# Computing homomorphic program invariants

by

**Benjamin Robert Holland**

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering (Secure and Reliable Computing)

Program of Study Committee:
Suraj Kothari, Co-major Professor
Yong Guan, Co-major Professor
Srikanta Tirthapura
Hridesh Rajan
Wei Le

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

Copyright © Benjamin Robert Holland, 2019. All rights reserved.

## DEDICATION

I would like to dedicate this work to my family, friends, colleagues, mentors, coworkers, and Amber. Without any of you this would not have been possible. All of you have pushed me at times when I needed pushing, held high expectations I felt obligated to rise to, and given support when it was needed most. This has been an unexpected journey for me. My conviction was that I wondered through life to this point by following around interesting people with interesting ideas, so in retrospect it is not surprising that I ended up where I am today.

# TABLE OF CONTENTS


**Page**

LIST OF TABLES . . . vi

LIST OF FIGURES . . . vii

ACKNOWLEDGMENTS . . . ix

ABSTRACT . . . xi

CHAPTER 1. OVERVIEW . . . 1

- 1.1 Fundamental Challenges of Program Analysis . . . 2
- 1.2 Contemporary Approaches to Program Analysis . . . 5
- 1.3 Research Theme . . . 10
- 1.4 Research Goals . . . 11
- 1.5 Organization . . . 13

CHAPTER 2. A "HUMAN-IN-THE-LOOP" APPROACH FOR RESOLVING COMPLEX SOFTWARE ANOMALIES . . . 15

CHAPTER 3. A SECURITY TOOLBOX FOR DETECTING NOVEL AND SOPHISTICATED SOFTWARE ANOMALIES . . . 20

- 3.1 General Purpose Program Indexers . . . 22
- 3.2 Domain-specific Semantic Indexers . . . 23
- 3.3 Automatic Program Analyzers . . . 24
- 3.4 Interactive User Interface . . . 25
- 3.5 Interactive On-demand Analysis Visualizations . . . 26

CHAPTER 4. STATICALLY-INFORMED DYNAMIC ANALYSIS . . . 27
4.1 Loop Call Graph . . . 29
4.2 Probe Instrumentation . . . 30
4.3 Dynamically-informed Static Analysis . . . 31
4.4 Time Complexity Analyzer . . . 32
CHAPTER 5. TARGETED DYNAMIC ANALYSIS . . . 34
5.1 A Motivating Example for Mocking . . . 35
5.2 Mockingbird Framework . . . 36
CHAPTER 6. COMPUTING RELEVANT PROGRAM BEHAVIORS . . . 40
6.1 The PCG Abstraction . . . 42
6.2 Loop Behaviors Model . . . 43
6.3 CFG to PCG Mapping as a Graph Homomorphism . . . 45
6.4 Inter-procedural Projected Control Graphs . . . 45
CHAPTER 7. BENCHMARKING STATE-OF-THE-ART IMMUTABILITY ANALYSES . . . 50
7.1 Overview of Existing Approaches and Tools . . . 52
7.2 Benchmark Design . . . 53
7.3 Benchmark Results . . . 58
CHAPTER 8. HOMOMORPHIC PROGRAM INVARIANTS . . . 60
CHAPTER 9. SIDIS FRAMEWORK . . . 66
9.1 Aborting Irrelevant Path Execution . . . 67
9.2 Eliding Irrelevant Statement Execution . . . 68
9.3 Injecting Fail Early Assertions . . . 69
CHAPTER 10. EVALUATION . . . 75
10.1 DARPA STAC Engagement Case Study I . . . 75
10.2 DARPA STAC Engagement Case Study II . . . 83
10.3 DARPA STAC Engagement Case Study III . . . 87

CHAPTER 11. CONCLUSIONS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

BIBLIOGRAPHY . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

# LIST OF TABLES

Page

Table 1.1 Elemental Language Features . . . . . . . . . . . . . . . . . . . . . . . . 4

Table 1.2 Classification of Elemental Language Feature Analysis Challenges . . 5

Table 9.1 Toy Example Expected Outcomes . . . . . . . . . . . . . . . . . . . . . 73

Table 9.2 Toy Example Fuzzing Speed . . . . . . . . . . . . . . . . . . . . . . . . 73

Table 9.3 Toy Example Invariant Detection . . . . . . . . . . . . . . . . . . . . . 73

Table 10.1 Loop Execution Counts for the Blogger Challenge Application . . . . 78

# LIST OF FIGURES


| | | Page |
|---|---|---|
| Figure 2.1 | Computation Cost Escalation Beyond Automation Wall | 17 |
| Figure 2.2 | Traditional Automation vs. ART-based Approach | 18 |
| Figure 3.1 | Security Toolbox Dashboard Interface | 25 |
| Figure 3.2 | Smart View - Interactive On-demand Analysis Visualization | 26 |
| Figure 4.1 | LCG for TimSort along with the CFG of `minRunLength` | 30 |
| Figure 4.2 | Control Flow Graph Execution Count Heat Map Overlay | 33 |
| Figure 5.1 | Mockingbird Framework Architecture and Workflow | 37 |
| Figure 6.1 | CFG and PCG for Example DBZ Vulnerability | 42 |
| Figure 6.2 | Illustrative Example of Loop Behavior Model | 44 |
| Figure 6.3 | Illustrative Example of IPCG Transformation of a Simple ICFG | 48 |
| Figure 6.4 | Illustrative Example of IPCG transformation of a Recursive ICFG | 49 |
| Figure 7.1 | Immutability Assignment Graph | 56 |
| Figure 8.1 | Toy Example - Control Flow Graph (CFG) | 64 |
| Figure 8.2 | Toy Example - Projected Control Graph (PCG) of Division Operation | 65 |
| Figure 9.1 | SIDIS High Level Architecture Diagram | 67 |
| Figure 9.2 | Toy Example - Eliding Irrelevant Program Statements | 74 |
| Figure 10.1 | LCG for the Blogger Challenge Application | 77 |
| Figure 10.2 | Blogger Challenge Application CFG to PCG Transformation | 79 |

Figure 10.3 Log-Log Plot of Loop Execution Counts by Input Size . . . . . . . . . 81

Figure 10.4 Image Processor Challenge Application ICFG to IPCG Transformation 85

Figure 10.5 Image Processor Challenge Application `accuracy` Array Contents . . 86

Figure 10.6 Image Processor Challenge Application Expensive Pixel Colors . . . . 86

Figure 10.7 Screenshot of Loop Call Graph of `Plait.normalizeCompletely` . . . 88

Figure 10.8 BraidIt Challenge Application Case Study IPCG . . . . . . . . . . . 90

## ACKNOWLEDGMENTS

First, I would like to thank my parents who set me on this course in life from a very young age. I doubt that they expected I would be here either, but they never missed an opportunity to provide me with an opportunity to succeed. If they had not supported me during my undergraduate degree, I doubt that I would be where I am now.

Second, I owe a great debt to the many mentors that I have had throughout my life. Beginning in Cub Scouts to my Eagle Scout award there have been Scoutmasters such as Tom Stott and parents who were meaningfully involved with our lives at a very young age. My teachers through school and to Mr. Whitney in my High School vocational computer courses who took special interest in my education. During my undergraduate, I met many professors that helped to develop my thinking. Each of the members on my committee played a big role in my life whether they knew it or not. Hridesh Rajan completely opened my eyes to Computer Science courses at a time when I was on the fence of about the program with his excitement and passion for teaching during his first year teaching at Iowa State University. In fact, if it were not for him, I would not have gone back to obtain a second undergraduate degree in Computer Science. Srikanta Tirthapura and Wei Le both passionately taught excellent courses that helped to lay the foundation for this work. Yong Guan was perhaps the first professor to engage with me on an individual basis and helped me to enter and succeed in my Master's program. Just as I thought I was done with academia, Suresh Kothari changed my life in a single afternoon by convincing me to stay for a summer and try my hand at an entirely different line of research in program analysis. That summer turned into three years and with his mentoring I stopped saying I was a "security guy" and started saying I was a "program analysis guy with an interest in security". Suresh Kothari has changed the way I think, and I feel very privileged to have been given the chance to learn from him. His ability to see different strengths in individuals and look past what is traditionally thought of as a "good student" (which is not me)

is a credit to him. I remember telling my mother that my advisor was a mathematician to which she replied: “But you’re not good at math!” and I replied “I agree.”.

I would like to take this opportunity to acknowledge that many of the ideas in this work are the result of a team of intellectuals that I was privileged to interact with both inside and outside of Iowa State University. There are too many names to be complete, but many of the ideas in this work were byproducts of long conversations and the technical support of Suresh Kothari, Payas Awadhutkar, Ganesh Ram Santhanam, Ahmed Tamrawi, Tom Deering, Jon Mathews, Nikhil Ranade, and Jeremias Sauceda.

Finally, I’ve heard that a PhD takes a bigger toll on the people around you than it does on the student. I know the toll it has taken on me, so I cannot imagine the toll it has taken on Amber, who has literally put her life on hold for me. For me, a PhD has been a test of perseverance, perhaps with a required minor study in Political Science, so if I am to be awarded this degree, I have to think that she has somehow earned an equivalent degree.

Interestingly, I also should acknowledge the Defense Advanced Research Projects Agency (DARPA), which allocated the funding and set the ambitious high-level research goals for what this work only begins to address and by doing so I must also acknowledge that:

> *“This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.”*

# ABSTRACT

Program invariants are properties that are true at a particular program point or points. Program invariants are often undocumented assertions made by a programmer that hold the key to reasoning correctly about a software verification task. Unlike the contemporary research in which program invariants are defined to hold for all control flow paths, we propose *homomorphic program invariants*, which hold with respect to a relevant equivalence class of control flow paths. For a problem-specific task, homomorphic program invariants can form stricter assertions. This work demonstrates that the novelty of computing homomorphic program invariants is both useful and practical.

Towards our goal of computing homomorphic program invariants, we deal with the challenge of the astronomical number of paths in programs. Since reasoning about a class of program paths must be efficient in order to scale to real-world programs, we extend prior work to efficiently divide program paths into equivalence classes with respect to control flow events of interest. Our technique reasons about inter-procedural paths, which we then use to determine how to modify a program binary to abort execution at the start of an irrelevant program path. With off-the-shelf components, we employ the state-of-the-art in fuzzing and dynamic invariant detection tools to mine homomorphic program invariants.

To aid in the task of identifying likely software anomalies, we develop human-in-the-loop analysis methodologies and a toolbox of human-centric static analysis tools. We present work to perform a statically-informed dynamic analysis to efficiently transition from static analysis to dynamic analysis and leverage the strengths of each approach. To evaluate our approach, we apply our techniques to three case study audits of challenge applications from DARPA's Space/Time Analysis for Cybersecurity (STAC) program. In the final case study, we discover an unintentional vulnerability that causes a denial of service (DoS) in space and time, despite the challenge application having been hardened against static and dynamic analysis techniques.

# CHAPTER 1. OVERVIEW

The pervasiveness of software makes the security and reliability of software a concern of everyone. Ever since the first programs were written there have been bugs both logically [1] and literally [2]. Ada Lovelace, largely credited with being the world's first programmer, added to her translations of Charles Babbage's sketch for an analytical engine her own thoughts on proving correctness of programs and noted that the task was complicated by "numerous and inter-complicated conditions" [1]. Godel's incompleteness theorems [3] and later work by Turing [4] seem to indicate that we may never be able to completely prove the correctness of software, however a glance at recent software vulnerabilities tracked by MITRE's CVE database [5] shows that our analysis capabilities are nowhere near any theoretical limits since we routinely miss even relatively simple problems in high profile software projects [6, 7, 8]. What's worse is the difference between software bugs and malware may only be the intent of the programmer [9]. Both bugs and malware can have catastrophic consequences, yet malware disguised as a bug has plausible deniability, which means that security auditors must consider both types of software anomalies. An exploitable bug is a vulnerability. The ability to discover existing vulnerabilities in an adversary's software systems is perhaps the most important arms race. With the rise of the Internet and its mainstream adoption in the 1990's, the rapid integration of software into a globally inter-connected network has exposed an amazing number of software systems. The true scale of the Internet is unknown, but an anonymous 2012 Internet census project that leveraged an illegal botnet to exploit vulnerable network routers put the estimate at around 2.3 billion active systems [10]. Recent malicious probing of power grid systems [11] and the sharp increase in military and industrial cyber espionage by nation state adversaries [12] has made the situation quite dire. The development of practical program analysis tools to discover and verify software anomalies is of the utmost importance.

In the following sections we examine some of the fundamental challenges of program analysis (Section 1.1), the contemporary approaches to program analysis (Section 1.2), our research theme (Section 1.3) and goals (Section 1.4), and the organization of the rest of this work (Section 1.5).

## 1.1 Fundamental Challenges of Program Analysis

To explore some of the fundamental challenges in program analysis that manifest in modern programming languages it is useful to distill modern programming languages down to their most basic elements. To aid in our exploration, we present a small Turing complete language called *Elemental*[1]. *Elemental* is a backwards compatible extension to Urban Müller's *Brainf*ck* [13], which is a minor variation on Corrado Böhm's Turing complete language *P"* [14]. The design goal of *Brainf*ck* was to write the smallest possible compiler. A *Brainf*ck* to x86 machine code compiler exists that is only 100 bytes[2]. The minimalism of the *Brainf*ck* language is a good starting point, but it conflates branching and looping and lacks some features that are analogous to modern languages. *Elemental* explicitly separates the language's branching and looping mechanisms and introduces rudimentary versions of modern concepts such as goto's, assignments, and functions. Table 1.1 outlines the 16 language features of *Elemental* and Table 1.2 describes a classification by the types of program analysis challenges that are introduced with each feature set. That classification is hierarchical, each level introduces additional program analysis challenges. Classification *T0* presents no analysis challenges while classification *T7* contains every analysis challenges.

*Elemental* includes a single line comment, which aside from an additional parser rule does not introduce any program analysis challenges. We assign the comment feature a classification type of *T0* because it cannot impact the execution of a program in any way. Like a conceptual Turing machine, *Elemental* reads and writes symbols to a single tape consisting of cells that store data. The data pointer to the current tape cell can be moved left or right with the respective `<` and `>`

[1] `https://github.com/benjholla/Elemental`
[2] `https://github.com/peterferrie/brainfuck`

instructions. Symbols consist of integer values that can be incremented or decremented with the `+` and `-` instructions.

Unlike the formal definition of a Turing machine [15], which includes a partial function called the *transition function* that can map a program state to any other program state (unmapped states being halting states), *Elemental* and most modern programming languages define a stricter sequential ordering of state transitions for most instructions. In *Elemental*, language features in type *T1* always progress sequentially to the next instruction in the program instructions. *T1* instruction types always have a single behavior. *T2* introduces a branching for multiple behaviors. An *Elemental* program of `(>(>(>)))` has four possible behaviors while `(>)(>)(>)` has eight possible behaviors depending on the initialization of the tape. An *Elemental* program consisting only of $n$ branch statements could have anywhere from $n + 1$ to $2^n$ behaviors.

The introduction of type *T3* adds Turing completeness to the language and the possibility that programs may no longer halt. The looping behavior of *T3* could be implemented with a while loop style language feature or with recursion through functions. In both features there is an unconditional jump that returns to a previous program instruction that enables looping. Type *T4* forces an analysis to reason about the current tape cell value to identify the next instruction. While type *T4* mixes analysis of control flow (instruction orderings) with an analysis of data, the return jump is still a fixed location unlike type *T5* instructions that can be used to write unstructured code and blur control and data without fixed return jumps.

The store and recall functions supported by *Elemental* are not required for Turing completeness. Traditionally, input for a Turing machine is provided by initializing the tape cell values before execution and output is read after the machine halts. The store and recall functions provided by *Brainf*ck* and included in *Elemental* are convenience features. We group a store as type *T6* because unlike its type *T1* counterparts, store can introduce data at runtime that originates from outside of the program, which forces an analysis to consider all possible integer inputs. *Elemental* has no concept of data variables, except for the values on the tape cells. For type *T6* and lower programs, there is no way to move data contained in one cell to another cell without conditionally copying

data. The assignment introduced in type *T7* enables the explicit flow of data, which an analysis must track to reason about future program states and further mixing the inter-dependence of control and data analysis.

