for augmenting kernels.

Clusters of [MASK] tokens could allow us to insert multiple tokens in each iteration. This is still unsatisfactory. The number of [MASK] tokens in the cluster biases the kind of code that will be generated: if we ask such a generator to produce five tokens, it will give us a five token statement that could be expected to close this gap, not a five token sequence that could be the start of a much longer statement. We could place the left and right context to the edges of a sequence and ll intermediate positions with [MASK] tokens. BenchPresscould predict a vocabulary or a stop token for a [MASK], allowing for arbitrary sequences. We test this con guration and sample a trained model with a xed input feed. BenchPressis unable to learn the [MASK]s' left and right context conditionally, when many [MASK]s are in a sequence, which leads to zero samples to compile or even resemble reasonable code.

What we do instead is to extend BERT's functionality with a new pair of learnt tokens, the [HOLE] and the [ENDHOLE]. [HOLE] follows the same logic with [MASK], however the number of tokens that have been hidden behind it is unknown to the model during training. The model only learns to predict the rst token of an arbitrarily long missing sequence. At inference-time, we iteratively predict the rst token of the remaining sequence and re-insert it just before the [HOLE]. This way BenchPresslearns to generate arbitrarily large code sequences within any part of a sequence.

Figure 5.3: When a [HOLE] is inserted to a kernel at a random index, it hides a random number of tokens, unknown to BenchPress. On this example, BenchPress learns to predict the rst hidden token, p.

Figure 5.3 shows how a [HOLE] is inserted into a function to create a datapoint. A random starting index and a random length are selected. The choice of index and length are only restricted by a potential overlap of the prospective hidden sequence with any of the other meta token or the maximum hole length that is de ned as a training parameter for the architecture as a percentage of each function's length. When the speci cations of a hole have been settled, the hidden sequence is discarded. Only the rst token of it is kept as the target prediction for that hole. A hole can also represent an empty sequence, i.e. hiding 0 tokens. In this case, the target prediction during training is