The Dissertation Committee for Jia Chen
certifies that this is the approved version of the following dissertation:

A Study of Program Analysis Techniques for Resource Usage Vulnerability Detection

Committee:

Isil Dillig, Supervisor

Calvin Lin, Supervisor

Vijay Chidambaram

Mohit Tiwari

A Study of Program Analysis Techniques for Resource Usage Vulnerability Detection

by

Jia Chen, B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2019

To my parents and my grandmother.

Acknowledgments

Thou thought'st to help me
and such thanks I give.
As one near death to those
that wish him live.

William Shakespeare
All's Well That Ends Well

Life has its ups and downs, of that I was very well aware. But I would never have expected that the ups and downs in my own life as a PhD student to be like such a big roller coaster. Countless times in the past seven years I had the thought that a PhD degree is, without doubt, out of reach for my humble level of intelligence and creativity. Nothing looked bright in those desperate moments.

The grim clouds that lowered upon my fate would never have been dispelled, were it not for the help from the people that I am about to mention. To this day I am still amazed by how significantly my fortune has changes for the better because of them. I sincerely thank every single person listed in this section, and the gratitude will always be part of me for the rest of my life.

Words are powerless to express my gratitude to my advisor, Isil Dillig. It is not too far off to say that Isil almost single-handedly lifted me up to the level where a PhD degree is within my reach. Her hands-on guidances on how to pick a good research topic, how to approach a research problem, and how to efficiently evaluate a research idea are invaluable assets that helped me get through the tough challenges in my own work. When I was in need of help, she never held back her time and efforts to go over my problems with me and lend me a hand in finding a good solution. She is truly one of the most caring and supportive person I have ever met.

I also owe a great debt to my other advior, Calvin Lin. Calvin introduced me to the world of research, and throughout my PhD years he has been giving me important instructions on writing, on teaching, and on my presentation skills. He is an exemplary role model for me both as a researcher and as an educator. I feel both lucky and honored to start my PhD under his supervision and I do not regret my choice.

None of my research publications would have existed without the aids of my collaborators. My deepest appreciation goes to Yu Feng. He taught me what dilligence and perseverance means with his own hard works. His passion, optimism and encouragement have helped me overcome numerous obstacles over the years. I thank Jiayi Wei for the kindness of sharing his creative ideas with me, absence of which I would not accomplish half of what I do today. I am grateful to Obsert Bastani, who offers valuable insights and technical assistances during our collaboration.

I am thankful to my fellow PhD students Ashay Rane, Akanksha Jain, Oswaldo Olivo, Curtis Dunham, Hao Wu and Zhan Shi. Discussions with them are always enlightening, and their numerous feedbacks on my talks and my paper drafts were indispensable in helping me grow intellectually. Of course, life as a researcher would be way less fun if not for the accompany of my labmates in the Utopia group — My gratitude goes to Yuepeng Wang, Xinyu Wang, Kostas Ferles, Navid Yaghmazadeh, Greg Anderson, Rong Pan, Shankara Pailoor, Jon Stephens, Jocelyn Chen, Jacob Van Geffen, and Rushi Shah.

I want to thank Jeremy Dubreil, who offered me an intership position at Facebook and mentored me for the entire summer. Thanks to his exceptional guidance as well as the support from other members in the Infer group, it was the most pleasant and satisfying summer I ever had in the US.

I thank my friends in Austin who grew up with me in the past seven years: Jinru Hua, Fei Xue, Xiang Li, Kai Hao, Liuyang Sun, Keren Wang, Bangguo Xiong and Lingyuan Gao. It was them that made my days joyful while I was not thinking about my research.

Finally, I wish to dedicate this thesis with affection to my family. Time has taught me that life is fickle, but my family's unconditional love to me stays all the same irrespective of this fickleness. It was very fortunate for me to be born where I was well educated and always supported no matter who I am or what I do. I'm proud of them, and I wish that one day the skills I learned from this graduate program could help me accomplish something that makes them be proud of me as well.

A Study of Program Analysis Techniques for Resource Usage Vulnerability Detection

Publication No. ____________________

Jia Chen, Ph.D.
The University of Texas at Austin, 2019

Supervisors: Isil Dillig
Calvin Lin

In the program analysis community, security vulnerabilities casued by defective implementations of the desired functionality have been extensively studied. As it becomes more difficult to find and exploit defective implementations, adversaries now start to turn their attentions to vulnerabilities that arise from the resource usage characteristics of a program.

Two classes of resource usage vulnerabilities frequently get exploited in recent years. One class is called algorithmic complexity vulnerability, where the vulnerable system consumes an unexpectedly large amount of resource when it receives a carefully crafted input from an adversary. The other class is called side channel vulnerability, where the resource usage of the vulnerable system is controlled by computations that involve confidential information stored in the system. By observing the varying resource usage characteristics, this class of vulnerabilities permits the attacker to deduce the secret information given sufficient knowledge of how these computations are conducted.

Automatic detection of these resource usage related vulnerabilities is challenging because (1) it is not straightforward to characterize them formally, and (2) they are less well-studied and therefore existing program analysis techniques are not immediately applicable.

In this thesis, we present solutions that address these challenges. More specifically, we discuss (1) how to leverage dynamic input fuzzing to detect algorithmic complexity vulnerabilities, and (2) how to adapt and augment existing program verification techniques to detect application level side channel vulnerabilities. We implement our ideas and demonstrate in our evaluation that our implementations not only outperform the state-of-the-art but also find developer-acknowledged issues in real-world open source projects.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software vulnerability, as defined by the *Internet Engineering Task Force* (IETF), is "a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy" [153]. Various factors may introduce vulnerabilities to a system. Traditionally, software vulnerability comes from defective implementations of the desired functionality. Exploitation techniques such as stack smashing [122], heap metadata corruption [90, 55], return oriented programming [140] and code injection [43, 66] all requires the victim application to contain one or more defects in their input-handling logic. Due to their prevalence, defensive measures against such vulnerabilities have been extensively studied in the program analysis community. Techniques such as random testing [163], runtime instrumentation [147], information flow analysis [25], and program verification [159] have been developed to either detect or mitigate the implementation defects.

However, in recent years we have seen an increase in number of other types of vulnerability that are not caused by software bugs. In this thesis, our primary concerns are the vulnerabilities that come from the *resource usage characteristics* of the application. In contrast to the aforementioned defect-based vulnerabilities, resource usage-based vulnerabilities are less well-studied in the literature. As a result, existing techniques program analysis techniques that are tailored towards detecting defect-based vulnerabilities are less effective on detecting resource usage-based vulnerabilities as a program may still contain the latter while being 100% conformant to the functional specification and having no defects at all.

Two classes of resource usage related vulnerabilities are studied in this thesis. One class is called algorithmic complexity vulnerability, where the vulnerable system consumes an unexpectedly large amount of resource when it receives a carefully crafted input from an adversary. This class of vulnerabilities permits the attacker to disrupt or deny the system's service to benign users. Many popular systems and websites have become the victims of algorithmic complexity attacks in recent years. For instance, DoS attacks could take down an entire website [44], bypass rules for access controls [111], downgrade system performance [112], and cause financial loss [179].

The other class is called resource usage side channel vulnerability, where the resource usage of the vulnerable system is controlled by computations that involve confidential information stored in the system. By observing the varying resource usage characteristics, this class of vulnerabilities permits the attacker to deduce the secret information given sufficient knowledge of how these computations are conducted. Numerous research papers and several real-world exploits have shown that such side channel attacks are both practical and harmful. For instance, side channels have been used to infer confidential data involving user accounts [80, 58], cryptographic keys [92, 33, 6], geographic locations [154], and medical data [39]. Recent work has shown that side channels can also lead to information leakage in cyber-physical systems [40].

1.2 Challenges

As mentioned in the last section, existing program analysis techniques for defect-based vulnerability detection are often inadequate when directly applied to finding resource usage related vulnerabilities. One challenge comes from the formalization of the problem. In defect-based vulnerability detection, the techniques are typically given a specification of what defective program states look like. Examples of such a specification could be "a null pointer is dereferenced" [52], "data in a dynamically allocated memory block is reused after being free" [100], or "a critical system API is invoked on a data buffer that stores untrusted and unsanitized user input" [79]. Detecting whether a defect-based vulnerability exists in a given program can mostly be reduced to the problem of deciding whether a defective program state is reachable or not, which is a well-defined criteria. Unfortunately, vulnerabilities related to resource usages are harder to specify in the same manner: There is hardly one or two "defective" program states that are responsible for the vulnerability, as usage behavior for a given program typically depends on *all* reachable program states. Hence the old way of formalizing vulnerability detection problem needs to be adapted for resource usage related vulnerabilities.

Even if one can precisely write down the formal specification of the resource usage related vulnerabilities she is interested in, detecting the violation of such specification remains a challenging problem. For algorithmic complexity vulnerabilities, one need to be able to quickly identify whether there exists a potential input that, when fed into the application, lead to a program path of excessive length. Both the number of program inputs and the number of program paths could be infinite, which makes it hard to solve the problem in a resonable amount of time. For side channel vulnerabilities, reasoning about the absense of them requires us to solve a *2-safety* verification problem [166], which is significantly different from the kind of problems traditionally studied in the verification literature. Not many researches have been dedicated to the study of 2-safety verification, and the related techinques have not been applied to the domain of vulnerability detection.

1.3 Contribution

This thesis is about the study of program analysis techinques with the application of resource usage vulnerability detection. To explore the design space and evaluate our research hypothesis, we build and evaluate three different systems in our study:

- Singularity, an algorithmic complexity vulnerability detector based on dynamic fuzzing. This will be the topic of chapter 2.

- Themis, a static analyzer that tries to verify the absence of side channel vulnerability. Section 3 covers the verifier in more detail.

- Coeus, an extension of Themis' verification algorithm to handle a wider variety of programs. We discuss the enhancement and its implications in chapter 4 .

With the presentation of these systems, we hope to address the challenged mentioned before with the following contributions:

- We provide our formulation to a broad class of interesting resource usage related vulnerabilities.

- For the reformulated problems, we show how to adapt and augment existing program analysis techinques to better fit the new problem domain.

Chapter 2

Dynamic Detection of Algorithmic Complexity Vulnerabilities

For algorithmic compelxity attacks, adversaries tend to exploit the fact that the resource consumption of the application is rarely uniform across all possible inputs. It is often the case that an algorithm exhibits a significantly different worst-case behavior (w.r.t. a computational resource) than the average-case behavior. A typical example of such an algorithm is hash table insertion, which has an average case run time complexity of $O(1)$ and a worst case complexity of $O(n)$ where n is the current size of the table (the worst case occurs when all entries collide to the same bucket). The asymptotic difference in the worst and average case behavior is what allows the attacker to effectively exhaust the computation resource of the target system without incurring exorbitant cost on his own side.

While the problem of automatically detecting availability vulnerabilities has attracted some attention from the research community in recent years, existing techniques suffer from one of two shortcomings. In particular, some techniques only apply to *specific* classes of vulnerabilities, such as those related to regular expression matching [26, 172]. On the other hand, more recent work (e.g., SlowFuzz [127]) aims to be domain-agnostic, but fails to scale to scenarios where a relatively large input is needed to cause the program's resource usage to exceed a specified threshold.

In this chapter, we propose a scalable and domain-independent dynamic program analysis technique for automatically detecting algorithmic complexity vulnerabilities.

2.1 Overview

In reality, to launch an effective algorithmic complexity attack, the adversary must be able to generate a *worst performance input*, henceforth called *WPI*s, that trigger the worst-case performance behavior of the target program. The key insight underlying our approach is that WPIs almost always follow a *specific pattern* that can be expressed as a simple program. For instance, to trigger the worst-case performance of an insertion sort algorithm, the input array must be in reverse sorted order, which can be programmatically generated by appending larger and larger numbers to an empty list. Once we obtain a set of potential WPIs of certain size, we could check whether a target program has an algorithmic complexity vulnerability by running the program on the WPIs and see if the resource usage is above a pre-defined threshold.

Based on this observation, we transform the algorithmic complexity vulnerbility detection problem to a *program synthesis* problem, where the goal is to find a *program* that expresses the common pattern shared by all WPIs. In particular, given a *target program* $\mathcal{P}$ whose resource usage we want to maximize, our algorithm synthesizes another program $\mathcal{G}$, called a *generator*, such that the outputs of $\mathcal{G}$ correspond precisely to the WPIs of $\mathcal{P}$. Since the common pattern underlying WPIs can often be represented using *small* generator programs, this approach allows us to discover WPIs very efficiently.

The problem of finding patterns that characterize WPIs corresponds to an *optimal synthesis problem*, where the goal is to synthesize a generator $\mathcal{G}$ such that the values produced by $\mathcal{G}$ maximize the target program's resource usage. Our method solves this optimal synthesis problem by performing feedback-guided optimization using genetic programming. Specifically, we represent generators using a new programming model called *Recurrent Computation Graphs (RCG)* that are (a) expressive enough to model most input patterns of interest and yet (b) restrictive enough to make the search space manageable. Given this representation, our method looks for an optimal RCG by applying genetic operators (e.g., mutation, crossover) to existing RCGs and biasing the search towards generators that maximize the target program's resource usage.

2.2 Motivating Example

We now informally describe our complexity testing technique on the simple `quickSort` example shown in Figure 2.1 as Python code. For concreteness, let us assume that generators are expressed in the simple DSL shown in Figure 2.2. Specifically, a program $\mathcal{G}$ in this language is a tuple (c, e) where c is a constant *seed* value and e is an expression that operates over lists and integers. In particular, the semantics of program $\mathcal{G}$ is an infinite sequence of

```
def quick_sort(xs):
  if(xs.length <= 1):
    return xs
  pivot = xs[xs.length/2]
  left, middle, right = []
  for x in xs:
    if(x==pivot):
      middle.append(x)
    elif(x<pivot):
      left.append(x)
    else:
      right.append(x)
  left = quick_sort(left)
  right = quick_sort(right)
  return concat(left, middle, right)
```

Figure 2.1: QuickSort with middle pivot selection

values where the i'th value in the sequence is given by $e^i(c)$, denoting i successive applications of expression e to value c. As we will see in Section 2.3, this DSL is a simple instantiation of the recurrent computation graph model that we use to express generators.

The input pattern that triggers the worst-case complexity of this `quickSort` implementation can be described using the following generator in the DSL of Figure 2.2:

$$\mathcal{G}^* = \big([0], \lambda x.\mathit{append}(\mathit{prepend}(\mathit{length}(x)+1, x), \mathit{length}(x))\big)$$

which produces the following sequence of inputs:

$$[0], [2,0,1], [4,2,0,1,3], [6,4,2,0,1,5], \ldots$$

Observe that these inputs indeed trigger the worst-case behavior of the `quickSort`

$$\begin{array}{lcl}
P & := & (C, \lambda x.LE) \\
E & := & IE \mid LE \\
C & := & \mathit{Int} \mid \mathit{List} \\
IE & := & \mathit{Int} \mid x \mid \mathit{plus}(IE, IE) \mid \mathit{minus}(IE, IE) \\
 & & \mid \mathit{times}(IE, IE) \mid \mathit{length}(LE) \\
LE & := & \mathit{List} \mid x \mid \mathit{append}(LE, E) \mid \mathit{prepend}(E, LE) \\
 & & \mid \mathit{concat}(LE, LE)
\end{array}$$

Figure 2.2: A simple DSL for expressing generators in motivating example

implementation from Figure 2.1 because (a) the smallest value in each list of the sequence is the middle element, and (b) the `quicksort` implementation Figure 2.1 chooses the middle element as its pivot.

We now explain how SINGULARITY triggers this $O(n^2)$ behavior of this `quicksort` implementation by solving an optimal program synthesis problem. Specifically, our goal is to find a generator $\mathcal{G}$ in the DSL of Figure 2.2 that maximizes the running time of `quicksort` for sufficiently large inputs. As mentioned earlier, SINGULARITY solves this optimization problem using genetic programming (GP).

Specifically, SINGULARITY starts with a population of randomly-generated programs that conform to the context-free grammar given in Figure 2.2 and evaluates the fitness of each program. Since our goal is to maximize running time, the fitness function assigns a higher score to programs that take longer. For simplicity, let us assume that we evaluate running time on some particular input size, such as arrays of length 100.

Even though it is highly unlikely that the target generator $\mathcal{G}^*$ occurs in the initial population P, it might be the case that P contains several useful, albeit suboptimal, functions such as $f_1 = \lambda x.\mathit{append}(x, \mathit{length}(x))$ and $f_2 = \lambda x.\mathit{prepend}(\mathit{length}(x), x)$. These functions are useful in the sense that the desired pattern can be obtained by mixing these functions using genetic operators.

For the next iteration, the genetic programming algorithm randomly picks "fit" generators from the previous iteration. For example, the input patterns $([0], f_1)$ and $([0], f_2)$ are likely to be selected because they have higher than average resource usage. Singularity then uses these input patterns to generate a new population by combining them using genetic operators, such as mutation and crossover. For example, we can obtain the following program f_3 from f_1 and f_2 by using the crossover operation:

$$\lambda x.\mathit{append}(\mathit{prepend}(\mathit{length}(x), x), \mathit{length}(x))$$

In particular, crossover replaces a random sub-expression from one program with another sub-expression from another program. In this case, we can obtain f_3 from f_1, f_2 by replacing the sub-expression x in f_1 with f_2. Furthermore, f_3 results in higher resource consumption compared to f_1 and f_2.

We continue the process of generating new populations and monitor both their maximal and average performance. In general, average performance will keep increasing over generations and, at some point, Singularity will generate the desired program $\mathcal{G}^*$ from $([0], f_3)$ by mutating the sub-expression *length*(x) to *length*(x) + 1. Since $([0], f^*)$ can be used to generate an input of size 100 that achieves the maximal possible resource usage, our algorithm will terminate with the desired input pattern $\mathcal{G}^*$. Observe that we can now determine the worst-case complexity of this `quicksort` implementation by measuring the running time of `quickSort` on the input values generated by $\mathcal{G}^*$ and using standard techniques to fit a curve through these data points.

2.3 Formal Description

In this section, we formally present the problem as well as out solution.

2.3.1 Problem Definition

Given a target program $\mathcal{P}$, our goal is to find an input pattern that triggers $\mathcal{P}$'s worst-case resource usage. As mentioned in Section 2.1, we represent input patterns as generator programs $\mathcal{G}$ that produce an infinite sequence of increasingly large inputs for $\mathcal{P}$.

Definition 1. (Generator) Given a program $\mathcal{P}$ with signature $\tau \to \tau'$, a generator $\mathcal{G}$ for $\mathcal{P}$ is a program with signature `unit` $\to$ `Stream`(τ). We write $\mathcal{G}_i$ to indicate the i'th element in the stream produced by $\mathcal{G}$ and require that *size*$(\mathcal{G}_i) >$ *size*$(\mathcal{G}_j)$ whenever $i > j$.

Because our goal is to maximize the resource usage of a given program, we need a metric for measuring the size of an input and its corresponding resource usage. Thus, a *problem configuration* in our setting consists of a triple $(\mathcal{P}, \Sigma, \Psi)$, where $\mathcal{P}$ is the target program with signature $\tau \to \tau'$, Σ is a metric for measuring the size of any value of type τ, and Ψ is a function of type $\tau \to \mathbb{R}$ that measures the resource usage of $\mathcal{P}$ on any input of type τ. In particular, we write $\Psi(s)$ to denote the resource usage of $\mathcal{P}$ on a concrete input s of type τ. We also use the notation $\mathcal{G}_{\leq n}$ to denote the largest element $\mathcal{G}_i$ such $\Sigma(\mathcal{G}_i) \leq n$.

To compare the resource usage of two patterns, we define the following binary relation $\succ$ on a pair of generators:

Definition 2. (**Relation** $\succ$) A generator $\mathcal{G}$ is *asymptotically better than* another generator $\mathcal{G}'$, written $\mathcal{G} \succ \mathcal{G}'$, iff the resource usage of $\mathcal{G}$ on the target program exceeds that of $\mathcal{G}'$ for *all sufficiently large sizes*:

$$\exists \hat{n}. \forall n > \hat{n}.\ \Psi(\mathcal{G}_{\leq n}) > \Psi(\mathcal{G}'_{\leq n})$$

Given a problem configuration $(\mathcal{P}, \Sigma, \Psi)$, we now formalize our goal as follows:

Definition 3. The goal of the *algorithmic complexity vulnerability testing* problem is to find an input pattern whose asymptotic resource usage on $\mathcal{P}$ is *not worse than* any other pattern. That is, we want to find a $\mathcal{G}$ such that:

$$\forall \mathcal{G}'.\ \mathcal{G}' \not\succ \mathcal{G}$$

Our problem definition implicitly assumes the existence of a programming language over which generators are defined. While we can, in principle,

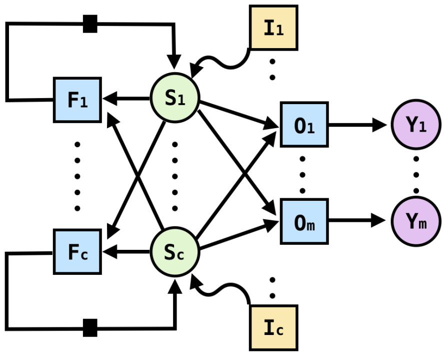


Figure 2.3: Recurrent computation graph with c internal states and m output states.

use any language for defining generators, we will restrict our attention to generators that can be expressed as *recurrent computation graphs* (RCG). Intuitively, we choose RCGs as the underlying representation because they are expressive enough to capture any input pattern of interest, but also restrictive enough to keep the search space manageable.

2.3.2 Recurrence Computation Graphs

Definition 4. (Recurrent Computation Graph) A *recurrent computation graph* $\mathcal{G}$ is a triple $(\mathcal{I}, \mathcal{F}, \mathcal{O})$ where $\mathcal{I}$ is a tuple of initialization expressions, $\mathcal{F}$ is a tuple of update expressions (where $|\mathcal{I}| = |\mathcal{F}|$), and $\mathcal{O}$ is a tuple of output expressions.

Before considering the formal semantics of RCGs, we first explain them informally: An RCG $(\mathcal{I}, \mathcal{F}, \mathcal{O})$ generates an infinite sequence of values by maintaining $|\mathcal{I}|$ internal states that are initialized using $\mathcal{I}$ and updated using $\mathcal{F}$.

$$\begin{aligned} s_i[0] &= [\![\mathcal{I}_i]\!] \\ s_i[t+1] &= [\![\mathcal{F}_i]\!][s_1 \mapsto s_1[t], \ldots, s_c \mapsto s_c[t]] \\ y_j[t] &= [\![\mathcal{O}_j]\!][s_1 \mapsto s_1[t], \ldots, s_c \mapsto s_c[t]] \\ &\quad \text{where } 1 \leq i \leq c = |\mathcal{I}| \text{ and } 1 \leq j \leq m = |\mathcal{O}| \\ [\![(\mathcal{I}, \mathcal{F}, \mathcal{O})]\!] &= \big[(y_1[t], \ldots, y_m(t)) \mid t \in [0, \infty]\big] \end{aligned}$$

Figure 2.4: RCG semantics

Since the number of internal states of the RCG may be different from the number of arguments that the target program takes, the RCG converts the $|\mathcal{I}|$ internal states to $|\mathcal{O}|$ output states using $\mathcal{O}$. As illustrated schematically in Figure 2.3, we can generate the k'th value in the infinite sequence by applying the update expression $\mathcal{F}$ exactly k times.

RCG semantics. More formally, the semantics of an RCG $(\mathcal{I}, \mathcal{F}, \mathcal{O})$ is given by the rules shown in Figure 2.4. Here, $s_i[t]$ represents the i'th internal state at time step t, and $y_i[t]$ corresponds to the i'th output value at time t. As shown in Figure 2.4, $s_i[0]$ is computed using the i'th initialization expression in $\mathcal{I}$, and $s_i[t+1]$ is obtained from $(s_1[t], \ldots, s_c[t])$ by applying the update function $\mathcal{F}_i$. Finally, $y_j[t]$ is obtained from the internal state at time t by applying the output expression $\mathcal{O}_j$ to $(s_1[t], \ldots, s_c[t])$. The semantics of the RCG is then given by the infinite sequence of values $(y_1[t], \ldots, y_m[t])$ for $t = 0, 1, 2, \ldots$ Given an RCG $\mathcal{G}$ and a value y, we say that y is in the language of $\mathcal{G}$, written $\mathcal{L}(\mathcal{G})$, if $y = (y_1[t], \ldots, y_m[t])$ for some time step t.

RCG expressions. Our definition of recurrent computation graphs intentionally does not fix the expression language over which $\mathcal{I}, \mathcal{F}, \mathcal{O}$ are specified. To maximize the flexibility of our approach, RCGs are parametrized by a set of components $\mathcal{C}$ over which the initialization, update, and output expressions are constructed. Recall that both $\mathcal{F}$ and $\mathcal{O}$ are functions, and their arguments correspond to the RCG's internal states. Hence, expressions e for $\mathcal{F}$ and $\mathcal{O}$ can be generated according to the following grammar:

$$e \coloneqq s_i \mid c \mid f(e_1, \dots, e_k)$$

where s_i represents the i'th internal state, c is a constant value, and $f \in \mathcal{C}$ is a function of arity k. Since initialization expressions are required to be constants, *init* follows a similar grammar except that we do not allow initialization expressions to refer to the RCG's internal states.

Example 2.3.1. The `quickSort` pattern from Section 2.2 can be expressed as the following 2-state RCG using the components `plus`, `append`, `prepend`, `inc`, as well as integer constants $\{0, 1, 2\}$.

$$\begin{aligned}
&\mathcal{I} = (1, [0]) \\
&\mathcal{F} = (\text{plus}(s_1, 2), \text{append}(\text{prepend}(\text{inc}(s_1), s_2), s_1)) \\
&\mathcal{O} = s_2
\end{aligned}$$

The first few iterations of the pattern's evaluation are shown below, where we use $(\triangleright), (\triangleleft), (+)$ to denote *append*, *prepend*, and *plus* respectively:

$$
\begin{aligned}
&s_1[0] = 1 \\
&s_2[0] = [0] \\
&s_1[1] = 1 + 2 = 3 \\
&s_2[1] = (\text{inc}(1) \triangleleft [0]) \triangleright 1 = [2, 0, 1] \\
&s_1[2] = 3 + 2 = 5 \\
&s_2[2] = (\text{inc}(3) \triangleleft [2, 0, 1]) \triangleright 3)) = [4, 2, 0, 1, 3]
\end{aligned}
$$

In the previous example, the output state was exactly the same as one of the internal states. However, as illustrated by the following example, this is not always the case:

Example 2.3.2. Consider the following sequence of inputs: $[\,]$, $[1, 1]$, $[1, 2, 1, 2]$, $[1, 2, 3, 1, 2, 3]$, $[1, 2, 3, 4, 1, 2, 3, 4]$, ... This input pattern can be represented using the following RCG:

$$
\begin{aligned}
\mathcal{I} &= (0, [\,]) \\
\mathcal{F} &= (\text{plus}(s_1, 1), \text{append}(s_2, s_1)) \\
\mathcal{O} &= \text{concat}(s_2, s_2)
\end{aligned}
$$

Note that the output here is obtained by concatanating two copies of the input state s_2. For this example, there is no simple way to express this pattern without distinguishing between internal and output states.