*Elemental* shows that the number of paths in software can be exponential without loops, non-halting with loops, and that the challenges of both data flow analysis and control flow analysis can be conflated. The challenges presented in *Elemental* are common place in modern languages such as C, C++, and Java.

Table 1.1 Elemental Language Features

| Instruction | Type | Description |
|---|---|---|
| # | T0 | A one line comment |
| + | T1 | Increment the byte at the current tape cell by 1 |
| - | T1 | Decrement the byte at the current tape cell by 1 |
| < | T1 | Move the tape one cell to the left |
| > | T1 | Move the tape one cell to the right |
| ( | T2 | (Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching ), else execute the next instruction |
| [ | T3 | (While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching ], else execute instructions until the matching ] and then unconditionally return to the [ |
| *[0-9]+*: | T3 | (Function) Declares a uniquely named function (named [0-9]+ within range 0-255) |
| {*[0-9]+*} | T3 | (Static Dispatch) Jump to a named function |
| ? | T4 | (Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell |
| "*[0-9]+*" | T5 | (Label) Sets a unique label (named [0-9]+ within range 0-255) within a function |
| '*[0-9]+*' | T5 | (GOTO) Jumps to a named label within the current function |
| & | T5 | (Computed GOTO) Jumps to the named label within the current function with the value of the current cell |
| . | T1 | (Recall) Write byte value to output from current tape cell |
| , | T6 | (Store) Read byte value from input into current tape cell |
| = | T7 | (Assignment) Replace the value of the current cell with the value of the cell at the address indicated by the current cell |

Table 1.2 Classification of Elemental Language Feature Analysis Challenges

| Type | Description |
|---|---|
| 0 | Type 0 is language convenience and does not add any analysis challenges |
| 1 | Type 1 provides the basic architecture for a Turing machine (update tape cell and move tape cell). Programs limited to type 1 and lower instruction types are sub-Turing machines and will always halt and have a single program behavior. |
| 2 | Programs limited to type 2 and lower instructions are sub-Turing machines, which must halt but can contain an exponential number of program behaviors. |
| 3 | Programming languages that only contain instruction of Type 3 or lower are Turing complete. While type 4+ programs introduce new analysis challenges, they do not increase computing power. |
| 4 | Type 4+ instructions obscure control flow with data |
| 5 | Type 5 instructions allow for the formation of unstructured code |
| 6 | Type 6 introduces program inputs at runtime |
| 7 | Type 7 introduces data flow through pointer based assignments |

## 1.2 Contemporary Approaches to Program Analysis

Program analysis involves the analysis of a program's control flow (CF) and data flow (DF) to reason about various program properties such as correctness, reliability, and security. Control flow analysis reasons about the order of statements in a program's execution while data flow analysis reasons about the possible data values during a program's execution. Data flow tends to influence control flow, and vice versa. Eventually program analysis must reason about both control and data flow because data drives program execution. Taylor Hornby, a judge for the Underhanded Crypto Contest succinctly states this challenge [16, 17]:

> *"The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program. When you sort a deck of cards, you're moving them around, but it's the numbers on the cards that are telling you where to move them."*

Generally attackers are unable to change a program's coded control flow, but they can drive the program in malicious ways by crafting software exploits that consist of malicious data, which influence control flow. As an aside, a buffer overflow attack famously described in [18] first uses

malicious data to corrupt stack memory using faulty control flow and then influences the control flow to execute the malicious data giving the attacker the ability to execute arbitrary code. Similarly, the earlier versions of Stuxnet found targeting Iran's nuclear program facilities consisted of a malicious configuration file that caused the target software to execute arbitrary code with administrative privileges [19, 20]. Section 1.1 explored the ways that CF and DF mix in modern languages. The remainder of this section explores contemporary approaches to analyzing a program's control and data flow.

A common approach to model control and data relationships in program analysis involves modeling the program as a graph data structure. A *control flow graph* (CFG), proposed by Frances Allen in 1970 [21], models control flow within a function as a directed graph where nodes represent program statements or basic blocks and the edges represent control flow transitions. Each possible behavior of a function can be described as a path from the CFG root to a CFG leaf. Even without considering loops, the number of paths (behaviors) a function may exhibit is exponential in the case of non-nested branches ($n$ branches could exhibit $2^n$ distinct behaviors). An examination of the Linux kernel reveals a single function named `lustre_assert_wire_constants` that alone has $2^{656}$ paths [22]! The number of paths in a program are further multiplied by each callsite to a function [23]. With the addition of loops the number of program paths are potentially infinite. The astronomical number of paths in software is often referred to as the *path explosion problem*. A specific program behavior can be described as the path traced from the root CFG node of the program main method until system termination. A program *execution trace* is a feasible path through a program's control flow that is possible at runtime for a given program input. The challenge of determining if a path is feasible, in other words if a program behavior could occur at runtime, is often referred to as the *path feasibility problem*.

The range of possible input values to a program is also exponential in the sense that all program inputs could be ordered and serialized into binary string of $n$ bits giving $2^n$ unique inputs. A program's data relationships can be modeled as a *data flow graph* (DFG) [24]. A DFG is a directed graph of program variable, primitive value, and operator nodes where edges represent atomic level

data dependencies such as assignments [25]. To get precise results for data flow analysis, it is necessary to account for control flow and to get precise results for control flow analysis it is necessary to account for data flow. As the demand for analysis precision increases, control flow analysis and data flow analysis become codependent. By considering control flow, a DFG can be transformed to a *static single assignment* (SSA) form so that every variable is defined exactly once. This enhancement was first proposed in 1988 [26] and effectively disambiguates the reaching definitions of variables at various program points with some exceptions for branches and loops. The problem of inter-dependent control and data flows becomes very apparent when an analysis becomes inter-procedural (e.g., function pointers, dynamic dispatch, global program state).

Inter-procedural control flow can be modeled as a *call graph* (CG) with the addition of control flow edges from the function callsite to the target function (or summarized as a caller-callee relationship between functions). While efficient algorithms such as class hierarchy analysis [27] and rapid type analysis based algorithms [28, 29] exist to conservatively compute possible inter-procedural behaviors, additional precision comes at a cost. For example, consider a points-to analysis [30], which is a widely employed iterative data flow analysis that reasons about the values a variable may or must point-to at a given program point. The points-to analysis result can be used to refine control flow and construct precise call graphs, but at a cost of $O(n^3)$, where $n$ is the number of variable assignments [31]. There are points-to analysis algorithms that sacrifice precision for reduced computation such as [32], but many modern precise points-to analysis implementations further increase the $O(n^3)$ cost by disambiguating inter-procedural calling context and/or object instances. Both sensitivities can exponentially increase the value of $n$ [30]. In the general sense, a perfect points-to analysis is impossible because the problem can be reduced to the halting problem [33, 34, 35]. To combat the high cost demanded by increased precision, modern points-to analysis approaches have employed binary decision diagrams [36, 37], declarative approaches [38], and on-demand computation [39] with relatively modest improvements.

In contemporary program analysis approaches there are *composite* and *decompositional* approaches. Composite approaches combine reasoning about control and data flow. Decompositional

approaches reason about control and data flow independently. A points-to based call graph construction is a composite approach to call graph construction. In the remainder of this section we briefly describe both types of approaches.

Composite approaches range in the spectrum of static to dynamic analysis. Static based approaches range from abstract interpretation that seeks to map the space of data to a smaller abstract domain [40] to symbolic execution that computes operations on symbolic inputs in place of concrete values, forking the symbolic logic at each program branch [41, 42, 43]. Symbolic computation must deal with the path explosion problem when forking. Additionally, symbolic computation often employs expensive constraint solvers that necessarily reason about boolean satisfaction in addition to other domains such as integers, strings, etc., to realize the values required to reproduce the program behavior in question. On the dynamic side of the spectrum are approaches such as concolic testing that combine symbolic execution and concrete execution [44, 45, 46] ranging to invariant detection of instrumented programs executed with an input workload [47]. Fully automatic approaches on the static end of the spectrum tend to suffer from tradeoffs made between imprecise results or the high cost of computation required for precise results. Fully automatic approaches on the dynamic end of the spectrum suffer from challenges in input generation that result in poor coverage of relevant program paths. There has been some promising work in hybrid static and dynamic analysis [48, 49, 50], but in practice the majority of bugs found currently can be attributed to fuzzing [51, 52, 53]. This result appears to be purely an economical result in that a “dumb” tool can make more guesses at what an interesting program input could be in the time a “smart” tool can compute the interesting program inputs. With future improvements to SAT/SMT solvers this situation could change, however the SAT problem is the first proven NP-complete problem [54]. Since resolving the question of whether or not SAT has a polynomial-time algorithm would be equivalent to solving the most famous open problem in the theory of computing, the *P versus NP* problem, one shouldn’t hold their breath.

Before control flow analysis and data flow analysis become tangled there is an opportunity to reason about relevant program statements by computing relationships on the CFG and DFG inde-

pendently. Program slicing, first proposed by Mark Weiser in 1981 [55], is a decompositional technique that extracts control and data dependencies to answer whether or not a particular statement could impact the computation of a variable at a given statement. A *data dependence graph* (DDG) is formed of relationships that are derived from a summarization of variable DFG relationships at the statement defining or using the variable. A *control dependence graph* (CDG) is formed of relationships such that if a statement $X$ determines whether a statement $Y$ can be executed then $Y$ is control dependent on $X$. The union of the DDG and CDG forms a *program dependence graph* (PDG). The F.O.W. algorithm computes the PDG efficiently in nearly linear time [25] by leveraging Tarjan's dominance algorithm [56]. A program slice is then computed using a traversal from a statement or set of statements $S$ within the PDG. A forward traversal within the PDG from the slicing criterion $S$ computes a relevant set of statements that could be impacted by a change to the initial statement $S$. A reverse traversal from the slicing criterion computes a set of relevant statements that may impact the initial statement $S$ if changed. Statements not included in the program slice are considered irrelevant to the selected slicing criterion. In [57], Horwitz describes a conservative approach for inter-procedural program slicing referred to as the *system dependence graph* (SDG).

In a similar vein to program slicing, a CFG's paths (each representing a unique program behavior) can be divided into equivalence classes of relevant and irrelevant paths with respect to a set of events. An event is a program statement that if executed classifies a CFG path as relevant. A *projected control graph* (PCG) is a compact representation of a CFG containing only statements in the set of events of interest and the control dependent branches reachable in the CDG from the events of interest [58]. Imaginary top and bottom nodes are added as placeholders for elided CFG root and leaf nodes. The edges in a CFG are collapsed to representative edges between the nodes in the PCG where each path in the PCG corresponds to an equivalent set of paths in the CFG with respect to the PCG events of interest. A PCG is a purely decompositional approach because it does not consider data flow. Previous work [59] proposed the PCG abstraction and presents an efficient

algorithm to compute the PCG within a function that was later implemented in a general purpose tool [60] for this work.

## 1.3 Research Theme

The goal of this research is to build practical program analysis tools and methodologies to detect software security anomalies. We seek a means to an end to detect malware and security vulnerabilities in software. A core philosophy of this research is to leverage human reasoning and machine automation wherever possible to avoid intractable problems in program analysis. It is generally accepted that human reasoning is necessary for developing a program, but by assuming that humans should not be involved in the analysis of a program we have created an asymmetric property in the analysis of programs.

The sentiments towards leveraging human reasoning in program analysis have historically not been very positive in academic circles, which have aggressively sought and promoted fully automatic approaches. Similar aspirations have been reflected in government sponsored program's such as DARPA's Automated Program Analysis for Cybersecurity (APAC) [61] program with the idea that eliminating a human's role eliminates the opportunity for mistakes. Earlier, in 1979, De Millo and Perlis (the first Turing Award winner) argued that we should value human intuition of verification of software over a human-incomprehensible "proof" [62]. Recent work has even begun to question some of the assumptions of previous formal verification work of the seL4 Linux kernel and other projects [63]. Due to the fact that significant breakthroughs to overcome the intractable problems in program analysis that appear again and again have failed to come forward, the idea of including a human-in-the-loop has started to become more widely accepted.

This shift in research to include human reasoning is observable in DARPA's recent programs for Explainable Artificial Intelligence (XAI) [64] and Computers and Humans Exploring Software Security (CHESS) [65]. A human-in-the-loop approach is also echoed by some forward thinkers, perhaps most succinctly by cryptography and computer security expert Bruce Schneier who said that: "Security is a process, not a product" [66]. USAF Colonel John Boyd described the *OODA*

*loop* as an iterative decision cycle of *observe*, *orient*, *decide*, and *act* [67]. Boyd developed this framework as a way to explain the unanticipated, superior agility of US fighter pilots in aerial combat situations. The pilot must iterate the OODA loop faster than his opponent in order to *decide*, and *act* before his opponent has a chance to *observe*, *orient* himself to new information. Both pilots are aided by machines and a superior pilot may still lose the race if his instruments fail to him at any point in the cycle. The paradigm of OODA loops applies equally well to the context of program analysis and there is no reason that a human cannot be included in the cycle. Fred Brooks makes a stronger statement in his Allen Newell address "Computer Scientist as a Toolsmith" [68]:

> *"If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself."*

In the context of this research, we seek to recover the underlying program invariants that are relevant to human understanding of a software security anomaly. More specifically, we seek to uncover program invariants with respect to only the code relevant to a software anomaly and not the entire program. In this work, program invariants are computed for the data computations that are performed on program paths that are relevant to the vulnerable code. As we will show later, these invariants can be valuable to gain a holistic understanding of the root cause of a software anomaly, which enables a human to succinctly understand the problem. This human comprehension could be used to assess the impact of malware, develop a proof of concept exploit for a software vulnerability, or create a software patch to fix the core issue.

## 1.4 Research Goals

A program invariant is defined as "a property that is true at a particular program point or points" [69]. Program invariants are often undocumented assertions made by a programmer that hold the key to reasoning correctly about a software verification task [70]. Ernst relates program invariants to human program comprehension when he writes [69]:

*"I contend that most programmers have invariants in mind, consciously or unconsciously, when they write or otherwise manipulate programs: they have an idea of how the system works or is intended to work, how the data structures are laid out and related to one another, and the like. Regrettably, these notions are rarely written down, and so most programs are completely lacking in formal or informal invariants and other documentation."*

Daikon is a publicly available dynamic invariant detector that has pioneered the research of invariant detection in both the academic and commercial world [47]. Like Ernst, we say an invariant is relevant if it is useful to a programming activity [70]. Daikon detects program invariants by instrumenting a program's variables on all program paths. The instrumented program is then executed with a set of program inputs and the resulting execution traces are used to compute likely program invariants. Daikon does not dictate how program inputs should be chosen, but the authors note that the accuracy of the detected invariants depend highly on the selection of program inputs and code coverage of execution traces [47]. The automatic input generation provided by program fuzzers are a natural complement to Daikon. The current state-of-the-art in automated fuzzing is the American Fuzzy Lop (AFL) fuzzer [52, 51, 53]. AFL employs a lightweight instrumentation to guide a genetic algorithm to generate inputs that result in high code coverage.

Employing a fuzzer or a similar randomized testing approach proposed by Daikon's authors [71] to drive program execution addresses the problem of input selection and code coverage, however these techniques produce a large amount of execution traces that become computationally burdensome for invariant detection. To scale the approach several optimizations have been made to Daikon that avoid computation of redundant invariants [70], in the process defining the remaining invariants as *relevant invariants*. Ernst broadly defines an invariant to be relevant "if it assists a programmer in a programming activity" [70].

This research begins with a simple observation that invariants that hold on a subset of program paths may not hold on all program paths. For example, in the toy example presented in Chapter 8, the invariant "`x < 5`" holds for the subset of the program paths where a *division by zero* (DBZ)

error occurs, however the same invariant is invalidated and does not hold for all program paths. The invariant "`x < 5`" represents a restriction on the program input required to cause a DBZ error and is necessary information to replicate the bug. Previous work [72], makes a similar observation and proposes detecting *conditional invariants*, which are predicates that hold before and after conditions, however the inter-dependence of conditions is not considered. By restricting invariant detection to only the homomorphic program behaviors (defined later) the relevance and strictness of the program invariant assertions are increased while the computational burden on invariant detectors is reduced.

Homomorphic program behaviors (e.g., behaviors relevant to a division by zero) are a partitioning of the program's paths into equivalence classes of relevant and irrelevant paths with respect to the events of interest (e.g., a division operation) [59, 60]. We will refer to invariants computed for relevant program behaviors as *homomorphic invariants*. By the original definition of relevant invariants, homomorphic invariants are necessarily *relevant invariants* that assist a programmer in reasoning about a specific class of program behaviors.

Unlike previous work that seeks to remove irrelevant invariants by removing redundant invariants [70], our goal is to compute homomorphic invariants to aid human program comprehension of software security anomalies. In Chapter 9, we present a framework to dynamically compute homomorphic program invariants. As a byproduct of the approach, we significantly increase the speed of the state-of-the-art in program fuzzing by targeting the dynamic analysis to a specific region of code and aborting execution on paths that are irrelevant to the analysis task. We also significantly reduce the computational burden on the state-of-the-art in dynamic invariant detection by purging irrelevant execution traces from the set of all execution traces to be analyzed.

## 1.5 Organization

To practically compute homomorphic program invariants for understanding software anomalies, a significant amount of supporting infrastructure must be established. While developing our approach we encountered several difficult research questions. Since our ultimate goal was to answer research question *RQ7* below, we first had to address research questions *RQ1-RQ6*.

The rest of this thesis is organized as follows:

RQ1 - ***How can we facilitate both automation and human comprehension to deal with intractable problems in program analysis?*** (Chapter 2)

RQ2 - ***How can we statically identify likely candidates for novel and sophisticated software anomalies?*** (Chapter 3)

RQ3 - ***How can we effectively leverage the strengths of both static and dynamic program analysis to verify and validate software anomalies?*** (Chapter 4)

RQ4 - ***How can we systematically drive a targeted dynamic program analysis to explore relevant program paths?*** (Chapter 5)

RQ5 - ***How can we statically compute a program's relevant behaviors to deal with the path explosion problem?*** (Chapter 6)

RQ6 - ***How can we evaluate static program analyses to detect function side-effects and aid in selection of relevant control flow events?*** (Chapter 7)

RQ7 - ***How can we compute homomorphic program invariants to aid in human program comprehension?*** (Chapters 8, 9, 10)

Chapter 11 provides general concluding remarks.

# CHAPTER 2. A "HUMAN-IN-THE-LOOP" APPROACH FOR RESOLVING COMPLEX SOFTWARE ANOMALIES

This chapter presents a summary of key concepts from our paper *A "Human-in-the-loop" Approach for Resolving Complex Software Anomalies* by Suresh Kothari, Akshay Deepak, Ahmed Tamrawi, Benjamin Holland, and Sandeep Krishnan. The paper was published at the IEEE International Conference on Systems, Man, and Cybernetics (SMC) in October of 2014 in San Diego, California. This work addresses our first research question: ***How can we facilitate both automation and human comprehension to deal with intractable problems in program analysis?***

A purely static analysis aims to analyze software without executing it. Automated static analysis tools are widely used in detecting software anomalies, such as non-compliance with coding and documentation guidelines, dead code and unused data, software vulnerabilities like unsafe thread synchronization or memory leaks, and malware in smartphones. Generally, an automated tool runs in three steps: (1) a human specifies the software to be analyzed and analysis parameters, (2) the tools runs on the input and outputs a report of potential anomalies in the software, (3) an analyst goes through the report. A tool is considered sound if it reports all anomalies in the software with no false positives or false negatives. However, quite often, it not possible to build a sound tool. Balancing coverage vs. accuracy in an analysis strategy involves an inherent trade-off: one can list only true-positives (low coverage, high accuracy) or one can output all potential anomalies (high coverage, low accuracy). Achieving high coverage and high accuracy in a fully automated tool can be impossible or incur prohibitive cost in terms of implementing the automation and/or sifting through the large number of erroneous results manually. To motivate the case for an alternative to such an automated approach, we classify the set of anomaly-prone scenarios into two groups: "ordinary" and "complex". Ordinary scenarios correspond to the scenarios that are amenable to automation and do not pose extraordinary analysis challenges. On the contrary, complex scenarios are the ones

that pose significant barriers to automation; they involve hard-to-analyze programming paradigms and programming constructs. For example, a fully automated analysis may be intractable because of consumer-producer paradigm in which related events happen in different threads (e.g., a memory allocation in a producer thread and a corresponding memory deallocation in a consumer thread). Programming constructs such as function pointers also make analysis difficult by obscuring the control or data flow.

Even if automation for a complex scenario is possible, it may well be infeasible due to economics of time and effort. Malware analysis of applications in smartphones is a good example of this. What is considered malicious in one application may be considered benign in another because of difference in the purpose and the intended contextual uses of those applications. For example, accessing of contacts by an e-mail client is a legitimate action, but it is illegitimate, and probably malicious, for a weather application to do so. Further, malicious applications frequently blend their overt and malicious purposes. For example, an application meant for sharing pictures can also leak those same pictures to a malicious website without the knowledge of the user. Automating identification of malicious activities in such cases can mean writing a lot of application-specific code. Clearly, this will incur significant additional cost and seriously affect the viability and general applicability of the automated tool. On the other hand, leaving all the complex scenarios to be resolved by a human analyst alone after the automation run can also be prohibitive for reasons similar to why automation was approached in the first place. However, there can be some initial help from the automation run in such cases, but because the roles played by the automated tool and the human analysts are segregated, this help can be limited; leaving the human analyst to do most of the work.

As summarized in Figure 2.1, static analysis tools hit an "automation wall" and the cost for resolving complex scenarios escalates beyond the automation wall. We appeal for an "Amplified Reasoning Technique" (ART) as an alternative approach to resolve such complex scenarios. The ART philosophy is: instead of resolving complex anomalies as definitive "Yes/No" answers through a fully automated tool, bring the human in a human-machine loop and use the tool to amplify human reasoning to resolve such anomalies faster and efficiently. For example, to check the possibility of

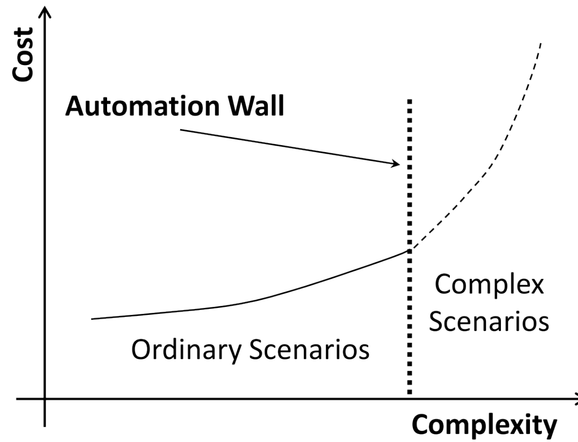
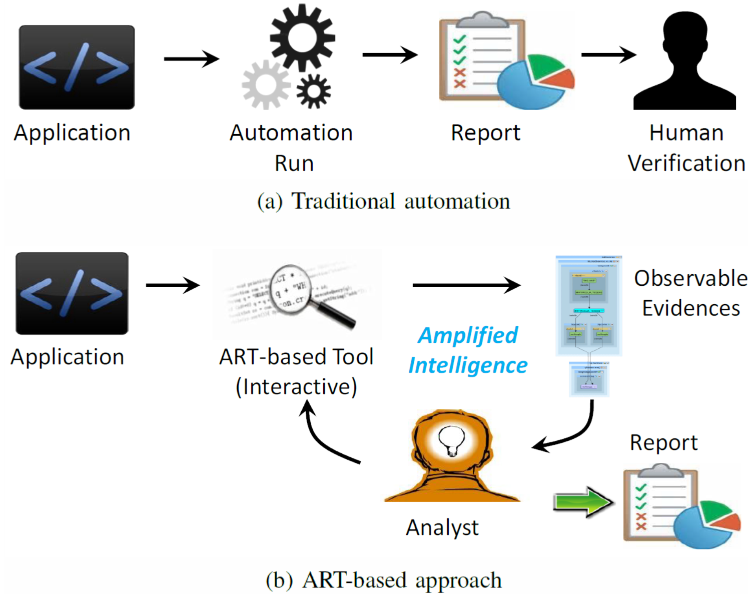



Figure 2.1 Computation Cost Escalation Beyond Automation Wall

information leaks in an Android application, the human analyst would want to examine all the occurrences of Android APIs that are relevant in the context of a given application. Any reasoning of the type "examine all" is powerful but it is quite difficult to apply it to large software without an appropriate tool. Figure 2.2 brings out the difference between the traditional approach to automation vs. the ART-based technique that has been discussed in this paper. In the traditional automation, the role of human is to sift through the false positives and unresolved cases generated from the automation run, and is segregated from the role played by the machine. Whereas, the ART-based approach puts the human and machine in an interactive loop. Each iteration involves human intelligence guiding the tool to generate refined evidences that bring closer to the final conclusion. Thus, an ART-based approach amplifies the human intelligence in resolving complex scenarios. The guiding principle is what Fred Brooks points out in his Allen Newell address "Computer Scientist as a Toolsmith" [68]:

> *"If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself."*




(a) Traditional automation

(b) ART-based approach

Figure 2.2 Traditional Automation vs. ART-based Approach

The paper presented three case studies of complex scenarios from real-world software (lock/unlock pairing in Linux operating system code, memory leak analysis in XINU operating system code, and malware analysis of an Android application) to show how human reasoning amplified by a tool can be an effective and practical solution. These studies showcased different barriers to automation and their varying complexity, and the kind of role an ART-based tool plays. The complex anomaly scenarios presented in the paper are not sporadic, but frequently occurring real-world cases. For example, for checking lock/unlock pairing in Linux kernel out of 1255 scenarios, 403 are complex (32.1%). Specifically, out of the 403 complex scenarios: 68 have large execution paths (greater than 500 nodes), 112 have infeasible paths (paths not feasible at runtime), 11 call functions via pointers, and 212 have unstructured code (e.g., jump statements like goto).

The complex scenarios involve data and control relationships that require graph representations. Thus, a powerful ART tool needs the capability to represent and refine program semantics as graphs and provide a query language to search and manipulate the graphs. In general, a fully automated and completely accurate data flow analysis is known to be NP-complete [33]. The number of execution paths grow exponentially with the number of non-nested branch nodes [21]. Apart from

theoretical infeasibility, full automation may not be pragmatic because of programming constructs such as flow of data through containers such as linked lists, interrupt-driven routines, multi-threaded programs, function pointers, and the fact that a program may not be a closed system because of its dependence on external library or operating system calls. Given the infeasibility or impracticality of full automation, the ART approach makes sense because it is often relatively easier for a human analyst to reason: both in terms of the reasoning ability as well as a richer contextual understanding of the software. The analyst's difficulty is extracting relevant information from large software. That difficulty can be addressed with an apt tool with query interface. It is important that the analyst be not restricted to canned queries. Instead, the analyst should be able to compose new queries necessary to address varying contexts of complex anomaly scenarios.

Program comprehension is a central activity during software maintenance, evolution, and reuse [73]. Some reports estimate that up 60-70% of the maintenance effort is spent of understanding code [74]. It also plays an important role in an ART-based approach. The purpose of the paper was to draw researchers' attention involved in resolving complex scenarios towards making their analysis ART-based by illustrating how it can be more cost-effective and productive. The case studies discussed in the paper did not compare the results of an ART-based vs. the typical automated techniques "statistically" to prove the clear superiority of one over the other, but they instead highlighted the advantages an ART based analysis can offer by comparing the process of analysis. That is, the focus is on the process of analysis – rather than the results – and let the researcher make an informed decision if making their analysis ART based would be helpful.

# CHAPTER 3. A SECURITY TOOLBOX FOR DETECTING NOVEL AND SOPHISTICATED SOFTWARE ANOMALIES

This chapter presents a summary of key concepts from four of our papers as well as additional work performed for this thesis that does not appear in the original publications. These works all address our second research question: ***How can we statically identify likely candidates for novel and sophisticated software anomalies?***

1. *Security Toolbox for Detecting Novel and Sophisticated Android Malware* by Benjamin Holland, Tom Deering, Suresh Kothari, Jon Mathews, Nikhil Ranade. The paper was published in the demonstration track at the 37th International Conference on Software Engineering (ICSE) in May of 2015 in Firenze, Italy.

2. *Interactive Visualization Toolbox to Detect Sophisticated Android Malware* by Ganesh Ram Santhanam, Benjamin Holland, Suresh Kothari, Jon Mathews. The paper was published at the 14th IEEE Symposium on Visualization for Cyber Security (VizSec) in October of 2017 in Phoenix, Arizona.

3. *Human-on-the-loop Automation for Detecting Software Side-Channel Vulnerabilities* by Ganesh Ram Santhanam, Benjamin Holland, Suresh Kothari, Nikhil Ranade. The paper was published at the 13th International Conference on Information System Security (ICISS) in December 2017 in Mumbai, India.

4. *Intelligence Amplifying Loop Characterizations for Detecting Algorithmic Complexity Vulnerabilities* by Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, Suresh Kothari. The paper was published at the 24th Asia-Pacific Software Engineering Conference (APSEC) in December 2017 in Nanjing, China.

Searching for novel malware can be like looking for a needle in the haystack, but without knowing what a needle is or having ever seen one. In 2010 we learned of Stuxnet [75], a targeted nation-state level attack against an Iranian nuclear research site. The attack was only detected, some speculate intentionally, when it began to utilize noisy traditional attack vectors such as USB malware propagation. Recently we have seen a proliferation of high-level logic bugs in SSL [6, 7] and even a 25-year-old logic bug in the Bash shell [8]. While most would agree that these bugs were honest mistakes, a few have speculated that some may have been added with malicious intent [76]. Since we have no way to determine intent solely by examining code, a security analyst must consider software bugs as potential malice. In either case, the consequences can be catastrophic. When the stakes are high, the current practices for malware detection are far from adequate. The DARPA APAC program [61] was created to create new techniques and tools to detect sophisticated Android malware capable of causing serious damage in a Department of Defense scenario.

USAF Colonel John Boyd described the *OODA loop* as an iterative decision cycle of *observe*, *orient*, *decide*, and *act* [67]. Boyd developed this framework as a way to explain the unanticipated, superior agility of US fighter pilots in aerial combat situations. The paradigm of OODA loops applies equally well to the APAC context. To detect malware, our tools must be able to outmaneuver the capabilities of adversaries who will continue to develop new varieties of Android malware. Our Security Toolbox for Android is designed to utilize best-in-class automation and iteration techniques to maximize the odds of emerging victorious from this confrontation.

Leveraging this approach, we completed Phase I of the DARPA APAC program [61] as the top performing Blue team. Our team audited 77 Android source code applications developed by the Red team, of which 62 contained novel malware able to evade current automatic detection techniques. DARPA employed a control team to use current state-of-the-art tools to audit the applications along side Blue team performers. Our process correctly classified 66 (85.7%) applications as malicious or benign, found unintended malicious behaviors in 6 (7.8%) applications, and missed malware in only 5 (6.5%) of the applications consistently beating the control team. At the end of APAC's Phase II we ranked closely among the top three performing R&D teams. A demonstration of the Security

Toolbox developed for APAC is available online[1] as well as a timelapse audit of using the Security Toolbox on a Phase II challenge application[2].