2.3.3 Algorithmic Vulnerability Testing as Discrete Optimization

We now formulate the algorithmic complexity vulnerability testing problem introduced earlier as an *optimal program synthesis problem* [30]. Towards this goal, we first introduce the concept of a *measurement model* for assigning scores to recurrent computation graphs:

Definition 5. (Ideal measurement model) Given an RCG $\mathcal{G}$, an *ideal measurement model* $\mathcal{M}$ maps $\mathcal{G}$ to a numeric value such that:

$$\forall \mathcal{G}, \mathcal{G}'. \ (\mathcal{G} \succ \mathcal{G}' \rightarrow \mathcal{M}(\mathcal{G}) > \mathcal{M}(\mathcal{G}')) \tag{2.1}$$

In other words, an ideal measurement model $\mathcal{M}$ assigns a higher score to $\mathcal{G}$ compared to $\mathcal{G}'$ if $\mathcal{G}$ induces asymptotically worse behavior of the target program compared to $\mathcal{G}'$. Using this notion, we now formulate complexity testing in terms of the following pattern optimization problem:

Definition 6. (Pattern Optimization) Given an ideal measurement model $\mathcal{M}$, the *pattern optimization* problem is to find an RCG that maximizes $\mathcal{M}$, i.e., find the solution of:

$$\operatorname*{argmax}_{\mathcal{G}} \mathcal{M}(\mathcal{G}) \tag{2.2}$$

Because RCGs correspond to programs, Definition 6 is a form of optimal program synthesis problem, where the goal is to maximize asymptotic resource usage. The following theorem states that the pattern optimization problem is equivalent to our definition of the algorithmic complexity vulnerability testing problem:

Theorem 1. *Eqn. 2.2 gives a solution to Definition 3.*

Proof: Suppose pattern $\mathcal{G}$ satisfies Eqn. 2.2. If $\mathcal{G}$ is not a solution to Definition 3, then we have some $\mathcal{G}'$ such that $\mathcal{G}' \succ \mathcal{G}$. Using Eqn. 2.1, we know that $\mathcal{M}(\mathcal{G}') > \mathcal{M}(\mathcal{G})$, which means $\mathcal{G}$ is not the solution to Eqn. 2.2 (i.e., contradiction). $\square$

Theorem 1 is useful because it allows us to turn the algorithmic complexity vulnerability testing problem into a discrete optimization problem, assuming that we have access to an ideal measurement model $\mathcal{M}$. However, due to the black-box nature of our approach, $\mathcal{M}$ is difficult to obtain in practice. In particular, the ideal measurement model requires reasoning about the asymptotic resource usage of the program on all inputs of a given shape, but this is clearly a very difficult static analysis problem. Thus, as a proxy to this idealized metric, we instead estimate the quality of an input pattern by using an *empirical* measurement model $\mathcal{M}^{\hat{n}}$. Specifically, a measurement model $\mathcal{M}^{\hat{n}}$ evaluates the quality of a generator $\mathcal{G}$ by running the input program $\mathcal{P}$ on inputs up to size $\hat{n}$. In the remainder of this chapter, we use the following empirical model as a proxy for Definition 5:

Definition 7. (Empirical Measurement Model) Our *empirical measurement model*, denoted $\mathcal{M}^{\hat{n}}$, evaluates an input pattern by returning the maximum resource usage among all inputs whose size does not exceed bound $\hat{n}$. More formally:

$$\mathcal{M}^{\hat{n}}(\mathcal{G}) = \max_{x \in \mathcal{L}(\mathcal{G}) \wedge \Sigma(x) \leq \hat{n}} \Psi(x) \tag{2.3}$$

The following theorem states the conditions under which $\mathcal{M}^{\hat{n}}$ is a good approximation of the ideal model:

Theorem 2. $\mathcal{M}^{\hat{n}}$ *is an ideal measurement model (i.e., satisfies equation 2.1) if* $\hat{n}$ *is sufficiently large and we have:*

$$\lim_{n\to\infty} \Psi(\mathcal{G}_{\leq n}) = \infty$$

Proof: We show that $\mathcal{G} \succ \mathcal{G}'$ *implies* $\mathcal{M}^{\hat{n}}(\mathcal{G}) \succ \mathcal{M}^{\hat{n}}(\mathcal{G}')$ *under the conditions stated in the theorem. Suppose* $\mathcal{G} \succ \mathcal{G}'$. *From Definition 2, this means there exists* n_1 *such that* $\forall n \geq n_1.\ \Psi(\mathcal{G}_{\leq n}) > \Psi(\mathcal{G}'_{\leq n})$. *Because we assume all patterns' resource usage increase to infinity as the input size grows, we can show that there exists some* n_2 *such that* $\forall n \geq n_2.\mathcal{M}^{n}(\mathcal{G}) = \Psi(\mathcal{G}_{\leq n})$ *and* $\mathcal{M}^{n}(\mathcal{G}') = \Psi(\mathcal{G}'_{\leq n})$ *using Eqn. 2.3. Thus, for* $\hat{n} \geq \max(n_1, n_2)$, *we have* $\mathcal{M}^{\hat{n}}(\mathcal{G}) > \mathcal{M}^{\hat{n}}(\mathcal{G}')$. □

2.3.4 Genetic Programming

We now describe a genetic programming (GP) algorithm for solving the discrete optimization problem from Section 2.3.3. We first present the top-level algorithm and then explain its subroutines.

Our pattern maximization algorithm is summarized in Algorithm 1 and follows the typical structure of genetic programming. Specifically, we start with a randomly-generated initial population of RCGs (lines 2-3) and repeatedly create a new population by combining the fittest individuals from the old population.

Algorithm 1 Pattern Maximization using GP

```
Input: $gpOps$ - the set of generic operators to use
Input: $m$ - population size
Input: $K$ - tournament size
Input: $\hat{n}$ - size bound for performance measurement.
Input: $\mu, \alpha$ - hyper-parameters used for calculating fitness
Output: the pattern with the highest fitness score so far
1: procedure FindOptimalRCG($gpOps, m, K, \hat{n}, \mu, \alpha$)
2:     $pop \leftarrow initPopulation(m)$
3:     $best \leftarrow findBest(pop)$
4:     while not $converged()$ do
5:         $pop' \leftarrow \emptyset$
6:         for $i$ from 1 to $m$ do
7:             $op \leftarrow randomPick(gpOps)$
8:             for $j$ from 1 to $op.arity$ do
9:                 $args_j \leftarrow tournament(pop, K)$
10:            $\mathcal{G} \leftarrow op(args)$
11:            $\mathcal{G}.fitness \leftarrow \mathcal{M}^{\hat{n}}(\mathcal{G}) \cdot e^{-(size(\mathcal{G})/\mu)^4} \cdot \alpha^{cost(\mathcal{G})}$
12:            if $\mathcal{G}.fitness > best.fitness$ then
13:                $best \leftarrow \mathcal{G}$
14:            $pop' \leftarrow pop' \cup \{\mathcal{G}\}$
15:        $pop \leftarrow pop'$
16:    return $best$
```

To create a new population *pop'*, we create m new RCGs by combining individuals from the existing population *pop* — this corresponds to the `for` loop at lines 6-14. A new individual $\mathcal{G}$ is created by randomly choosing a genetic operator *op* (line 7) and combining *op.arity* individuals from the current population. While there are several different techniques that can be used to select individuals from the population, our algorithm uses the so-called *deterministic tournament method* (lines 8-9). Specifically, we sample K RCGs and choose the RCG with the best fitness as the winner. [1]

Given the new RCG $\mathcal{G}$ created at line 10, we evaluate $\mathcal{G}$'s fitness (line 11) using a fitness function that we discuss in more detail later. If $\mathcal{G}$ is fitter than the previously fittest RCG, we then update *best* to be $\mathcal{G}$. The algorithm terminates with solution *best* if there has been no fitness improvement on *best* for many generations (line 4).

The genetic operators used in Algorithm 1 are described as follows:

Mutation operator. The mutation operator is used to maintain diversity from one generation to the next and prevents the algorithm from converging on a local – rather than global – optimum. It creates an RCG $\mathcal{G}'$ from an existing RCG $\mathcal{G}$ by applying modifications to a node in the abstract-syntax tree (AST) representation of $\mathcal{G}$. Specifically, we first randomly choose an initialization, update, or output expression e and then select a random node n, called the *mutation point*, in e. Our mutation operator then replaces the subtree T rooted at n with a randomly generated AST with the same type as T. Figure 2.5 illustrates this process.

Crossover operator. The crossover operator is used to combine existing members of a population into new individuals. Specifically, given RCGs $\mathcal{G}_1$

[1] K is a hyper-parameter called *tournament size* and controls the *evolution pressure* of the GP process: When K is set to 1, there is no evolution pressure and all individuals from the population, regardless of their fitness, have the same chance to be picked by the tournament method; hence, in this case, GP degenerates to random search. When K is set to the size of the whole population, only the best individual of each population can be selected to participate in the creation of new individuals.

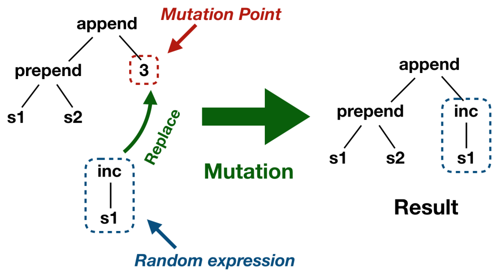


Figure 2.5: Mutation operator

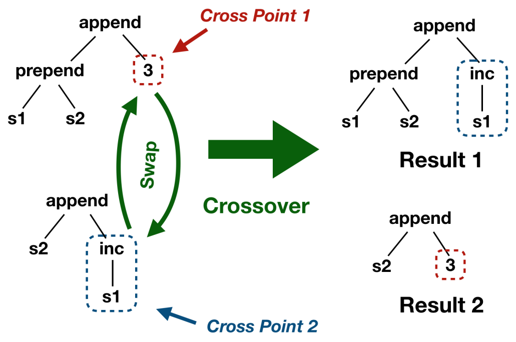


Figure 2.6: Crossover operator

and $\mathcal{G}_2$, we choose a mutation point n_1 of type τ in $\mathcal{G}_1$ as well as another mutation point n_2 of the same type τ from $\mathcal{G}_2$. We then create two new RCGs by swapping the sub-trees rooted at n_1 and n_2 and randomly pick one of the two new RCGs. The crossover operation is illustrated in Figure 2.6.

Reproduction operator. The reproduction operator is just an identity function – it simply copies the selected individual into the next generation. Reproduction is used to maintain stability between generations by preserving the fittest individuals.

ConstFold operator. The *ConstFold* operator is similar to reproduction except that it also performs light-weight constant folding on the AST. Using *ConstFold* allows continuous evolution of constants used in the RCGs without growing total AST size.

2.3.5 Fitness Function

Since our goal is to find an RCG that maximizes the target program's resource usage, the simplest implementation of the fitness function simply uses the measurement model $\mathcal{M}$. However, as standard in genetic programming, the fitness function does not have to be exactly the same as the optimization objective. We design our fitness function to have the following three properties:

1. It should be consistent with the measurement model $\mathcal{M}$, meaning that $\mathcal{G}$ is considered fitter than $\mathcal{G}'$ if $\mathcal{M}(\mathcal{G}) > \mathcal{M}(\mathcal{G}')$.
2. It should prevent individuals from evolving to unboundedly large programs by penalizing RCGs with very large AST size.
3. When two RCGs have similar size and resource usage, it should use the Occam's razor principle to prefer the simpler one.

Based on these criteria, our fitness function F is defined as follows:

$$F(\mathcal{G}) = \mathcal{M}^{\hat{n}}(\mathcal{G}) \cdot e^{-(size(\mathcal{G})/\mu)^4} \cdot \alpha^{cost(\mathcal{G})}$$

where *size* measures the total AST size of $\mathcal{G}$, and *cost* is a measure of the complexity of the RCG [2]. Both μ and α are tunable hyper-parameters. Specifically, μ is used for bloat control: If the AST size of $\mathcal{G}$ is smaller than μ, then $e^{-(size(\mathcal{G})/\mu)^4}$ is close to 1; but, when $size(\mathcal{G}) > \mu$, the fitness quickly decays to 0. The hyper-parameter α must be chosen as a value less than 1 and determines the penalty factor associated with complexity.

2.4 Implementation

We have implemented the proposed method in a tool called Singularity, which consists of approximately 6,000 lines of Scala code. In what follows, we discuss important design and implementation choices underlying Singularity.

Resource usage measurement. Recall that our problem definition and fitness evaluation function use a resource measurement function Ψ. We implement Ψ by counting the number of executed instructions rather than measuring absolute running time, as the latter strategy is too noisy due to factors such as cache warm-up, context switching, garbage collection etc.

To measure the executed number of instructions, we perform static instrumentation using the Soot framework [168] for Java programs and the LLVM framework [99] for C/C++ programs. In more detail, we initialize an integer counter when the application starts and increment it by one after each instruction. Our implementation also provides a lighter-weight version

[2]We define complexity in terms of the constants used in the RCG. Intuitively, the larger the constants used in the RCG, the higher the cost.

of this instrumentation that only increments the counter at method entry points and loop headers. In practice, we found this alternative strategy to work quite well, as it strikes a good balance between precision and overhead. Unless stated otherwise, all of our benchmarks are instrumented using this lightweight strategy.

RCG components. Recall from Section 2.3.2 that our recurrent computation graphs are parametrized by a set of components that are used to construct expressions. Our implementation comes with a library of such components for most built-in types and collections. For instance, the component library for integers include methods such as `inc`, `dec`, `plus`, `minus`, `times`, `mod` etc. Similarly, for lists, we have generic components such as `append`, `prepend`, `access`, `concat`, `length` and so forth. For graphs, we have components that represent empty graphs as well as operations that add nodes and edges. Since our framework is generic and extensible, the user can also define new components for custom data types.

Parameter tuning. As mentioned earlier, genetic programming algorithms have many tunable parameters such as population size, tournament size, threshold μ and cost penalty factor α used in the fitness function etc. Unfortunately, these parameters are often hard to configure manually due to the complex dynamics of genetic programming and the intricate interaction between different parameters. To address this problem, we developed an automatic parameter generator which samples these parameters from a joint distribution. When we run SINGULARITY multiple times on a problem, we always use different parameter sets sampled from this joint distribution. In our experience, this strategy increases the likelihood that Singularity will find the desired worst-case pattern.

2.5 Evaluation

To evaluate the usefulness of Singularity, we design a series of experiments that are intended to address the following questions:

1. Is Singularity useful for revealing the worst-case complexity of a given program?
2. How does Singularity compare with state-of-the-art testing tools that address the same problem?
3. Is Singularity useful for detecting algorithmic complexity vulnerabilities and performance bugs in real world systems?

Unless stated otherwise, experiments are conducted on an Intel Xeon(R) computer with an E5-1620 v3 CPU and 64G of memory running on Ubuntu 16.04.

2.5.1 Asymptotic Bound Analysis

In this section, we evaluate Singularity on standard algorithms taken from a widely-used algorithms textbook by Sedgewick and Wayne [146]. The goal of this experiment is to determine whether Singularity can identify the

Algorithm Name	**Best Case**	**Worst Case**	**Found Worst?**
Optimized Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	✓
Quick Sort	$\Theta(n \log n)$	$\Theta(n^2)$	✓
Optimized Quick Sort	$\Theta(n \log n)$	$\Theta(n^2)$	✓
3-way Quick Sort	$\Theta(n \log n)$	$\Theta(n^2)$	✓
Sequential Search	$\Theta(1)$	$\Theta(n)$	✓
Binary Search	$\Theta(1)$	$\Theta(\log n)$	✓
Binary Search Tree Lookup	$\Theta(1)$	$\Theta(n)$	✓
Red-Black Tree Lookup	$\Theta(1)$	$\Theta(\log n)$	✓
Separate Chaining Hash Table Lookup	$\Theta(1)$	$\Theta(n)$	✓
Linear Probing Hash Table Lookup	$\Theta(1)$	$\Theta(n)$	✓
NFA regex match	$\Theta(m + n)$	$\Theta(mn)$	✓
Booyer-Moore substring search	$\Theta(m + n)$	$\Theta(mn)$	✓
Prim Minimum Spanning Tree	$\Theta(V + E)$	$\Theta(E \log V)$	✓
Bellman-Ford shortest path	$\Theta(1)$	$\Theta(V(V + E))$	✓
Dijkstra shortest path	$\Theta(1)$	$\Theta(E \log V)$	✓
Bipartite matching (alternating path)	$\Theta(V)$	$\Theta(V(V + E))$	✓
Bipartite matching (Hopcroft-Karp)	$\Theta(V)$	$\Theta(E\sqrt{V})$	✗

Table 2.1: Evaluation on textbook algorithms.

worst-case asymptotic complexity of a wide range of algorithms that operate over different data structures.

We use the following methodology for selecting algorithms on which to evaluate SINGULARITY: First, to ensure that the benchmarks are not trivial, we only focus on algorithms whose worst-case complexity is different from their best-case behavior. Second, we only focus on algorithms whose worst-case input is known to us. Third, we only select algorithms that are related to sorting, search, graphs, and string matching. Overall, we obtain a total of 17 algorithms and evaluate SINGULARITY on the corresponding implementation from the textbook website [145]. For each algorithm, we run SINGULARITY for three hours, restarting the fuzzer with a different random seed whenever the fitness metric stays unchanged for more than 150 generations. Finally, we determine worst-case complexity by using the input patterns that maximize resource usage at $\hat{n} = 250$.

The results of this experiments are summarized in Table 2.1. The first three columns of this table provide the name of the algorithm along with its corresponding best-case and worst-case asymptotic performance, and the final column shows whether SINGULARITY is able to trigger the expected worst-case complexity. To obtain the data shown in the last column, we fit a curve through data points (x, y) where the x-value represents the theoretical worst case and the y-value corresponds to the actual performance metric of the pattern. If the fitted curve shows a linear trend with the R^2 metric being no less than 0.95, we conclude that SINGULARITY is able to generate the desired inputs that cause worst-case behavior. [3]

As we can see from this table, SINGULARITY can trigger the worst-case behavior of these algorithms in 16 of the 17 cases (i.e., 94% of our benchmarks). For the remaining benchmark (i.e., Hopcroft-Karp bipartite matching), the inputs generated by SINGULARITY trigger $O(V + E)$ complexity rather than the expected $O(E\sqrt{V})$ complexity. For this algorithm, the worst-case pattern is quite tricky to construct [110] and cannot be represented using our simple DSL for manipulating graphs.

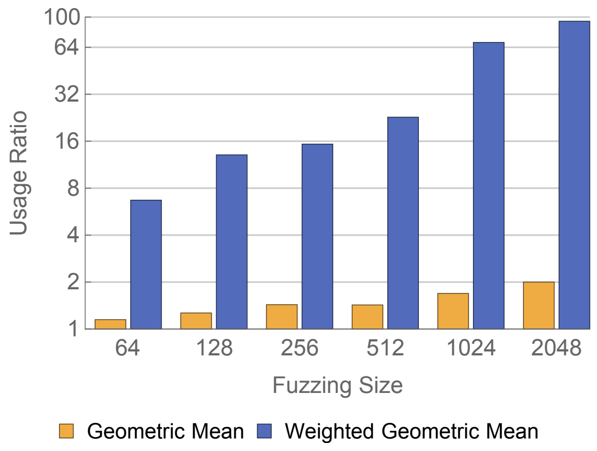


Figure 2.7: Comparison against SLOWFUZZ
The usage ratio represents the ratio between the worst case resource usage found by SINGULARITY and by SLOWFUZZ. Thus, a ratio greater than 1 indicates that SINGULARITY triggers higher resource usage. We display both the weighted and unweighted geometric mean of these ratios over the entire benchmark set.

2.5.2 Comparison Against SlowFuzz

In our next experiment, we compare Singularity against SlowFuzz [127], a state-of-the-art fuzzing tool for finding availability vulnerabilities. Similar to our approach, SlowFuzz performs resource-usage-guided evolutionary search but generates *concrete inputs*, as opposed to *input patterns*, that maximize resource usage.

We compare Singularity with SlowFuzz in terms of scalability as well as the quality of the generated inputs. We assess scalability by running each tool on increasing input sizes ranging from 64 bytes to 2K bytes. To evaluate the quality of the result, we run each tool 30 times (with a 2-hour time limit on each run) and pick the inputs I, I' from Singularity and SlowFuzz that cause the target program to run the longest time. We then compare the running time of the corresponding program on I and I'.

We perform this experiment on the same set of benchmarks reported in the SlowFuzz paper [127]. These benchmarks include several sorting algorithms from different libraries, a hash table implementation from PHP, 19 regular expression matching problems, and a zip utility from the `bzip2` application. We do not use the `bzip2` example in our evaluation since the vulnerability is triggered only when certain bits in the input file header are set; hence, this benchmark is not related to the input pattern generation problem addressed in this paper. Due to the heavy computational workload required

[3]In addition, we also manually inspect the data to further confirm there is a good linear fit.

for this experiment, we run both tools on an HPC cluster with Intel Xeon Phi 7250 CPU (68 cores at 1.4GHz) and 96G RAM running CentOS 6.3.

Since this experiment involves 27 benchmarks and 6 different configurations (one for each input size), we report the aggregate results across all benchmarks for each input size. For each benchmark b and input size n, we use the inputs I, I' generated by SINGULARITY and SLOWFUZZ to compute the following usage ratio r_b^n:

$$r_b^n = \frac{\Psi_b(I)}{\Psi_b(I')}$$

where $\Psi_b(I)$ denotes the running time (in terms of instruction count) of benchmark b on input I. Observe that $r_b^n > 1$ indicates that SINGULARITY outperforms SLOWFUZZ (i.e., SINGULARITY-generated inputs cause worse performance).

To aggregate over all benchmarks for each input size, we consider two different metrics:

- *Geometric mean:* For each input size s and benchmarks $b_1, \ldots, b_k$, we compute the geometric mean, denoted $GM(r_{b_1}^n, \ldots, r_{b_k}^n)$, of ratios $r_{b_1}^n, \ldots, r_{b_k}^n$.
- *Weighted geometric mean:* Since the usage ratio r_b^n is close to 1 for about half of the benchmarks, the geometric mean does not convey the full story. Thus, we also compute the following *weighted geometric mean*:

$$WGM(r_{b_1}^n, \ldots, r_{b_k}^n) = \exp\big(\frac{\sum_{i=1}^k \ln(r_{b_i}^n)^3}{\sum_{i=1}^k \ln(r_{b_i}^n)^2}\big)$$

Intuitively, this weighted geometric mean assigns a weight close to 0 to all ratios that are close to 1 and a higher weight to those where there is a significant performance difference. [4]

The results of this comparison are summarized in Figure 2.7, where the left bar in each size group shows the geometric mean and the right bar indicates the weighted geometric mean. We can observe two main trends based on this figure: First, SINGULARITY is able to generate inputs that cause the applications to run significantly longer within the time frame, showing that SINGULARITY is more efficient than SLOWFUZZ in terms of fuzzing efficiency. Second, the performance ratios grow as n increases, showing that SINGULARITY scales better compared to SLOWFUZZ. Hence, these results highlight the scalability advantage of pattern fuzzing over concrete input fuzzing.

2.5.3 Availability Vulnerability Detection

To demonstrate that SINGULARITY can generate inputs that exercise non-trivial algorithmic complexity vulnerabilities, we evaluate SINGULARITY on ten benchmarks from the DARPA STAC program. Specifically, we choose exactly those benchmarks that (a) exhibit an availability vulnerability, and (b) where it is possible to construct an exploit using a malicious input pattern.

In more detail, each STAC benchmark is a Java application containing

[4]Like the geometric mean, this metric is fair because if we switch SINGULARITY and SLOWFUZZ (i.e., replace $r_{b_i}^n$ with $1/r_{b_i}^n$ for all i), $WGM(\vec{r^n})$ becomes $1/WGM(\vec{r^n})$. Many other common averaging functions (e.g., arithmetic or quadratic mean) do not have this property.

Benchmark	Description	Input Type	DSL Used	Input budget	Target time	AV found?
blogger	Blogging web application	URL	string	5KB	300s	✓
graphAnalyzer	DOT to PNG/PS converter	DOT file	graph	5KB	3600s	✓
imageProcessor	Image classifier	PNG file	array	70KB	1080s	✓
textCrunchr	Text analyzer	text file	string	400KB	300s	✗
linearAlgebra	Matrix computation service	Matrix	array	15.25KB	230s	✓
airplan1	Online airline scheduler	Graph	graph	25KB	500s	✓
airplan2	Online airline scheduler	Graph	graph	25KB	500s	✓
airplan3	Online airline scheduler	Graph	graph	25KB	500s	✗
searchableBlog	Webpage search engine	Matrix	array	1KB	10s	✓
braidit1	Online multiplayer game	String	string	2KB	300s	✓

Table 2.2: Evaluation on STAC Benchmarks.

between 500 to 20,000 lines of code. Furthermore, each benchmark comes with a pre-defined input budget b and a target running time t such that the goal is to craft an attack vector that causes the running time of the application to exceed t using an input of size at most b. Table 2.2 provides more detailed information about these STAC benchmarks.

To perform this experiment, we run Singularity for a total of three hours on each benchmark. If the specified input budget b is below some threshold, we parametrize the measurement model with b; otherwise, we use a default size of $\hat{n} = 1$KB.

The results of this experiment are summarized in Table 2.2. At a high level, Singularity is able to generate the desired attack vector for eight out of these ten benchmarks. In particular, given a benchmark with input budget b and target running time t, Singularity can find inputs of size at most b that cause the application to exceed the specified time t.

To understand the limitations of Singularity, we manually investigate the *textCrunchr* and *airplan3* benchmarks for which Singularity fails to find an attack vector. For *textCrunchr*, the root cause of the problem is the empirical measurement model. In particular, Singularity evaluates the fitness of an individual based on its performance on inputs with size 1KB, but since this is much smaller than the input budget of $400KB$, our empirical measurement model does not end up being an ideal one. While we could circumvent this problem by using a much larger input size, this would significantly increase the time to evaluate the fitness of a given input pattern, thereby slowing down the fuzzing algorithm.

For the *airplan3* benchmark, the reason Singularity fails to find the desired attack vector is the three hour time limit. In particular, running the application on an input of size of 1KB takes more than 3 minutes after roughly 90 generations of the GP algorithm; thus, Singularity does not converge to the fittest input pattern within the provided time limit.

2.5.4 Performance Bug Detection

In addition to vulnerability detection, Singularity can also help with discovering *unknown* performance bugs in real-world projects. These performance bugs do not correspond to any vulnerability since they reside in general-purpose libraries that are not directly related to any security-critical applications, but finding them is still as hard as finding zero-day vulnerabilities. We run Singularity on three popular Java libraries, namely Google Guava [74], Vavr [170], and JGraphT [89]. All of these libraries have more than 1000 stars on Github and are used by more than 70 other projects on Maven Central. Hence, any performance issue in these libraries is likely to have significant real-world impact.

For each library, we manually identify APIs that are related to containers or graphs, and write driver code to invoke these APIs with data generated by SINGULARITY. We then use the input patterns generated by SINGULARITY to determine worst-case complexity by (a) generating inputs of different sizes, and (b) fitting a curve through these data points. If the complexity reported by SINGULARITY does not match the expected worst-case, we manually inspect the code to determine whether there is a performance bug.

Using this methodology, we identified five *previously unknown* performance bugs, all of which have been confirmed by the developers. In what follows, we include brief descriptions of the performance problems uncovered by SINGULARITY:

Performance bugs in Guava. SINGULARITY identified two performance bugs in the `ImmutableBiMap` and
`ImmutableSet` container classes in the Guava library. Specifically, both of these classes provide a method called `copyOf` that returns an `ImmutableBiMap` or
`ImmutableSet` that contains the same elements as the input collection. While both of these `copyOf` methods are expected to take linear time, the inputs generated by SINGULARITY cause $O(n^2)$ performance. In particular, SINGU-

LARITY triggers this worst-case behavior by causing hash collisions despite the existence of a mechanism that tries to protect against hash collisions. The inputs generated by SINGULARITY are complex enough to bypass these existing mitigation mechanisms. Both of these performance bugs have been acknowledged by the developers and already been fixed.

Performance bug in JGraphT SINGULARITY identified a serious performance bug in the JGraphT implementation of the push-relabel maximum flow algorithm [72]. While the theoretical worst-case behavior of this algorithm is $O(n^3)$, SINGULARITY is able to find inputs that trigger $O(n^5)$ running time. The developers have acknowledged this bug and are currently investigating its root cause.

Performance bug in Vavr SINGULARITY also identified two performance problems in the Vavr library that provides immutable and persistent collections. In particular, while the `addAll` and `union` methods of `LinkedHashSet` are supposed to have worst-case linear complexity, SINGULARITY found inputs that trigger quadratic behavior. The developers have acknowledged this issue and added a caveat to the corresponding JavaDocs that these methods have quadratic rather than the (expected) linear complexity.

Chapter 3

Static Detection of Side Channel Vulnerabilities

We now shift our focus from algorithmic complexity vulnerabilities to the more challenging problem of side-channel vulnerabilities. In general, the most robust defense against side-channel attacks is to eradicate the underlying vulnerabilities by ensuring that the resource usage of the program (time, space, power etc.) does not vary with respect to the secret. Unfortunately, it can be challenging to write programs in a way that follows this discipline, and side-channel vulnerabilities continue to be uncovered on a regular basis in real-world security-critical systems [33, 169, 76, 6].

Our goal in this thesis is to help programmers develop side-channel-free applications by automatically analyzing correlations between variations in resource usage and differences in security-sensitive data. In particular, given a program P and a "tolerable" resource deviation ϵ, we would like to *verify* that the resource usage of P does not vary by more than ϵ no matter what the value of the secret. Following the terminology of Goguen and Meseguer [71], we refer to this property as *ϵ-bounded non-interference*. Intuitively, a program that violates ϵ-bounded non-interference for even large values of ϵ exhibits significant secret-induced differences in resource usage.

In this chapter, we propose a program verification technique to automatically detect violations of ϵ-bounded non-interference in recursion-free programs.

3.1 Overview

The problem of verifying ϵ-bounded non-interference is challenging for at least two reasons: First, the property that we would like to verify is an instance of a so-called *2-safety property* [166] that requires reasoning about all possible interactions between *pairs* of program executions. Said differently, a *witness* to the violation of ϵ-bounded interference consists of a *pair* of program runs on two different secrets. Unlike standard safety properties that have been well-studied in verification literature and for which many automated tools exist, checking 2-safety is known to be a much harder problem. Furthermore, while checking 2-safety can in principle be reduced to standard safety via so-called *product construction* [18, 14] such a transformation either causes a blow-up in program size [14], thereby resulting in scalability problems, or yields a program that is practically very difficult to verify [18].

In this chapter, we solve these challenges by combining relatively lightweight static taint analysis with more precise *relational verification* techniques for reasoning about *k-safety* (i.e., properties that concern interactions between k program runs). Specifically, our approach first uses taint information to identify so-called *hot spots*, which are program fragments that have the potential to exhibit a secret-induced imbalance in resource usage. We then use much more precise relational reasoning techniques to automatically verify that such hot spots do not violate ϵ-bounded non-interference.