The human-in-the-loop approach was later transitioned to DARPA's STAC program [77]. Instead of detecting malware, STAC sought to discover *algorithmic complexity* (AC) and *side channel* (SC) vulnerabilities in compiled code. During Phase I of the program, we attempted 30.95% of the engagement challenges with a 60% accuracy in discovering AC and SC vulnerabilities. We found that our process correctly identified the vulnerable code segments for the majority of applications, but we failed to collect enough evidence that an attack was within the given attack budget. Since the definition of AC and SC vulnerabilities is closely related to an attacker budget in terms of time or space, this evidence is most straightforwardly collected from a dynamic analysis, for which our approach lacked strength. This thesis was one of the primary works created to address this weakness and as tools matured our performance increased. In the beginning of Phase II we attempted 33.9% of the challenges with a 75% accuracy increasing to attempting 83.87% of the challenges with a 62% accuracy. The end of Phase II also brought a chance of collaboration with other R&D teams that did have strengths in dynamic analysis. During collaboration we were the only R&D team to achieve a 100% accuracy. During Phase III we were ranked as the most accurate R&D team individually attempting 88.37% of the challenges with a 76% accuracy and maintaining a 100% accuracy during collaboration.

The Security Toolbox is built on top of the Atlas program analysis framework [78] and is logically separated into five main components as detailed in the following sections.

## 3.1 General Purpose Program Indexers

Indexers are automatic program analyzers that run after the initial creation of the Atlas program graph, which seek to add to or refine the program artifacts in the graph. The Atlas program graph provides much of the information needed to analyze programs, but some information is a conservative estimate. One example is type inference where the dynamic dispatch edges may be

[1] https://youtu.be/WhcoAX3HiNU

[2] https://youtu.be/p2mhfOMmgKI

conservatively resolved to many potential targets. To address this problem, the Security Toolbox implements a Rapid Type Analysis (RTA) [28] strategy to exclude call edges to methods that should not be possible at runtime based on observed constructor calls. The Rapid Type Analysis Indexer is one of nine call graph refinement indexers[3] implemented as a part of this thesis. In addition, general purpose indexers implemented for this thesis include a 0-CFA points-to analysis[4], control flow dominance analysis[5], program slicing[6], and control and data flow interactions with standard language libraries[7,8] indexers. Since indexers may be also be clients of indexers, this work includes automation for prioritizing the execution of indexers based on a self-documenting dependency graph.

## 3.2 Domain-specific Semantic Indexers

Domain-specific semantic indexers provide support for the creation of nodes and edges that correspond to non-standard code artifacts. For example, to support Java JSP web servlets compiled as WAR files a special domain-specific indexer[9] was created during this work to support mapping the program artifacts specific to JSP applications.

Android provides a more illustrative example of the need for domain-specific indexers. Since Android makes extensive use of XML for its user interface, manifest, and other resources, many important program artifacts are missing in the Java program graph produced by Atlas. The Security Toolbox provides indexers to annotate and add missing program elements from these resources to the Atlas program graph[10]. Android's sensitive functionalities such as sending and receiving text messages, accessing geo-location information, or accessing user contacts are protected by runtime checks that enforce whether or not an application has been granted permission to invoke such functionalities. The Security Toolbox leverages the permission mappings produced by [79, 80] to provide additional indexer support to identify callsites to Android's permission protected methods.

[3] https://github.com/EnSoftCorp/call-graph-toolbox
[4] https://github.com/EnSoftCorp/points-to-toolbox
[5] https://github.com/EnSoftCorp/toolbox-commons
[6] https://github.com/EnSoftCorp/slicing-toolbox
[7] https://github.com/EnSoftCorp/java-toolbox-commons
[8] https://github.com/EnSoftCorp/c-toolbox-commons
[9] https://github.com/benjholla/AtlasWBP
[10] https://github.com/EnSoftCorp/android-essentials-toolbox

The indexer identifies the corresponding set of permission protected methods for each API version of Android then parses an Application's manifest and automatically annotates the correct API mapping onto the Atlas program graph. We have automatically scraped and encoded into Java objects the Google developer documentation for permissions, permission groups, and protection levels to aid in developing analyzers. Additionally we have recovered mappings for Android permissions to protection levels, and permissions to permission groups by mining their relationships from the Android source.

Domain-specific indexers can also be used to extend the Atlas XCSG[11] schema to support new languages. As a part of this thesis a complete language indexer was created for the *Elemental*[12] language described in Chapter 1.1.

## 3.3 Automatic Program Analyzers

The Security Toolbox defines an `Analyzer` programming interface that encapsulates the logic for traversing a program graph to extract an "envelope" (a subgraph that is either empty if the security property is satisfied or non-empty containing the necessary information to locate the violation of the security property). Analyzers encapsulate their descriptions, assumptions, and possible continuations to refine results or broaden a traversal. For example, one possible continuation for a data flow based taint analysis between a sensitive source and a sensitive sink that produced a graph that is too large to interpret would be to perform the same taint analysis with call, object, type, and flow sensitivities enabled. Analyzers have been subdivided into property, smell, confidentiality, integrity, and availability analyzers. A property is something the analyst should be aware of, but does not necessarily indicate malice, such as uses of native code. A smell is a heuristic similar to a property that indicates a stronger suspicion, which demands a justification such as using Java reflection to invoke a private API. The confidentiality, integrity, and availability (CIA) analyzers detect violations of CIA properties using taint analysis of sources and sinks, modification operations on sensitive mutables, and loop detection of expensive resources respectively.

[11] https://ensoftatlas.com/wiki/Extensible_Common_Software_Graph
[12] https://github.com/benjholla/Elemental

## 3.4 Interactive User Interface

The Dashboard (shown in Figure 3.1) is an interface for automating the execution and managing results of the Toolbox's automated analyzers. The Dashboard accounts for analyzer dependencies to enable the highest amount of parallel computation while running a multitude of analyzers. As results are computed, they are presented to the analyst in the work item queue on the right of the Dashboard. Results can be filtered by category and marked as reviewed. Optionally an analyst can make additional notes on a work item. Since work items correspond to subgraphs of the program graph, they can be named and even colored to help identify separate program subsystems. Program artifacts can be manually added or removed from a work item based on the colors given to program artifacts. The security toolbox includes additional interfaces to perform `what-if` style experiments for tracking/naming program artifacts, filtering program artifacts by applicable program artifact relationships, and cataloging a program's loops and branches.

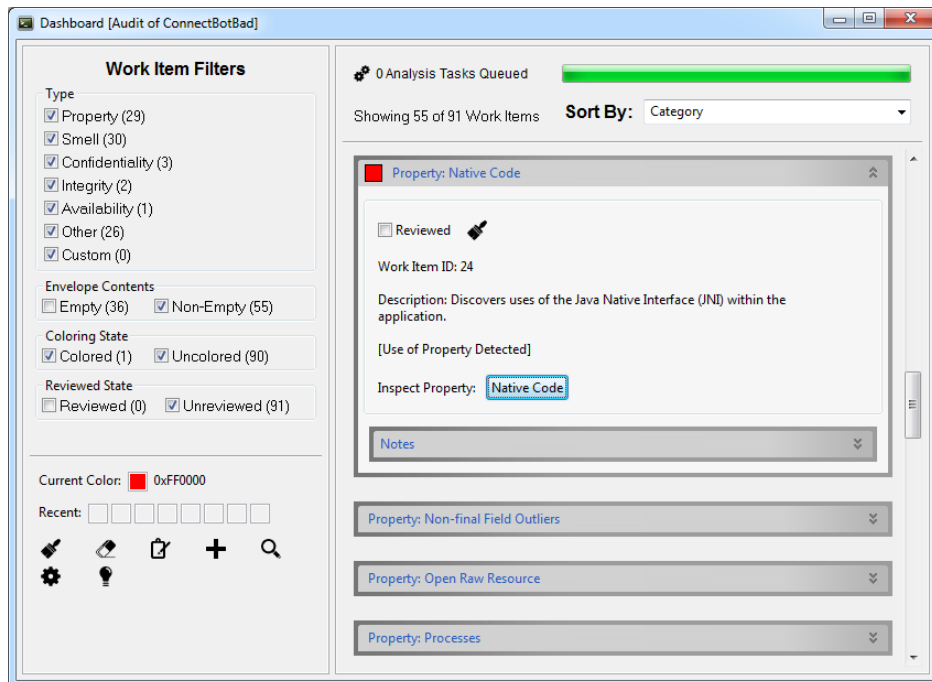



Figure 3.1 Security Toolbox Dashboard Interface

### 3.5 Interactive On-demand Analysis Visualizations

Smart Views are developed from the observation that there are several graph traversal queries that analysts use over and over again during audits, such as forward and reverse control and data flows, or discovering the declarative structures and instantiations of an object. To speed up such tasks, a graph for each of these queries can be automatically generated in response to mouse selection events on relevant source code or existing graph components. Smart Views can be customized for particular analysis tasks, such as showing a program slice or an Android user interface XML button event callbacks. Figure 3.2 shows a customized Smart View showing a data flow program slice that includes program artifacts in the Android XML resources. The graph was generated on-demand when the user clicked on the “destination” field in the source window to inspect its value.

The Security Toolbox includes Smart Views for a variety of tasks related to auditing applications for malware or security vulnerabilities. Our papers [81, 82] include visualizations for Android malware analysis, [83] includes visualizations for detecting side channel information leakage vulnerabilities, and [84] contains visualizations for detecting algorithmic complexity vulnerabilities.

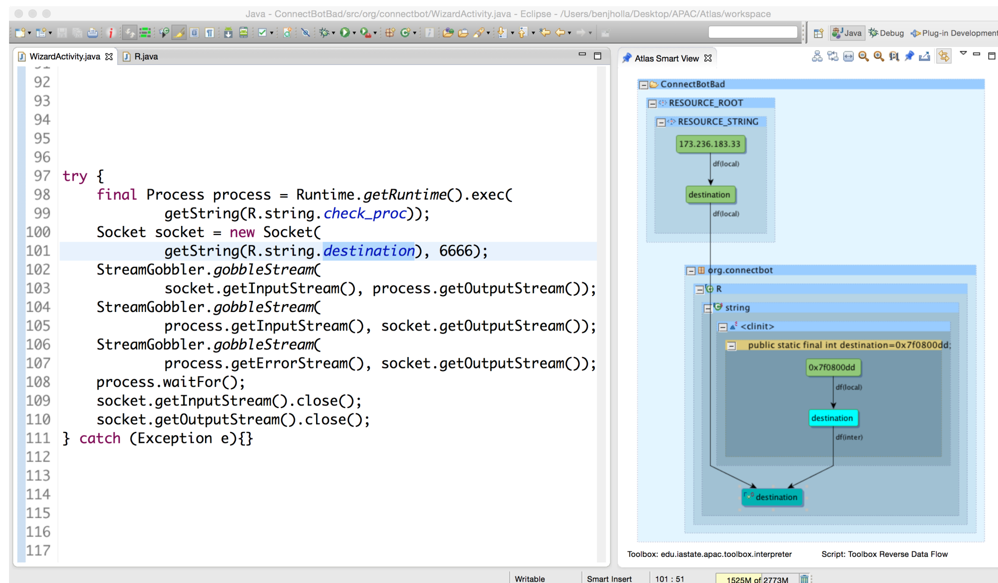



Figure 3.2 Smart View - Interactive On-demand Analysis Visualization

# CHAPTER 4. STATICALLY-INFORMED DYNAMIC ANALYSIS

This section presents a summary of key concepts from our paper *Statically-informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities* by Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. The paper was published in the engineering track at the 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) in October of 2016 in Raleigh, North Carolina. This work addresses our third research question: ***How can we effectively leverage the strengths of both static and dynamic program analysis to verify and validate software anomalies?***

Algorithmic complexity (AC) vulnerabilities [85, 86] are rooted in the space and time complexities of externally-controlled execution paths with loops. In the AC vulnerability commonly known as the "billion laughs attack" or an XML bomb [87], parsing a specially crafted input file of less than a kilobyte creates a string of $10^9$ concatenated "lol" strings requiring approximately 3 gigabytes of memory. At the extreme, a completely automated detection of AC vulnerabilities amounts to solving the intractable halting problem [88]. Existing tools for computing the loop complexity [89, 90] are limited and cannot prove termination for several classes of loops [91]. Based on our ongoing research for the DARPA Space/Time Analysis for Cybersecurity (STAC) [77] program, this paper describes a pragmatic engineering approach to detect AC vulnerabilities.

Our approach to detect AC vulnerabilities involves: (1) *Automated Exploration:* Identify loops, precompute their crucial attributes such as intra-procedural and inter-procedural nesting structures and depths, and termination conditions. (2) *Hypothesis Generation:* Through an interactive inspection of the precomputed information the analyst hypothesizes plausible AC vulnerabilities and selects candidate loops for further examination using dynamic analysis. (3) *Hypothesis Validation:* The analyst inserts dynamic analysis probes and creates a driver to exercise the program by feeding workloads to measure resource consumption for the selected loops.

Since detecting AC vulnerabilities is an open-ended problem, our approach strives to combine human intelligence with static and dynamic analysis to achieve scalability and accuracy. A lightweight static analysis is used for a scalable exploration of loops in bytecode from large software, and an analyst selects a small subset of these loops for further evaluation using a dynamic analysis for accuracy. This paper describes *statically-informed dynamic* (SID) analysis, which is applied using the above 3-step process, however the same 3-step process can be generalized to finding a broader class of software anomalies. That is human-in-the-loop reasoning and the strengths of both static and dynamic analysis can be combined to: (1) lightweight static analysis can be used to quickly scan potential program artifacts of interest, (2) an interactive static analysis can help to formulate a testable hypothesis and compute relevant program behaviors with respect to the hypothesis, and (3) dynamic analysis can be applied to validate or refute the hypothesis.

Our paper presented a pragmatic engineering approach to detect AC vulnerabilities. It demonstrated a case study application of *statically-informed dynamic* (SID) analysis and two tools that provided critical capabilities for detecting AC vulnerabilities. The first is a static analysis tool for exploring the software to find loops as the potential candidates for AC vulnerabilities. The second is a dynamic analysis tool that can try many different inputs to evaluate the selected loops for excessive resource consumption. The two tools are built and integrated together using the interactive software analysis, transformation, and visualization capabilities provided by the Atlas [78] platform.

The tools presented in this paper were demonstrated[1] and open sourced online, but have been succeeded by our more capable open source SIDIS toolbox[2,3] developed as a part of this thesis and described in Chapter 9. The SIDIS toolbox was named such because its supports *statically-informed dynamic* (SID) analysis and *dynamically-informed static* (DIS) analysis. The additional capability for DIS analysis was added for this thesis.

[1] https://youtu.be/8dH7q9aPD44
[2] https://github.com/kcsl/sidis-toolbox
[3] https://youtu.be/fgqQy1xPzao

## 4.1 Loop Call Graph

The *loop call graph* (LCG) is a succinct representation of crucial information necessary to hypothesize AC vulnerabilities, namely the inter- and intra-procedural nesting structure of loops and how they are reached through a call chain. The LCG represents the information compactly as a multi-attributed directed graph. The nodes in the LCG are methods containing loops and methods that call other methods which also contain loops. There is an edge from method $m_1$ to method $m_2$ in the LCG if $m_1$ calls $m_2$. Further, nodes and edges have color attributes. A node is colored blue if it contains loops, and grey otherwise. An edge is colored yellow if the callsite of $m_2$ is located within a loop in $m_1$ (indicates that loops in $m_2$ are inter-procedurally nested), and black otherwise. Explicit loops as well as loops due to recursion are identified in the LCG. The accuracy of LCG depends on the accuracy of the underlying call graph construction algorithm. Our implementation of LCG allows the flexibility to use different algorithms to manage the accuracy by a suitable choice of an algorithm. Typically, we use a class hierarchy analysis for computing the call graph, which can be substituted with more expensive algorithms supported by the Security Toolbox for increased accuracy. The loops in the bytecode are identified using the DLI algorithm [92]. The information about loops is computed and added to the program graph database stored by Atlas [78].

The LCG *Smart View* is an interactive tool that enables the analyst to explore and understand loops nested inter-procedurally. First, the view itself serves as succinct visual index of loops and their inter-procedural nesting. This is essential for the analyst to hypothesize AC vulnerabilities because if $m_1$ declares a loop $l$, and $m_1$ is called from a loop $l'$ in $m_2$, the LCG helps the analyst to reason that $l$ can be nested in $l'$ in that calling context. Second, the LCG Smart View is interactive. The LCG Smart View inherits 2-way source correspondence from Atlas. The analyst can double click on a method in the LCG Smart View to go to the code for that method. Alternately, when an analyst clicks on a method name in the source code window, the LCG window instantly updates to show the LCG for the selected method. Since the LCG for real-world programs can be large, to scale the visualization and make large graphs comprehensible, the Smart View allows users to incrementally expand and collapse nodes at any level of the graph, and offers a textual search to easily locate

methods. Third, the LCG Smart View seamlessly integrates with other Atlas Smart Views (e.g., control flow and call graph Smart Views). The analyst can compose analyses by feeding a selection in the LCG Smart View as input to another Atlas Smart View. For example, if the analyst opens the control flow Smart View alongside the LCG Smart View and clicks on a method in the LCG, the method's control flow graph (CFG) is shown in another window. This is illustrated in Figure 4.1. Along with the LCG view for Java's implementation of `TimSort` [93] it shows the CFG for a method selected in the LCG window. Overall, the LCG helps the analyst to explore and understand the loops interactively in order to hypothesize AC vulnerabilities.

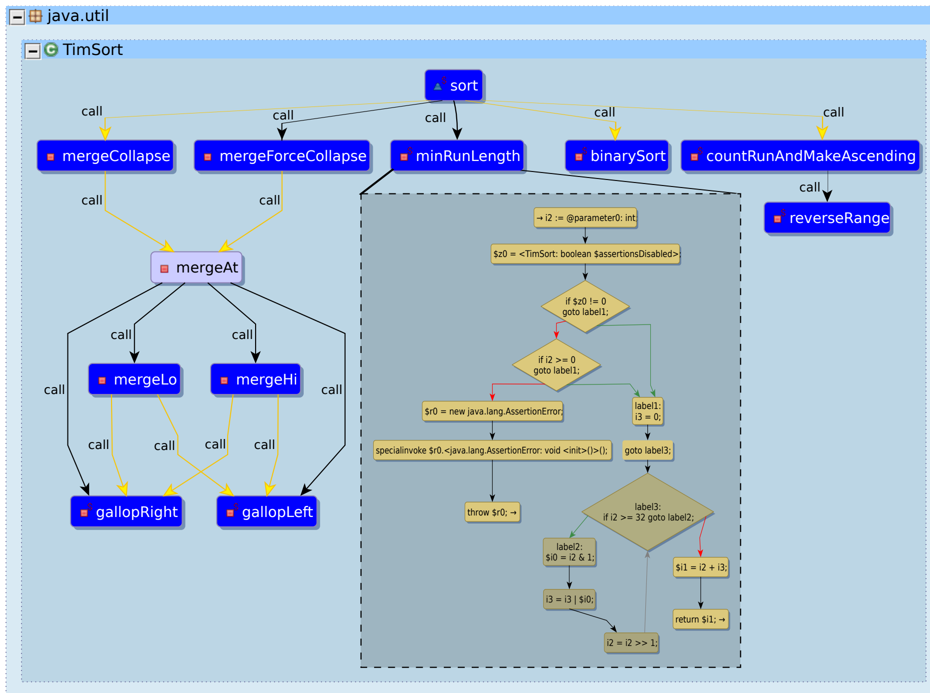



Figure 4.1 LCG for TimSort along with the CFG of `minRunLength`

## 4.2 Probe Instrumentation

The analyst can click on a LCG node to select a method and then click the SIDIS toolbox UI menu button to instrument the methods for a dynamic analysis of loops in the selected method. The SIDIS toolbox automates the insertion of probes into bytecode, and records the runtime measure-

ments for loops. We refer to dynamic analysis instrumentation that records runtime measurements as *probes*.

The SIDIS toolbox defines an extensible interface to implement new probe types and currently supports the following probes types:

1. *Execution Counters.* An iteration counter tracks the number of times a statement such as an assignment, branch, or loop header (entry point of a loop) is executed. Execution counter measurements are platform independent and produce repeatable results for deterministic code.

2. *Wall Clock Timers.* A wall clock timer uses timestamps to measure the cumulative time spent between program points. Time-based measurements, although noisy and inaccurate [94], are nevertheless useful. For example, caching or garbage collection side effects on the runtime are captured by the time measurements but not through iteration counters.

3. *Value Probes.* Measure the runtime value of a primitive or Object variable at a particular program point.

4. *Cardinality Probes.* Measure the size of a Java collection at a particular program point.

The SIDIS toolbox instruments Jimple [95] (an intermediate representation for Java bytecode) produced from the bytecode of the application. TCA computes the list of probes to be inserted and the corresponding offsets (locations within the method) where they should be inserted. When the driver is executed, it invokes the instrumented bytecode that collects the measurements made by the probes for each workload.

## 4.3 Dynamically-informed Static Analysis

The instrumentation results can be optionally imported back into the program graph as graph attributes to enable a *dynamically-informed static* (DIS) analysis. For example, inserting iteration counters on the control flow graph for the code shown in Listing 4.1 could be collected and imported as a new attribute on control flow nodes for the number of times the statement was executed. The

Security Toolbox includes a Smart View overlay that can visualize execution count information as a heat map on control flow graphs. Figure 4.2 shows a heat map overlay with a color gradient ranging from low execution counts colored cold (blue) to relatively high execution counts colored hot (red).

Listing 4.1 Nested Loop Execution Count Probe Example

```
1  public static int run(int n) {
2    int c = 0;
3    for (int i = 1; i <= n; i++) {
4      for (int j = 1; j <= i; j++) {
5        for (int k = 1; k <= j; k++) {
6          for (int l = 1; l <= j; l++) {
7            c = c + 1;
8          }
9        }
10     }
11   }
12   return c;
13 }
```

## 4.4 Time Complexity Analyzer

For measurements collected via the wall clock timer, TCA produces a plot of the workload size versus the measured time. For measurements collected via the iteration counters, TCA includes an engine for calculating a regression fit of the observed number of iteration counts. Similarly to existing tools [96, 94], TCA uses linear or power-law models to fit the observed measurements and estimates empirical complexity. In particular, TCA plots a graph of actual measurements against the workload size on a log-log scale using JFreeChart[4] The plot also reports the slope and the $R^2$ error for the best fit obtained by the linear regression engine. A slope of $m$ on the log-log plot indicates the measured empirical complexity of $n^m$ as shown in Figure 10.3.

The worst case complexity for arbitrary programs is not always computable (amounts to the halting problem). Static analysis tools such as COSTA [89] and AProVE [90] formulate and solve recurrence relations to compute the asymptotic complexity or termination of loops. However, they only work for a restricted class of loops [91, 97, 98], or do not support Java bytecode [99].

[4] www.jfree.org/jfreechart

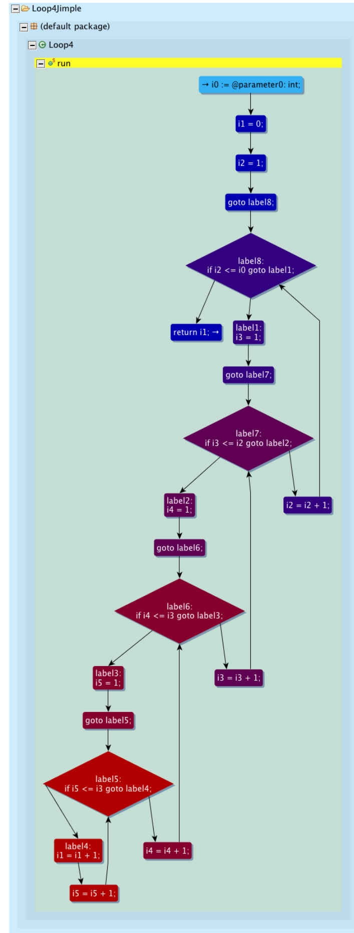



Figure 4.2 Control Flow Graph Execution Count Heat Map Overlay

Dynamic analysis approaches typically use benchmarks or fuzzing techniques to provide input workload for the program. Some dynamic analysis tools require the source code (not always available) to add profiling information [94]. Recent tools and techniques for *input-sensitive profiling* [96] automatically estimate the size of the input for generating the workload for individual routines in a program. The algorithmic profiling tool *AlgoProf* [100] introduced the idea of algorithm profiling as supported by SID tools. *AlgoProf* does not support several important capabilities we have introduced such as selective instrumentation or interactive visualization. The most severe limitation of *AlgoProf* is its overhead, both in terms of space and time. A SID analysis to insert selective instrumentation helps overcome this limitation.

# CHAPTER 5. TARGETED DYNAMIC ANALYSIS

This chapter presents a summary of key concepts from our *Mockingbird* framework that enables targeted dynamic analysis of Java programs. This work addresses our fourth research question: ***How can we systematically drive a targeted dynamic program analysis to explore relevant program paths?***

We present the Mockingbird framework; it combines static and dynamic analyses using a *statically-informed dynamic analysis* approach described in Chapter 4. This approach suits analysis tasks that can be subdivided into: (a) extracting relevant code buried in large and complex software, and (b) verifying specified properties with respect to the relevant code. An important application is detecting *side channel* (SC) and *algorithmic complexity* (AC) vulnerabilities [77]. The paper [83] describes how statically-informed dynamic analysis can be applied for detecting such open-ended vulnerabilities. We demonstrate the use of the Mockingbird framework for detecting AC vulnerabilities (e.g., Billion Laughs [101]). Combining static and dynamic analyses opens the possibility to have the best of both worlds - cover all execution paths and yield precise results. However, how to do it effectively is a challenge [48, 49, 50].

The Mockingbird framework is an innovative integration of existing static and dynamic analysis tools and a newly developed component *Object Mocker* that enables the integration. Specifically, the Mockingbird framework incorporates:

- Existing static analyzers built using the Atlas platform [78, 102, 103, 60]. These analyzers are used to extract the relevant code - the code that is potentially vulnerable.
- The existing AFL fuzzer [52] supplemented by the Kelinci, a tool that interfaces AFL to operate on Java programs [104].
- The Object Mocker that can automatically create harnesses to apply the AFL fuzzer to dynamically analyze the relevant code extracted using the static analyzers.

The key novelty of the Mockingbird framework is its ability to enable a *targeted dynamic analysis* (TDA). TDA performs dynamic analysis of just the relevant code using binary inputs. The framework provides the capability to mock any Java object type and produces a testing harness that transforms binary inputs to appropriate object types. For fuzzers such as AFL, users manually create a test harness that translates the binary inputs from the fuzzer to the data structures expected by the target program. Developing the harnesses manually for just the relevant code is difficult and laborious because the program state must be considered as the input. The program state is communicated to the relevant function via the *stack memory* using function parameters and return values or through the *heap memory* using reads and writes to global variables. The harness incorporates mocked objects that mimic the program artifacts that carry the inputs into the relevant code. The Mockingbird framework creates the harnesses automatically. The framework earns its name from the term "mock objects" as used by the testing community [105].

The Mockingbird framework optionally allows constraints to be placed on the values of object fields that store the encapsulated program data. With this enhancement, it is possible to systematically incorporate domain-specific knowledge into the automatically generated test harnesses. For example, if a program only operates on byte arrays that begin with a magic sequence, such as `0xFFD8` in the case of JPEG file formats, then the test harness can simply prefix the program input with the known sequence to increase the chances of quickly driving the program execution in meaningful ways. The Mockingbird framework supports configurable input generation constraints to be specified for test harnesses as a means for injecting domain knowledge.

## 5.1 A Motivating Example for Mocking

Mock objects are simulated objects that mimic the behavior of real objects in controlled ways. Mock objects serve two important purposes: (1) they allow direct control of the data values that could have reached the relevant code, and (2) they effectively isolate the relevant code by breaking existing control and data dependencies (replacing an object with a mock object removes dependencies since mock objects have no program dependencies).

The need for mocking is illustrated with a simple code example shown in Listing 5.1. We shall also use this example to illustrate how the test harness can be generated automatically.

Listing 5.1  Motivating Example for Mocking

```
public class Example {
  public static boolean isVowel(char c) {
    return c == 'a' || c == 'e' || c == 'i'
        || c == 'o' || c == 'u' || c == 'y';
  }
  class Pet {
    private String name;
    public Pet(String name) {
      this.name = name;
      sleep(5000);
    }
    public String getName() {
      return name;
    }
    public double getVowelRatio() {
      double vowels = 0;
      String name = getName().toLowerCase();
      for(char c : name.toCharArray()) {
        if(isVowel(c)) {
          vowels++;
        }
      }
      return vowels / (name.length() - vowels);
    }
  }
}
```

The `getVowelRatio` method has a division-by-zero vulnerability that occurs when the input consists of only vowels. In order to perform a TDA of the `getVowelRatio` method using existing tools such as Kelinci [104], which adapts the AFL fuzzer [52] to execute Java programs, a test harness would have to be developed that constructs instances of the `Pet` class with desired string values to test. Furthermore, constructing the objects could involve code that is unnecessary for mocking but it consumes significant execution time. The `sleep` on line 10 symbolizes such unnecessary code. By mocking the instance variable "`name`", we can directly control the value of "`name`" and elide the expensive constructor logic.

## 5.2 Mockingbird Framework

Figure 5.1 depicts an overview of the Mockingbird framework architecture and workflow.

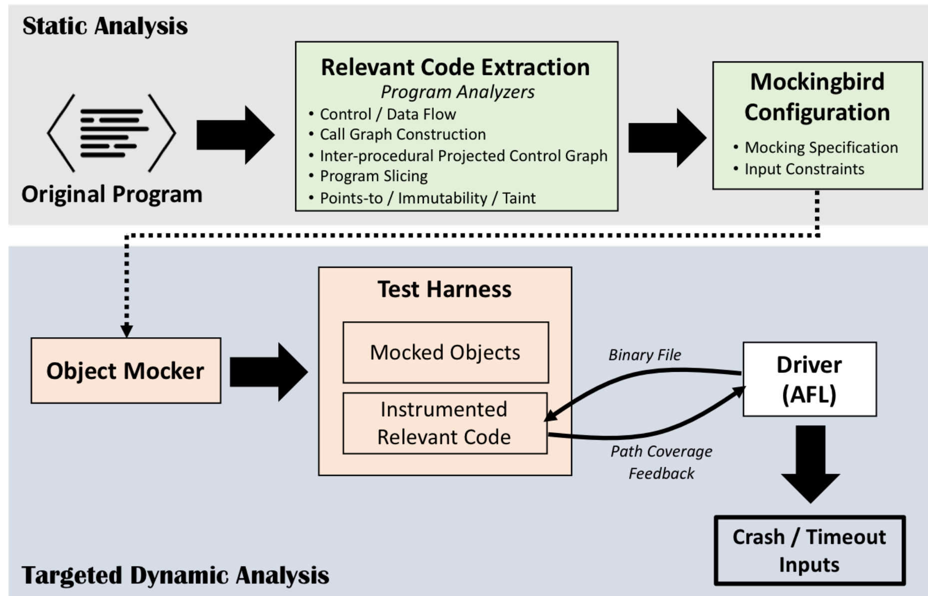



Figure 5.1 Mockingbird Framework Architecture and Workflow

As shown in Figure 5.1, a variety of program analyzers are used to extract the relevant code given a class of vulnerabilities. The static program analyzers are designed to be configurable to cater to different classes of software vulnerabilities and the analysis results in the inter-procedural projected control graph (PCG) (described in Chapter 6) that succinctly captures relevant program behaviors [60, 59, 106].

These program analyzers are built on top of Atlas [78], a graph database platform with graphs as the unifying abstraction to build program analyzers for a variety of languages including Java and Java bytecode. We performed a case study to illustrate how these program analyzers are used to extract the relevant code from large and complex software.

As shown in Figure 5.1, the relevant code feeds into the Mockingbird's configuration component. This component builds a configuration file, which serves as the specification for creating the harness. For the code shown in Listing 5.1, the `getVowelRatio` instance method in the `Pet` class is extracted as the relevant code. The corresponding configuration file is shown in Listing 5.2.