At the core of our technique is a new program logic called *Quantitative Cartesian Hoare Logic (QCHL)* for verifying the ϵ-bounded non-interference property. QCHL leverages recent advances in relational verification by building on top of *Cartesian Hoare Logic (CHL)* [161] for verifying k-safety properties. Specifically, QCHL allows us to prove triples of the form $\langle\phi\rangle\ S\ \langle\psi\rangle$, where S is a program fragment and ϕ, ψ are first-order formulas that relate the program's resource usage (e.g., execution time) between an arbitrary pair of program runs. Starting with the precondition that two runs have the same public input but different values of the secret, QCHL proof rules allow us to prove that the difference in resource usage is bounded from above by some (user-provided) constant ϵ. Similar to CHL, our QCHL logic allows effective relational verification by symbolically executing two copies of the program in lockstep. However, QCHL differs from CHL in that it reasons about the program's resource usage behavior and exploits domain-specific assumptions to improve both analysis precision and scalability. Furthermore, since the QCHL proof rules are deterministic (modulo an oracle for finding loop invariants and proving standard Hoare triples), QCHL immediately lends itself to a fully automated verification algorithm.

3.2 Motivating Example

```java
1  BigInteger modPow(BigInteger base,
2    BigInteger exponent, BigInteger modulus) {
3    BigInteger s = BigInteger.valueOf(1);
4    // BigInteger r;
5    int width = exponent.bitLength();
6    for (int i = 0; i < width; i++) {
7      s = s.multiply(s).mod(modulus);
8      if(exponent.testBit(width - i - 1))
9          s = s.multiply(base).mod(modulus);
10     //else r = s.multiply(base).mod(modulus);
11   }
12   return s;
13 }
```

Figure 3.1: Gabfeed code snippet that contains a timing side channel
A possible fix can be obtained by commenting in lines 4 and 10.

```json
{
  "epsilon":"0",
  "costModel":"time",
  "secrets": [ "<com.cyberpointllc.stac.auth.KeyExchangeServer:
                java.math.BigInteger secretKey>" ]
}
```

Figure 3.2: THEMIS configuration file for Gabfeed.

Suppose that Bob, a security analyst at a government agency, receives a Java web application called Gabfeed, which implements a web forum that allows community members to post and search messages[1]. In this context, both the user names and passwords are considered confidential and are therefore encrypted before being stored in the database. Bob's task is to vet this application and verify that it does not contain timing side-channel vulnerabilities that may compromise user name or password information. However, Gabfeed contains around 30,000 lines of application code (not including *any* libraries); hence, manually searching for a vulnerability in the application is akin to finding a needle in the haystack.

A security analyst like Bob can greatly benefit from THEMIS by using it to automatically verify the absence of side-channel vulnerabilities in the target application. To use THEMIS, Bob first identifies application-specific confidential data (in this case, secretKey) and annotates them as such in a THEMIS-specific configuration file, as shown in Figure 3.2. In the same configuration file, Bob also tells THEMIS the type of side channel to look for (in this case, timing) by specifying the `costModel` field and provides a reasonable value of ϵ, using the `epsilon` field. Here, Bob wants to be conservative and initially sets the value of ϵ to zero.

Using the information provided by Bob in the configuration file, THEMIS

[1]Gabfeed is one of the challenge problems from the DARPA STAC project. Please see http://www.darpa.mil/program/space-time-analysis-for-cybersecurity for more details about the STAC project.

first performs static taint analysis to identify methods that are dependent on confidential data. In this case, one of the methods that access confidential data is `modPow`, shown in Figure 3.1. Specifically, THEMIS determines that the second argument (`exponent`) of `modPow` is tainted and marks it as a "hot spot" that should be analyzed more precisely using relational verification techniques.

In the next phase, THEMIS uses its Quantitative Cartesian Hoare Logic (QCHL) verifier to analyze `modPow` in more detail. Specifically, the QCHL verifier considers two executions of `modPow` that have the same values of `base` and `modulus` but that differ in the value of `exponent`. In this case, the QCHL verifier fails to prove that the resource usage of any such two runs is identical and therefore issues a warning about a possible timing side channel in the `modPow` procedure.

Next, Bob wonders whether the imbalance in resource usage is large enough to be actually exploitable in practice. For this reason, he plays around with different values of the bound ϵ, gradually increasing it to larger and larger constants. In the case of timing side channels, ϵ represents the difference in the executed number of Java bytecode instructions. However, no matter what value of ϵ Bob picks, THEMIS complains about a possible timing side channel. This observation indeed makes sense because the difference in resource usage is proportional to the secret and can therefore not be bounded by a constant.

Bob now inspects the source code of `modPow` and realizes that a possible vulnerability arises due to the resource imbalance in the secret-dependent branch from line 8. To fix the vulnerability, Bob adds the code from lines 4 and 10, with the goal of ensuring that the timing behavior of the program is not dependent on `exponent`. To confirm that his fix is valid, Bob now runs THEMIS one more time and verifies that his repair eliminates the original vulnerability.

In this section, we formally present the problem as well as out solution.

3.3 Threat Model

We assume that an adversary can observe a program's total resource usage, such as timing, memory, and response size. When measuring resource usage, we further assume that any variations are caused at the *application software* level. Hence, side channels caused by the microarchitecture such as cache contention [175] and branch prediction [3] are out of the scope of this work. Physical side channels (including power and electromagnetic radiation [67]) can, in principle, be handled by our our system as long as a precise model of the corresponding resource usage is given. We assume that the attacker is not able to observe anything else about the program other than its resource usage.

One possible real-world setting in which the aforementioned assumptions hold could be that the attacker and the victim are connected through a network, and the victim runs a server or P2P software that interacts with other machines through encrypted communications. In this scenario, the attacker and the victim are physically separated; hence, the attacker cannot exploit physical side channels, such as power usage. Furthermore, the attacker does not have a co-resident process or VM running on the victim's machine, thus it is hard to passively observe or actively manipulate OS and hardware-level side channels. What the attacker can do is to either interact with the server and measure the time it takes for the server to respond, or observe the network traffic and measure request and response sizes. In our setting, we assume that data encryption has been properly implemented and the attacker cannot directly read the contents of any packet.

3.4 Side-Channels and Bounded Non-interference

In this section, we introduce the property of *ϵ-bounded non-interference*, which is the security policy that will be subsequently verified using the THEMIS system.

Let P be a program that takes a list of input values $\vec{a}$, and let $R_P(\vec{a})$ denote the resource usage of P on input $\vec{a}$. Following prior work in the literature [75, 141, 65], we assume that each input is marked as either *high* or *low*, where high inputs denote security-sensitive data and low inputs denote public data. Let $\vec{a}^h$ (resp. $\vec{a}^l$) be the sublist of the inputs that are marked as *high* (resp. *low*). Prior work in the literature [166, 53, 7] considers a program to be side-channel-free if the following condition is satisfied:

Definition 3.4.1. A program P is free of resource-related side-channel vulnerabilities if

$$\forall \vec{a_1}, \vec{a_2}.\ (\vec{a_1}^{\,l} = \vec{a_2}^{\,l} \wedge \vec{a_1}^{\,h} \neq \vec{a_2}^{\,h}) \Rightarrow R_P(\vec{a_1}) = R_P(\vec{a_2})$$

The above definition, which is a direct adaptation of the classical notion of *non-interference* [71], states that a program is free of side channels if the resource usage of the program is deterministic with respect to the public inputs. In other words, the program's resource usage does not correlate with any of its secret inputs.

3.4.1 Language

One of the key technical contributions of this chapter is a new method for verifying ϵ-bounded non-interference using QCHL, a variant of Cartesian Hoare Logic introduced in recent work for verifying k-safety [161]. QCHL proves triples of the form $\langle\phi\rangle\ S\ \langle\psi\rangle$, where S is a program fragment and ϕ, ψ are first-order formulas that relate the program's resource usage between an arbitrary pair of program runs. Starting with the precondition that the program's low inputs are the same for a pair of program runs, QCHL tries to derive a post-condition that logically implies ϵ-bounded non-interference.

We will describe our program logic, QCHL, using the simplified imperative language shown in Figure 3.3. In this language, program inputs are annotated as *high* or *low*, indicating private and public data respectively. Atomic statements include skip (i.e., a no-op), assignments of the form $x := e$, and *consume* statements, where "**consume**(e)"indicates the consumption of e units of resource. Our language also supports standard control-flow constructs, including sequential composition, if statements, and loops.

Figure 3.4 defines the cost-instrumented operational semantics of this language using judgments of the form $\Gamma \vdash S : \Gamma', r$. The meaning of this judg-

$$
\begin{array}{l}
\langle \mathit{expr} \rangle ::= \langle \mathit{const} \rangle \mid \langle \mathit{var} \rangle \mid \langle \mathit{expr} \rangle \circ \langle \mathit{expr} \rangle \\
\qquad (\circ \in \{+, -, \times, \vee, \wedge, ...\}) \\[1em]
\langle \mathit{stmt} \rangle ::= \textbf{skip} \mid \textbf{consume}(\langle \mathit{expr} \rangle) \mid \langle \mathit{var} \rangle := \langle \mathit{expr} \rangle \\[1em]
\langle \mathit{stmts} \rangle ::= \langle \mathit{stmt} \rangle \mid \langle \mathit{stmt} \rangle ; \langle \mathit{stmts} \rangle \\
\quad \mid \ \ \textbf{if } \langle \mathit{expr} \rangle \textbf{ then } \langle \mathit{stmts} \rangle \textbf{ else } \langle \mathit{stmts} \rangle \\
\quad \mid \ \ \textbf{while } \langle \mathit{expr} \rangle \textbf{ do } \langle \mathit{stmts} \rangle \\[1em]
\langle \mathit{params} \rangle ::= \langle \mathit{param} \rangle \mid \langle \mathit{param} \rangle , \langle \mathit{params} \rangle \\[1em]
\langle \mathit{param} \rangle ::= \langle \mathit{annot} \rangle \ \langle \mathit{var} \rangle \\[1em]
\langle \mathit{annot} \rangle ::= \textbf{low} \mid \textbf{high} \\[1em]
\langle \mathit{prog} \rangle ::= \lambda \langle \mathit{params} \rangle . \ \langle \mathit{stmts} \rangle
\end{array}
$$

Figure 3.3: Language used in our formalization

ment is that, assuming we execute S under environment Γ (mapping variables to values), then S consumes r units of resource and the new environment is Γ'. As shown in Figure 3.4, we use the notation $R_P(\vec{a})$ to denote the resource usage of program P on input vector $\vec{a}$. In cases where the resource usage is irrelevant, we simply omit the cost and write $\Gamma \vdash S : \Gamma'$.

3.4.2 QCHL Proof Rules

We now turn our attention to the proof rules of Quantitative Cartesian Hoare Logic (QCHL), which forms the basis of our verification methodology. Similar to CHL [161], QCHL is a relational program logic that allows proving relationships between multiple runs of the program. However, unlike CHL,

$$\frac{P = \lambda\vec{p}.S \qquad \forall p_i \in \vec{p}.\ \Gamma(p_i) = a_i \qquad \Gamma \vdash S : \Gamma', r}{R_P(\vec{a}) = r}$$

$$\frac{S = \mathbf{skip}}{\Gamma \vdash S : \Gamma, 0}$$

$$\frac{S = (x := e) \qquad \Gamma \vdash e : v \qquad \Gamma' = \Gamma[x \leftarrow v]}{\Gamma' \vdash S : \Gamma', 0}$$

$$\frac{S = \mathbf{consume}\ (e) \qquad \Gamma \vdash e : v}{\Gamma \vdash S : \Gamma, v}$$

$$\frac{S = S_1; S_2 \qquad \begin{array}{c}\Gamma \vdash S_1 : \Gamma_1, r_1 \\ \Gamma_1 \vdash S_2 : \Gamma_2, r_2\end{array}}{\Gamma \vdash S : \Gamma_2, r_1 + r_2}$$

$$\frac{S = \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \qquad \begin{array}{c}\Gamma \vdash e : \mathbf{true} \\ \Gamma \vdash S_1 : \Gamma', r'\end{array}}{\Gamma \vdash S : \Gamma', r'}$$

$$\frac{S = \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \qquad \begin{array}{c}\Gamma \vdash e : \mathbf{false} \\ \Gamma \vdash S_2 : \Gamma', r'\end{array}}{\Gamma \vdash S : \Gamma', r'}$$

$$\frac{S = \mathbf{while}\ e\ \mathbf{do}\ S' \qquad \Gamma \vdash e : \mathbf{false}}{\Gamma \vdash S : \Gamma, r}$$

$$\frac{\begin{array}{cc} S = \mathbf{while}\ e\ \mathbf{do}\ S' & \Gamma \vdash S' : \Gamma_1, r_1 \\ \Gamma \vdash e : \mathbf{true} & \Gamma_1 \vdash S : \Gamma_2, r_2\end{array}}{\Gamma \vdash S : \Gamma_2, r_1 + r_2}$$

Figure 3.4: Rules for computing resource usage

QCHL is concerned with proving properties about the difference in *resource usage* across multiple runs. Towards this goal, QCHL performs cost instrumentation and explicitly tracks the program's resource usage. Furthermore, since our goal is to prove the specific property of ϵ-bounded non-interference, QCHL exploits domain-specific assumptions by incorporating taint information into the proof rules. Finally, since the QCHL proof rules we describe here are deterministic, our program logic can be immediately translated into a verification algorithm (modulo an oracle for providing loop invariants and proving standard Hoare triples).

Figure 3.5 presents the proof rules of QCHL. Here, all proof rules, with the exception of Rule (0), derive judgments of the form $\Sigma \vdash \langle \Phi \rangle\ S_1 \circledast S_2\ \langle \Psi \rangle$, where S_1 and S_2 contain a disjoint set of variables and Σ is a *taint environment* mapping variables to a taint value drawn from the set $\{low, high\}$. The notation $S_1 \circledast S_2$ describes a program that is semantically equivalent to $S_1; S_2$ but that is somehow easier to verify (because it tries to execute loops from different executions in lock step). Hence, we have $\Sigma \vdash \langle \Phi \rangle\ S_1 \circledast S_2\ \langle \Psi \rangle$ if $\{\Phi\} S_1; S_2 \{\Psi\}$ is a valid Hoare triple. As we will see shortly, the taint environment Σ is used as a way of increasing the precision and scalability of the analysis. In the remainder of this section, we assume that Σ is sound, i.e., if $\Sigma(x)$ is *low*, then the value of x does not depend (either explicitly or implicitly) on any high inputs. We now explain each of the rules from Figure 3.5 in more detail.

The first rule labeled (0) corresponds to the top-level verification procedure. If we can derive $\Sigma \vdash \mathit{SideChannelFree}(P, \epsilon)$, then P obeys the ϵ-bounded non-interference property. In this rule, we use the notation S^τ to denote the *cost-instrumented* version of S, defined as follows:

Definition 3.4.2. Given a program $P = \lambda \vec{p}.S$, its cost-instrumented version is another program P^τ obtained by instrumenting P with a counter variable τ that tracks its resource usage. More formally, $P^\tau = \gamma(P)$ where the instrumentation procedure γ is defined as:

- $\gamma(\lambda \vec{p}.S) = \lambda \vec{p}.(\tau := 0; \gamma(S))$
- $\gamma(\texttt{skip}) = \texttt{skip}$
- $\gamma(x := e) = (x := e)$
- $\gamma(\texttt{consume}\ (e)) = (\tau := \tau + e)$
- $\gamma(S_1; S_2) = \gamma(S_1); \gamma(S_2)$
- $\gamma(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2) =$
  $\texttt{if}\ e\ \texttt{then}\ \gamma(S_1)\ \texttt{else}\ \gamma(S_2)$
- $\gamma(\texttt{while}\ e\ \texttt{do}\ S) = \texttt{while}\ e\ \texttt{do}\ \gamma(S)$

Essentially, the program P^τ is the same as P except that it contains an additional variable τ that tracks the program's resource usage. As stated by the following lemma, our instrumentation is correct with respect to the operational semantics from Figure 3.4.

Lemma 1. *Let program $P = \lambda \vec{p}.S$ and let $P^\tau = \lambda \vec{p}.S^\tau$. We have*

- *S^τ does not contain any* `consume` *statement.*
- *If $\Gamma(\vec{p}) = \vec{a}$ and $\Gamma \vdash S^\tau : \Gamma'$, then $R_P(\vec{a}) = \Gamma'(\tau)$.*

Hence, rule (0) from Figure 3.5 instruments the original program $\lambda \vec{p}.\ S$ to obtain a new program $\lambda \vec{p}.\ S^\tau$ that uses a fresh variable τ to track the program's resource usage. Since bounded non-interference is a 2-safety property, it then creates two α-renamed copies S_1^τ and S_2^τ of S^τ that have no shared variables and uses the remaining QCHL proof rules to derive a triple

$$\langle \vec{p_1}^{\,l} = \vec{p_2}^{\,l} \land \vec{p_1}^{\,h} \neq \vec{p_2}^{\,h} \rangle\ S_1^\tau \circledast S_2^\tau\ \langle \Psi \rangle$$

If the post-condition Ψ logically implies $|\tau_1 - \tau_2| \leq \epsilon$, we have a proof that the program obeys bounded non-interference. Intuitively, this proof rule considers an arbitrary pair of executions of S where the low inputs are the same and tries to prove that the resource usage of the two runs differs by at most ϵ.

The remaining rules from Figure 3.5 derive QCHL triples of the form $\langle \Phi \rangle\ S_1 \circledast S_2\ \langle \Psi \rangle$. Our verification algorithm applies these rules in the reverse order shown in Figure 3.5. That is, we only use rule labeled i if no rule with label $j > i$ is applicable. Hence, unlike standard CHL, our verification method does not perform backtracking search over the proof rules.

Let us now consider the remaining rules in more detail: Rule (1) is the same as commutativity rule in CHL and states that the $\circledast$ operator is

$$\frac{\begin{array}{cc} & \Phi = (\vec{p_1}^{\,l} = \vec{p_2}^{\,l} \ \wedge \vec{p_1}^{\,h} \neq \vec{p_2}^{\,h}) \\ \lambda\vec{p_1}.S_1^\tau = \alpha(\lambda\vec{p}.S^\tau) & \Sigma \vdash \langle\Phi\rangle\ S_1^\tau \circledast S_2^\tau\ \langle\Psi\rangle \\ \lambda\vec{p_2}.S_2^\tau = \alpha(\lambda\vec{p}.S^\tau) & \models \Psi \to |\tau_1 - \tau_2| \leq \epsilon \end{array}}{\Sigma \vdash SideChannelFree(\lambda\vec{p}.S,\ \epsilon)}\ (0)$$

$$\frac{\Sigma \vdash \langle\Phi\rangle\ S_2 \circledast S_1\ \langle\Psi\rangle}{\Sigma \vdash \langle\Phi\rangle\ S_1 \circledast S_2\ \langle\Psi\rangle}\ (1)$$

$$\frac{\begin{array}{c} S \neq (S_1; S_2) \\ \Sigma \vdash \langle\Phi\rangle\ S; \mathbf{skip} \circledast S'\ \langle\Psi\rangle \end{array}}{\Sigma \vdash \langle\Phi\rangle\ S \circledast S'\ \langle\Psi\rangle}\ (2)$$

$$\frac{\begin{array}{c} \vdash \{\Phi\}\ S_1\ \{\Phi'\} \\ \Sigma \vdash \langle\Phi'\rangle\ S_2 \circledast S_3\ \langle\Psi\rangle \\ S_1 = \mathbf{skip} \vee S_1 = (v := e) \end{array}}{\Sigma \vdash \langle\Phi\rangle\ S_1; S_2 \circledast S_3 \langle\Psi\rangle}\ (3)$$

$$\frac{\vdash \{\Phi\}\ S\ \{\Psi\}}{\Sigma \vdash \langle\Phi\rangle\ S \circledast \mathbf{skip}\ \langle\Psi\rangle}\ (4)$$

$$\frac{\begin{array}{c} \Sigma \vdash \langle\Phi \wedge e\rangle\ S_1; S \circledast S_3 \langle\Psi_1\rangle \\ \Sigma \vdash \langle\Phi \wedge \neg e\rangle\ S_2; S \circledast S_3 \langle\Psi_2\rangle \end{array}}{\Sigma \vdash \langle\Phi\rangle\ \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2; S \circledast S_3 \langle\Psi_1 \vee \Psi_2\rangle}\ (5)$$

$$\frac{\begin{array}{c} \vdash \{\Phi\}\mathbf{while}\ e_1\ \mathbf{do}\ S_1\{\Phi'\} \\ \vdash \{\Phi'\}\mathbf{while}\ e_2\ \mathbf{do}\ S_2\{\Psi'\} \\ \Sigma \vdash \langle\Psi'\rangle\ S \circledast S'\ \langle\Psi\rangle \end{array}}{\Sigma \vdash \langle\Phi\rangle\ \mathbf{while}\ e_1\ \mathbf{do}\ S_1; S \circledast \mathbf{while}\ e_2\ \mathbf{do}\ S_2; S' \langle\Psi\rangle}\ (6)$$

$$\frac{\begin{array}{c} \Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I) \\ \Sigma \vdash \langle I \wedge e_1 \wedge e_2\rangle S_1 \circledast S_2 \langle I'\rangle \\ \Sigma \vdash \langle I \wedge \neg e_1 \wedge \neg e_2\rangle S \circledast S' \langle\Psi\rangle \\ \models \Phi \to I \qquad \models I' \to I \end{array}}{\Sigma \vdash \langle\Phi\rangle\ \mathbf{while}\ e_1\ \mathbf{do}\ S_1; S \circledast \mathbf{while}\ e_2\ \mathbf{do}\ S_2; S' \langle\Psi\rangle}\ (7)$$

Figure 3.5: QCHL proof rules

The notation $\alpha(S)$ denotes an α-renamed version of statement S.

symmetric. Intuitively, since S_1 and S_2 do not share variables, any interleaving of S_1 and S_2 will yield the same result, and we can therefore commute the two operands when deriving QCHL triples. As will become clear shortly, the commutativity rule ensures that our verification algorithm makes progress when none of the other rules are applicable.

The next rule states that we are free to append a skip statement to any non-sequential statement without affecting its meaning. While this rule may not seem very useful on its own, it allows us to avoid redundancies in the proof system by bringing each $S_1 \circledast S_2$ to a canonical form where S_1 is always of the form $S; S'$ or S_2 is skip.

Rule (3) specifies the verification logic for $S_1 \circledast S_2$ when S_1 is of the form $A; S$ where A is an atomic statement. In this case, we simply "consume" A by deriving the Hoare triple $\{\Phi\}A\{\Phi'\}$ and then use Φ' as a precondition for $S \circledast S_2$.

Rule (4) serves as the base case for our logic. When we want to prove $\langle\Phi\rangle\ S \circledast \textbf{skip}\ \langle\Psi\rangle$, we immediately reduce this judgement to the standard Hoare triple $\{\Phi\}\ S\ \{\Psi\}$ because skip is just a no-op.

Example. Suppose we want to prove (0-bounded) non-interference for the following program:

```
λ(low x). consume(x); skip;
```

First we apply transformation γ and get the resource instrumented program:

```
λ(low x). τ=0; τ = τ + x; skip;
```

Ignore the taint environment for now, as we will not use it in this example. According to rule (0), we only need to prove

$$\langle x_1 = x_2 \rangle\ \tau_1 = 0; \tau_1 = \tau_1 + x_1; \textbf{skip};\ \circledast\ \tau_2 = 0; \tau_2 = \tau_2 + x_2; \textbf{skip};\ \langle \tau_1 = \tau_2 \rangle$$

Applying rule (3) twice, we can reduce the above judgement to the following one:

$$\langle x_1 = x_2 \wedge \tau_1 = x_1 \rangle\ \textbf{skip};\ \circledast\ \tau_2 = 0; \tau_2 = \tau_2 + x_2; \textbf{skip};\ \langle \tau_1 = \tau_2 \rangle$$

Swapping the two operands of $\circledast$ with rule (1), we get

$$\langle x_1 = x_2 \wedge \tau_1 = x_1 \rangle\ \tau_2 = 0; \tau_2 = \tau_2 + x_2; \textbf{skip};\ \circledast\ \textbf{skip}; \langle \tau_1 = \tau_2 \rangle$$

After applying rule (4), we get

$$\{x_1 = x_2 \wedge \tau_1 = x_1\}\ \tau_2 = 0; \tau_2 = \tau_2 + x_2; \textbf{skip};\ \{\tau_1 = \tau_2\}$$

Applying Hoare-style strongest postcondition computation, the above Hoare triple can be reduced to

$$\{x_1 = x_2 \wedge \tau_1 = x_1 \wedge \tau_2 = x_2\}\ \textbf{skip};\ \{\tau_1 = \tau_2\}$$

Since this Hoare triple is clearly valid, we have proven non-interference using the QCHL proof rules. □

Rule (5) specifies the general verification logic for branch statements. This rule is an analog of the conditional rule in standard Hoare logic: we

$$\frac{e_1 = \alpha(e) \qquad e_2 = \alpha(e)}{e_1 \equiv_\alpha e_2}$$

$$\frac{S_1 = \alpha(S) \qquad S_2 = \alpha(S)}{S_1 \equiv_\alpha S_2}$$

$$\frac{\begin{array}{cc} e_1 \equiv_\alpha e_2 & \Sigma \vdash e_1 : \textbf{low} \\ S_1 \equiv_\alpha S_2 & \Sigma \vdash e_2 : \textbf{low} \end{array}}{\Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I)}$$

$$\frac{\models I \rightarrow (e_1 \leftrightarrow e_2)}{\Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I)}$$

Figure 3.6: Helper rules for figure 3.5

can verify an if statement by embedding the branch condition e into the true branch and its negation $\neg e$ into the false branch and carry out the proof for both branches accordingly.

Rule (6) specifies the general verification logic for loops. Without loss of generality, this rule requires both sides of the $\circledast$ operator to be loops: If one side is a not a loop, we can always apply one of the other rules, using rule (1) to swap the loop to the other side if necessary. The idea here is to apply self-composition [18]: we run the loop on the left-hand side first, followed by the loop on the right-hand side, and try to derive the proof as if the two loops are sequentially composed.

While rule (6) is sound, it is typically difficult to prove 2-safety using rule (6) alone. In particular, rule (6) does not allow us to synchronize executions between the two loops, so the resulting Hoare triples are often hard to verify. The following example illustrates this issue:

Example. Consider the following code snippet:

```
λ(low n, low k).
  i = 0;
  while (i < n) {
    consume(i); i = i + k;
  }
```

To prove that this program obeys ϵ-bounded non-interference, we need to show that the difference in resource consumption after executing the two copies of the loop is at most ϵ. However, to prove this property using rule (6), we would need to infer a precise post-condition about resource consumption. Unfortunately, this requires inferring a complex *non-linear* loop invariant involving i, n, k. Since such loop invariants are difficult to infer, we cannot prove non-interference using rule (6). □

Rule (7) is one of the most important rules underlying QCHL, as it allows us to execute loops from different executions in lockstep. This loop can be applied only when the *CanSynchronize* predicate is true, meaning that the two loops are guaranteed to execute the same number of times. The definition of the *CanSynchronize* predicate is shown in Figure 3.6: Given two loops $L_1 \equiv while(e_1)\ do\ S_1$ and $L_2 \equiv while(e_2)\ do\ S_2$, and a loop invariant I for the "fused" loop $while(e_1 \wedge e_2)\ do\ S_1; S_2$, *CanSynchronize* determines if L_1 and L_2 must execute the same number of times. In the easy case, this information can be determined using only taint information: Specifically, suppose that L_1, L_2 are identical modulo variable renaming and e_1, e_2 contains only untainted (low) variables. Since we prove bounded non-interference under the assumption that low variables from the two runs have the same value, this assumption implies L_1 and L_2 must execute the same number of times. If we cannot prove the *CanSynchronize* predicate using taint information alone, we may still be able to prove it using the invariant I for the fused loop. Specifically, if I logically implies $e_1 \leftrightarrow e_2$, we know that after each iteration e_1, e_2 have the same truth value; hence, the loops must again execute the same number of times.

Now, suppose we can prove that *CanSynchronize* evaluates to true. In this case, rule (7) conceptually executes the two loops in lock-step. Specifically, the premise $\Sigma \vdash \langle I \wedge e_1 \wedge e_2 \rangle S_1 \circledast S_2 \langle I' \rangle$, together with $\models I' \rightarrow I$, ensures that I is an inductive invariant of the fused loop *while*$(e_1 \wedge e_2)$ *do* $S_1; S_2$. Thus, I must hold when the both loops terminate. Thus, we can safely use the predicate $I \wedge \neg e_1 \wedge \neg e_2$ as a precondition when reasoning about the "continuations" S and S'.

Example. In the previous example, we illustrated that it is difficult to prove non-interference using rule (6) even for a relatively simple example. Let us now see why rule (7) makes verifying 2-safety easier. Since i and n are both low according to the taint environment Σ, we can show that the the *CanSynchronize* predicate evaluates to true. To prove that the program obeys non-interference, we use the relational loop invariant $I = (i_1 = i_2 \wedge \tau_1 = \tau_2 \wedge k_1 = k_2)$. It is easy to see that I is a suitable inductive relational loop invariant, because:

- i_1, i_2, τ_1, τ_2 are all set to 0 before the loop starts.

- We know $k_1 = k_2$ from the precondition (since they are low inputs)
- i_1 and i_2 are increased by the same amount in each iteration of the loop since $k_1 = k_2$.
- τ_1 and τ_2 are also increased by the same amount in each iteration of the loop since $i_1 = i_2$.
- I implies the post condition $|\tau_1 - \tau_2| \leq 0$.