Listing 5.2 An Example of Mockingbird Configuration

```
1  {
2    "definition": {
3      "class": "Example$Pet",
4      "method": "getVowelRatio",
5      "instance_variables": [
6          {
7              "name": "name"
8          }
9      ]
10   },
11   "config": {
12     "timeout": 1000
13   }
14 }
```

The configuration file human-readable on purpose so that a human operator can easily refine the mocking specifications. As mentioned earlier, one purpose is to enable injection of domain knowledge to constrain the inputs to improve the TDA efficiency. It also provides an opportunity to manually verify and alter the mocking.

Figure 5.1 shows the Object Mocker, which creates the test harness that incorporates the mocked objects - as specified by the configuration file. Internally, the Object Mocker leverages the bytecode manipulation capabilities provided by ByteBuddy [107] and the ubiquitous ASM [108] bytecode manipulation library to generate runtime compatible mock objects containing only the fields and methods required to execute the relevant code. With the configuration file shown in Listing 5.2, the `Pet` object is mocked, without calling the constructor, by providing a simulated instance variable called “`name`” (the fact that “`name`” is a `java.lang.String` type is picked up automatically by the framework using Java reflection).

Pure methods (methods that do not mutate pre-existing program state) as identified by [103] are not mocked. For the motivating example, the `isVowel` method is correctly identified as pure and automatically excluded from mocking. By default, the JDK is not mocked.

As shown in Figure 5.1, after the harness is created the rest of TDA can be performed with AFL. AFL iteratively generates new binary inputs guided by a genetic algorithm to maximize path coverage.

Importantly, the Mockingbird framework can use another fuzzer in place of AFL. Moreover, a fuzzer such as AFL can be further enhanced by creating custom tools that can work with the fuzzer. The Mockingbird capability to programmatically control the data of each mocked field, is valuable for developing custom dynamic analysis tools. A tool called afl-unicorn [109] creates test harnesses and performs targeted dynamic analysis but it works only for assembly languages and does not address the problem of mocking objects. Godefroid introduced Micro execution in [110], which enables targeted dynamic analysis for low level languages by introducing a specialized virtual machine. Several object mocking tools exist, such as TestNG and Mockito [111], but we found that these tools made many low level assumptions that prevented the direct control of object states that we required for generalized dynamic analysis.

Broadly speaking, program analysis techniques can be divided into two main camps: static analysis and dynamic analysis. Static analysis can be efficient but it is imprecise. Dynamic analysis is precise but it is inefficient. Because of diametrically opposite tradeoffs, combining static and dynamic analyses opens the possibility to have the best of both worlds. However, how to do combine static and dynamic analyses effectively is a challenge. The Mockingbird framework is designed to address this challenge. The framework represents a major engineering effort involving integration of existing static analysis and fuzzing tools and the development of an innovative component Object Mocker. The Object Mocker can mock any Java object. Case studies included in Chapter 10 illustrate how the framework can be effectively applied in practice.

Our implementation is open source and freely available online[1].

[1] https://github.com/kcsl/Mockingbird

# CHAPTER 6. COMPUTING RELEVANT PROGRAM BEHAVIORS

This section presents a summary of key concepts from our paper *COMB: Computing Relevant Program Behaviors* by Benjamin Holland, Payas Awadhutkar, Suresh Kothari, Ahmed Tamrawi and Jon Mathews. The paper was published in the demonstration track at the 40th International Conference on Software Engineering (ICSE) in May of 2018 in Gothenburg, Sweden. Material in Section 6.3 is summarized from [58], which describes additional properties and applications of the *projected control graph* (PCG). Material in Section 6.4 has been added for this thesis and has not appeared previously. This work addresses our fifth research question: ***How can we statically compute a program's relevant behaviors to deal with the path explosion problem?***

Many software engineering tasks require analysis and verification of all program behaviors relevant to the task. For example, all relevant behaviors must be analyzed to verify software for safety or security. We will discuss the capabilities of our COMB tool to compute relevant program behaviors and the underlying concepts with the help of examples. Consider the problem of verifying software for *Division By Zero* (DBZ) vulnerabilities. Line 24 of the code in Listing 6.1 involves division by `d`, which must be checked for a DBZ vulnerability.

As described in [59], each Control Flow (CF) path yields a behavior represented by the corresponding *trace*. Each trace is a regular expression of program statements along a path. In this simple example without loops, each trace is a sequence of statements. There are six behaviors and their corresponding traces are: (B1) 8, 9, 10, 11, 15, 18, 19, 24; (B2) 8, 9, 12, 13, 15, 18, 19, 24; (B3) 8, 9, 10, 11, 15, 16, 17, 24; (B4) 8, 9, 12, 13, 15, 16, 17, 24; (B5) 8, 9, 10, 11, 21, 22, 24; (B6) 8, 9, 12, 13, 21, 22, 24. The first two traces exhibit the vulnerability.

Verifying any vulnerability poses two challenges: (1) computing the vulnerable behaviors out of all behaviors and, (2) path feasibility, i.e., checking feasibility of CF paths for the vulnerable behaviors. The large number of behaviors ($2^n$ behaviors for $n$ non-nested branch conditions) make the

problem computationally intractable. The path feasibility problem is equivalent to the satisfiability problem [54].

An efficient algorithm must compute the vulnerable behaviors without computing all the behaviors. A closer inspection reveals that multiple behaviors can be grouped so that each group correspond to a unique *relevant behavior* defined by a sub-trace that retains only the relevant statements and conditions. The relevant statements for the DBZ vulnerability are: 9, 19, 22, and 24. The relevant behaviors are: (RB1) 9, 15, 18, 19, 24; (RB2) 9, 15, 16, 24; (RB3) 9, 21, 22, 24 with the grouping: RB1: {B1, B2}, RB2: {B3, B4}, RB3: {B5, B6}. Note `if(C1)` is irrelevant because taking the true and false branches produce the same relevant behaviors.

Listing 6.1  Motivating Example for Projected Control Graphs

```
1   public class DBZ {
2     public static int a1 = 0;
3     public static int a2 = 1;
4     public static boolean c1 = true;
5     public static boolean c2 = false;
6     public static boolean c3 = true;
7     public static void main(String[] args) {
8       int x = a1 + a2;
9       int d = a1;
10      if(c1) {
11        x = a1;
12      } else {
13        x = a1;
14      }
15      if(c2) {
16        if(c3) {
17          int y = a1;
18        } else {
19          d = d - a1;
20        }
21      } else {
22        d = d + 1;
23      }
24      int z = x / d;
25    }
26  }
```

COMB uses the PCG abstraction [106] to compute the relevant behaviors directly and efficiently. The abstraction amounts to an efficient way to form the equivalence classes of behaviors to yield relevant behaviors. The Control Flow Graph (CFG) fuses together multiple facets of the program logic. Unlike the CFG, the PCG distills the behaviors for a particular facet of the program logic.

The CFG and its corresponding PCG for the DBZ example are shown in Figure 6.1. As borne out by the case studies, the use of the PCG can circumvent the challenges of analyzing a large number of paths and path feasibility.

## 6.1 The PCG Abstraction

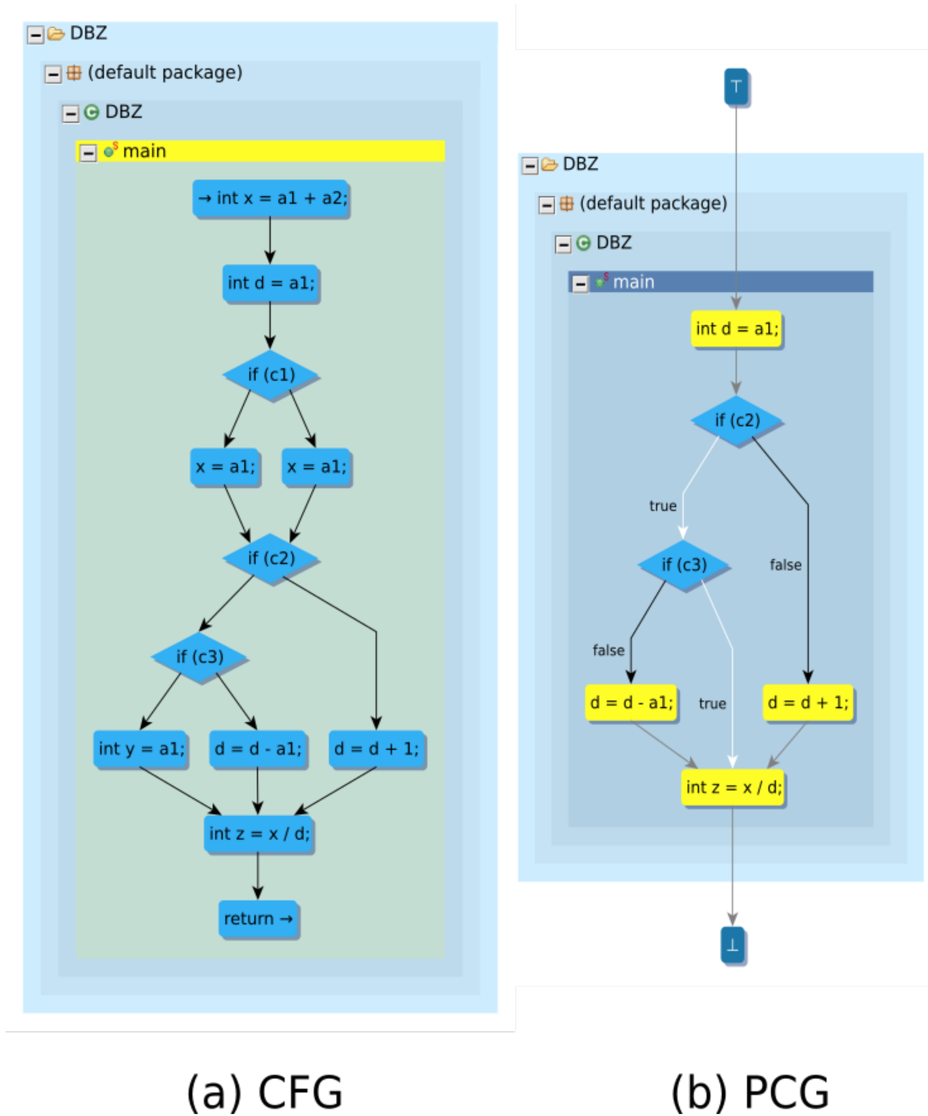



Figure 6.1 CFG and PCG for Example DBZ Vulnerability

The PCG abstraction and an efficient algorithm to compute it are originally described in [59]. Figure 6.1 shows the PCG for the DBZ vulnerability discussed in this study. Nodes for relevant statements are colored yellow and the `true` edges from branch conditions are colored white in Figure 6.1.

With respect to this work, key points about the PCG are:

- The PCG is obtained by transforming the corresponding CFG. In all our studies, we have found the PCGs are often much smaller than the corresponding CFGs. In an extreme example from our Linux verification study [106] the CFG for `client_common_fill_super` has 1,101 nodes, 1,179 edges, and 249 branch nodes. The corresponding PCG has 15 nodes, 28 edges, and 13 branch nodes.

- Filtering out unnecessary details is important to facilitate program comprehension. This is brought out in our demonstration video[1] by an example of a Linux bug that is easily spotted using the PCG.

- Relevant behaviors are derived by traversing the PCG paths. The video demonstration also shows a utility to count and compare the number of paths in a CFG and the corresponding PCG.

## 6.2 Loop Behaviors Model

To compute behaviors in the presence of loops, a model of loop behaviors is required as loops may iterate several times. Our loop behavior model [59] is intended for a class of problems in which the validity of safety or security properties does not depend on the exact number of times the loop iterates. This model is motivated by an empirical study of loops in open-source C and Java software and the need for a practical approach to account for loop behaviors to verify safety and security properties.

[1] https://youtu.be/YoOJ7avBIdk

Figure 6.2 shows an illustrative example with the loop-back edge cross marked. Consider the problem where a `Lock` must be followed by `Unlock` and it must not be followed by another `Lock`. Consider a loop with `Lock`, `Unlock`, and a `break` in the body. The model yields three relevant behaviors with traces: (1) $(c1, L, \overline{c2}, U)^+$, (2) $(c1, L, \overline{c2}, U)^* c1, L, c2,$ `break`, (3) $\overline{c1}$ with $L$ for `Lock` and $U$ for `Unlock`. $ci$ and $\overline{ci}$ denotes whether the condition $ci$ evaluates to true or false. The second behavior implies that the property is violated. The model requires a unique entry for each loop but allows multiple exits to account for `break` or `return`.

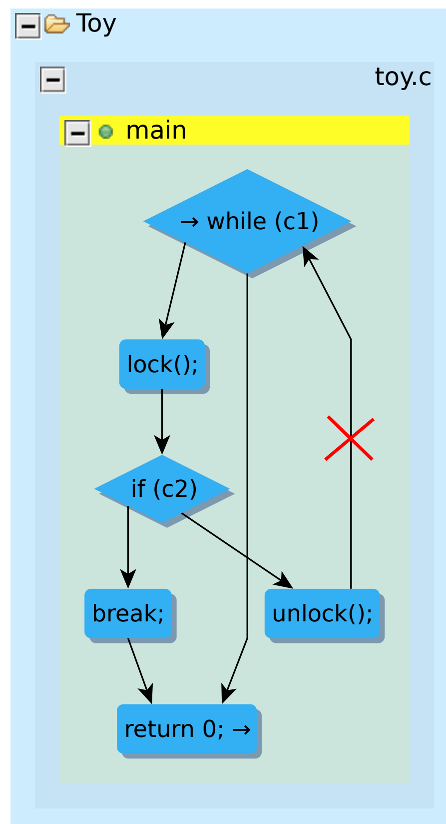



Figure 6.2 Illustrative Example of Loop Behavior Model

## 6.3 CFG to PCG Mapping as a Graph Homomorphism

Graph homomorphisms are structure preserving mappings, which preserve the adjacency of graph nodes and are widely used in applications involving graph coloring problems [112]. In 1981, Tarjan pointed out that many problems including finding shortest paths and solving sparse systems of linear equations can be translated to directed graphs. Tarjan then showed that the directed graphs represented a general path problem that could be solved efficiently given an appropriate graph homomorphism [113]. In 2017, Tamrawi and Kothari [58] showed that the CFG to PCG transformation is a graph homomorphism for control flow graphs. In the transformation of a CFG to PCG, the homomorphism is a one-to-one mapping between the nodes for relevant statements and relevant conditions and the other nodes in the CFG are mapped to their relevant successors in the PCG. Since the PCG collapses CFG paths into equivalence paths with respect to control flow events of interest and their relevant conditions, program invariants that hold true on a particular PCG path are homomorphic to program invariants that hold true on corresponding CFG paths.

## 6.4 Inter-procedural Projected Control Graphs

The PCG abstraction collapses control flow paths within a function, but the abstraction must be extended to consider inter-procedural flows. In this work we propose the Inter-procedural Projected Control Graph (IPCG) to consider control flow events of interest that span across multiple functions.

Atlas [78] represents inter-procedural control flow between functions as a *Call* relationship. A *Call* relationship exists between two functions and summarizes the existence of a callsite in the caller method that resolves to the target callee method in a Class Hierarchy Analysis [27]. To allow for more advanced call graph construction algorithms Atlas provides `Invoked Function` and `Invoked Signature` relationships as raw material for resolving static and dynamic dispatches. As a part of this work, we implemented nine different call graph construction algorithms[2] ranging from Class Hierarchy Analysis [27], to Rapid Type Analysis [28] and its variants [29], to an on-the-fly call graph construction algorithm using a 0-CFA points-to analysis.

[2] https://github.com/EnSoftCorp/call-graph-toolbox

Each call graph construction algorithm implemented for this work can be used to compute the target call functions for a callsite with $call(C)$. A *control flow graph* (CFG) can be formally defined as a graph $G = (N, E)$, where each node in $N$ is a program statement and each edge in $E$ is a control flow edge that represents the transfer of control from the predecessor to the successor statement. We define an *inter-procedural control flow graph* (ICFG) to be a set of each graphs $\{G_i\}$ such that each $G_i$ is a CFG. Let each callsite $C_i$ in each statement in $\{G_i\}$ have a unique callsite id computable with $callid(C_i)$. For each callsite $C_i$ in each statement node $N_j$ of each $G_i$, remove the outgoing control flow edges from $N_j$ in $G_i$ and add an inter-procedural control flow entry edge from $N_j$ to the CFG root node of $call(C_i)$. For each CFG leaf in $call(C_i)$ we add an inter-procedural control flow exit edge to each successor of $N_j$ along removed control flow edges from $N_j$ in $G_i$. For each pairing of an inter-procedural control flow entry edge and its corresponding inter-procedural control flow exit edge set assign an edge attribute of $callid(C_i)$.

The ICFG captures inter-procedural flow for functions and models the flow into and out of functions. Similar to the inter-procedural program slicing work proposed by Horwitz in [57], a client analysis performed on an ICFG can simulate the call stack by ensuring that a traversal of a *Inter-procedural Control Flow Entry* is eventually matched with the corresponding *Inter-procedural Control Flow Exit* edge identified by the callsite ID attribute, whereas ignoring the callsite ID attributes would be an over-approximation of the program's runtime behaviors. With respect to computing inter-procedural projected control graphs, an ICFG is a natural analog to a standard CFG. Since an ICFG captures looping due to recursive function calls, recursion is automatically handled by the existing PCG loop abstraction model. To compute an IPCG on an ICFG we:

1. Augment the ICFG entry function with a single master entry ($\top$) node and an edge to the entry function's control flow root to represent the start of relevant behavior

2. Augment the ICFG entry function with a single exit ($\bot$) node to and edges from each entry function control flow exit to represent the end of relevant behavior

3. Consider *Inter-procedural Control Flow Entry* and *Inter-procedural Control Flow Exit* to be unconditional control flow edges and compute the PCG as normal

To explore IPCGs further, we examine two small motivating examples shown in Listing 6.2 and Listing 6.3 and the corresponding ICFG to IPCG transformations of each shown in Figure 6.3 and Figure 6.4 respectively. In the figures, ICFG inter-procedural edges are dotted, where black is the entry edge and gray is the exit edge. Local control flow edges are colored black for `false`, white for `true`, and gray for unconditional control flows. Note that Listing 6.2 and Listing 6.3 are identical except that Listing 6.3 has an additional recursive callsite to `bar()` on line 13. In both examples, we choose control flow events of interest (the declaration of `a` and the declaration of `c`) that are spread between the two functions. In Figure 6.3 the ICFG shows there are 3 callsites to bar() that could lead to a control flow event interest. The transformation of the ICFG to an IPCG preserves the unique call stack contexts by producing two elided control flow paths from the return of `bar()`. In one context the next relevant control flow node is the control flow condition guarding the declaration of the variable `c` and in another context the return node is last relevant control flow node before the master exit. Figure 6.4 shows how recursion call also be handled by the IPCG transformation. The addition of the fourth callsite to `bar()` adds another unique call stack context relevant to the retained IPCG behaviors. In general, an IPCG path added as a result of a callsite will only be added if the callsite introduces a path to an event of interest.

Listing 6.2 Motivating simple example for ICFG

```
1  public class ICFGToy {
2    public void foo() {
3      int a;
4      bar();
5      bar();
6      bar();
7      return;
8    }
9    public void bar() {
10     int b;
11     if(new Random().nextBoolean()) {
12       int c;
13     }
14     return;
15   }
16 }
```

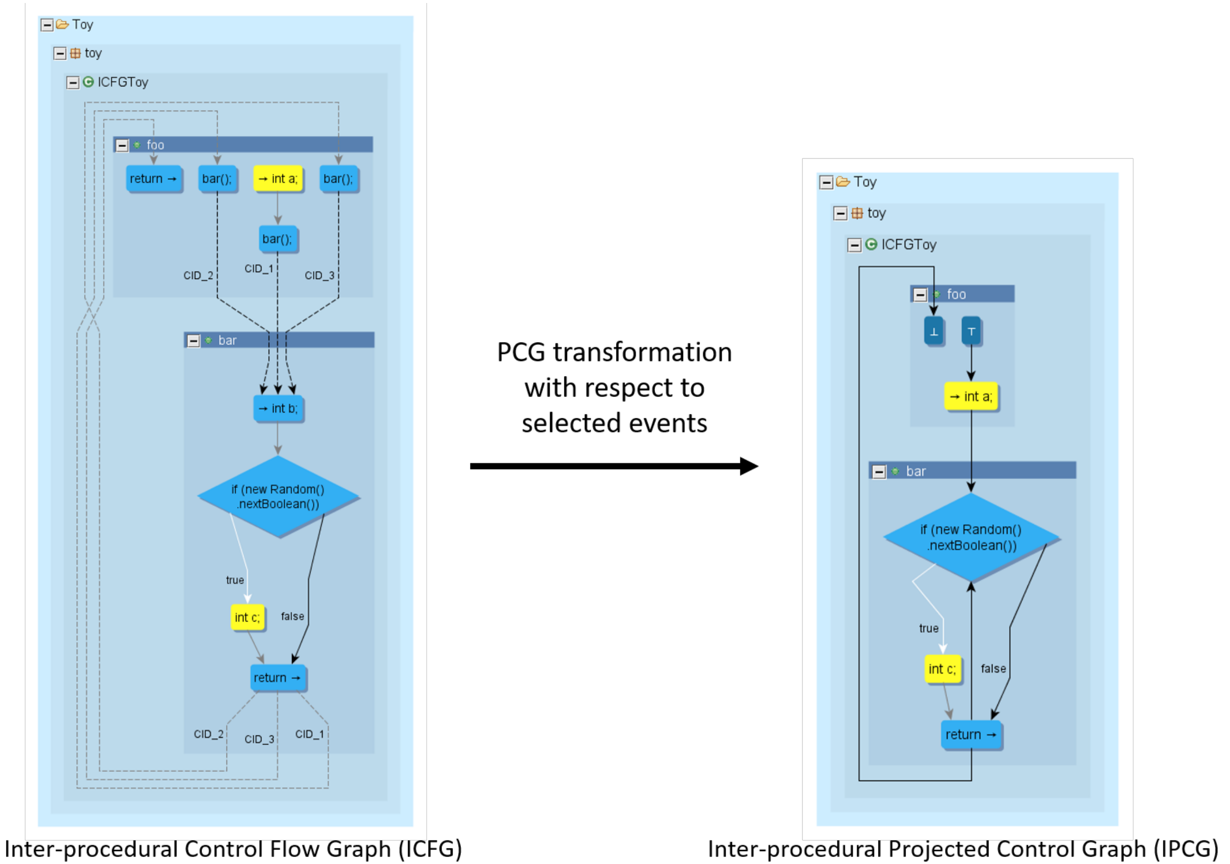
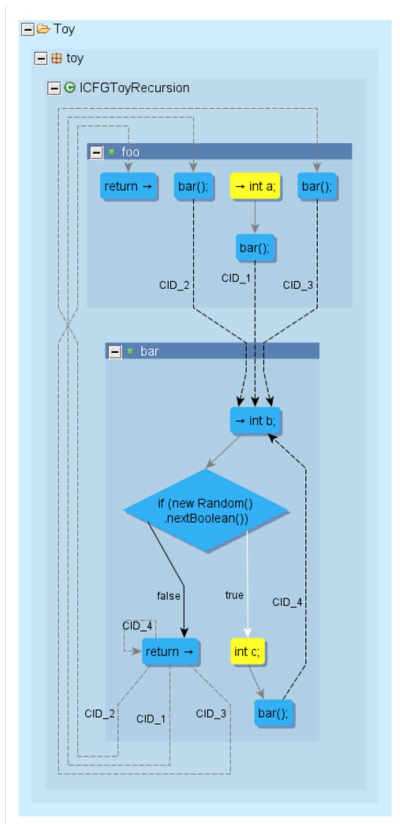


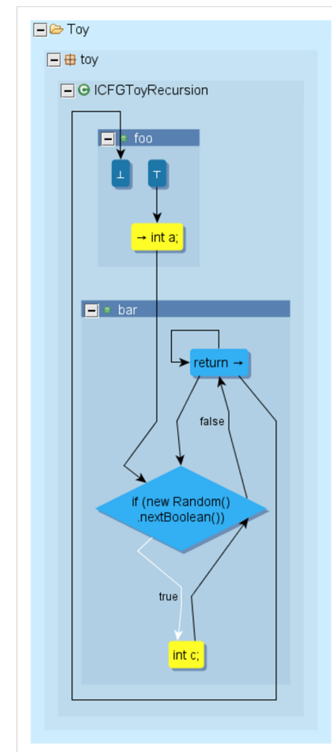

Figure 6.3 Illustrative Example of IPCG Transformation of a Simple ICFG

Listing 6.3 Motivating recursive example for ICFG

```
1  public class ICFGToyRecursion {
2    public void foo() {
3      int a;
4      bar();
5      bar();
6      bar();
7      return;
8    }
9    public void bar() {
10     int b;
11     if(new Random().nextBoolean()) {
12       int c;
13       bar();
14     }
15     return;
16   }
17 }
```




PCG transformation with respect to selected events




Inter-procedural Control Flow Graph (ICFG)

Inter-procedural Projected Control Graph (IPCG)

Figure 6.4 Illustrative Example of IPCG transformation of a Recursive ICFG

# CHAPTER 7. BENCHMARKING STATE-OF-THE-ART IMMUTABILITY ANALYSES

This section presents a summary of key concepts from our paper *Transferring State-of-the-art Immutability Analyses: Experimentation Toolbox and Accuracy Benchmark* by Benjamin Holland, Ganesh Ram Santhanam, Suresh Kothari. The paper was published at the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST) in March of 2017 in Tokyo, Japan. This work addresses our sixth research question: ***How can we evaluate static program analyses to detect function side-effects and aid in selection of relevant control flow events?***

Many applications in software analysis, testing and verification call for tools that analyze the immutability of objects. For instance, in software testing it is useful to first identify independently testable functions that do not have side-effects (according to Sălcianu and Rinard [114], a function has a side-effect if it mutates an object that existed prior to its invocation), as it allows the tester to focus on more complex areas of the codebase. Knowing that an object is immutable is particularly useful in verifying concurrent programs because an immutable object cannot be corrupted by thread interference. Knowledge of function side-effects is important for human comprehension and for automatically improving the accuracy of Daikon's likely invariant detection [47]. Finally, for security verification, immutability analysis is important because information leakage via space or time based side channels is primarily enabled through the presence of functions with observable side-effects [77, 115]. By identifying callsites to functions with side-effects, an immutability analysis can be useful for selecting control flow events of interest to reason about side channel vulnerabilities.

An object is mutated when any of the object's fields are updated, i.e., a field assignment is made after the object is initialized. In Java, an object can only be accessed through a reference, so

*immutability analysis* boils down to answering the question: *given code C and reference R, is the object O referenced by R mutated in the code C?*

An important question to ask is whether a dedicated immutability analysis is needed, given that Java already offers a mechanism for enforcing object immutability using the `final` keyword, which would require all fields (and the fields of fields) of an object to be marked `final`. First, marking an object immutable using the `final` keyword is impractical for general purpose programming, because it would lock the object's initialized state and prevent any further updates to the initialized object. Instead, it is more common for developers to define mutable object types, but treat particular instances of the object as immutable objects by simply not updating the object's fields. In fact, many IDEs encourage this design pattern by offering automatic generation of getter and setter functions for object fields. Second, our survey of 274,504 Java projects on Github using the Boa [116] framework showed that 38.45% of all types are potentially mutable having at least one non-final field [117]. Finally, there is no language mechanism for enforcing the immutability of array components. Hence, there is a need for reliable tools for recovering the immutability of objects, with applications to software testing as well as verification and validation.

Tools published in academic research related to immutability analysis [114, 118, 119, 120, 121] use one of two prominent approaches to immutability analysis: one employs a points-to analysis to infer immutability and another uses a type based inference system. A points-to based immutability analysis takes a top-down approach by first resolving aliasing relationships then using them to identify mutations to the aliased objects. The type inference based immutability analysis takes a bottom-up approach by identifying mutation at field assignments, and inferring the objects that may be affected by the mutation.

Our survey of the state-of-the-art for these two approaches to immutability analysis corroborates the findings of others [118, 119] that many tools only work small examples or cannot scale to the demands of realistic software. Moreover, the existing tools are largely academic prototypes that have not been rigorously tested for accuracy or have not been maintained and are unable to analyze programs written in later versions of Java. In particular, like many static analyses, a fully-

precise immutability is intractable in general (e.g., it may require a fully-precise pointer or alias analysis [34]), which forces tool implementers to make design choices or choose abstractions that necessarily affect the accuracy of their results.

The status of the current tooling prompted a need for (a) a robust implementation of the immutability analysis approaches that supports Java 8 and compiled Java bytecode, as well as (b) a benchmark that rigorously tests each approach for accuracy in the presence of various program features and analysis challenges. In this paper we re-implemented two prominent approaches to immutability analysis, namely a points-to based and a type-inference based approach. We presented the challenges we faced in the academic transfer and how we addressed them. In particular, our re-implementation of the points-to based immutability approach involved modifying a well known static analysis algorithm (0-CFA Andersen style pointer analysis [31]), our re-implementation of ReImInfer [118, 119] was more challenging because it required running several experiments and directly conversing with the original authors for clarifications to unstated assumptions made in their papers. Our re-implementations are built on top of a robust and well maintained program analysis platform [78] that provides native support for handling the necessary language features for Java 8 source and Java bytecode. We developed a benchmark to validate the basic correctness of the approaches implemented, and to test the accuracy boundaries inherent to each approach. Our micro-benchmark consisted of 250 basic assignment test cases, where each test case was also an executable proof showing the correct analysis result.

## 7.1 Overview of Existing Approaches and Tools

In this section we briefly summarize a type inference based approach and a points-to based approach to performing immutability analysis.

**Type Inference Based Approach:** The approach implemented by ReImInfer leverages a small auxiliary type system, a set of inference rules, and a type checker to infer the immutability of each object referenced in a program. The type system consists of three types with a hierarchy of READONLY :> POLYREAD :> MUTABLE, with READONLY as the most generic (super)

type and MUTABLE as the most specific (sub) type. A type can only be assigned to the same type or a supertype. The POLYREAD type acts as a halfway point between the READONLY and MUTABLE types to account for mutations made in one calling context but not another. For instance a mutation to a field in one method does not imply that the field is mutated in every method that accesses the field. By default references are assigned a set of all three types, except for instance variables which are initialized with the set READONLY and POLYREAD. The inference rules are iteratively applied to each assignment statement in the program using a type checker to remove unsatisfiable types until a fixed point is reached. For instance in a basic assignment $x = y$, the TASSIGN inference rule asserts the constraint that the types on $y$ should be less than or equal to the types on $x$. After fixed point is reached, the maximal type is selected from each set as the final type of the reference. This approach supports whole and partial program analysis with $O(3*n)$ worst case number of iterations, where $n$ is the number of assignments [118].

**Points-to Based Approach:** Given the results of a points-to analysis it is a straight-forward task to implement an immutability analysis. There are many approaches to implementing points-to analyses, but for this work we implement a standard 0-CFA Andersen-style [31] pointer analysis (a context-insensitive subset constraint based approach). This choice is motivated by the fact that adding context sensitivity significantly increases cost without necessarily significantly improving the precision [122]. Given the pointer analysis, an immutability analysis only requires a single iteration through each new allocation in the program. For each new allocation the analysis determines if any of the references to the new allocation were used to update fields. If the a mutation occurred then all references to the new allocation are marked mutable. Updates to array components are treated as mutations to the array itself.

## 7.2 Benchmark Design

Our approach to creating a benchmark that brings out the accuracy boundaries of the tools and embeds the ground truth within the test programs to evaluate the accuracy of the immutability analysis approaches was to design benchmark test cases with the following desirable properties.