Observe that the use of rule (7) allows us to prove the desired property without reasoning about the total resource consumption of the loop. Hence, we do not need complicated non-linear loop invariants, and the verification task becomes much easier to automate. □

Theorem 3 (Soundness)**.** *Assuming soundness of taint environment* Σ*, if* $\Sigma \vdash SideChannelFree(\lambda\vec{p}.S,\ \epsilon)$*, then the program* $\lambda\vec{p}.S$ *does not have an* ϵ*-bounded resource side-channel.*

3.4.3 Loop Invariant Generation

In the previous subsection, we assumed the existence of an oracle for finding suitable relational loop invariants (recall rule 7). Here, by "relational loop invariant", we mean a simulation relation over variables in programs S_1, S_2. Specifically, we use such relational loop invariants in two ways: First, we use them to check whether two loops execute the same number of times. Second, we use the relational loop invariant to compute the precondition for

Algorithm 2 Relational Invariant Generation

```
Input: $\Sigma$, the taint environment.
Input: $\Phi$, the pre-condition of the loop.
Input: $e, S$, loop condition and loop body.
Input: $V$, the set of all variables appeared in the loop.
Output: An inductive relational loop invariant
 1: function RelationalInvGen($\Sigma, \Phi, e, S, V$)
 2:     $(e_1, S_1) \leftarrow \alpha(e, S)$
 3:     $(e_2, S_2) \leftarrow \alpha(e, S)$
 4:     $Guesses \leftarrow \{v_1 = v_2 \mid v \in V\}$
 5:     for $g \in Guesses$ do
 6:         if $\not\models \Phi \rightarrow g$ then
 7:             $Guesses \leftarrow Guesses \backslash \{g\}$
 8:     $inductive \leftarrow$ false
 9:     while $\neg$ $inductive$ do
10:         $I \leftarrow \bigwedge_{g \in Guesses} g$
11:         $assume$ $\Sigma \vdash \langle I \wedge e_1 \wedge e_2 \rangle S_1 \circledast S_2 \langle I' \rangle$
12:         $inductive \leftarrow$ true
13:         for $g \in Guesses$ do
14:             if $\not\models I' \rightarrow g$ then
15:                 $Guesses \leftarrow Guesses \backslash \{g\}$
16:                 $inductive \leftarrow$ false
17:     return $\bigwedge_{g \in Guesses} g$
```

the continuations of the two programs. Hence, to apply rule 7, we need an algorithm for computing such relational loop invariants.

Algorithm 2 shows our inference engine for computing relational loop invariants. This algorithm can be viewed as an instance of monomial predicate abstraction (i.e., *guess-and-check*) [98, 61, 149]. Specifically, we consider the universe *Guesses* of predicates $v_1 = v_2$ relating variables from the two loops. Because synchronizable loops execute the same number of times, they typically

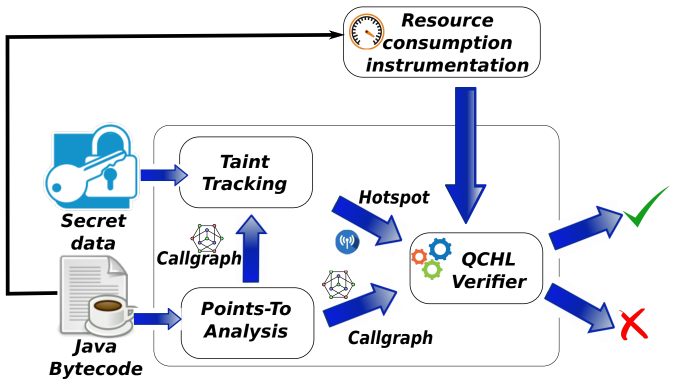


Figure 3.7: Workflow of the THEMIS tool

contain one or more “anchor” variables that are pairwise equal. Thus, we can often find useful relational invariants over this universe of predicates.

Considering Algorithm 2 in more detail, we first filter our those predicates that are not implied by the precondition Φ (lines 5-7). In lines 9-16, we then further filter out those predicates that are not preserved in the loop body. In particular, on line 10, we construct the candidate invariant by conjoining all remaining predicates in our guess set, and, on line 11, we compute the post condition I' of the loop using the proof rules shown in figure 3.5. Since we have $\Sigma \vdash \langle I \wedge e_1 \wedge e_2 \rangle S_1 \circledast S_2 \langle I' \rangle$ and $\not\models I' \rightarrow g$, this means predicate g is not preserved by the loop body and is therefore removed from our set of predicates. When the loop in lines 9-16 terminates, we have $\Sigma \vdash \langle I \wedge e_1 \wedge e_2 \rangle S_1 \circledast S_2 \langle I' \rangle$ and $\models I' \rightarrow I$; thus, I is an inductive relational loop invariant.

3.5 Implementation

Figure 3.7 gives a high-level schematic overview of Themis's architecture. In addition to the QCHL verifier discussed in detail in Section 3.4.1, Themis also incorporates pointer and taint analyses and instruments the program to explicitly track resource usage. We now give a brief overview of each of these components.

Pointer analysis. Given the bytecode of a Java application, Themis performs (field- and object-sensitive) pointer analysis to build a precise call graph and identify all variables that may alias each other. The resulting call graph and alias information are used by the subsequent taint analysis as well as the QCHL verifier.

Taint analysis. The use of taint analysis in Themis serves two goals: First, the QCHL verifier uses the results of the taint analysis to determine whether two loops can be synchronized. Second, we use taint analysis to identify hotspots that need to be analyzed more precisely using the QCHL verifier.

The taint analyzer uses the annotations in Themis's configuration file to determine taint sources (i.e., high inputs) and propagates taint using a field- and object-sensitive analysis. Our taint analyzer tracks both explicit and implicit flows. That is, a variable v is considered tainted if (a) there is an assignment $v := e$ such that e is tainted (explicit flow), or (b) a write to v occurs inside a branch whose predicate is tainted (implicit flow).

We use the results of the taint analysis to identify methods that should be analyzed by the QCHL verifier. A method m is referred to as *hot spot* if it reads from a tainted variable. We say that a hot spot m *dominates* another hot spot m' if m' is a transitive callee of m but not the other way around. Any hot spot that does not have dominators is given as an entry point to the QCHL verifier. In principle, this strategy of running the QCHL verifier on only hot spots can cause our analysis to report false positives. For instance, consider the following example:

```
main(...) { foo(); bar(); }

foo() {
  int x = readSecret();
  if(x > 0) consume(1); else consume(100);
}

bar() {
  int y = readSecret();
  if(y <= 0) consume(1); else consume(100);
}
```

While this program does not have any secret-dependent imbalance in resource usage, `foo` and `bar` individually do not obey non-interference, causing our analysis to report false positives. However, in practice, we have not observed any such false positives, and this strategy greatly increases the scalability of the tool.

Resource usage instrumentation The language we considered for our formalization in Section 3.4.1 is equipped with a *consume*(x) statement that models consumption of x units of resource. Unfortunately, since Java programs do not come with such statements, our implementation uses a *cost model* to instrument the program with such *consume* statements. In principle, our framework can detect different classes of side channels, provided that the tool is given a suitable cost model for the corresponding resource type.

Our current implementation provides cost models for two kinds of resource usage, namely, timing and response size. For timing, we use a coarse cost model where every byte code instruction is assumed to have unit cost. For response size, each string s that is appended to the response consumes $s.length()$ units of resource.

Counterexample generation If THEMIS fails to verify the bounded non-interference property for a given ϵ, it can also generate counterexamples by using the models provided by the underlying SMT solver. In particular, when the verification condition (VC) generated by THEMIS is invalid, the tool asks the SMT solver for a falsifying assignment and pretty-prints the model returned from Z3 by replacing Z3 symbols with their corresponding variable names. Since the VCs depend on automatically inferred loop invariants, the counterexamples generated by THEMIS may be spurious if the inference engine does not infer sufficiently strong loop invariants.

The THEMIS system is implemented in a combination of Java and OCaml and leverages multiple existing tools, such as Soot [168], Z3 [49], and Apron [88]. Specifically, our pointer analysis builds on top of Soot [168], and we extend the taint analysis provided by FlowDroid [10], which is a state-of-the-art context-, field-, flow-, and object-sensitive taint analyzer, to also track implicit flows. Our QCHL verifier is implemented in OCaml and uses the Z3 SMT solver [49] to discharge the generated verification conditions. To prove the Hoare triples that arise as premises in the QCHL proof rules, we perform standard weakest precondition computation, leveraging the Apron numerical abstract domain library [88] to infer standard loop invariants. Recall that we infer relational loop invariants using the monomial predicate abstraction technique described in Section 3.4.3.

Our formal description of QCHL in chapter 3.4.1 uses a simplified programming language that does not have many of the complexities of Java. THEMIS handles these complexities by first leveraging the Soot framework to parse the Java bytecode to Soot IR, and then using an in-house "front-end" that further lowers Soot IR into a form closer to what is presented in chapter 3.4.1. In particular, the transformation from Soot to our IR recovers program structures (loops, conditionals etc.) and encodes heap accesses in terms of arrays. The verifier performs strongest postcondition calculation over our internal IR and encodes verification conditions with SMT formulae.

In the remainder of this section, we explain how we handle various challenges that we encountered while building the THEMIS frontend.

Object encoding Since objects are pervasive in Java applications, their encoding has a significant impact on the precision and scalability of the approach.

In THEMIS, we adopt a heap encoding that is similar to ESC-Java [63]. Specifically, instance fields of objects are represented as maps from object references (modeled as integer variables) to the value of the corresponding field. Reads and writes to the map are modeled using select and update functions defined by the theory of arrays in SMT solvers. If two object references are known not to be the same (according to the results of the pointer analysis), we then add a disequality constraint between the corresponding variables.

Method invocation Since the simplified language from Section 3.4.1 did not allow function calls, we only described an intraprocedural version of the QCHL verifier. We currently perform interprocedural analysis by function inlining, which is performed as a preprocessing step at the internal IR level before the analysis takes place. Since the QCHL verifier only needs to analyze hot spots (which typically constitute a small fraction of the program), we do not find inlining to be a major scalability bottleneck. However, since recursive procedures cannot be handled using function inlining, our current implementation requires models for recursive procedures that correspond to hot spots.

Virtual calls and instanceof encoding The result of certain operations in the Java language, such as virtual calls and the `instanceof` operator, depends on the runtime values of their operands. To faithfully model those operations , we encode the type of each allocation site as one of its field, and we transform virtual calls and `instanceof` to a series of if statements that branch on this field. For example, if variable `a` may point to either allocation A_1 of type `T1` or allocation A_2 of type `T2`, then the polymorphic call site `a.foo()` will be modeled as:

```
if (a.type == T1)
  ((T1)a).foo();
else if (a.type == T2)
  ((T2)a).foo();
```

We handle the `instanceof` operator in a similar way.

3.6 Evaluation

In this section, we describe our evaluation of Themis on a set of security-critical Java applications. Our evaluation is designed to answer the following research questions:

Q1. How does Themis compare with state-of-the-art tools for side channel detection in terms of accuracy and scalability?

Q2. Is Themis able to detect known vulnerabilities in real-world Java applications, and can Themis verify their repaired versions?

Q3. Is Themis useful for detecting zero-day vulnerabilities from the real world?

In what follows, we describe a series of experiments that are designed to answer the above questions. All experiments are conducted on an Intel

Xeon(R) computer with an E5-1620 v3 CPU and 64G of memory running on Ubuntu 16.04.

3.6.1 Comparison Against Blazer

To evaluate how competitive Themis is with existing tools, we compare Themis against Blazer [9], a state-of-the-art tool for detecting timing side channels in Java bytecode. Blazer is a static analyzer that uses a novel decomposition technique for proving non-interference properties. Since the Blazer tool is not publicly available, we compare Themis against Blazer on the same 22 benchmarks that are used to evaluate Blazer in their PLDI'17 paper [9]. These benchmarks include a combination of challenge problems from the DARPA STAC program, classic examples from previous literature[69, 93, 126], and some microbenchmarks constructed by the developers of Blazer. Since Blazer verifies standard non-interference (rather than our proposed ϵ-bounded variant), we set the value of ϵ to be 0 when running Themis.

We summarize the results of our comparison against Blazer in Table 3.8. [2] One of the key points here is that Themis is able to automatically verify all 25 programs from the Blazer data set. Moreover, we see that Themis is consistently faster than Blazer except for a few benchmarks that take a very short time to analyze. On average, Themis takes a median of 7.73

[2] The Blazer paper reports two sets of numbers for running time, namely time for safety verification alone, and time including attack specification search. Since Themis does not perform the latter, we only compare time for safety verification. For the "Size" column in the table, we use the original metric from Blazer, which indicates the number of basic blocks.

Benchmark	Version	Size	Time (s)	
			BLAZER	THEMIS
MicroBench				
array	Safe	16	1.60	0.28
array	Unsafe	14	0.16	0.23
loopAndbranch	Safe	15	0.23	0.33
loopAndbranch	Unsafe	15	0.65	0.16
nosecret	Safe	7	0.35	0.20
notaint	Unsafe	9	0.28	0.12
sanity	Safe	10	0.63	0.41
sanity	Unsafe	9	0.30	0.17
straightline	Safe	7	0.21	0.49
straightline	Unsafe	7	22.20	5.30
STAC				
modPow1	Safe	18	1.47	0.61
modPow1	Unsafe	58	218.54	14.16
modPow2	Safe	20	1.62	0.75
modPow2	Unsafe	106	7813.68	141.36
passwordEq	Safe	16	2.70	1.10
passwordEq	Unsafe	15	1.30	0.39
Literature				
k96	Safe	17	0.70	0.61
k96	Unsafe	15	1.29	0.54
gpt14	Safe	15	1.43	0.46
gpt14	Unsafe	26	219.30	1.25
login	Safe	16	1.77	0.54
login	Unsafe	11	1.79	0.70

Figure 3.8: Comparison between THEMIS and BLAZER.

Benchmark	Version	LOC	LOC'	$\epsilon = 64$	$\epsilon = 0$	Time (s)
Spring-Security	Safe	1630	41	✓	✓	1.70
Spring-Security	Unsafe	1602	32	✓	✓	1.09
JDK7-MsgDigest	Safe	633	30	✓	✓	1.27
JDK6-MsgDigest	Unsafe	619	27	✓	✓	1.33
Picketbox	Safe	208	73	✓	✗	1.79
Picketbox	Unsafe	180	65	✓	✓	1.55
Tomcat	Safe	12221	100	✓	✗	9.93
Tomcat	Unsafe	12173	96	✓	✓	8.64
Jetty	Safe	2667	77	✓	✓	2.50
Jetty	Unsafe	2619	76	✓	✓	2.07
orientdb	Safe	19564	134	✓	✗	37.99
orientdb	Unsafe	19413	131	✓	✓	38.09
pac4j	Safe	1978	104	✓	✗	3.97
pac4j	Unsafe	1900	105	✓	✓	1.85
boot-auth	Safe	7106	74	✓	✗	9.12
boot-auth	Unsafe	6977	69	✓	✓	8.31
tourPlanner	Safe	7735	46	✓	✓	22.22
tourPlanner	Unsafe	7660	34	✓	✓	22.01
Dyna_table	Unsafe	175	40	✓	✓	1.165
Advanced_table	Unsafe	232	55	✓	✓	2.01

Figure 3.9: Evaluation on existing vulnerabilities
A checkmark (✓) indicates that THEMIS gives the correct result, while ✗ indicates a false positive.

seconds to verify a benchmark, whereas the median running time of BLAZER is 376.92 seconds.

In summary, while both BLAZER and THEMIS are sound, this comparison shows that THEMIS can verify more programs than BLAZER in a fraction of the time.

3.6.2 Detection of Known Vulnerabilities

To demonstrate that Themis can be used to detect non-trivial vulnerabilities in real-world Java programs, we further evaluate Themis on security-sensitive Java frameworks. The benchmarks we collect come from the following sources:

1. Response-size side-channel benchmarks from existing publication [181][3].
2. One benchmark that contains a response-size side channel from the DARPA STAC project.
3. A well-known timing side channel in the MessageDigest class from JDK6.
4. Seven other benchmarks with known vulnerabilities collected from Github.

Benchmarks that fall in the first two categories contain response-size side-channel vulnerabilities, and all other benchmarks contain timing side-channels. All benchmarks except for those in category (1) also come with a repaired version that does not exhibit the original vulnerability.

Before running Themis, we need to specify the entry points of each application. Since most applications come with test cases, we use these test harnesses as entry points. For those applications for which we do not have

[3]We are only able to obtain the source codes for 2 of 3 benchmarks mentioned in the paper.

suitable drivers, we manually construct a harness and specify it as the entry point.

Main results. The table in Figure 3.9 shows the accuracy and running time of THEMIS on these benchmarks. Using a value of $\epsilon = 64$, THEMIS successfully finds vulnerabilities in the original vulnerable versions of these frameworks and is able to verify that the original vulnerability is no longer present in the repaired versions. The running time of THEMIS is also quite reasonable, taking an average 8.81 seconds to analyze each benchmark.

Benefit of taint analysis. Recall from Sections 3.1 and 3.5 that THEMIS performs taint analysis to identify hot spots, which overapproximate program fragments that may contain a side-channel vulnerability. The QCHL verifier only analyzes such hot spots rather than the entire program. To demonstrate the usefulness of taint analysis, we compare the lines of code (in Soot IR) in the original application (reported in the LOC column) with the lines of code (also in Soot IR) with those analyzed by the QCHL verifier (reported in the LOC' column). As we can see from Figure 3.9, taint analysis significantly prunes security-irrelevant parts of the application in terms of lines of codes. This pruning effect can also be observed using other statistics. For example, the number of reachable methods ranges from 15 to 1487, with an average of 479, before taint analysius, whereas the number of reachable methods after taint analysis ranges from 6 to 35, with an average of 15, after taint analysis. Thus, pruning using taint information makes the job of the QCHL verifier significantly easier.

Benefit of ϵ. To justify the need for our relaxed notion of non-interference, Figure 3.9 also shows the results of the same experiment using an ϵ value of 0. Hence, the $\epsilon = 0$ column from Figure 3.9 corresponds to the standard notion of non-interference. As we can see from the table, THEMIS reports several false positives using an ϵ value of 0. In particular, the repaired versions of some programs still exhibit a minor resource usage imbalance but this difference is practically infeasible to exploit, so the developers consider these versions to be side-channel-free. However, these programs are deemed unsafe using standard non-interference. We believe this comparison shows that our relaxed policy of ϵ-bounded non-interference is useful in practice and allows security analysts to understand the severity of the side channel.

Benefit of relational analysis. To investigate the benefit of relational invariants, we analyze the safe versions of the 20 benchmarks from Figures 8 and 9 with relational invariant generation disabled. In this case, THEMIS can only verify the safety of 10 of the benchmarks.

Although this number can potentially be increased by using a more sophisticated non-relational loop invariant generation algorithm, THEMIS circumvents this need, instead using simple *relational* in- variants that are conjunctions of simple equality constraints. This experiment corroborates the hypothesis that QCHL makes verification easier by requiring simpler loop invariants compared to other techniques like self-composition.

Benchmark	LOC	Category	#Reports	Time (s)
Jetty	2619	Server	4	10.17
Tomcat	12173	Server	1	5.86
OpenMRS	10721	Healthcare	1	9.71
OACC	78	Authentication	1	1.83
Apache Shiro	4043	Authentication	0	6.54
Apache Crypto	4505	Crypto	0	4.33
bc-java	5759	Crypto	0	6.89

Figure 3.10: Evaluation THEMIS on identifying zero-day vulnerabilities from popular Java applications

3.6.3 Discovery of Zero-Day Vulnerabilities

To evaluate whether THEMIS can discover *unknown* vulnerabilities in real world Java applications, we conduct an experiment on seven popular Java frameworks. Our data set covers a wide range of Java applications from different domains such as HTTP servers, health care platforms, authentication frameworks, etc. For example, Eclipse Jetty is a well-known web server that is embedded in products such as Apache Spark, Google App Engine, and Twitter's Streaming API. OpenMRS is the world's leading open source enterprise electronic medical record system platform; OACC is a well-known Java application security framework, and bc-java is an implementation of the Bounty Castle crypto API in Java.

As in our previous experiment, we first manually annotate each application to indicate the sources of confidential information. We then use THEMIS to find timing side channels in these applications using an ϵ value of 10. The results of this experiment are summarized in Figure 3.10. As we can see from this figure, THEMIS reports a total of seven vulnerabilities in four of the analyzed applications. We manually inspected each report and confirmed that the detected vulnerabilities are indeed true positives. We also reported the vulnerabilities detected by Themis to the developers, and the majority of these vulnerabilities were confirmed and fixed by the developers in less than 24 hours. However, the vulnerability that we reported for OpenMRS was rejected by the developers. The reason for this false positive is that the leaked password is actually hashed and salted in the database, but, because the logic for hashing and salting is not part of the Java implementation, Themis was not able to reason about this aspect.

To give the reader some intuition about the kinds of side channels detected by Themis, Figure 3.11 shows a security vulnerability from the Eclipse Jetty web server. The `check` procedure from Figure 3.11 checks whether the password provided by the user matches the expected password (`_cooked`). The original code performs this check by calling the built-in equality method provided by the `java.lang.String` library. Since the built-in equality method returns false *as soon as* it finds a mismatch between two characters, line 10 in the `check` method introduces a timing side-channel vulnerability.

The developers have fixed the vulnerability [1] in this code snippet by replacing line 10 with the (commented out) code shown in line 9. In particular, the fix involves calling the safe version of equals, called `stringEquals`, which checks for equality between *all* characters in the strings. This repaired version of the `check` method no longer contains a vulnerability for any $\epsilon > 1$, and Themis can verify that the `check` procedure is now safe.

```
1  public boolean check(Object credentials)
2  {
3    if (credentials instanceof char[])
4      credentials = new String((char[])credentials);
5    if (!(credentials instanceof String) &&
6        !(credentials instanceof Password))
7      LOG.warn("Can't check " +
8          credentials.getClass() + " against CRYPT");
9
10   String passwd = credentials.toString();
11   // FIX: return stringEquals(_cooked, UnixCrypt.crypt(passwd,_cooked));
12   return _cooked.equals(
13       UnixCrypt.crypt(passwd,_cooked));
14 }
15
16 /**
17  * <p>Utility method that replaces String.equals()
18  * to avoid timing attacks.</p>
19  */
20 static boolean stringEquals(String s1, String s2)
21 {
22   boolean result = true;
23   int l1 = s1.length();
24   int l2 = s2.length();
25   if(l1 != l2) result = false;
26   int n = (l1 < l2) ? l1 : l2;
27   for (int i = 0; i < n; i++)
28     result &= s1.charAt(i) == s2.charAt(i);
29   return result;
30 }
```

Figure 3.11: Eclipse Jetty code snippet that contains a timing side channel Line 12 is the original buggy code. This vulnerability can be fixed by implementing `stringEquals` (lines 20 – 30) and calling it instead of the built-in `String.equals` method.

3.7 Limitations

Like any other program analysis tool, Themis has a number of limitations. First, due to the fundamental undecidability of the underlying static analysis problem, Themis is incomplete and may report false positives (e.g., due to imprecision in pointer analysis or loop invariant generation). For example, our method for inferring relational invariants is based on monomial predicate abstraction using a fixed set of pre-defined templates, and we restrict our templates to equalities between variables. In addition, our non-relational invariant generator is based on traditional abstract interpretation, which does not distinguish array elements precisely.

Second, dynamic features of the Java language, such as reflective calls, dynamic class loading, and exceptional handling, pose challenges for Themis. Our current implementation can handle some cases of reflection (e.g., reflective calls with string constants), but reflection can, in general, cause Themis to have false negatives.

Finally, Themis unconditionally trusts all human inputs into the system, which may result in false negatives if the user inputs are not accurate. Said user inputs include application entry points, taint sources, cost instrumentations, and models of library methods.

Chapter 4

Enhanced Static Verification Using Reinforcement Learning

In Chapter 3 we introduced the techinque of verifying a 2-safety property with a proof system that reduces the original problem into discharging a set of standard Hoare triples. Despite the power and conceptual simplicity of this approach, a key challenge is that there are typically *many* ways to reduce a relational verification problem to proving standard safety. While each reduction method corresponds to a valid, provably-sound proof strategy, some of these strategies are much more amenable to automation than others.

For example, consider the problem of verifying noninterference of program P_1 and P_2 shown in Figure 4.1 under the assumption that all input arguments are marked as low inputs. Although it is relatively straightforward for a human to recongnize that the easiest way to prove the noninterference property is to swap the order of the two loops in P_2 and synchronize each of them with the corresponding loops in P_1, the verification strategy described in Chapter 3.4.1 is not intelligent enough to spot the pattern. Instead, it will naïvely try to synchronize the first loop in P_1 with the first loop in P_2, recognizing that synchronization is not feasible due to the difference of resource

```
void P1(int n1, int k1) {
  for (int i = 0; i < n1; ++i)
    consume(k1);
  for (int i = 0; i < n1; ++i) {
    consume(k1);
    consume(k1);
  }
}
```

```
void P2(int n2, int k2) {
  for (int i = 0; i < n2; ++i) {
    consume(k2);
    consume(k2);
  }
  for (int i = 0; i < n2; ++i)
    consume(k2);
}
```

Figure 4.1: Example programs

consumption in the loop body, and fall back to self-composition where a much harder loop invariant is required from the underlying invariant generation algorithm.

The problem of finding a good reduction strategy most amenable to automation is not unique to the domain of noninterference verification. It also arises in the context of equivalence checking [57], software evolution [96, 97] and version control [162]. In fact, the challenge is shared among all *relational verification* problem in general. Interestingly, another popular approach [15, 17, 56, 176, 162] for relational verification is to construct a so-called *product program* P such that the relational property is valid if P obeys some safety property. In a similar vein, there are often many ways where P can be constructed, and we still face the problem of picking a construction whose corresponding safety property is easy to discharge.

Due to the significance of relational verification across many application domains, and due to the fact that most commonly used approach for relational verification would inevitably have to handle the problem of reduction strategy selection, we are going to study the problem in detail in this chapter. Note that although we motivate the problem as an extention of Themis, the majority of this chapter will be devoted to the study of the general technique without restricting ourselves into any particular domain.

4.1 Overview

In principle, there is a simple way to deal with the challenge of reduction strategy selection: We could simply try *all* possible ways of reducing the relational verification problem to standard safety and conclude that the relational property holds if *any* of the corresponding safety problems can be verified. Unfortunately, this naïve strategy is not feasible in practice because there are simply too many reduction strategies to try. As a result, prior techniques either use domain-specific heuristics (e.g., [176, 161, 38, 162]) or require the user to manually specify a suitable reduction strategy (e.g., [15, 57]). Both of these approaches are sub-optimal in that the latter one lacks automation whereas the first one, by exploring a limited subset of possible reduction strategies, may fail to prove the property.

We aims to address this challenge by guiding relational proof search using machine learning. That is, our goal is to learn a probability distribution over possible relational verification strategies such that those that are deemed more likely-to-be-successful by the machine learning model are explored first. While such an approach would allow us to prioritize different reduction strategies based on the relational verification task at hand, a key problem is the lack of *labeled* training data in the form of successful relational proof strategies. In this chapter, we deal with this problem by using *reinforcement learning (RL)* which essentially allows the relational verifier to learn over time from its *own* failed proof attempts and from successful reduction strategies.

In order to apply reinforcement learning in this context, we first formalize the relational verification problem as a *Markov Decision Process (MDP)* where states correspond to steps in the derivation process and actions correspond to proof rules in a relational program logic [1]. Then, a *policy* of this MDP tells us which relational proof rule to apply at a given step in the derivation process, and our goal is to learn an *optimal policy* that can be used to determine the most promising proof rule to use at each step.

Unfortunately, there are several challenges to applying reinforcement learning to find an optimal policy in this context. First, unlike standard reinforcement learning where the goal is to maximize expected reward for a single trial, we want to maximize our chances of finding a successful proof strategy during a *search algorithm* where we explore multiple proof attempts. Second, we want to learn a policy that is applicable to *any* relational verification problem, meaning that the state space of the underlying MDP is infinite. In this chapter, we overcome the first challenge by adapting the well-known *policy gradient algorithm* to our unique setting, and we address the second challenge by using *approximate reinforcement learning* where proof strategies are mapped

[1]The same formulation can be easily applied to relational verifiers based on product programs – specifically, actions would correspond to product construction rules instead of proof rules.

into n-dimensional feature vectors and policies are represented using a deep neural network.

In addition to learning an optimal probability distribution over relational proof strategies using RL, another contribution of this work is a novel relational verification algorithm that uses the optimal policy to guide search. Specifically, our proposed verification algorithm enumerates proof strategies according to their probability of being sampled from the optimal (stochastic) policy, but it further prunes the search space by extracting a *minimal failing proof strategy* from failed proof attempts.

4.2 Background on Relational Verification

As mentioned in Section 4.1, existing techniques reduce relational verification to safety checking either by explicitly constructing a product program [15, 17, 56] or introducing a proof system where certain proof obligations can be discharged by an off-the-shelf safety checker [24, 19, 161]. In this paper, we adopt the latter approach and think of relational verification as the problem of searching for a proof within a relational program logic.

Following prior work [161, 38, 24], we assume a relational program logic that derives judgments of the form

$$\vdash \ \langle \Phi \rangle \ S_1 \circledast S_2 \ \langle \Psi \rangle$$

where S_1 and S_2 are programs containing disjoint sets of variables and Φ (resp. Ψ) is a relational precondition (resp. post-condition). In the rest of this paper,

$$\frac{\vdash \{\Phi\}S\{\Psi\}}{\vdash \langle\Phi\rangle \mathit{skip} \circledast S\langle\Psi\rangle}\ \text{(Lift)}$$

$$\frac{\vdash \{\Phi\}S\{\Phi'\} \qquad \vdash \langle\Phi'\rangle S_1 \circledast S_2\langle\Psi\rangle}{\vdash \langle\Phi\rangle S; S_1 \circledast S_2\langle\Psi\rangle}\ \text{(Seq)}$$

$$\frac{\begin{array}{c}\Phi \Rightarrow (e_1 \leftrightarrow e_2) \qquad \Phi \Rightarrow \mathbb{I} \\ \vdash \langle \mathbb{I} \wedge e_1 \wedge e_2\rangle S_1 \circledast S_2\langle\mathbb{I}\rangle \\ \vdash \langle \mathbb{I} \wedge \neg e_1 \wedge \neg e_2\rangle S \circledast S'\langle\Psi\rangle\end{array}}{\vdash \langle\Phi\rangle \textbf{while } e_1 \textbf{ do } S_1; S \circledast \textbf{while } e_2 \textbf{ do } S_2; S'\langle\Psi\rangle}\ \text{(Sync)}$$

$$\frac{\begin{array}{c}\vdash \langle \Phi \wedge e\rangle S; \textbf{while } e \textbf{ do } S; S_1 \circledast S_2\langle I\rangle \\ \vdash \langle \Phi \wedge \neg e\rangle S_1 \circledast S_2\langle\Psi\rangle\end{array}}{\vdash \langle\Phi\rangle \textbf{while } e \textbf{ do } S; S_1 \circledast S_2\langle\Psi\rangle}\ \text{(Peel)}$$

$$\frac{\begin{array}{c}f_1 = \lambda\vec{p_1}.\ S_1' \qquad f_2 = \lambda\vec{p_2}.\ S_2' \qquad \vdash \langle\mathcal{P}\rangle S_1' \circledast S_2'\langle\mathcal{Q}\rangle \\ \Phi \Rightarrow \mathcal{P}[\vec{a_1}/\vec{p_1}, \vec{a_2}/\vec{p_2}] \\ \vdash \langle \mathcal{Q}[\vec{a_1}/\vec{p_1}, \vec{a_2}/\vec{p_2}]\rangle S_1 \circledast S_2\langle\Psi\rangle\end{array}}{\vdash \langle\Phi\rangle \textbf{call } f_1(\vec{a_1}); S_1 \circledast \textbf{call } f_2(\vec{a_2}); S_2\langle\Psi\rangle}\ \text{(Call)}$$

Figure 4.2: Selected rules for reducing 2-safety verification problem to standard Hoare triples

we refer to triples of the form $\langle\Phi\rangle\ S_1 \circledast S_2\ \langle\Psi\rangle$ as *relational Hoare triples.*

By studying prior work on relational program verification [24, 161, 15, 17, 57], we built a library of 37 different proof rules and tactics, of which five representative ones are shown in Figure 4.2. While a detailed discussion of these proof rules is out of scope for this chapter, we highlight some of their salient features below.

Reduction to safety. As illustrated by the Lift and Seq rules from Figure 4.2, the premises of a relational proof rule can involve proving standard Hoare triples of the form $\{P\}S\{Q\}$. Thus, relational program logics eventually reduce the problem to standard safety checking.

Non-determinism. Given a proof goal $\mathcal{G} = \langle\Phi\rangle\ S_1 \circledast S_2\ \langle\Psi\rangle$, there are typically many rules that can be used to prove $\mathcal{G}$. For example, if S_1 and S_2 are both while loops, we can apply three different rules (namely, Seq, Sync, and Peel) even for the small subset of proof rules shown in Figure 4.2.

Sensitivity to proof strategy. Let us define a *proof strategy* to be a mapping from each proof subgoal to a proof rule that can be used for discharging it. Because the base cases of a relational proof require invoking an off-the-shelf safety checker, the success of a particular proof strategy depends on how easy or difficult the corresponding safety checking problems are. Thus, some proof strategies may lead to successful proofs, while others may not.

Large search space. Since there are many proof rules that can be used to discharge a relational Hoare triple, the search space of proof strategies is very large. For example, suppose we have m rules with k subgoals. Then, given two programs S_1, S_2 of size n, there are up to $O((mk)^n)$ possible applications of the proof rules.

Shape of the rules. As we can see from Figure 4.2, each relational proof rule $\mathcal{R}$ consists of (i) a goal $\mathcal{G}$ (i.e., a relational Hoare triple), (ii) a set of subgoals $\Omega = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$, where each $\mathcal{G}_i$ is also a relational Hoare triple, and (iii) a set of verification conditions (VCs) (e.g., $\Phi \rightarrow (e_1 \leftrightarrow e_2)$ and $\Phi \rightarrow \mathbb{I}$ in rule Sync). Thus, we can represent each relational proof rule $\mathcal{R}$ as a quadruple $\mathcal{R} = (\mathcal{R}_{id}, \mathcal{R}_{\mathcal{G}}, \mathcal{R}_{\Omega}, \mathcal{R}_{\varphi})$, where $\mathcal{R}_{id}$ is the name of the rule, $\mathcal{R}_{\mathcal{G}}, \mathcal{R}_{\Omega}$ represent the goal and subgoals respectively, and $\mathcal{R}_{\varphi}$ is a formula that corresponds to the conjunction of all VCs. Observe that the VCs can involve unknown predicates such as $\mathbb{I}$ in rule Sync or pre- and post-conditions $\mathcal{P}, \mathcal{Q}$ in rule Call; thus we represent VCs as a system of *Constrained Horn Clauses (CHCs)* [48, 115]. Furthermore, since standard Hoare triples can also be encoded as CHCs [29, 47], we also think of the standard Hoare triples that occur in the premises as part of the VC of the corresponding rule.

4.3 Representing Proof Strategies

Our goal in the rest of the paper is to automate relational verification by efficiently searching through a large space of possible proof strategies. In this chapter, we will describe how we represent proof strategies and formalize what we mean by a strategy being successful.

Intuitively, a proof strategy specifies which rule to apply to discharge each proof subgoal. In the rest of this paper, we represent proof strategies as trees where nodes correspond to proof subgoals and edges represent the application of some proof rule.

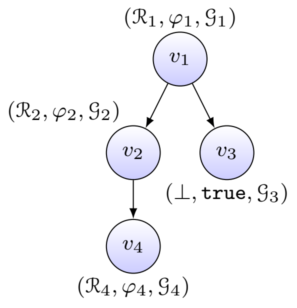
Definition 8 (Proof strategy). A *proof strategy* is a tuple $\Upsilon = (V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$ where

- V is a set of nodes.
- E is a set of arcs.
- $A_{\mathcal{R}}$ maps each node to either a proof rule $\mathcal{R}$ or $\bot$.
- A_{φ} maps each node to a verification condition.
- $A_{\mathcal{G}}$ maps each node to the corresponding proof goal $\langle \Phi \rangle\ S_1 \circledast S_2\ \langle \Psi \rangle$ for its subtree.

We refer to $A_{\mathcal{R}}, A_{\varphi}, and A_{\mathcal{G}}$ as the *rule*, *VC*, and *goal* annotations respectively, and we use the symbol $\bot$ to indicate *open branches* of the proof. That is, if $A_{\mathcal{R}}(v)$ is $\bot$, this means that we have not yet chosen a proof rule for proving the subgoal associated with v. Thus, we also differentiate between *complete* and *incomplete* proof strategies:

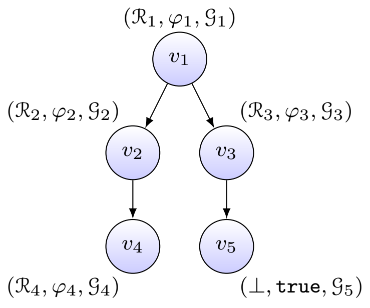
Definition 9 (Complete proof strategy). We say that $v \in V$ is an *open branch* of proof strategy S if $A_{\mathcal{R}}(v) = \bot$. A proof strategy is *complete* if it does not have any open branches and *incomplete* otherwise.

Example 4.3.1. Figure 4.3a shows an example proof strategy Υ. Based on the tree structure, we see that nodes v_2 and v_3 correspond to subgoals of v_1, which represents the proof goal $\mathcal{G}_1$. Furthermore, since v_1 is annotated with



$(\mathcal{R}_1, \varphi_1, \mathcal{G}_1)$
v_1
$(\mathcal{R}_2, \varphi_2, \mathcal{G}_2)$
v_2
v_3
$(\perp, \texttt{true}, \mathcal{G}_3)$
v_4
$(\mathcal{R}_4, \varphi_4, \mathcal{G}_4)$

(a) Before applying $\mathcal{R}_3$



$(\mathcal{R}_1, \varphi_1, \mathcal{G}_1)$
v_1
$(\mathcal{R}_2, \varphi_2, \mathcal{G}_2)$
$(\mathcal{R}_3, \varphi_3, \mathcal{G}_3)$
v_2
v_3
v_4
v_5
$(\mathcal{R}_4, \varphi_4, \mathcal{G}_4)$
$(\perp, \texttt{true}, \mathcal{G}_5)$

(b) After applying $\mathcal{R}_3$

Figure 4.3: Example proof strategies

rule $\mathcal{R}_1$, we can tell that proof subgoals v_2 and v_3 were obtained by applying proof rule $\mathcal{R}_1$. Also, node v_1 is annotated with verification condition φ_1; this means φ_1 must be discharged for the application of rule $\mathcal{R}_1$ to be valid. Finally, note that v_3 is an open branch of the proof since we have $A_{\mathcal{R}}(v_3) = \perp$. Thus, Υ is *incomplete*.

Since our verification algorithm starts with a completely unconstrained strategy and iteratively refines it, we define the notion of *initial strategy* for a given proof goal $\mathcal{G}$:

Definition 10 (Initial strategy). Given a proof goal $\mathcal{G}$, the *initial strategy* for $\mathcal{G}$, denoted $\Upsilon_0(\mathcal{G})$, is given by:

$$(\{v_1\}, \emptyset, \{v_1 \mapsto \perp\}, \{v_1 \mapsto \mathit{true}\}, \{v_1 \mapsto \mathcal{G}\})$$

Thus, $\Upsilon_0(\mathcal{G})$ encodes all possible ways of proving the goal $\mathcal{G}$ within the given proof system. Since our verification algorithm will iteratively refine its strategy by expanding an open branch, Algorithm 3 desribes how we apply a proof rule $\mathcal{R}$ to strategy Υ. Given a proof rule $\mathcal{R}$ and incomplete strategy Υ, ApplyProofRule yields a refined strategy by generating (a) verification conditions as prescribed by $\mathcal{R}_\varphi$, and (b) new proof subgoals $\mathcal{G}_1, \ldots, \mathcal{G}_n$ according to $\mathcal{R}_\mathcal{G}$. Note that each proof subgoal leads to the addition of an open branch in the refined strategy. [2]

Example 4.3.2. Figure 4.3b shows the result of applying rule $\mathcal{R}_3$ to the open branch of Figure 4.3a. Here, $\mathcal{R}_3$ generates one new subgoal $\mathcal{G}_5$ with associated verification conditions φ_3. The rule application introduces a new open branch v_5 below v_3, with $A_\varphi(v_5)$ initialized to `true`.

Definition 11 (Strategy refinement). We say that a strategy Υ' *directly refines* another strategy Υ, written $\Upsilon' \preceq^1 \Upsilon$, if Υ' is the result of calling ApplyProofRule on Υ for *some* proof rule $\mathcal{R}$. We define $\preceq$ as the reflexive transitive closure of $\preceq^1$ and say that Υ' *refines* Υ whenever $\Upsilon' \preceq \Upsilon$.

Given a proof strategy, we need a way of determining whether it results in a valid proof. Towards this goal, we define a *successful* proof strategy as follows:

Definition 12 (Successful proof strategy). A proof strategy $\Upsilon = (V, E, A_\mathcal{R}, A_\varphi, A_\mathcal{G})$ is *successful* if

[2] In Algorithm 3, `GenVC` and `GenSubgoals` take a proof rule and a proof goal as input and generate new VCs and new subgoals according to the proof rules in Figure 4.2, respectively. The `FirstOpenBranch` function returns the first open branch of the given strategy. Since every open branch must be closed eventually, we assume a canonical order for simplicity.

Algorithm 3 Rule application

```
Input: $\Upsilon = (V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$: incomplete proof strategy
Input: $\mathcal{R} = (\mathcal{R}_{id}, \mathcal{R}_{\mathcal{G}}, \mathcal{R}_{\Omega}, \mathcal{R}_{\varphi})$: rule to apply
Output: A refined proof strategy
1: procedure ApplyProofRule($\Upsilon$, $\mathcal{R}$)
2:     $v \leftarrow$ FirstOpenBranch($\Upsilon$)
3:     $A_{\mathcal{R}}(v) \leftarrow \mathcal{R}_{id}$
4:     $A_{\varphi}(v) \leftarrow$ GenVC($\mathcal{R}_{\varphi}, A_{\mathcal{G}}(v)$)
5:     $\Omega \leftarrow$ GenSubgoals($\mathcal{R}_{\Omega}, A_{\mathcal{G}}(v)$)
6:     for $\mathcal{G}_i \in \Omega$ do
7:         $v' \leftarrow$ fresh node
8:         $(V,\ E) \leftarrow (V \cup \{\ v'\ \},\ E \cup \{\ v \rightarrow v'\ \})$
9:         $(A_{\mathcal{R}},\ A_{\varphi}) \leftarrow (A_{\mathcal{R}}[v' \leftarrow \bot],\ A_{\varphi}[v' \leftarrow true])$
10:        $A_{\mathcal{G}} \leftarrow A_{\mathcal{G}}[v' \leftarrow \mathcal{G}_i]$
11:    return $(V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$
```

- Υ is complete.

- The formula $\bigwedge_{v \in V} A_{\varphi}(v)$ can be proven satisfiable.

Recall from Section 4.2 that we represent verification conditions as Constrained Horn Clauses (CHCs). Thus, the satisfiability of the formula $\bigwedge_{v \in V} A_{\varphi}(v)$ means that there exists an interpretation of the unknown relations under which the formula evaluates to true.

Definition 13 (**Failing proof strategy).** A proof strategy $\Upsilon = (V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$ is failing if the conjunction $\bigwedge_{v \in V} A_{\varphi}(v)$ is unsatisfiable.

Note that, unlike successful proof strategies, failing strategies need not be complete. In particular, the formula can become unsatisfiable $\bigwedge_{v \in V} A_{\varphi}(v)$

even when the proof contains open branches. Our proof search algorithm will take advantage of this observation in Section 4.6.

4.4 Learning Algorithm Overview

In this section, we give a high-level overview of our relational verification algorithm and highlight its salient features.

Searching for relational proofs. As mentioned earlier, our verification algorithm performs backtracking search over proof strategies, prioritizing those that are most promising. To this end, we use reinforcement learning to predict which proof strategies are most likely to be successful. Specifically, our reinforcement learning algorithm produces a distribution p over complete proof strategies such that, if $p(\Upsilon_1) > p(\Upsilon_2)$, then Υ_1 is more likely to be a successful strategy compared to Υ_2 according to the learned model.

Given a specific relational verification task t, we use the notation $p^{(t)}$ to denote the distribution of complete proof strategies Υ that are *applicable* to verifying t (i.e., the root node of Υ is annotated with the initial proof goal for t). Now, to solve a relational verification problem t, our search algorithm initializes $p_0 = p^{(t)}$. Then, on each iteration $i = 0, 1, 2, ...$ (up to some upper bound r)[3], it chooses a complete proof strategy Υ_i that has high probability

[3]While the value of r used by the search algorithm is large (it corresponds to the timeout set on the search algorithm), during training we choose r to be small. By doing so, we encourage the the search algorithm to discover a successful proof strategy earlier in the search.

according to p_i, and checks whether Υ_i is successful. If so, the verification algorithm terminates and returns Υ_i. Otherwise, based on feedback explaining why Υ_i was unsuccessful, our algorithm constrains the support of p_i to obtain a new distribution p_{i+1} that avoids making mistakes similar to those in Υ_i. In Section 4.6, we describe how our search strategy constructs p_{i+1} given p_i and a failing proof strategy Υ_i.

Learning objective. The goal of our learning algorithm is to generate a distribution p that places high probability mass on successful proof strategies. In particular, it aims to solve the following optimization problem:

$$p^* = \arg\max_p \Pr_{t\sim\mathcal{T},\Upsilon\sim\xi_{r,p}^{(t)}}[\mathcal{O}(\Upsilon) = 1] \tag{4.1}$$

where $t \sim \mathcal{T}$ is a uniformly random task, $\mathcal{O}(\Upsilon)$ is 1 if Υ is successful and 0 otherwise, and $\xi_{r,p}^{(t)}$ is a distribution of proof strategies checked by the search algorithm, i.e.,

$$\xi_{r,p}^{(t)}(\Upsilon) = \frac{1}{r}\sum_{i=1}^{r} p_i^{(t)}(\Upsilon).$$

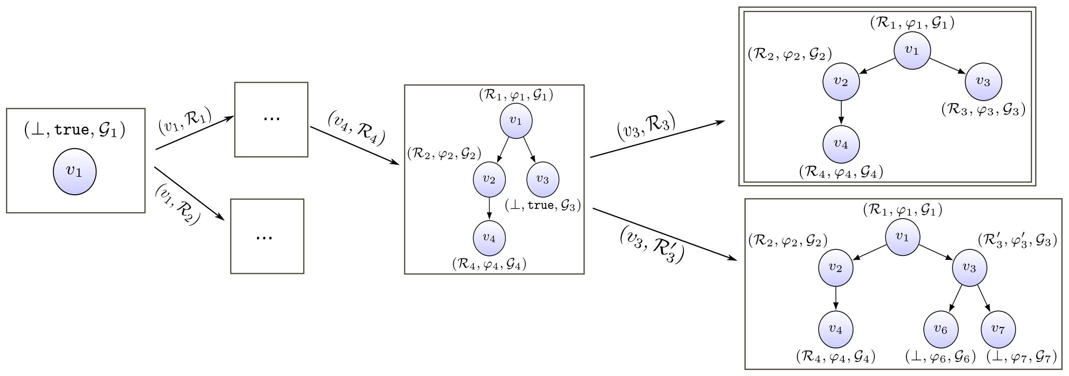
Essentially, the objective in Eq. (4.1) is to maximize the probability that our search algorithm discovers a successful proof strategy for a uniformly random task within r iterations.

However, there are three challenges to solving the optimization problem from Eq. (4.1): First, we do not have positive examples of successful proof strategies. Second, we only have a finite training set of tasks $\mathcal{T}_{\text{train}}$.

Finally, standard reinforcement learning algorithms cannot be applied to optimize Eq. (4.1) due to the modified distribution $\xi_{r,p}^{(t)}$. Below, we discuss how we address these challenges.

Reinforcement learning. Since we do not have positive examples of successful proof strategies, we cannot use standard supervised learning algorithms to optimize Eq. (4.1). Instead, we have oracle access to $\mathcal{O}$ in the form of our proof checker, which makes it possible to use reinforcement learning. In Section 4.5.2, we describe how to formulate the optimization problem from Eq. (4.1) as a reinforcement learning problem.

Function approximation. Since we are only given a finite subset of tasks $\mathcal{T}_{\text{train}} \subseteq \mathcal{T}$, we can only approximate the samples $t \sim \mathcal{T}$ from Eq. (4.1) with uniformly random samples $t \sim \mathcal{T}_{\text{train}}$. However, the solution to the approximate objective may not generalize to all of $\mathcal{T}$. Thus, we use a *feature map* to improve generalization. The essential idea is to restrict the search space to distributions $p(\Upsilon)$ that only depend on Υ through a handcrafted feature map $\phi(\Upsilon) \in \mathcal{X} = \mathbb{R}^d$, which is designed to map similar proof strategies to similar features. In particular, given two strategies Υ and Υ', we should have $\phi(\Upsilon) \approx \phi(\Upsilon')$ if the proof goals labeling their roots are similar, and $\phi(\Upsilon) \not\approx \phi(\Upsilon')$ otherwise. Then, if the optimal distribution p^* assigns high probability mass to Υ, it similarly assigns high probability mass to Υ' (assuming p^* is reasonably smooth). Thus, knowledge can be transferred to new



$(\perp, \texttt{true}, \mathcal{G}_1)$
v_1
$(v_1, \mathcal{R}_1)$
$(v_1, \mathcal{R}_2)$
...
...
$(v_4, \mathcal{R}_4)$
$(\mathcal{R}_1, \varphi_1, \mathcal{G}_1)$
$(\mathcal{R}_2, \varphi_2, \mathcal{G}_2)$
$(\perp, \texttt{true}, \mathcal{G}_3)$
$(\mathcal{R}_4, \varphi_4, \mathcal{G}_4)$
$(v_3, \mathcal{R}_3)$
$(v_3, \mathcal{R}'_3)$
$(\mathcal{R}_3, \varphi_3, \mathcal{G}_3)$
$(\mathcal{R}'_3, \varphi'_3, \mathcal{G}_3)$
$(\perp, \varphi_6, \mathcal{G}_6)$
$(\perp, \varphi_7, \mathcal{G}_7)$

Figure 4.4: An example of an MDP constructed for a relational verification problem.

tasks with proof goals that are different from those for training tasks $t \in \mathcal{T}$. We describe this approach in Section 4.5.3.

Reinforcement learning algorithm. Standard reinforcement learning algorithms can only be applied to optimizing Eq. (4.1) for the case $\xi^{(t)}_{r,p} = p^{(t)}$, i.e., where $r = 1$. In other words, these algorithms can only optimize for the case where the search algorithm only considers a single proof strategy, so they are not directly applicable to our setting where the search algorithm tries multiple consecutive proof strategies. In Section 4.5.4, we describe how we adapt the standard policy gradient algorithm to our setting.

4.5 Reinforcement Learning

4.5.1 Background on Reinforcement Learning

A reinforcement learning problem is typically specified as a *Markov decision process (MDP)*. Informally, an MDP is a transition system where the process is in some state S_i at each time step, and a decision maker can take any of the actions $A_1, \ldots, A_n$ that is available at state S_i and collects some reward R. The goal of reinforcement learning is to find the optimal action to take in each state to maximize the expected long-term reward.

Definition 14. A *Markov decision process* is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{S}_0, \mathcal{S}_F, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ is the set of states, $\mathcal{S}_0$ is the initial distribution over states, $\mathcal{S}_F$ is a set of terminal states, $\mathcal{A}$ is the set of actions, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the (possibly stochastic) transition function, and $\mathcal{R} : \mathcal{S} \to \mathbb{R}$ is the (possibly stochastic) reward function.[4]

Definition 15. A *policy* π for an MDP $\mathcal{M}$ is a (possibly stochastic) function $\pi : \mathcal{S} \to \mathcal{A}$ specifying which action to take in each state.

When we use a stochastic policy π to choose which actions to take, this results in a random trajectory, referred to as a *rollout*, through the state space:

Definition 16. A *rollout* $\zeta \sim \pi$ is a random sequence of tuples $\zeta \in (\mathcal{S} \times (\mathcal{A} \cup \{\varnothing\}) \times \mathcal{R})^*$ constructed as follows:

- sample a random state $S_0 \sim \mathcal{S}_0$
- sample actions $A_i = \pi(S_i)$, random transitions $S_{i+1} = \mathcal{P}(S_i, A_i)$, and rewards $R_i = \mathcal{R}(S_i)$ for each $i \in \{1, ..., T\}$ until a terminal state $S_T \in \mathcal{S}_F$ is reached.

[4]Oftentimes, a *discount factor* $\gamma \in (0, 1)$ is needed to ensure that the learning problem for the MDP is well-defined; however, in our setting, the MDP always terminates after a finite number of steps.

Then, rollout ζ is the sequence:

$$((S_0, A_0, R_0), ..., (S_{T-1}, A_{T-1}, R_{T-1}), (S_T, \varnothing, R_T))$$

Note that there is no action A_T for the last element since S_T is a terminal state.

As mentioned earlier, the objective in reinforcement learning is to maximize expected long-term reward:

Definition 17. Given an MDP $\mathcal{M}$, the *reinforcement learning problem* is to find the optimal policy $\pi^* = \arg\max_\pi \mathcal{R}^{(\pi)}$, where $\mathcal{R}^{(\pi)}$ denotes the *cumulative reward* of π:

$$\mathcal{R}^{(\pi)} = \mathbb{E}_{\zeta\sim\pi}\left[\sum_{i=0}^{T} R_i\right].$$

4.5.2 MDP for Relational Verification

To use reinforcement learning in our setting, we need to formulate an MDP $\mathcal{M}_{\text{proof}}$ encoding relational verification problems. Intuitively, given an (incomplete) proof strategy Υ, we want to learn a policy that chooses a proof rule $\mathcal{R}$ to apply to Υ that maximizes the chance of eventually constructing a successful (complete) proof strategy. Thus, states in our MDP are proof strategies, and actions are proof rules that can be applied to the current strategy. More formally, we construct the MDP $\mathcal{M}_{\text{proof}} = (\mathcal{S}, \mathcal{S}_0, \mathcal{S}_F, \mathcal{A}, \mathcal{P}, \mathcal{R})$ as follows:

- The states $\mathcal{S}$ are proof strategies Υ.

- $\mathcal{S}_0$ corresponds to initial proof stategies (recall Def. 10)
- The terminal states $\mathcal{S}_F$ are complete proof strategies.
- The actions $A \in \mathcal{A}$ are all pairs $(v, \mathcal{R})$, where $\mathcal{R}$ is a proof rule that can be applied to node v in the current proof strategy Υ.
- The (deterministic) transitions are $\mathcal{P}(S, A) = S'$, where S' is the proof strategy obtained from S by applying the proof rule A to the first open branch of S.
- The reward function is $\mathcal{R}(S) = \mathcal{O}(S)$ (i.e., the reward is 1 if S is successful and 0 otherwise).

Intuitively, the actions in $\mathcal{M}_{\text{proof}}$ incrementally construct a complete proof strategy $S_T \in \mathcal{S}_F$ from the initial proof strategy $S_0^{(t)}$, where $t \sim \mathcal{T}$ is a uniformly random task, and the reward is whether S_T is successful.

Example 4.5.1. Figure 4.4 shows an example of an MDP for a relational verification problem $\mathcal{G}_1$. Each state is a proof strategy Υ, and each action is a pair $(v, \mathcal{R})$ consisting of a node v in the current proof strategy and a proof rule $\mathcal{R}$ that can be applied to v. The initial state is the left-most state. For each action, an arrow shows the state transition that would occur if that action is taken. The right-most state on the top is a final state with reward 1 since it represents a successful proof strategy; all other states have reward 0.

Connection to our objective. Now, we describe the connection between the optimal policy for $\mathcal{M}_{\text{proof}}$ and the optimization problem from Eq. (4.1). First, we define a correspondence between distributions p over complete proof strategies and MDP policies π:

Definition 18. Given a policy π for $\mathcal{M}_{\text{proof}}$, its *terminal state distribution* is

$$p_\pi(\Upsilon) = \Pr_{\zeta\sim\pi}(S_T = \Upsilon),$$

where S_T is the terminal state of ζ.

In other words, $p_\pi(\Upsilon)$ is the probability that a rollout $\zeta \sim \pi$ ends in terminal state $S_T = \Upsilon$. Since the terminal states in $\mathcal{M}_{\text{proof}}$ are complete proof strategies, p_π is a distribution over complete proof strategies. Then, we have the following theorem, which relates the problem of maximizing Eq. (4.1) to the reinforcement learning problem for our MDP $\mathcal{M}_{\text{proof}}$:

Theorem 4. *Let π^* be the optimal policy for $\mathcal{M}_{proof}$, and*

$$p^* = \arg\max_p Pr_{t\sim\mathcal{T},\Upsilon\sim p^{(t)}}[\mathcal{O}(\Upsilon) = 1], \tag{4.2}$$

where p is a distribution over complete proof strategies. Then, we have $p^* = p_{\pi^*}$.

There is a key difference between our objective Eq. (4.1) and the objective Eq. (4.2) from Theorem 4: In Eq. (4.1), the probability is taken with respect to complete proof strategies $\Upsilon \sim \xi_{r,p}^{(t)}$ (i.e., the distribution of proof strategies tried by our search algorithm given guiding distribution $p^{(t)}$), whereas in Eq. (4.2), the probability is taken with respect to $\Upsilon \sim p^{(t)}$ (i.e., a single proof strategy according to $p^{(t)}$). In other words, our objective optimizes over a sequence of complete proof strategies tried by the search algorithm, whereas Eq. (4.2) optimizes for a single randomly sampled proof strategy. In Section 4.5.4, we describe how to modify an existing reinforcement learning algorithm to optimize Eq. (4.1) instead of Eq. (4.2).

4.5.3 Function Approximation

Recall that, when we only have a limited set of training tasks available, then the solution to Eq. (4.1) may not generalize well beyond tasks in the training set. As standard, we use approximate reinforcement learning to improve generalization power [164]. We first give background on the approximate RL and then describe our design choices within this framework

Background on approximate reinforcement learning. In approximate reinforcement learning, one needs to provide:

- A *feature map* $\phi : \mathcal{S} \to \mathcal{X}$, where $\mathcal{X} = \mathbb{R}^d$, which maps each state S to a feature vector $\phi(S)$ representing S.
- A *function family* $f_\theta : \mathcal{X} \to \mathcal{A}$, parameterized by $\theta \in \Theta = \mathbb{R}^m$, which maps feature vectors to actions.

Then, rather than search over all possible policies, the reinforcement learning algorithm restricts to policies of the form $f_\theta(\phi(S))$ (for $\theta \in \Theta$):

Definition 19. Given a feature map $\phi : \mathcal{S} \to \mathcal{X}$ and function family f_θ, the *approximate reinforcement learning problem* is to compute the optimal parameters

$$\theta^* = \arg\max_{\theta \in \Theta} \mathcal{R}^{(\theta)},$$

where $\mathcal{R}^{(\theta)} = \mathcal{R}^{(\pi_\theta)}$ and $\pi_\theta(S) = f_\theta(\phi(S))$.

In other words, the goal of approximate reinforcement learning is to find a policy within function family f_θ that maximizes expected cumulative reward.

In order for approximate reinforcement learning to be effective, the feature map ϕ must be constructed using domain expertise to balance two competing goals: First, given two states S and S', if the most promising actions to take in S and S' are similar, then we should have $\phi(S) \approx \phi(S')$. On the other hand, if the most promising actions are very different, then we should have $\phi(S) \not\approx \phi(S')$. Then, if the reinforcement learning algorithm learns the best actions to take in state S, this knowledge is automatically transferred to taking good actions in state S' (assuming smoothness of f_θ).

Approximating our objective. Given a feature map and function family, we can approximate Eq. (4.1) as follows. First, given parameters $\theta \in \Theta$, we have the following corresponding distribution over complete proof strategies:

Definition 20. Given parameters $\theta \in \Theta$, its *terminal state distribution* is $p_\theta = p_{\pi_\theta}$.

Then, rather than optimize over all distributions p, we restrict to optimizing over proof strategies of the form p_θ (for $\theta \in \Theta$):

$$\theta^* = \arg\max_{\theta \in \Theta} \Pr_{t \sim \mathcal{T}, S \sim \xi_{r,\theta}^{(t)}}[\mathcal{O}(S) = 1], \tag{4.3}$$

where $\xi_{r,\theta} = \xi_{r,p_\theta}$.

Feature map. In order to apply the approximation framework to our problem, we need to map each proof strategy to a feature vector. Since proof strategies are complex tree-structured objects involving many relational Hoare triples, our feature map grossly over-approximates the states in $\mathcal{M}_{\text{proof}}$. Specifically, we design $\phi(\Upsilon)$ to take into account both (a) the global aspects of Υ (e.g., depth and breadth of its tree structure, number of open/closed branches, etc.) as well as (b) local properties of the first open branch of Υ. For (b), suppose that the active open branch is labeled with the proof goal $\mathcal{G} = \langle \Phi \rangle \ S_1 \circledast S_2 \ \langle \Psi \rangle$. We featurize this relational Hoare triple by both considering which proof rules are (syntactically) applicable for discharging $\mathcal{G}$ and also performing a lightweight "diff" between S_1 and S_2. In particular, our differencing algorithm considers features such as whether both S_1, S_2 start with the same type of statement, whether they involve loops or recursive functions, the ratio between their trip count and step size (if both start with loops) etc. Thus, intuitively two strategies Υ_1 and Υ_2 will be deemed similar under ϕ if (a) their tree structures are similar, and (b) the same proof rule is likely to be successful for discharging the first open branches of Υ_1 and Υ_2.

Function family. In addition to the feature map, we also need a function family f_θ for mapping features (i.e., proof strategies) to actions (i.e., proof rules). For this, we use a standard choice in the reinforcement learning literature, namely the function family f_θ of neural networks with two (fully-connected) hidden layers and ReLU activations. Then, θ is the concatenation of all the weight and bias parameters of the layers in the neural network [144, 114, 173, 20].

4.5.4 Reinforcement Learning Algorithm

Recall from Section 4.5.2 that an optimal policy for our MDP does not yield an optimal solution to Eq. 4.1. In particular, standard RL maximizes expected reward under the assumption that we will explore a *single* rollout of the learned policy, whereas we want to maximize expected reward when exploring multiple rollouts during a backtracking search algorithm. Towards this goal, we describe a modified reinforcement learning algorithm that directly optimizes for our objective.

Our proposed optimization method builds on the *policy gradient algorithm*, which optimizes the cumulative reward $\mathcal{R}^{(\theta)}$ as a function of the policy parameters $\theta \in \Theta$ using stochastic gradient descent. There are two key reasons for building on top of the policy gradient algorithm: First, as we discuss in the rest of this chapter, policy gradient is easy to adapt to directly optimize Eq. (4.3). Second, because our feature vector $\phi(\Upsilon)$ grossly overapproximates Υ, we run into the so-called *perceptual aliasing* problem [41, 109], where two states that are different look the same under ϕ. In contrast to alternative algorithms like Q-learning, it is well-known that policy gradient works better in this scenario.

Background on policy gradient. The key challenge solved by the policy gradient algorithm is how to compute an estimate of the gradient $\frac{\mathrm{d}}{\mathrm{d}\theta}\mathcal{R}^{(\theta)}$. This algorithm is based on the the following well-known *policy gradient theorem* [165]:

Theorem 5. *We have*

$$\frac{\mathrm{d}}{\mathrm{d}\theta}\mathcal{R}^{(\theta)} = \mathbb{E}_{\zeta\sim\pi_\theta}[\ell(\zeta)],$$

where

$$\ell(\zeta) = \sum_{i=0}^{T-1}\left(\sum_{j=i+1}^{T} R_j\right)\frac{\mathrm{d}}{\mathrm{d}\theta}\log\pi_\theta(S_i, A_i).$$

In particular, the policy gradient theorem gives the gradient of the objective $\mathcal{R}^{(\theta)}$, making it possible to use gradient descent to optimize $\mathcal{R}^{(\theta)}$ as a function of θ.

Our algorithm. We now describe our algorithm for optimizing our objective in Eq. (4.3), i.e.,

$$J(\theta) = \Pr_{t\sim\mathcal{T}, S\sim\xi^{(t)}_{r,\theta}}[\mathcal{O}(S) = 1].$$

To solve this problem, we leverage additional structure of our search algorithm: Recall that, given guiding distribution p over complete proof strategies, our search algorithm initializes $p_0 = p^{(t)}$, and then iteratively constructs a sequence of distributions $p_0, p_1, p_2, ..., p_r$. As we describe in Section 4.6, this sequence of distributions corresponds to a sequence of policies $\pi_{\theta,0},\ \pi_{\theta,1},\ \pi_{\theta,2},\ \ldots,\ \pi_{\theta,r}$, where $p_i = p^{(t)}_{\pi_{\theta,i}}$. Then, we have the following theorem:

Theorem 6. *We have*

$$\frac{\mathrm{d}J}{\mathrm{d}\theta}(\theta) = \frac{1}{r}\sum_{i=1}^{r} \mathbb{E}_{\zeta\sim\pi_{\theta,i}}[\ell(\zeta)],$$

where $\ell(\zeta)$ is the same as in Theorem 5.

As in standard policy gradient, we can use known techniques [165] to approximate the gradient of $J(\theta)$ as follows:

$$\frac{\mathrm{d}J}{\mathrm{d}\theta}(\theta) \approx \frac{1}{r}\sum_{i=1}^{r}\frac{1}{n}\sum_{k=1}^{n}\hat{\ell}(\zeta^{(i,k)}),$$

where $\zeta^{(i,k)} \sim \pi_{i,\theta}$. Thus, we can use this approximate gradient in conjunction with gradient descent to compute the optimal parameters:

$$\theta^* = \arg\max_{\theta\in\Theta} J(\theta).$$

4.6 Policy-Guided Proof Search

In this section, we show how to use the optimal policy π synthesized using reinforcement learning to perform backtracking search over proof strategies. Towards this goal, a naïve approach would be to explore rollouts of π according to according to their probability of ending in a successful proof strategy. Essentially, such an approach corresponds to never updating the guiding distribution $p^{(t)}$, i.e., $p_{i+1} = p_i$.

However, the drawback of this approach is that failures may be correlated—two proof strategies Υ and Υ' that are very probable according to π may be unsuccessful for similar reasons. Even after discovering that Υ is unsuccessful, the naïve approach would not learn from this failure and would likely try Υ' on a subsequent iteration.

Our search algorithm addresses this issue by integrating logical learning into the statistical learning approach described in Section 4.5. Specifically, in addition to using the stochastic policy to guide the search, we also use the feedback provided by the verifier (i.e., CHC solver) to block proof strategies that are *guaranteed* to result in failure. The main idea is to analyze the root cause of an unsuccessful proof attempt and use this information to prune the search space. Based on these ideas, we propose a backtracking search algorithm that (a) uses the policy to decide which proof strategy to refine next, and (b) leverages feedback from the verifier to avoid the exploration of proof strategies that share the same root cause of failure as a previously explored one.

Our relational verification algorithm, called RelVerif, is shown in Algorithm 4. Given a relational Hoare triple $\mathcal{G}$ and the stochastic policy π learned from the training examples, RelVerif returns a successful proof strategy if one exists and $\perp$ otherwise. At a high level, the algorithm works as follows: It maintains a worklist W of (incomplete) proof strategies, which initially contains the unconstrained strategy $\Upsilon_0(\mathcal{G})$ (recall Def. 10). During each iteration, the algorithm invokes a procedure called `ChooseStrategy`, discussed in Section 4.6.1, to pick the most promising strategy according to policy π (line 5)

Algorithm 4 Policy-guided backtracking proof search

Input: $\mathcal{G}$ - target proof goal
Input: π - learned stochastic policy
Input: Δ - available proof rules
Output: A successful proof strategy for $\mathcal{G}$, or $\bot$ if it does not exist

```
1: procedure RelVerif($\mathcal{G}$, $\pi$, $\Delta$)
2:     $W \leftarrow \{\Upsilon_0(\mathcal{G})\}$                         ▷ worklist of proof strategies
3:     $B \leftarrow \emptyset$                                    ▷ blocked proof strategies
4:     while $W \neq \emptyset$ do
5:         $\Upsilon \leftarrow$ ChooseStrategy($\pi, W$)                       ▷ Use policy
6:         $W \leftarrow W \setminus \{\Upsilon\}$
7:         for $\mathcal{R}_i \in \Delta$ do
8:             if $\neg$Applies($\mathcal{R}_i, \Upsilon$) then
9:                 continue
10:            $\Upsilon_i \leftarrow$ ApplyProofRule($\Upsilon, \mathcal{R}_i$)
11:            if $\exists \Upsilon' \in B.\ \Upsilon_i \preceq \Upsilon'$ then
12:                continue
13:            if IsSuccessful($\Upsilon_i$) then return $\Upsilon_i$
14:            if IsFailing($\Upsilon_i$) then
15:                $B \leftarrow B \cup \{$ Minimize($\Upsilon_i$) $\}$
16:            else if $\neg$IsComplete($\Upsilon_i$) then
17:                $W \leftarrow W \cup \{ \Upsilon_i \}$
18:     return $\bot$
```

and constructs a series of refinements $\Upsilon_1, \ldots, \Upsilon_n$ by applying each one of the applicable proof rules $\mathcal{R}_i$ in the relational proof system Δ (line 10). If we are guaranteed that Υ_i is a failing strategy (i.e., Υ_i is a refinement of one of the *blocked strategies* B), then we move on the next proof rule without adding Υ_i to the worklist W (lines 11-12). On the other hand, if Υ_i is successful (i.e., it is complete and the corresponding CHCs are satisfiable), then we return Υ_i as a solution to the relational verification problem (line 13). Otherwise, if Υ_i is failing, we compute an unsatisfiable core of the VCs used in Υ_i and add the corresponding *minimal failing strategy* to the blocked strategies B (lines 14-15). As mentioned earlier, this blocking set B allows the algorithm to prune strategies that are guaranteed to be unsuccessful.

In what follows, we explain in more detail (a) how to use π to find the most promising proof strategy, and (b) how to compute minimal failing proof strategies.

4.6.1 Using policy to guide search

In order to use policy π to guide search, we need a suitable way to prioritize which states to explore first. Intuitively, we want our search algorithm to have two desired properties: First, complete proof strategies that have a higher probability of being successful according to p_π should be explored first. Second, the search must be exhaustive. That is, given a large enough time limit, the algorithm should return a successful proof strategy if one exists.

One straightforward way to utilize π is to use a *stochastic* search algorithm that repeatedly samples complete proof strategies according to the distribution given by p_π. However, implementing an efficient random sampling algorithm that guarantees exhaustiveness is a challenging task. Instead, we use a *deterministic* search algorithm that simply enumerates complete proof strategies in decreasing order of their probability according p_π. The intuition is that strategies that are more probable under p_π are more likely to lead to a successful proof; thus, they should be investigated first.

To ensure that the algorithm prioritizes complete strategies that correspond to more likely rollouts of π, we introduce a prioritization function ℓ_π as follows:

$$\ell_\pi(\Upsilon) = \begin{cases} 1 & \text{if } \Upsilon = \Upsilon_0(\mathcal{G}) \\ \ell_\pi(\Upsilon') - \log \pi(\Upsilon', \mathcal{R}) & \text{otherwise,} \end{cases}$$

where $\Upsilon =$ApplyProofRule$(\Upsilon', \mathcal{R})$.

Note that for a complete proof strategy Υ, we have $\ell_\pi(\Upsilon) = -\log p_\pi(\Upsilon)$. Thus, complete proof strategies that are more likely to be successful according to p_π are assigned a lower value according to ℓ_π.

Going back to Algorithm 4, the function `ChooseStrategy` simply uses the function ℓ_π to figure out which proof strategy to dequeue from W. In particular, `ChooseStrategy` dequeues the strategy with the lowest ℓ_π value.

Theorem 7. *Let Υ_1 and Υ_2 be two complete non-failing proof strategies. If $p_\pi(\Upsilon_1) > p_\pi(\Upsilon_2)$, then Υ_1 will be explored (i.e., dequed from W) before Υ_2 by Algorithm 4.*

4.6.2 Finding minimal failing strategies

To prevent the search algorithm from exploring failing strategies that share the same *root cause* of failure as previously explored ones, we introduce the following notion of *minimal failing proof strategy*:

Definition 21 (Minimal failing proof strategy). Given a failing proof strategy Υ, we say that Υ' is a minimally failing proof strategy of Υ if the following conditions hold:

- $\Upsilon \preceq \Upsilon'$

Algorithm 5 Failing strategy minimization

```
Input: $\Upsilon = (V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$ - a failing proof strategy
Output: The corresponding minimal failing proof strategy
1: procedure MINIMIZE($\Upsilon$)
2:     $\mathcal{U} \leftarrow \texttt{MinimalUnsatCore}(\{A_{\varphi}(v) \mid v \in V\})$
3:     $V_{\perp} \leftarrow \{v \mid A_{\varphi}(v) \in \mathcal{U}\}$
4:     $V_c \leftarrow \{v \in V \mid \exists u \in V_{\perp}.\ v \rightsquigarrow u\}$
5:     $V_f \leftarrow \{v \in V \mid v \notin V_c \wedge \exists u \in V_c.\ (u \rightarrow v) \in E\}$
6:     $V' \leftarrow V_c \cup V_f$
7:     $E' \leftarrow \{u \rightarrow v \mid u \in V' \wedge v \in V' \wedge (u \rightarrow v) \in E\}$
8:     $(A'_{\mathcal{R}}, A'_{\varphi}, A'_{\mathcal{G}}) \leftarrow (A_{\mathcal{R}} \downarrow V', A_{\varphi} \downarrow V', A_{\mathcal{G}} \downarrow V')$
9:     for $v \in V_f$ do
10:        $(A'_{\mathcal{R}},\ A'_{\varphi}) \leftarrow (A'_{\mathcal{R}}[v \leftarrow \perp],\ A'_{\varphi}[v \leftarrow \texttt{true}])$
11:    return $(V', E', A'_{\mathcal{R}}, A'_{\varphi}, A'_{\mathcal{G}})$
```

- Υ' is failing

- There does not exist $\Upsilon'' \neq \Upsilon'$ such that $\Upsilon' \preceq \Upsilon''$.

Essentially, a minimally failing proof strategy Υ' for Υ captures the root cause of failure in the sense that every proof rule in Υ' is necessary for generating an unsatisfisfiable system of CHCs in Υ. Thus, any proof strategy that refines Υ' is also guaranteed to fail and can be pruned from the search space without losing completeness.

Now, going back to Algorithm 4, we use a procedure called `Minimize` (line 15) to compute a minimally failing strategy. This procedure is summarized in pseudo-code in Algorithm 5. Given a failing strategy Υ, `Minimize` first computes a *minimal unsatisfiable core* of the VCs for Υ. More concretely, for a failing strategy $\Upsilon = (V, E, A_{\mathcal{G}}, A_{\mathcal{R}}, A_{\varphi})$, we compute a subset of nodes

$V_\perp \subseteq V$ such that $\bigwedge_{v \in V_\perp} A_\varphi(v)$ is unsatisfiable but for every $U \subset V_\perp$ we have $\bigwedge_{v \in U} A_\varphi(v)$ is satisfiable. Hence, $V_\perp$ has the following key properties:

- If we remove nodes that are not in $V_\perp$ from Υ, we still get a failing strategy.
- Removing any node in $V_\perp$ from Υ will make it not failing.

In other words, we can view $V_\perp$ as the root cause of failure for strategy Υ; thus, all nodes that are descendants of $V_\perp$ can be removed from Υ while preserving unsatisfiability. The `Minimize` algorithm essentially removes all nodes $V_\perp$ from Υ but adds open branches as necessary to ensure that the resulting proof strategy is a valid one that can be refined. [5]

The following theorem states that our search algorithm does not prune any successful proof strategies:

Theorem 8. *If there exists a complete proof strategy Υ for goal $\mathcal{G}$ such that $\bigwedge_{v \in V} A_\varphi(v)$ can be proven satisfiable by the underlying CHC solver, then Algorithm 4 will produce a proof of correctness of $\mathcal{G}$.*

4.7 Implementation

We have implemented the proposed ideas in a prototype called Coeus. Our tool takes as input two C programs and a relational property and outputs a successful proof strategy if the property can be verified.

[5]In Algorithm 5, the notation $v \rightsquigarrow u$ indicates that there is a path from v to u in Υ, and the notation $A \downarrow V$ yields a new mapping A' that is the same as A except that its domain is restricted to V.

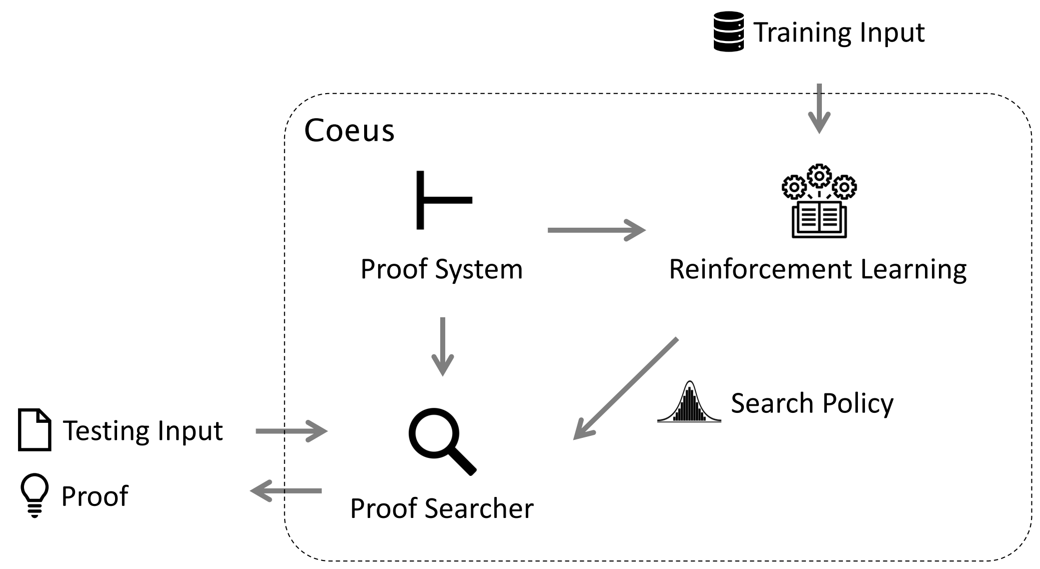


Figure 4.5: Coeus architecture

As depicted schematically in Figure 4.5, Coeus consists of three major components: First, the Proof System component serves as the basis of the entire system and implements the relational proof rules for reducing the relational verification task to standard safety. The Reinforcement Learning component implements the learning algorithm described in Section 4.5 and requires a set of representative training examples. Finally, the Proof Search component uses the learned policy to guide the search for successful proof strategies, as described in Section 4.6.

The Proof System and the Proof Search components are both implemented in OCaml and use the front-end of the CompCert compiler [102] for parsing the input C files. As mentioned in Section 4.2, our implementation uses a CHC solver to both (a) find relational loop invariants, and (b) discharge the resulting safety verification problems. For this purpose, our implementation leverages an enhanced version of the Spacer engine [94] distributed with the Z3 SMT solver [49]. [6]

The Reinforcement Learning module is implemented in Python and uses the TensorFlow library [2]. While our implementation largely follows the description from Section 4.5, it contains an optimization that is designed to speed up convergence during training. Specifically, rather than starting with a purely random policy, we bootstrap training using a policy learned via supervised learning. Specifically, supervised learning is conducted using exhaustive enumerative search on small, easy-to-solve training problems. The policy learned using supervised learning is far from optimal, but we have nonetheless found it useful for significantly speeding up the training process.

4.8 Evaluation

We evaluate the proposed approach by designing a series of experiments that address the following questions:

RQ1. How well does Coeus perform across different relational verification tasks?

RQ2. How does Coeus compare against state-of-the-art relational verification tools?

[6]Similar to the SeaHorn verifier [78], our implementation augments Spacer by incorporating a Houdini-style algorithm [62] for solving constrained Horn Clauses.

RQ3. What is the impact of learning compared to traditional proof search strategies?

RQ4. How important is it to combine the learned policy with backtracking search?

To answer these questions, we evaluate COEUS on two different benchmark suites and compare it against several baselines. For all experiments, we set a time limit of 300 seconds and a memory limit of 5GB for the proof search algorithm, and we set a time limit of 15 seconds per CHC solver invocation. All experiments are conducted on an Arch Linux workstation with an Intel Xeon E5-2630 CPU (2.6GHz) and 64GB of RAM.

4.8.1 Translation Validation Benchmarks

In our first experiment, we evaluate our approach in the context of *translation validation* [128]. Specifically, we use COEUS to check the correctness of various transformations performed by the ROSE compiler infrastructure [131] from the Lawrence Livermore Laboratory. Specifically, given the original C program P and its transformed version P', we use COEUS to prove equivalence between P and P'.

For the purposes of this experiment, we consider five (intra-procedural) transformation passes from the ROSE library. These transformations include loop unrolling, loop splitting, loop fission, constant propagation, and partial redundancy elimination. Given an original C program P, we obtain multiple transformed programs by applying all possible combinations of these transformations to P.

Training set. Recall that Coeus has an off-line training phase that is used for learning an optimal search policy via reinforcement learning. Towards this goal, we wrote a simple program generator that produces random, self-contained C functions. For each randomly generated program P, we obtain multiple transformed programs $P_1, \ldots, P_n$ as described above and use each (P, P_i) pair as a training example. Using this methodology, we trained Coeus on a total of 400 translation validation benchmarks.

Test set. The programs in our test set come from approximately 80 functions collected from popular Github repositories written in C. By applying various combinations of transformations to these functions and eliminating duplicates, we obtain a total of 153 benchmarks for our test set.

Results. Figure 4.6 summarizes the results of our evaluation on the translation validation domain. The x-axis shows the time limit per benchmark, and the y-axis shows the percentage of benchmarks that can be solved within that time limit. The different graphs in the figure correspond to the following several variants of Coeus:

- The blue line (with circles) is the full Coeus system.

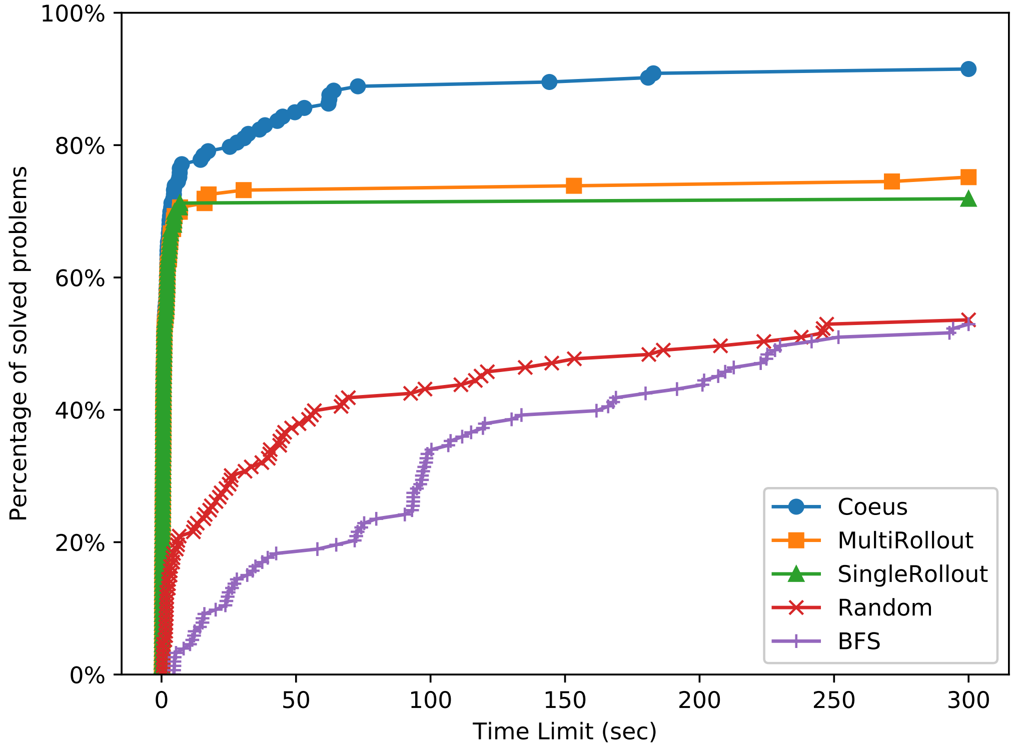


Figure 4.6: Comparison on translation validation benchmarks

- The orange and green lines (with squares and triangles respectively) correspond to variants of Coeus that use the learned policy but not our proposed search algorithm. Specifically, Single-Rollout only explores a single rollout of the learned policy and Multi-Rollout samples multiple rollouts until a time limit is reached.

- Both the red graph (with crosses) and the purple graph (with pluses) correspond to variants that do *not* use learning to guide search. The first variant (labelled Random) uses our search algorithm with a randomly generated policy, and the latter variant (BFS) uses breadth-first search.

One of the key conclusions to draw from Figure 4.6 is that learning-guided search significantly boosts the percentage of benchmarks that can be solved within a given time limit. In particular, both BFS and Random solve less than 56% of the benchmarks within a 5 minute time-limit whereas Coeus can solve 91.5% of the benchmarks within the same limit. The second important conclusion is that our proposed search algorithm allows us to effectively utilize the learned policy. Specifically, the Single-Rollout and Multi-Rollout variants plateau at 71.9% and 75.2% respectively, whereas Coeus can continue to solve more benchmarks as we increase the time limit.

Comparison against other tools. In addition to comparing Coeus against its own variants, we also compare it against two state-of-the-art relational verification tools, namely VeriMap [48] and a re-implementation of Descartes [161].

VeriMap is a relational verification tool that uses a method called *predicate pairing* for solving constrained Horn clauses that arise in relational proofs. In contrast, Descartes is based on the CHL program logic and performs heuristic-guided backtracking search over the CHL proof rules. Since the original version of Descartes is for Java programs, we re-implemented a version of Descartes for C that uses the same proof rules and search heuristics.

As summarized in Table 4.1, Coeus significantly outperforms both VeriMap and Descartes. Specifically, VeriMap can solve only 11% of these benchmarks within the 5-minute time limit . However, upon further inspection, the low success rate of VeriMap is caused, in part, by the benchmarks containing features (e.g., bitvectors, multi-dimensional arrays) that are not supported by this tool. If we exclude 130 out of 153 benchmarks that are not supported by VeriMap, the success rate increases to 73.9%. That is, VeriMap solves 17 out of these 23 benchmarks, whereas Coeus solves all. The success rate of Descartes on this benchmark set is around 50.3% compared to 91.5% for Coeus.

Bugs found in ROSE. During the process of running this experiment, Coeus uncovered two sources of unsoundness in the ROSE compiler. Specifically, since the accuracy of Coeus on the training set was initially lower than expected, we manually inspected the benchmarks that could not be verified using Coeus. Our inspection revealed two subtle bugs in the loop unrolling and fission transformation passes implemented in ROSE. Note that the results

	Translation Validation		
	Coeus	Descartes	VeriMap
Number of benchmarks	153		
Number of benchmarks supported by each tool	153	153	23
Number of solved benchmarks	140	77	17
Solved benchmarks / All benchmarks	91.5%	50.3%	11.1%
Solved benchmarks / Supported benchmarks	91.5%	50.3%	73.9%
Number of solved commonly supported benchmarks	23	20	17
Solved commonly suppored benchmarks / Commonly supported benchmarks	100%	87%	73.9%
Average running time for solved benchmarks (sec)	10.2	12.3	32.29

Table 4.1: Comparison with other relational verification tools on translation validation benchmarks .

shown in Figure 4.6 are obtained after fixing the loop unrolling bug and filtering out benchmarks that trigger the source of unsoundness in the loop fission pass. [7]

4.8.2 Medley of Relational Verification Benchmarks

In our second experiment, we evaluate our approach on a medley of relational verification benchmarks, where the properties to be checked include equivalence, inequality, monotonicity, injectivity, and others. The benchmarks used in this experiment come from two sources: First, we use all relational verification benchmarks evaluated in VeriMAP [48]. Second, we also collect pairs of independent solutions to programming challenge problems from LeetCode and HackerRank and use them to prove equivalence. Using this methodology, we obtain a total of 292 relational verification benchmarks. For this experiment, benchmarks were splitted into training vs. testing set based on their

[7]We did not fix the latter bug since it did not seem to admit an easy fix.

	Miscellaneous		
	Coeus	Descartes	VeriMap
Number of benchmarks	106		
Number of benchmarks supported by each tool	106	79	65
Number of solved benchmarks	89	46	35
Solved benchmarks / All benchmarks	84.0%	43.4%	33.0%
Solved benchmarks / Supported benchmarks	84.0%	58.2%	53.8%
Number of commonly supported benchmarks	52		
Number of solved commonly supported benchmarks	48	39	23
Solved commonly suppored benchmarks / Commonly supported benchmarks	92.3%	75%	44.2%
Average running time for solved benchmarks (sec)	23.9	17.4	66.52

Table 4.2: Comparison with other relational verification tools on miscellaneous benchmarks.

sizes. Specifically, programs whose size is smaller than a certain threshold were used for training whereas the larger programs were used for testing. Using this methodology, we obtained a training set of 186 benchmarks, and a testing set of 106 benchmarks.

Results. Figure 4.7 summarizes the results of our evaluation on this benchmark set. As in the previous subsection, the x-axis shows the time limit per benchmark, and the y-axis shows the percentage of benchmarks that can be solved within that time limit. Also as before, the different graphs from Figure 4.7 correspond to the Multi-Rollout, Single-Rollout, Random, and BFS variants of Coeus.

The trend we see in Figure 4.7 largely follows the one from Figure 4.6. Specifically, we observe that Coeus performs significantly better than both BFS and Random, highlighting the importance of guiding search using the

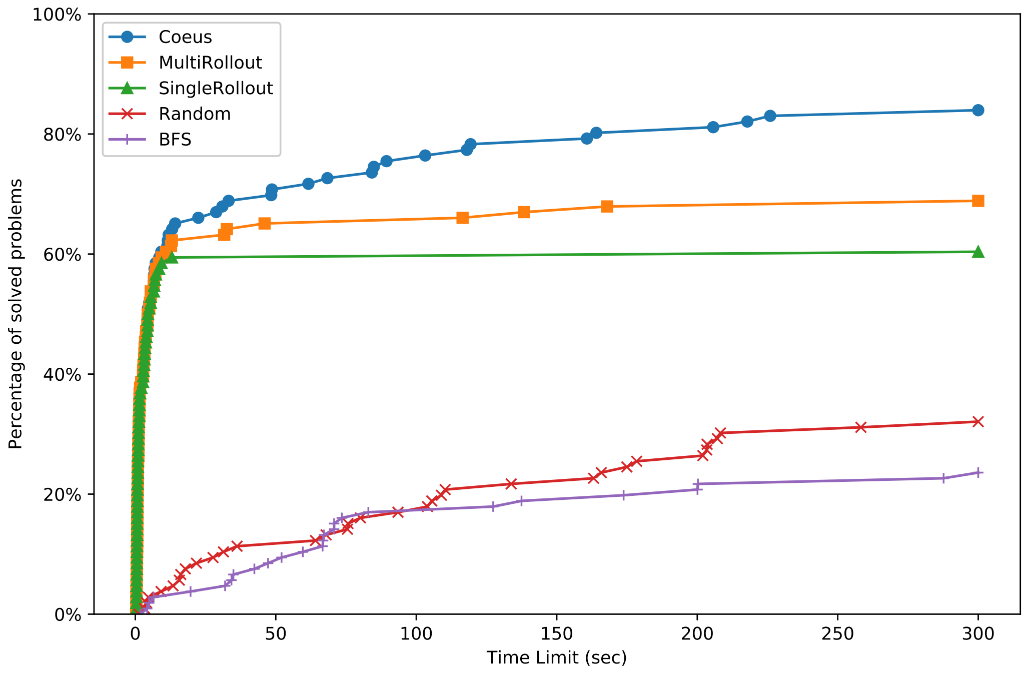


Figure 4.7: Comparison on benchmarks from [48] and equivalence checking problems obtained from solutions to exercises from LeetCode and HackerRank

RL-based policy. We also observe that Coeus can solve significantly more benchmarks compared to Single-Rollout and Multi-Rollout as we increase the time limit. This second observation again corroborates that our proposed policy-guided proof search algorithm from Section 4.6 allows us to use the policy much more effectively.

Comparison against VeriMap. As in Section 4.8.1, we also compare the performance of Coeus against Descartes and VeriMap on this benchmark set. As shown in Table 4.2, Coeus solves significantly more benchmarks. Specifically, VeriMap and Descartes are able to solve 35 and 46 of the 106 benchmarks respectively, whereas Coeus solves 89. However, if we exclude benchmarks that contain features not supported by both Descartes and VeriMap, we observe that Coeus solves 92.3% of the benchmarks whereas VeriMap solves 44.2% and Descartes solves 75%. We believe these results demonstrate that the proposed approach improves the state-of-the-art in relational verification.

Chapter 5

Related Works

In what follows, we discuss prior work that is most closely related to this dissertation.

Algorithmic complexity attacks Algorithmic complexity attacks have been actively studied during the past ten years. Some of these techniques target a specific class of vulnerabilities. For example, Crosby et al. [46] first presents a new class of algorithmic complexity attack that exploits the vulneralbility in hash tables. Wustholz et al. [172] designs a novel static analysis for automatically finding ReDoS vulnerabilities in Java programs. Cai et al. [35] intrduces a new algorithmic complexity attack that exploits data races in Unix file system. Smith et al. [158] explores NIDS (Network Intrusion Detection Systems) evasion through algorithmic complexity attacks by leveraging the pitfall in backtracking algorithms. Shenoy et al. [152] presents a performance throttling attack caused by an issue in the original string matching algorithm. Among approaches that target a broader class of AC vulnerabilities, SlowFuzz [127] is most closely related to the Singularity approach. In particular, SlowFuzz also uses evolutionary search for generating inputs but performs mutations at the byte level. In contrast, our method looks for input patterns rather than concrete inputs and can therefore scale better when large input sizes are required.

Performance bug detection. As demonstrated through our experiments in chapter 2.5, SINGULARITY can be useful for uncovering performance bugs. In this sense, our technique is related to a long line of work on performance bug detection. Most of these techniques target narrow classes of performance problems, such as redundant traversals [120, 121, 50, 124], loop inefficiencies [119, 51, 160], and unnecessary object creation [54]. Compared to these techniques, SINGULARITY can to detect a broader class of performance bugs but requires the user to decide whether the reported worst-case complexity corresponds to a performance bug.

Testing for performance There is a long line of work on automated testing techniques to uncover performance problems [34, 167, 182, 73, 130, 45, 178]. Among these prior techniques, WISE [34] is the first one to introduce the *complexity testing* problem, where the goal is to determine the complexity of a given program by constructing test cases that exhibit worst-case behavior. At a high level, WISE uses an optimized version of dynamic symbolic execution to guide the search towards execution paths with high resource usage. While WISE is a white-box testing technique, our approach is purely black-box and can scale to larger input sizes.

From a technical perspective, PerfSyn [167] is more similar to our approach in that it uses black-box evolutionary search to generate tests that cause performance bottlenecks. Specifically, PerfSyn starts with a minimal usage example of the method under test and applies a sequence of mutations that modify the original code. However, a key difference is that PerfSyn focuses on performance bottlenecks related to API usage, whereas our approach focuses on finding input patterns that trigger worst-case complexity.

Another idea related to performance testing is *empirical computational complexity* [73]. In particular, Goldsmith et al. propose a technique for measuring empirical complexity by running the program on workloads spanning several orders of magnitude in size and fitting these observations to a model that predicts performance as a function of input size. Since this technique requires the user to manually provide representative workloads, our approach is complementary to theirs.

Side channel attacks Side-channel attacks related to resource usage have been known for decades. Specifically, side channels have been used to leak private cryptographic keys [93, 4, 33], infer user accounts [31], steal cellphone and credit card numbers [68], obtain web browsing history [58], and recover the plaintext of encrypted TLS traffic [6]. Chen et al. presents a comprehensive study of side-channel leaks in web applications [39].

Verification for non-interference As mentioned in Section 3.4, we can prove that a program is free of side channel leaks by proving that it obeys a certain kind of non-interference property. There has been a significant body of work on proving non-interference. The simplest and most well-known technique for proving non-interference (and, in general, any 2-safety property) is *self-composition* [18]. The general idea underlying self-composition is as follows: Given a program P and 2-safety property ϕ, we create a new program P' which sequentially composes two α-renamed copies of P and then asserts that ϕ holds. Effectively, self-composition reduces verification of 2-safety to standard safety. While this self-composition technique is sound and relatively complete, successfully verifying the new program often requires the safety checker to come up with intricate invariants that are difficult to infer automatically [166]. Dufay et al. try to solve this problem by providing those invariants through JML annotations [53]; however, the resulting approach requires significant manual effort on the part of the developer or security analyst.

Another popular approach for proving k-safety is to construct so-called *product programs* [14, 16, 177]. Similar to self-composition, the product program method also reduces k-safety to standard safety by constructing a new program containing an assertion. While there are several different methods for constructing the product program, the central idea –shared in this work– is to execute the different copies of the program in lock step whenever possible. One disadvantage of this approach is that it can cause a blow-up in program size. As shown in the work of Sousa and Dillig [161], the product program approach can therefore suffer from scalability problems.

The approach advocated in this paper is most closely related to relational program logic, such as *Cartesian Hoare Logic* [161] and *Relational Hoare Logic* [24]. Specifically, the QCHL program logic introduced in Section 3.4.1 builds on top of CHL by instantiating it in the ϵ-bounded non-interference setting and augmenting it with additional rules for tracking resource usage and utilizing taint information. One advantage of this approach over explicit product construction is that we decompose the proof into smaller lemmas by constructing small product programs on-the-fly rather than constructing a monolithic program that is subsequently checked by an off-the-shelf verifier.

The approach described in this paper also shares similarities with the work of Terauchi and Aiken, in which they extend self-composition with *type-directed translation* [166, 117]. In particular, this technique uses a type system for secure information flow to guide product construction. Specifically, similar to our use of taint information to determine when two loops can be synchronized, Terauchi and Aiken use type information to construct a better product program than standard self-composition. Our verification technique differs from this approach in two major ways: First, our algorithm is not guided purely by taint information and uses other forms of semantic information (e.g., relational loop invariants) to determine when two loops can be executed in lock step. Second, similar to other approaches for product construction, the type-directed translation method generates a new program that is subsequently verified by an off-the-shelf verifier. In contrast, our method decomposes the proof into smaller lemmas by constructing mini-products on-the-fly, as needed.

Almeida et al. implement a tool named ct-verif based on aforementioned techniques (involving both product programs and self-composition) [7]. In particular, ct-verif is designed for verifying the *constant-time policy*, which roughly corresponds to our notion of 0-bounded non-interference instantiated with a timing cost model. In addition to using different techniques based on QCHL and taint analysis, THEMIS provides support for verifying a more general property, namely ϵ-bounded non-interference for any value of ϵ.

An alternative approach for verifying k-safety is the decomposition method used in BLAZER [9]: This method decomposes execution traces into different partitions using taint information and then verifies k-safety of the whole program by proving a standard safety property of each partition. One possible disadvantage of this approach is that, unlike our method and product construction techniques, BLAZER does not directly reason about the relationship between a pair of program executions. As illustrated through some of the examples in Section 3.4.1, such relational reasoning can greatly simplify the verification task in many cases.

In their recent work, Ngo et al. propose a language-based system for verifying and synthesizing synthesizes programs with *constant-resource usage*, meaning that every execution path of the program consumes the same amount of resource [118]. This technique uses a novel type system to reason both locally and globally about the resource usage bounds of a given program. Similar to work for verifying *constant-time policy*, this technique also does not allow proving ϵ-bounded non-interference for arbitrary values of ϵ. Furthermore, as a type-based solution for a functional language, this technique puts heavier annotation burden on the developer and is not immediately applicable to standard imperative languages like Java or C.

Secure information flow There has been a significant body of work on language-based solutions for enforcing information flow properties [180, 129, 116, 174]. For instance, Zhang et al. [180] propose a language-based approach that tracks side-channel leakage, and

Pottier et al. [129] design a type-based information flow analysis inside an ML-style language. Themis differs from these language-based solutions in that it requires minimal annotation effort and works on existing Java programs.

One of the most popular tools for tracking information flow in existing Java applications is FlowDroid [10], and Themis builds on top of FlowDroid to identify secret-tainted variables. FlowTracker [139] is another information flow analysis for C/C++ featuring efficient representation of implicit flow. We believe these techniques are complimentary to our approach, and a tool like Themis can directly benefit from advances in such static taint tracking tools.

There have also been attempts at verifying the constant-time policy directly using information-flow checking [13]. However, this approach is flow-insensitive (and therefore imprecise) and imposes a number of restrictions on the input program.

Automatic resource bound computation There has been a flurry of research on statically computing upper bounds for the resource usage of imperative programs. Existing techniques for this purpose leverage abstract interpretation [77], size-change abstraction [183], lossy vector addition systems [156], linear programming [36], difference constraints [157], recurrence relations [8, 64, 5], and term rewriting [32]. Another line of research, called AARA [86, 84, 82, 85, 83], performs bound analysis on functional languages.

THEMIS differs from these approaches in that we perform *relational* reasoning about resource usage. That is, rather than computing an upper bound on the resource usage of the program, we use QCHL to prove an upper bound on the *difference* between the resource usage of two program runs. Similar to our QCHL, recent work by Çiçek et al. performs *relational cost analysis* to reason about the difference in resource usage of a pair of programs [37]. Their work shares with us the insight that relational analysis may be simplified by exploiting the structural similarity between the inputs as well as the program codes. However, their non-relational reasoning relies on range analysis while THEMIS relies on Hoare-style weakest precondition computation; as a result THEMIS is more precise. Also, THEMIS analyzes real-world Java programs, while [37] is built on top of a hypothetical higher-order functional language.

Other defenses against side channels In this paper, we consider a purely static approach for detecting resource side channels. However, there are other possible ways of detecting vulnerabilities and preventing against side channel attacks. For instance, Bang et al. use symbolic execution and model counting to quantify leakage for a particular type of side channel [12]. Pasareanu et al. have recently implemented a symbolic execution based algorithm for generating inputs that maximize side channel measurements (namely timing and memory usage) [125]. Sidebuster [181] uses a hybrid static/dynamic analysis to detect side-channels based on irregularities in the One key advantage of our approach compared to these other techniques is that it can be used to verify the absence of side-channel vulnerabilities in programs.

There has also been a line of research that focuses for defending against side channels using runtime systems [108], compilers [113, 133, 134], or secure hardware [105]. Unlike these techniques, our approach does not result in runtime overhead.

Relational verification As stated in Section 4, relational verification problems are typically solved by reducing them to standard safety. Generally speaking, there are three different strategies for performing this reduction: Specifically, some approaches explicitly construct a product program that is safe iff the original relational verification problem is valid [18, 15, 17, 56]. Other approaches [24, 19, 38, 161] propose program logics for decomposing the relational verification task into a set of Hoare triples. Finally, some techniques [57, 48, 115] directly encode the relational verification problem as a set of constrained Horn clauses and propose new CHC solving techniques to deal with the resulting constraints [48, 115]. While these approaches define the space of strategies for reducing relational verification to safety checking, they do not propose algorithms for efficiently and intelligently exploring the large search space of different reduction strategies. In contast, the main contribution of this paper is to show how reinforcement learning can be used for effectively guiding the search for relational proofs.

Machine learning for program analysis. There has been a number of recent successes applying machine learning to programming languages research, for example, to infer program invariants [150, 148, 123], to improve program analysis [103, 107, 137, 132], program synthesis [142, 143, 11, 136, 60, 59, 91, 101], to build probabilistic models of code [138, 27, 135], to infer specifications [95, 106, 23, 81, 28, 21, 22], and to software testing [104, 42, 70]. There has also been a line of work on applying machine learning to theorem proving by predicting which lemmas might be useful to prove a given theorem, and then feeding those lemmas to a traditional theorem prover [87, 171]. However, these approaches treat the selection of promising lemmas as a one-shot problem rather than a sequential decision making problem. There has been work using machine learning (in particular, stochastic search) to perform equivalence checking [151]. Unlike our approach, theirs does not transfer knowledge across programs, and furthermore is tailored specifically to checking equivalence of two loops. Finally, there has been work using reinforcement learning to improve Polyhedral analysis [155]. In particular, they learn a policy for choosing parameters for approximating the join transformer. Unlike our setting, they observe rewards immediately, since they can directly assess the quality of a candidate approximation without regards to subsequent actions.

In contrast, we do not observe rewards until the very end of a rollout, which make reinforcement learning much more challenging. In particular, whereas the standard Q learning algorithm suffices in their setting, we use a variant of the policy gradient algorithm to address these challenges.

Chapter 6

Conclusion and Future Work

In this thesis, we discuss three programming analysis techniques: one dynamic fuzzing technique for algorithmic complexity vulnerability detection and two static verification techniques for resource usage side channel vulnerability. A common theme among all these solutions is to exploit problem-specific structures and adapt existing techniques to exploit those structures accordingly.

In the case of SINGULARITY, the critical observation we rely on is that worst-case inputs for a give program are often characterized by certain patterns, and for algorithmic complexity analysis pattern searching is more effective and scalable than directly searching for concrete inputs. Therefore, we adapt existing evolutionary technique for dynamic fuzzing by equipping it with a novel program representation, and shift the focus from fuzzing inputs to fuzzing this representation accordingly.

In the case of THEMIS, two key observations are: (1) Reasoning about cost difference is usually easier to automate than reasoning about cost values themselves directly. (2) Code that handles secret is typically small in large software projects. We therefore adapt existing off-the-shelf verifier with QCHL to fully utilize observation 1, and connect the verifier to a taint tracker based on observation 2.

In the case of Coeus, we observe that the sequences of proof rules that are most amenable to automation often follow certain patterns but sometimes deviate from them. Although these patterns can sometimes be hard to specify by a human, they can be empirically learned from data. The learned model can also provide us with valuable information to guide an exhaustive search, which effectively takes care of the aforementioned deviations if the model fails to predict the best strategy.

Looking to the future, this thesis opens up the following further research directions:

Applicability of Singularity. In chapter 2.5 we evaluate Singularity using a simple DSL with a small pool of components. Although our current selection of components is already good enough for a wise range of applications, certain problems (such as forcing hash collision on more sophisticated hash functions like MurmurHash and SipHash) seem to require more powerful components. Given a new problem domain, adapting the Singularity technique by designing new components that are likely to maximize the fuzzing efficiency will be an interesting challenge from a system's perspective.

Extending Themis with Coeus. Although we motivate the technique in Coeus with noninterference verification, we did not evaluate the technique in Themis' context. It would be interesting to prepare a sizable collection of noninterference verification benchmarks for Themis, and measure how Coeus could help in terms of false positive rates.

Coeus **on other domains.** The basic principle behind Coeus is not restricted to relational verification only: Any proof system with an proof-checking oracle, a fixed set of rules and a collection of training data can be a potential target where Coeus' principle may be used to improve the level of proof automation. Finding new domains and designing automation-friendly proof systems for it may help contributing to the adoption of formal methods in the real world.

Appendix

Appendix 1

Proofs of Selected Theorems

1.1 Proof of Theorem 3

Lemma 2. *Let program $P = \lambda\vec{p}.S$. If the following premises hold:*

- $\vec{p_1} = \alpha(\vec{p}), \vec{p_2} = \alpha(\vec{p})$
- $\models I \rightarrow \vec{p_1}^l = \vec{p_2}^l \wedge \vec{p_1}^h \neq \vec{p_2}^h$
- Σ *is sound*
- $\Sigma \vdash \mathit{CanSynchronize}(e_1, e_2, S_1, S_2, I)$

Then $\models I \rightarrow (e_1 \leftrightarrow e_2)$.

Proof. According to figure 3.6, if $\Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I)$, then at least one of the two conditions must be true:

- $\models I \rightarrow (e_1 \leftrightarrow e_2)$
- $e_1 \equiv_\alpha e_2 \wedge S_1 \equiv_\alpha S_2 \wedge \Sigma \vdash e_1 : \textbf{low} \wedge \Sigma \vdash e_2 : \textbf{low}$

If the first condition is true, then the conclusion trivially holds. Otherwise, since Σ is sound, we know that e_1 and e_2 depend solely on $\vec{p_1}^l$ and $\vec{p_2}^l$, respectively. According to the first two premises, $I \to \vec{p_1}^l = \vec{p_2}^l$. It follows that $I \to e_1 = e_2$ and therefore $\models I \to (e_1 \leftrightarrow e_2)$. $\square$

Lemma 3. *Let* $\texttt{vars}(S)$ *be the set of all free variables in* S*. If* $\texttt{vars}(S_1) \cap \texttt{vars}(S_2) = \emptyset$*, then* $S_1; S_2$ *is semantically equivalent to* $S_2; S_1$*.*

Proof. Suppose $\Gamma \vdash S_1; S_2 : \Gamma', r$. Since $\texttt{vars}(S_1)$ and $\texttt{vars}(S_2)$ are mutually disjoint, we could break Γ into three partitions $\Gamma = \Gamma_1 \sqcup \Gamma_2 \sqcup \Gamma_3$, where $dom(\Gamma_1) = \texttt{vars}(S_1)$, $dom(\Gamma_2) = \texttt{vars}(S_2)$ and $dom(\Gamma_3) = dom(\Gamma) \backslash \texttt{vars}(S_1) \backslash \texttt{vars}(S_2)$. Since S_i does not touch Γ_j where $i \neq j$, we have

$$\Gamma_1 \vdash S_1 : \Gamma'_1, r_1 \qquad \Gamma_2 \vdash S_2 : \Gamma'_2, r_2$$

It follows that

$$\Gamma \vdash S_1 : \Gamma'_1 \sqcup \Gamma_2 \sqcup \Gamma_3, r_1 \qquad \Gamma \vdash S_2 : \Gamma_1 \sqcup \Gamma'_2 \sqcup \Gamma_3, r_2$$

$$\Gamma'_1 \sqcup \Gamma_2 \sqcup \Gamma_3 \vdash S_2 : \Gamma'_1 \sqcup \Gamma'_2 \sqcup \Gamma_3, r_1 + r_2$$

$$\Gamma_1 \sqcup \Gamma'_2 \sqcup \Gamma_3 \vdash S_1 : \Gamma'_1 \sqcup \Gamma'_2 \sqcup \Gamma_3, r_2 + r_1$$

Using the operational semantics rule for sequential composition shown in figure 3.4, this means

$$\Gamma \vdash S_1; S_2 : \Gamma'_1 \sqcup \Gamma'_2 \sqcup \Gamma_3, r_1 + r_2$$

$$\Gamma \vdash S_2; S_1 : \Gamma'_1 \sqcup \Gamma'_2 \sqcup \Gamma_3, r_2 + r_1$$

As $S_1; S_2$ and $S_2; S_1$ both have the same effect on Γ and consume the same amount of resource, they are semantically equivalent. □

Lemma 4. *Let program $P = \lambda\vec{p}.S$. Under the assumption that the following premises hold:*

- $\vec{p_1} = \alpha(\vec{p}), \vec{p_2} = \alpha(\vec{p})$
- $\models \Phi \rightarrow \vec{p_1}^l = \vec{p_2}^l \wedge \vec{p_1}^h \neq \vec{p_2}^h$
- Σ *is sound*

If $\Sigma \vdash \langle\Phi\rangle\ S_1 \circledast S_2 \langle\Psi\rangle$, *then* $\vdash \{\Phi\}\ S_1; S_2\ \{\Psi\}$.

Proof. By structural induction on proof rules shown in figure 3.5.

- Rule (1).

  By inductive hypothesis, $\vdash \{\Phi\}S_2; S_1\{\Psi\}$. Since S_1 and S_2 belongs to two different alpha-renamed copies of the program, we have $\texttt{vars}(S_1) \cap \texttt{vars}(S_2) = \emptyset$. Using lemma 3, we get $\vdash \{\Phi\}S_1; S_2\{\Psi\}$

- Rule (2).

  By inductive hypothesis, $\vdash \{\Phi\}S_1; \textbf{skip}; S_2\{\Psi\}$. As $S_1; \textbf{skip}$ is semantically equivalent to S_1, we have $\vdash \{\Phi\}S_1; S_2\{\Psi\}$.

- Rule (3).

  By inductive hypothesis, $\vdash \{\Phi'\}S_2; S_3\{\Psi\}$. Also, we know $\{\Phi\}S_1\{\Phi'\}$. Using the sequence rule in standard Hoare logic, we derive $\vdash \{\Phi\}S_1; S_2; S_3\{\Psi\}$.

- Rule (4).

  By inductive hypothesis, $\vdash \{\Phi\}S\{\Psi\}$. As S is semantically equivalent to $S;\textbf{skip}$, we get $\{\Phi\}S;\textbf{skip}\{\Psi\}$.

- Rule (5).

  By inductive hypothesis, $\vdash \{\Phi\wedge e\}S_1;S;S_3\{\Psi_1\}$ and $\vdash \{\Phi\wedge\neg e\}S_2;S;S_3\{\Psi_2\}$. Since $\models \Psi_1 \rightarrow \Psi_1 \vee \Psi_2$ and $\models \Psi_2 \rightarrow \Psi_1 \vee \Psi_2$, according to the consequence rule in standard Hoare logic we have $\{\Phi \wedge e\}S_1;S;S_3\{\Psi_1 \vee \Psi_2\}$ and $\{\Phi \wedge \neg e\}S_2;S;S_3\{\Psi_1 \vee \Psi_2\}$. With the sequence rule in standard Hoare logic, assume

  1. $\vdash \{\Phi \wedge e\}S_1\{\Phi_1\}$
  2. $\vdash \{\Phi_1\}S;S_3\{\Psi_1 \vee \Psi_2\}$
  3. $\vdash \{\Phi \wedge \neg e\}S_2\{\Phi_2\}$
  4. $\vdash \{\Phi_2\}S;S_3\{\Psi_1 \vee \Psi_2\}$.

  Let $\Phi' = wp(\Psi_1 \vee \Psi_2)$. It follows immediately from (2) and (4) that $\Phi_1 \rightarrow \Phi'$ and $\Phi_2 \rightarrow \Phi'$. We could apply the consequence rule again to (1) and (3) and derive $\vdash \{\Phi \wedge e\}S_1\{\Phi'\}$ and $\vdash \{\Phi \wedge \neg e\}S_2\{\Phi'\}$. Using the condition rule in standard Hoare logic, we have $\{\Phi\}\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2\{\Phi'\}$. Combining (2), (4), sequence rule and the definition of wp, we could finally derive $\vdash \{\Phi\}\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2;S;S_3\{\Psi_1 \vee \Psi_2\}$.

- Rule (6).

  By inductive hypothesis, $\vdash \{\Psi'\}S;S'\{\Psi\}$. We also know that $\vdash \{\Phi\}\textbf{while } e_1 \textbf{ do } S_1\{\Phi'\}$

and $\vdash \{\Phi'\}\textbf{while } e_2 \textbf{ do } S_2\{\Psi'\}$. Applying the sequence rule in standard Hoare logic twice, we get $\vdash \{\Phi\}\textbf{while } e_1 \textbf{ do } S_1; \textbf{while } e_2 \textbf{ do } S_2; S; S'\{\Psi\}$. Additionally, S and $\textbf{while } e_2 \textbf{ do } S_2$ comes from two different alpha-renamed copies so $\texttt{vars}(S) \cap \texttt{vars}(\textbf{while } e_2 \textbf{ do } S_2) = \emptyset$. We could apply lemma 3 and get
$\vdash \{\Phi\}\textbf{while } e_1 \textbf{ do } S_1; S; \textbf{while } e_2 \textbf{ do } S_2; S'\{\Psi\}$

- Rule (7).

  By inductive hypothesis, $\vdash \{I \wedge e_1 \wedge e_2\}S_1; S_2\{I'\}$ and $\vdash \{I \wedge \neg e_1 \wedge e_2\}S; S'\{\Psi\}$. As $\models I' \rightarrow I$, we have $\vdash \{I \wedge e_1 \wedge e_2\}S_1; S_2\{I\}$ due to consequence rule. Now we may apply the while rule in standard Hoare logic to obtain $\vdash \{I\}\textbf{while } e_1 \wedge e_2 \textbf{ do } (S_1; S_2)\{I \wedge \neg(e_1 \wedge e_2)\}$.

  Now, as following two statements are semantically equivalent:

  - $\textbf{while } e_1 \wedge e_2 \textbf{ do } (S_1; S_2)$
  - $\textbf{while } e_1 \wedge e_2 \textbf{ do } (S_1; S_2); \textbf{while } e_1 \textbf{ do } S_1; \textbf{while } e_2 \textbf{ do } S_2$

  we could replace the former with the latter:

  $$\vdash \{I\}\textbf{while } e_1 \wedge e_2 \textbf{ do } (S_1; S_2); \textbf{while } e_1 \textbf{ do } S_1; \\ \textbf{while } e_2 \textbf{ do } S_2\{I \wedge \neg(e_1 \wedge e_2)\}$$

  According to lemma 2, $\models I \rightarrow (e_1 \leftrightarrow e_2)$. But we also know that the precondition $I \wedge \neg(e_1 \wedge e_2)$ holds before the second loop $\textbf{while } e_1 \textbf{ do } S_1$. This implies $I \neg e_1 \wedge \neg e_2$ and therefore both of the two loops $\textbf{while } e_1 \textbf{ do } S_1$ and $\textbf{while } e_2 \textbf{ do } S_2$ would not execute, which means $\vdash \{I \wedge \neg(e_1 \wedge$

$e_2)\}$**while** e_1 **do** $S_1;$**while** e_2 **do** $S_2\{I \wedge \neg(e_1 \wedge e_2)\}$. Applying the consequence rule here we end up with
$\vdash \{I\}$**while** e_1 **do** $S_1;$**while** e_2 **do** $S_2\{I \wedge \neg(e_1 \wedge e_2)\}$. Combining this with $\models \Phi \rightarrow I$ and the second inductive hypothesis we finally get
$\vdash \{\Phi\}$**while** e_1 **do** $S_1; S;$**while** e_2 **do** $S_2; S'\{\Psi\}$.

□

Theorem 9 (Soundness). *Assuming soundness of taint environment Σ, if $\Sigma \vdash SideChannelFree(\lambda\vec{p}.S,\ \epsilon)$, then the program $\lambda\vec{p}.S$ does not have an ϵ-bounded resource side-channel.*

Proof. We know that Σ is sound and $\models \Phi \rightarrow \vec{p_1}^l = \vec{p_2}^l \wedge \vec{p_1}^h \neq \vec{p_2}^h$. Therefore, lemma 4 applies, and we get $\vdash \{\Phi\}S_1^\tau; S_2^\tau\{\Psi\}$. Additionally, $\models \Psi \rightarrow |\tau_1 - \tau_2| \leq \epsilon|$. Using the consequence rule in standard Hoare logic, we obtain $\vdash \{\Phi\}S_1^\tau; S_2^\tau\{|\tau_1 - \tau_2| \leq \epsilon|\}$. By the soundness of Hoare logic, it follows that $\models \{\Phi\}S_1^\tau; S_2^\tau\{|\tau_1 - \tau_2| \leq \epsilon|\}$. By the soundness of self-composition, this means that

$$\forall \vec{a_1}, \vec{a_2}.\ \vec{a_1}^l = \vec{a_2}^l \wedge \vec{a_1}^h \neq \vec{a_2}^h \implies |\tau_1 - \tau_2| \leq \epsilon$$

By lemma 1, $\tau_1 = R_P(\vec{a_1})$ and $\tau_2 = R_P(\vec{a_2})$. Substitute τ with R_P we arrive at our conclusion

$$\forall \vec{a_1}, \vec{a_2}.\ (\vec{a_1}^l = \vec{a_2}^l \wedge \vec{a_1}^h \neq \vec{a_2}^h) \implies |R_P(\vec{a_1}) - R_P(\vec{a_2})| \leq \epsilon$$

□

1.2 Proof of Theorem 4

First, we show that the mapping from policies π to distributions $p^{(\pi)}$ is invertible:

Lemma 5. *Given a distribution p over complete proof strategies, we have $p^{(\pi)} = p$, where*

$$\pi(S, A) = \frac{\sum_{S' \preceq^* P(S,A)} \tilde{p}(S')}{\sum_{S' \preceq^* S} \tilde{p}(S')}.$$

Proof. First, because transitions are deterministic, we have

$$p^{(\pi)}(S) \propto \sum_{\zeta} \mathbb{I}[S_T = S] \cdot p(S_0 \mid \mathcal{S}_0) \cdot \prod_{i=0}^{T-1} \pi(S_i, A_i),$$

where $p(S_0 \mid \mathcal{S}_0)$ is the probability that the initial state is S_0. Furthermore, note that there is a unique way of constructing any given complete proof strategy $S \in \mathcal{S}_F$ using actions $A \in \mathcal{A}$. Letting $\zeta_S = ((S_0, A_0, R_0), ..., (S_T, \varnothing, R_T))$ denote the unique rollout with terminal state $S_T = S$, we have

$$p^{(\pi)}(S) = p(S_0 \mid \mathcal{S}_0) \cdot \prod_{i=0}^{T-1} \pi(S_i, A_i).$$

Expanding the right-hand side, we have

$$\begin{aligned} p^{(\pi)}(S) &= p(S_0 \mid \mathcal{S}_0) \cdot \prod_{i=0}^{T-1} \frac{\sum_{S' \preceq^* P(S_i,A_i)} p(S')}{\sum_{S' \preceq^* S_i} p(S')} \\ &= p(S_0 \mid \mathcal{S}_0) \cdot \prod_{i=0}^{T-1} \frac{\sum_{S' \preceq^* S_{i+1}} p(S')}{\sum_{S' \preceq^* S_i} p(S')} \\ &= p(S_0 \mid \mathcal{S}_0) \cdot \frac{\sum_{S' \preceq^* S_T} p(S')}{\sum_{S' \preceq^* S_0} p(S')}. \end{aligned}$$

Note that the numerator of the last line equals $p(S_T)$, since the only S' such that $S' \preceq^* S_T$ for a complete state S_T is S_T itself. Similarly, the denominator equals $p(S_0 \mid \mathcal{S}_0)$, since the sets of states $\{S' \mid S' \preceq^* S_0\}$ are disjoint for different initial states S_0. In other words, $p^{(\pi)}(S) = p(S)$, as claimed. $\square$

As a consequence, the space over policies (which reinforcement learning algorithms optimize over) and the space of distributions (which (4.1) optimizes over) are equal. Next, we prove that given a policy π, its cumulative reward of π equals the objective (4.1) evaluated at $\tilde{p} = p^{(\pi)}$:

Lemma 6. *For any policy π for $\mathcal{M}_{proof}$, we have*

$$\mathcal{R}^{(\pi)} = Pr_{t\sim\mathcal{T}, S\sim p_t^{(\pi)}}[\mathcal{O}(S)],$$

where $p_t^{(\pi)}$ is the distribution $p^{(\pi)}$ conditioned on task t:

$$p_t^{(\pi)} = p^{(\pi)} \mid S \textit{ is labeled with the initial proof goal for } t.$$

Proof. Note that since complete proofs are terminal states, and we only obtain reward on successful proofs (which are complete by definition). Thus, we have

$$\begin{aligned}\mathcal{R}^{(\pi)} = \mathbb{E}_{\zeta\sim\pi}\left[\sum_{i=0}^{T} R_i\right] &= \mathbb{E}_{\zeta\sim\pi}[R_T] \\ &= \mathbb{E}_{\zeta\sim\pi}[\mathcal{O}(S_T)].\end{aligned}$$

Finally, by definition, the distribution of S_T given a randomly sampled rollout $\zeta \sim \pi$ equals the distribution $p^{(\pi)}$, so

$$\mathcal{R}^{(\pi)} = \Pr_{S\sim p^{(\pi)}}[\mathcal{O}(S)],$$

as claimed. $\square$

The proof of Theorem 4 follows from Lemma 5 and Lemma 6.

1.3 Proof of Theorem 6

We can rewrite the objective $J(\theta)$ of (4.3) as follows:

Lemma 7. *We have*

$$J(\theta) = \frac{1}{r+1}\sum_{i=0}^{r} \mathcal{R}^{(\pi_{\theta,i})}.$$

Proof. Note that

$$\xi_{r,\theta}^{(t)}(S) = \frac{1}{r+1}\sum_{i=0}^{r} p_i,$$

where $p_i = p_{\pi_{\theta,i}}$. Thus, we have

$$\begin{aligned}
J(\theta) &= \Pr_{t\sim\mathcal{T},S\sim\xi_{r,\theta}^{(t)}}[\mathcal{O}(S)=1] \\
&= \mathbb{E}_{t\sim\mathcal{T},S\sim\xi_{r,\theta}^{(t)}}[\mathcal{O}(S)] \\
&= \mathbb{E}_{t\sim\mathcal{T}}\left[\frac{1}{r+1}\sum_{i=0}^{r}\mathbb{E}_{S\sim p_{\pi_{\theta,i}}^{(t)}}[\mathcal{O}(S)]\right] \\
&= \frac{1}{r+1}\sum_{i=0}^{r}\mathbb{E}_{t\sim\mathcal{T},S\sim p_{\pi_{\theta,i}}^{(t)}}[\mathcal{O}(S)] \\
&= \frac{1}{r+1}\sum_{i=0}^{r}\mathbb{E}_{S\sim p_{\pi_{\theta,i}}}[\mathcal{O}(S)] \\
&= \frac{1}{r+1}\sum_{i=0}^{r}\mathbb{E}_{\zeta\sim\pi_{\theta,i}}[\mathcal{O}(S_T)] \\
&= \frac{1}{r+1}\sum_{i=0}^{r}\mathcal{R}^{(\pi_{\theta,i})},
\end{aligned}$$

as claimed. □

Now, let τ denote the function by which our search algorithm constructs $\pi_{\theta,i+1}$ from $\pi_{\theta,i}$, i.e.,

$$\pi_{\theta,i} = \begin{cases} \pi_\theta & \text{if } i = 0 \\ \tau(\pi_{\theta,i-1}, \theta) & \text{otherwise.} \end{cases}$$

Then, consider the derivative of τ with respect to θ:

$$\frac{\mathrm{d}\tau}{\mathrm{d}\theta}(\pi, \theta) = \frac{\partial \tau}{\partial \pi}(\pi, \theta)\frac{\mathrm{d}\pi}{\mathrm{d}\theta} + \frac{\partial \tau}{\partial \theta}(\pi, \theta),$$

where the gradient with respect to π is the gradient with respect to the probabilities $\pi(S, A)$ of taking action A in state S. We have the following important fact about τ:

Lemma 8. *We have*

$$\frac{\partial \tau}{\partial \pi}(\pi, \theta) = 0,$$

except on a measure zero subset.

Proof. (sketch) Our search algorithm constructs $\pi_{\theta,i}$ from $\pi_{\theta,i-1}$ by first constructing the most probable rollout $\zeta_{\max}$ according to $\pi_{\theta,i-1}$, and constructing $\pi_{\theta,i}$ deterministically from $\zeta_{\max}$ and θ, i.e.,

$$\pi_{\theta,i} = \tilde{\tau}(\zeta_{\max}, \theta).$$

In other words, $\tau(\pi, \theta) = \tilde{\tau}(\zeta_{\max}, \theta)$, where $\zeta_{\max}$ is the most probable rollout according to π. However, note that $\zeta_{\max}$ is from a discrete set. Therefore, for fixed θ, τ must be a piecewise constant function of π, so the claim follows. $\square$

Intuitively, this lemma says that the way in which we construct the sequence of policies $\pi_{\theta,0}, \pi_{\theta,1}, ...$ is not affected by small changes to θ. An important consequence is that

$$\frac{\mathrm{d}\tau}{\mathrm{d}\theta}(\pi, \theta) = \frac{\partial\tau}{\partial\theta}(\pi, \theta).$$

Finally, Theorem 6 follows directly from Lemma 7, Theorem 5, and Lemma 8.

1.4 Proof of Theorem 7

First, we need to introduce the notion of the *length* of a proof strategy.

Definition 22. The length of a proof strategy Υ, written as $\mathcal{L}(\Upsilon)$, is defined as follows:

- For any proof goal $\mathcal{G}$, $\mathcal{L}\left(\Upsilon_0(\mathcal{G})\right) = 0$.
- If $\Upsilon \preceq^1 \Upsilon'$, then $\mathcal{L}(\Upsilon) = 1 + \mathcal{L}(\Upsilon')$.

Intuitively, proof length keeps track of how many proof rules have been applied. **In this paper, we only consider proof strategies of finite length.**

Lemma 9. *Given two proof strategies Υ_1 and Υ_2, if $\Upsilon_1 \preceq \Upsilon_2$, then $\ell_\pi(\Upsilon_1) \geq \ell_\pi(\Upsilon_2)$.*

Proof. The lemma can be proved by induction on the difference of length between Υ_1 and Υ_2. □

Lemma 10. *If a proof strategy Υ is non-failing, then for all strategy Υ' such that $\Upsilon \preceq \Upsilon'$, Υ' is non-failing.*

Proof. This lemma follows directly from the definition 13: for $\Upsilon = (V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$ and $\Upsilon' = (V', E', A'_{\mathcal{R}}, A'_{\varphi}, A'_{\mathcal{G}})$, if $\Upsilon \preceq \Upsilon'$, then $\bigwedge_{v \in V'} A'_{\varphi}(v)$ contains strictly less clauses than $\bigwedge_{v \in V} A_{\varphi}(v)$. If the latter is satisfiable, the former must also be satisfiable as it is strictly weaker. $\square$

We now prove Theorem 7 by contradiction. Let Υ_1 and Υ_2 be two complete non-failing proof strategies and $p_\pi(\Upsilon_1) > p_\pi(\Upsilon_2)$ (thus $\ell_\pi(\Upsilon_1) < \ell_\pi(\Upsilon_2)$). Suppose Υ_2 gets dequeued from W before Υ_1 on line 5 in Algorithm 4. Since `ChooseStrategy` always picks the strategy with the smallest value of ℓ_π, we know that Υ_1 must not be in W when Υ_2 gets dequeued.

We now consider the "predecessors" of Υ_1 in the search algorithm, i.e. $\mathcal{P} = \{\Upsilon^* | \Upsilon_1 \preceq \Upsilon^*\}$. We know that Υ_1 is non-failing, so according to Lemma 10 strategies in $\mathcal{P}$ will also be non-failing and thus will not be blocked by B on line 11 to 12. Since all proof strategies explored in Algorithm 4 refines the initial strategy $\Upsilon_0(\mathcal{G})$ for the initial goal $\mathcal{G}$, and the initial strategy is enqueued into W on line 2, there must exist one $\Upsilon^* \in \mathcal{P}$ such that Υ^* is in W when Υ_2 is dequeued.

According to lemma 9, we have $\ell_\pi(\Upsilon^*) \leq \ell_\pi(\Upsilon_1)$. Hence $\ell_\pi(\Upsilon^*) < \ell_\pi(\Upsilon_2)$, which means that when Υ^* and Υ_2 are both in W, Υ^* will be dequeued first. This contradicts our earlier assumption that Υ_2 is dequeued before Υ^*.

1.5 Proof Theorem 8

We only need to prove the proposition that when the function `RelVerif`$(\mathcal{G}, \pi, \Delta)$ returns $\perp$, every non-failing strategy must have been checked for successfulness on by Algorithm 4 on line 13. Theorem 8 is a direct corollary of this proposition.

The proof can be carried out by induction on the length of the non-failing strategies.

- When $\mathcal{L}(\Upsilon) = 0$, $\Upsilon = \Upsilon_0(\mathcal{G})$. The conclusion hold trivially as the initial strategy is guaranteed to reach line 13 in the first iteration of the for loop.

- Assume the proposition holds for non-failing strategies with length $n - 1$ where $n \geq 1$.
  Let $\Upsilon =$ `ApplyProofRule`$(\Upsilon', \mathcal{R})$ with $\mathcal{L}(\Upsilon) = n$. By inductive hypothesis we know that line 13 must have been reached with $\Upsilon_i = \Upsilon'$ before. As Υ' is both not failing and not complete by definition, line 17 will be reached and Υ' will be enqueued in W.
  Now consider the iteration when Υ' gets dequeued at line 5. Line 10 is guaranteed be reached with $\Upsilon_i = \Upsilon$. Since Υ is also non-failing, it will not be blocked by B on line 11 to 12. Therefore, Υ will be checked for successfulnesss on line 13 as well.

Bibliography

[1] A timing channel in jetty. `https://github.com/eclipse/jetty.project/commit/2baa1abe4b1c380a30deacca1ed367466a1a62ea`, 2017.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320. ACM, 2007.

[4] Onur Aciiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on

openssl. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'08, pages 256–273. Springer-Verlag, 2008.

[5] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. Non-cumulative resource analysis. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pages 85–100. Springer-Verlag New York, Inc., 2015.

[6] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540, 2013.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 53–70, 2016.

[8] Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 405–421. Springer-Verlag, 2012.

[9] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition

for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 362–375, New York, NY, USA, 2017. ACM.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269, 2014.

[11] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2016.

[12] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 193–204. ACM, 2016.

[13] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptogra-

phy. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1267–1279. ACM, 2014.

[14] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 200–214, 2011.

[15] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *International Symposium on Formal Methods*, pages 200–214. Springer, 2011.

[16] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*, volume 7734, pages 29–43. Springer, 2013.

[17] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Product programs and relational program logics. *Journal of Logical and Algebraic Methods in Programming*, 85(5):847–859, 2016.

[18] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[19] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *ACM SIGPLAN Notices*, volume 47, pages 97–110. ACM, 2012.

[20] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *NIPS*, 2018.

[21] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *PLDI*, volume 52, pages 95–110. ACM, 2017.

[22] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 678–692. ACM, 2018.

[23] Nels E Beckman and Aditya V Nori. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*, volume 46, pages 211–221. ACM, 2011.

[24] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25. ACM, 2004.

[25] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, January 1985.

[26] Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. In *AFL*, 2014.

[27] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942, 2016.

[28] Pavol Bielik, Veselin Raychev, and Martin Vechev. Learning a static analyzer from data. In *International Conference on Computer Aided Verification*, pages 233–253. Springer, 2017.

[29] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, pages 24–51. Springer, 2015.

[30] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 775–788, New York, NY, USA, 2016. ACM.

[31] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *World Wide Web*, pages 621–628. ACM, 2007.

[32] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer pro-

grams. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, August 2016.

[33] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[34] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 463–473, Washington, DC, USA, 2009. IEEE Computer Society.

[35] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting unix file-system races via algorithmic complexity attacks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 27–41, 2009.

[36] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 467–478. ACM, 2015.

[37] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 316–329. ACM, 2017.

[38] Jia Chen, Yu Feng, and Isil Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 875–890. ACM, 2017.

[39] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 191–206, 2010.

[40] Sujit Rokka Chhetri and Mohammad Abdullah Al Faruque. Side-channels of cyber-physical systems: Case study in additive manufacturing. *IEEE Design & Test*, 2017.

[41] Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *AAAI*, volume 1992, pages 183–188. Citeseer, 1992.

[42] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. Minimizing gui event traces. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 422–434. ACM, 2016.

[43] Justin Clarke. *SQL Injection Attacks and Defense.* Syngress Publishing, 1st edition, 2009.

[44] CNN. Widespread cyberattack takes down sites worldwide. `https://tinyurl.com/ycovsk7d`.

[45] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 89–98, New York, NY, USA, 2012. ACM.

[46] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[47] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Horn clause transformation for program verification. Technical report, 2016.

[48] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Relational verification through horn clause transformation. In *International Static Analysis Symposium*, pages 147–169. Springer, 2016.

[49] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[50] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities.

In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 607–622, New York, NY, USA, 2015. ACM.

[51] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 895–907, New York, NY, USA, 2016. ACM.

[52] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 435–445, New York, NY, USA, 2007. ACM.

[53] Guillaume Dufay, Amy Felty, and Stan Matwin. Privacy-sensitive information flow with jml. In *Proceedings of the 20th International Conference on Automated Deduction*, CADE' 20, pages 116–130. Springer-Verlag, 2005.

[54] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 118–128, New York, NY, USA, 2007. ACM.

[55] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded

model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, Baltimore, MD, 2018. USENIX Association.

[56] Marco Eilers, Peter Müller, and Samuel Hitz. Modular product programs. In *European Symposium on Programming*, pages 502–529. Springer, 2018.

[57] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 349–360, 2014.

[58] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 1-4, 2000.*, pages 25–32, 2000.

[59] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 420–435. ACM, 2018.

[60] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI*, volume 52, pages 422–436. ACM, 2017.

[61] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517. Springer-Verlag, 2001.

[62] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.

[63] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[64] Antonio Flores-Montoya and Reiner Hähnle. *Resource Analysis of Complex Programs with Cost Equations*, pages 275–295. Springer International Publishing, Cham, 2014.

[65] Riccardo Focardi and Roberto Gorrieri. A classification of security properties for process algebras1. *Journal of Computer security*, 3(1):5–33, 1995.

[66] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and

Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[67] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '01, pages 251–261. Springer-Verlag, 2001.

[68] Nethanel Gelernter and Amir Herzberg. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1394–1405. ACM, 2015.

[69] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, pages 242–260, 2014.

[70] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

[71] Jan Goguen and Meseguer Jose. Security policies and security models. In *Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[72] A V Goldberg and R E Tarjan. A new approach to the maximum flow problem. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 136–146, New York, NY, USA, 1986. ACM.

[73] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 395–404. ACM, 2007.

[74] Google. Google core libraries for java. `https://github.com/google/guava`.

[75] James W Gray III. Toward a mathematical foundation for information flow security. *Journal of Computer Security*, 1(3-4):255–294, 1992.

[76] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505, 2011.

[77] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 127–139. ACM, 2009.

[78] Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. Seahorn: A framework for verifying c programs (competition contribution). In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–450, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[79] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2):83–114, October 2005.

[80] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. Differential privacy under fire. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011.

[81] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: automatically learning the x86-64 instruction set. In *PLDI*, volume 51, pages 237–250. ACM, 2016.

[82] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, November 2012.

[83] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 359–373. ACM, 2017.

[84] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 287–306. Springer-Verlag, 2010.

[85] Jan Hoffmann and Zhong Shao. *Type-Based Amortized Resource Analysis with Integers and Arrays*, pages 152–168. Springer International Publishing, 2014.

[86] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 185–197. ACM, 2003.

[87] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath-deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2016.

[88] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 661–667. Springer-Verlag, 2009.

[89] JGraphT. A free java graph library. `http://jgrapht.org/`.

[90] Michel Kaempf. Vudo - An object superstitiously believed to embody magical powers. *Phrack Magazine*, 57(8), 2001.

[91] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *ICLR*, 2018.

[92] Paul Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[93] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 104–113, 1996.

[94] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, June 2016.

[95] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176, 2006.

[96] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative

programs. In *International Conference on Computer Aided Verification*, pages 712–717. Springer, 2012.

[97] Shuvendu K Lahiri, Kenneth L McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 345–355. ACM, 2013.

[98] Shuvendu K. Lahiri and Shaz Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *Proceedings of the 22Nd International Conference on Automated Deduction*, CADE-22, pages 214–229. Springer-Verlag, 2009.

[99] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–. IEEE Computer Society, 2004.

[100] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[101] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In

Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 436–449. ACM, 2018.

[102] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[103] Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *POPL*, volume 46, pages 31–42. ACM, 2011.

[104] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. 40(6):15–26, 2005.

[105] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 87–101. ACM, 2015.

[106] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. volume 44, pages 75–86. ACM, 2009.

[107] Ravi Mangal, Xin Zhang, Aditya V Nori, and Mayur Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 462–473. ACM, 2015.

[108] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129. IEEE Computer Society, 2012.

[109] R Andrew McCallum. Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 190–196, 1993.

[110] Kurt Mehlhorn and Stefan Näher. *LEDA: a platform for combinatorial and geometric computing*. Cambridge university press, 1999.

[111] mitre. Cve-2011-5021. `https://nvd.nist.gov/vuln/detail/CVE-2011-5021`.

[112] mitre. Cve-2015-3192. `https://pivotal.io/security/cve-2015-3192`.

[113] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, ICISC'05, pages 156–168. Springer-Verlag, 2006.

[114] William H Montgomery and Sergey Levine. Guided policy search via approximate mirror descent. In *Advances in Neural Information Processing Systems*, pages 4008–4016, 2016.

[115] Dmitry Mordvinov and Grigory Fedyukovich. Synchronizing constrained horn clauses. *LPAR, EPiC Series in Computing. EasyChair*, 2017.

[116] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005, 2001.

[117] David A. Naumann. *From Coupling Relations to Mated Invariants for Checking Information Flow*, pages 279–296. Springer Berlin Heidelberg, 2006.

[118] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 710–728, May 2017.

[119] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 902–912. IEEE Press, 2015.

[120] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571. IEEE Press, 2013.

[121] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 369–378, New York, NY, USA, 2015. ACM.

[122] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1998.

[123] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *PLDI*, volume 51, pages 42–56. ACM, 2016.

[124] Rohan Padhye and Koushik Sen. Travioli: A dynamic analysis for detecting data-structure traversals. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 473–483, Piscataway, NJ, USA, 2017. IEEE Press.

[125] Corina Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *Computer Security Foundations Symposium*. IEEE, 2016.

[126] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 387–400, 2016.

[127] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2155–2168, 2017.

[128] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, Berlin, Heidelberg, 1998. Springer-Verlag.

[129] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.

[130] Michael Pradel, Markus Huggler, and Thomas R Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 13–25. ACM, 2014.

[131] Dan Quinlan and Chunhua Liao. The rose source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[132] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 722–735. ACM, 2018.

[133] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., 2015. USENIX Association.

[134] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 71–86, Austin, TX, 2016. USENIX Association.

[135] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *OOPSLA*, volume 51, pages 731–747. ACM, 2016.

[136] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *POPL*, volume 51, pages 761–774. ACM, 2016.

[137] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *POPL*, volume 50, pages 111–124. ACM, 2015.

[138] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, volume 49, pages 419–428. ACM, 2014.

[139] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 110–120. ACM, 2016.

[140] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[141] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[142] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, volume 41, pages 305–316. ACM, 2013.

[143] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, volume 49, pages 53–64. ACM, 2014.

[144] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[145] Robert Sedgewick and Kevin Wayne. Java algorithms and clients. `https://algs4.cs.princeton.edu/code/`.

[146] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.

[147] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[148] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, 2014.

[149] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 574–592. Springer-Verlag, 2013.

[150] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. Verification as learning geometric concepts. In *International Static Analysis Symposium*, pages 388–411. Springer, 2013.

[151] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *OOPSLA*, volume 48, pages 391–406. ACM, 2013.

[152] Govind Sreekar Shenoy, Jordi Tubella, and Antonio González. Improving the resilience of an IDS against performance throttling attacks. In *Security and Privacy in Communication Networks - 8th International ICST Conference, SecureComm 2012, Padua, Italy, September 3-5, 2012. Revised Selected Papers*, pages 167–184, 2012.

[153] Robert W. Shirey. Internet Security Glossary, Version 2. RFC 4949, RFC Editor, Aug 2007.

[154] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1310–1321, 2015.

[155] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast numerical program analysis with reinforcement learning. In *International Conference on Computer Aided Verification*, pages 211–229. Springer, 2018.

[156] Moritz Sinn, Florian Zuleger, and Helmut Veith. *A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis*, pages 745–761. Springer International Publishing, 2014.

[157] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, pages 1–43, 2017.

[158] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking algorithmic complexity attacks against a NIDS. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*, pages 89–98, 2006.

[159] Dawn Song. Formal verification for computer security: Lessons learned and future directions. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD '16, pages 1–1, Austin, TX, 2016. FMCAD Inc.

[160] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 370–380, Piscataway, NJ, USA, 2017. IEEE Press.

[161] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 57–69, New York, NY, USA, 2016. ACM.

[162] Marcelo Sousa, Isil Dillig, and Shuvendu Lahiri. Verifying semantic conflict-freedom in three-way program merges. *arXiv preprint arXiv:1802.06551*, 2018.

[163] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

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

[165] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[166] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, pages 352–367, 2005.

[167] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 1–13, 2018.

[168] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[169] Serge Vaudenay. Security flaws induced by CBC padding - applications to ssl, ipsec, WTLS ... In *Advances in Cryptology - EUROCRYPT*

2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings, pages 534–546, 2002.

[170] Vavr. An object-functional language extension to java 8. `https://github.com/vavr-io/vavr`.

[171] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, pages 2786–2796, 2017.

[172] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, pages 3–20, 2017.

[173] Wenhan Xiong, Thien Hoang, and William Yang Wang. Deeppath: A reinforcement learning method for knowledge graph reasoning. In *EMNLP*, 2017.

[174] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *ACM SIGPLAN Notices*, volume 47, pages 85–96. ACM, 2012.

[175] Yuval Yarom, Daniel Genkin, and Nadia Heninger. *CacheBleed: A Timing Attack on OpenSSL Constant Time RSA*, pages 346–367. Springer Berlin Heidelberg, 2016.

[176] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *FM 2008: Formal Methods*, pages 35–51. Springer, 2008.

[177] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08, pages 35–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[178] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76, New York, NY, USA, 2012. ACM.

[179] ZDNet. The average dos attack cost for businesses rises to over 2.5 million. `https://tinyurl.com/m7mvzfc`.

[180] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110. ACM, 2012.

[181] Kehuan Zhang, Zhou Li, Rui Wang, XiaoFeng Wang, and Shuo Chen. Sidebuster: Automated detection and quantification of side-channel leaks in web application development. In *Computer and Communications Security*, pages 595–606. ACM, 2010.

[182] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.

[183] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 280–297. Springer-Verlag, 2011.