- **Independently Analyzable** Each test case should be individually testable, with all its dependencies encapsulated.
- **Human-comprehensible** The test cases should be simple enough that developers and users can understand the accuracy boundaries of a tool by browsing the tests cases in which the tool failed to produce correct analysis results.
- **Annotated** The test cases should be annotated to facilitate programmatic querying of the answer key. This makes it easy to write a test harness and evaluate whether a static analysis tool answered correctly for a given test case.
- **Executable** The test cases should also be executable, with its output serving as a proof of correctness of the test case's annotated answer key.
- **Reproducible** The test case should be deterministic and bundled with an environment that easily reproduces the results for a given implementation.
- **Individually Citable** To promote sharing of results, collaboration between researchers, and to support reproducibility, each test case should be individually referable. The DOI numbers for the benchmark (and its future versions) are available online[1]. To cite a testcase, authors can specify the benchmark version by citing its DOI, along with the testcase category and number.

We discussed a novel method to generate test cases for covering basic assignments in Java that any immutability analysis tool should be able to successfully analyze. These test cases do not require the tool to resolve aliases in order to pass. The novelty of our method of generating the basic assignment test cases that have the above desirable properties is twofold: (a) *systematic:* the test cases are generated by starting from the assignments allowed by the language model, and (b) *exhaustive:* every possible assignment within the Java language is tested. Since an object can be mutated only through field assignments, and aliasing is not considered, any immutability analysis tool should be expected to pass these test cases.

[1] https://kcsl.github.io/immutability-benchmark

To ensure systematic, exhaustive coverage of basic assignment cases, it is important to note that all mutations occur as a result of an update (assignment after initialization) to a field. Let us first consider updates to class variables, instance variables, and array components of fields. Array fields must be given special attention because there is no language mechanism to enforce the immutability of array components and updates to array components mutate the object containing the array field. Fields can be categorized into three groups, class variables, instance variables of another object instance, and instance variables of "this" object instance. Finally, we consider assignments involving stack variables, i.e. parameters passed and returned values. The high level assignment patterns can be represented succinctly as an *assignment graph* as shown in Figure 7.1.

Listing 7.1 Basic Assignment Testcase

```
1  public class AGT117_This_Parameter {
2    @READONLY
3    public Test test = new Test();
4    public static void main(String[] args) {
5      new AGT117_This_Parameter().foo();
6    }
7    public void foo(){
8      System.out.println(test);
9      test.bar(test);
10     System.out.println(test);
11   }
12 }
13
14 class Test {
15   public Object f = new Object();
16   public void bar(Object p){
17     p = this; // no mutation
18   }
19   @Override
20   public String toString() {
21     return "Test [f=" + f + "]";
22   }
23 }
```

Each node in the assignment graph corresponds to a set of program artifacts that can be on the left hand side or right hand side of an assignment[2]. Each edge in the assignment graph can be instantiated multiple times, one for each pair of program artifacts in the source and destination node to generate a valid Java assignment. For instance, the downward edge to the right side of the

[2]In the assignment graph, we intentionally do not consider inherited fields of an object, i.e., references through the "super" keyword to avoid introducing polymorphism challenges.

assignment graph generates assignments from a final object, an `Enum`, a 'this' reference, a primitive (byte, char, short, int, long, float, double, boolean), 'null' or a String literal to a parameter. We enumerate all possible assignment pairs from the assignment graph to produce a total of 250 test cases.

Listing 7.1 shows the basic assignment test case corresponding to the assignment of a 'this' reference to a parameter (line 17). Clearly, it is independently analyzable because there are no third party dependencies, and a human can quickly comprehend the specific language feature being tested. The test also contains a main method that if executed prints the state of the `test` object before and after the assignment on line 17 is made in the `bar(Object p)` method invocation. Executing the test case produces reproducible output that indicates indicate the `test` field was not mutated. In this test, the assignment to a parameter does not mutate the parameter and so the `test` field is annotated with `READONLY`, which serves as an answer key that can be readily verified. When possible, we have designed the test cases to use only the necessary language features and program artifacts required to create each test case. In the above example, removal of any reference or method would arguably compromise the desired properties of the test case such as executability or human comprehensibility.

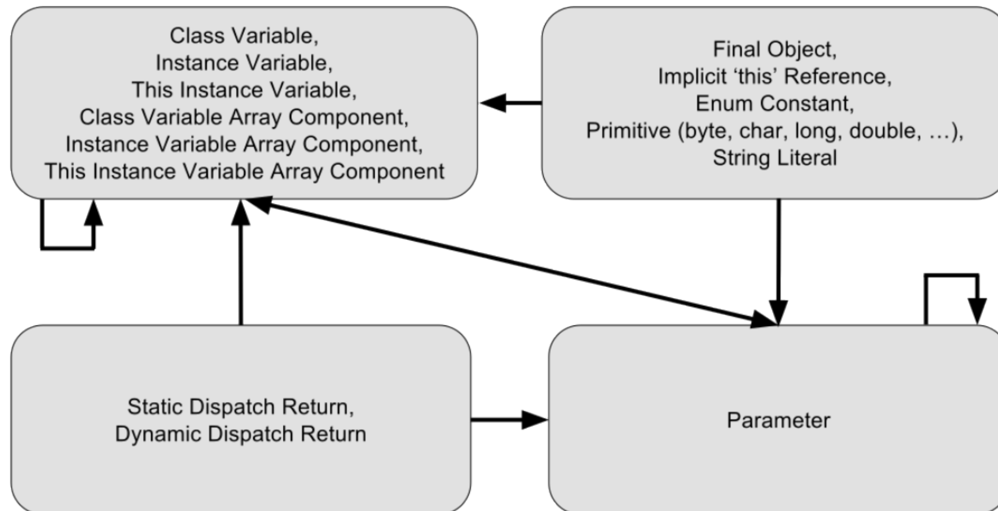



Figure 7.1 Immutability Assignment Graph

In addition to the basic assignment test cases, we created a set of manually curated test cases that involve complex program analysis challenges. These test cases test the accuracy of immutability analysis in the presence of aliasing, polymorphism and inheritance, open world assumptions, as well as issues that may arise from conservative assumptions to deal with array components and dynamic dispatch resolution. We next describe how we generated test cases that involve aliasing challenges. The other test cases involving polymorphism and inheritance, array index sensitivity and path sensitivity are not discussed here and can be found in the benchmark repository.

In the basic assignment test cases we defined a reference `test` that was explicitly mutated in the code and the analysis question was to correctly detect that `test` is mutated. In contrast, in the aliasing test cases, we create an alias to a reference `test`, and mutate the alias. The analysis question is to detect that `test` is indirectly mutated via the alias. In this category, there are four distinct aliasing patterns we test for, which can be explained using the concept of an aliasing chain. An aliasing chain is a sequence of explicit assignments of one reference to another, e.g., the statements `b = a; c = b;` creates the aliasing chain `c` → `b` → `a`. The four patterns of tests we perform are: (a) test case contains the aliasing chain `a` → `test`; (b) test case contains the aliasing chain `test` → `a`; (c) test case contains the aliasing chain `a` → `b` → `test`; and (d) test case contains the aliasing chain `test` → `b` → `a`. In all the above cases, the test case contains an explicit mutation to `a`, and analysis question asks whether `test` is mutable. The first two patterns (a) and (b) test whether mutations to an alias are propagated in either direction of the aliasing chain, while (c) and (d) test whether the analysis is also robust enough to detect mutations to aliases are propagated over levels of the aliasing chain. Listing 7.2 shows a test case corresponding to (b). Further these four basic aliasing patterns can be replicated by swapping the reference `a` with all the typable references such as a field, parameter, etc. Our test cases for aliasing are not exhaustive, and future work exists to expand these tests in a systematic way, as we have done for the basic assignment test cases.

Note that tests that address potential accuracy boundaries of points-to analysis implementations would also be applicable to testing immutability analysis implementations. In a feasibility study

we performed to create systematic tests for points-to analysis, available online[3], we discovered and reported a bug in the Soot program analysis framework that was accepted and later fixed [123]. In addition to the aliasing patterns, it is also possible to generate test cases for various kinds of dynamic dispatch and inheritance patterns, which would specifically test an analysis for correctness on propagating mutations through callsites. Our benchmark includes a few test cases in this category. This also opens another area of future work to create systematic test cases that systematically model inheritance and dynamic dispatch patterns.

Listing 7.2 Aliasing Testcase

```
1  public class AT_002 {
2    @MUTABLE
3    public Test test;
4    public static void main(String[] args) {
5      new AT_002().foo();
6    }
7    public void foo() {
8      // aliasing pattern: test -> a
9      Test a = new Test();
10     test = a;
11     System.out.println(test);
12     a.f = new Object(); // test is mutated
13     System.out.println(test);
14   }
15 }
16
17 class Test {
18   public Object f = new Object();
19   @Override
20   public String toString() {
21     return "Test [f=" + f.toString() + "]";
22   }
23 }
```

## 7.3 Benchmark Results

For our evaluation, we used our points-to based immutability implementation[4] and the latest releases of ReImInfer, which implemented the type-inference based approach. Both tools passed the basic assignment tests cases, but both versions of ReImInfer ([118, 119] and [120, 121]) failed a particular set of aliasing test cases, which we confirmed with the authors. From this result, we

[3] https://github.com/kcsl/JPATS
[4] https://github.com/EnSoftCorp/immutability-toolbox

derived the insight that one of the fundamental constraints for the assignment `x=y`, the constraint `x :> y` detailed in [118, 119], does not propagate mutations on `y` to `x`. With the understanding that longer aliasing chains offer more opportunities for such aliasing patterns to occur, we leveraged our points-to based implementation to compare the scalability of each approach and calculate the frequency of aliasing chains of different lengths in a real world application TinySQL (whose results were presented by ReImInfer's authors). The points-to based implementation, while more expensive, revealed that such cases are quite prevalent in practice. Specifically, for TinySQL we estimated that cases of aliasing chains of length 2 or more (47.12% of all aliasing chains detected by the points-to analysis) present opportunities for incorrect detection of mutations by ReImInfer.

Our experiments found that there was no clear winner between the approaches, but it did uncover an undocumented potential for false negative results in the current state-of-the-art implementation proposed in [118, 119]. The points-to based analysis was a winner in terms of accuracy, but the type based approach scaled better to large applications at the cost of potential false negatives. Our exercise of creating a benchmark brought out the accuracy boundaries inherent in each approach and provided precise examples of analysis challenges not handled by an approach. The approach agnostic benchmark created in this work can be used by industry practitioners to objectively assess current and future immutability analysis approaches.

# CHAPTER 8. HOMOMORPHIC PROGRAM INVARIANTS

As we learned in Chapter 1.1 the number of paths in software can be astronomical. There exists a function in the Linux kernel named `lustre_assert_wire_constants` that alone has $2^{656}$ paths [22]! At a first glance, it may seem amazing that humans could ever write something so complex as the Linux kernel, but a deeper examination reveals that humans don't think about software in this way. While the `lustre_assert_wire_constants` function may look complex, it consists of a very simple pattern of assertions repeated many times, which makes reasoning about its correctness feasible. Like an inductive proof, a human reasons about patterns in software and then reasons about the repetition of those patterns towards their final goal.

This work theorizes that when a programmer designs software, the programmer naturally breaks the astronomical number of control flow paths down into a relatively small sets of equivalent control flow paths to reason about a problem at hand. The programmer does this by (1) considering a segment of the overall path through a program such as the start and end of a function, and by (2) ignoring branches that are irrelevant to the task at hand. Similarly, a programmer reasons about relationships between program variables by considering classes of values that could occur during program executions of equivalent control flow paths.

A program invariant is defined as "a property that is true at a particular program point or points" [69]. We propose computing *homomorphic program invariants*, which we define to be program invariants that are computed with respect to a set of program paths homomorphically relevant to an analysis task. To explore this idea, we present a motivating toy example shown in Listing 8.1. If our goal is to determine if a *division by zero* (DBZ) error could occur in the toy example, we should consider all paths that contain a division operation. The example has a single division operation on line 24. Figure 8.1 shows the *control flow graph* (CFG) of the program. Note that there

are 7 paths through the CFG. As described in Chapter 6, each program control flow path yields a *behavior* represented by a corresponding *trace*.

Listing 8.1 Toy Example to Illustrate Homomorphic Program Invariants

```
1  public class Toy {
2    public static void main(String[] args) throws Exception {
3      FileInputStream input = new FileInputStream(args[0]);
4      int x = input.read() % 256; // x = -1 to 255
5      int y = input.read() % 256; // y = -1 to 255
6      int z = input.read() % 256; // z = -1 to 255
7
8      int a = x;
9      int b = y;
10     int c = z;
11
12     if(y < 128) {
13       if(x > 5) {
14         c = 0;
15         expensive();
16       }
17       expensive();
18       if(x < 5) {
19         b = a - b;
20       }
21       c = b;
22       int d = c + 1;
23       if(y % 2 == 0) {
24         System.out.println(x / d);
25       } else {
26         System.out.println(d);
27         expensive();
28       }
29     }
30     expensive();
31   }
32 }
```

In order to consider how control flow behaviors can be divided into equivalence classes with respect to the DBZ error, we will start with a decompositional approach and only consider control flow. The division operation on line 24 can be executed irrespective of the branch decisions made

on lines 13 and 18, however a division operation will only occur if the branch conditions at lines 12 and 22 are true, resulting in only 4 relevant paths that could perform a division operation.

In terms of control flow, the branches at lines 13 and 18 are irrelevant. To a programmer reasoning about whether or not the division operation would be executed, the details of two branches could be elided by merging the paths into a single path at each irrelevant branch. In 1981, Tarjan observed that many problems deal with intersecting sets of paths in a directed graph for which he proposed a general method for solving path problems by defining homomorphisms that map the regular expressions representing path sets into the given problem domain [113]. Prior work by Tamrawi showed that irrelevant branches in a CFG could be efficiently elided using a graph homomorphism to produce equivalence classes of control flow behaviors with respect to a set of control flow events [58]. The CFG shown in Figure 8.1 after eliding branches irrelevant to the division operation is shown in Figure 8.2 as a *projected control graph* (PCG).

In the PCG in Figure 8.2, the two `false` edges that lead to the master exit (⊥) indicate that if either of the PCG's conditional branches are `false` the control flow event of interest will not execute. By examining the CFG successor nodes of the corresponding `false` edges in the CFG we learn that if the statements at line 26 or 30 execute then the division operation will not be reached.

Program invariants that hold on a subset of program paths may not hold on all program paths. For example, in Listing 8.1, the invariant "`x < 5`" holds for the subset of the program paths where a division by zero error occurs, however the same invariant does not hold for all program paths. Homomorphic program invariants however, which are specific to a class of program behaviors, may form stricter assertions than program invariants computed on all paths. Since the homomorphic program invariant "`x < 5`" represents a restriction on the program input required to cause a division by zero error it is arguably more useful than an invariant of `x` that holds across all program paths when it comes to reasoning about a DBZ error. Because lines 26 and 30 represent the start of irrelevant statements for reasoning about a DBZ error, we should discard any program invariants that consider program paths through statements at lines 26 and 30. Ernst broadly defines an invariant to be relevant "if it assists a programmer in a programming activity" [70]. Since homomorphic

program invariants may form stricter assertions with respect to a class of control flow behaviors, homomorphic program invariants are arguably more useful to a programmer when reasoning about a specific programming task.

The toy example is contained entirely within a single function. The possible values of variables in the toy example were partitioned by conditional branches. Functions are used to create modular code with repeatable computations. Branches governing callsites to functions, function pointers, and dynamic dispatches can all be used to partition the values of data along inter-procedural paths. The endeavor to make modern languages highly modular and composable means that the relationships between data variables may be complex and spread inter-procedurally throughout a program since programmers constantly leverage vast libraries of reusable patterns to form more complex patterns of logic in software. To compute homomorphic program invariants in modern software we must consider inter-procedural control flows. Chapter 6 presents our work to extend prior work by Tamrawi and compute equivalence classes of inter-procedural control flow behaviors. Chapter 9 presents our SIDIS framework that computes homomorphic program invariants using off-the-shelf components for fuzzing and dynamic detection of likely invariants.

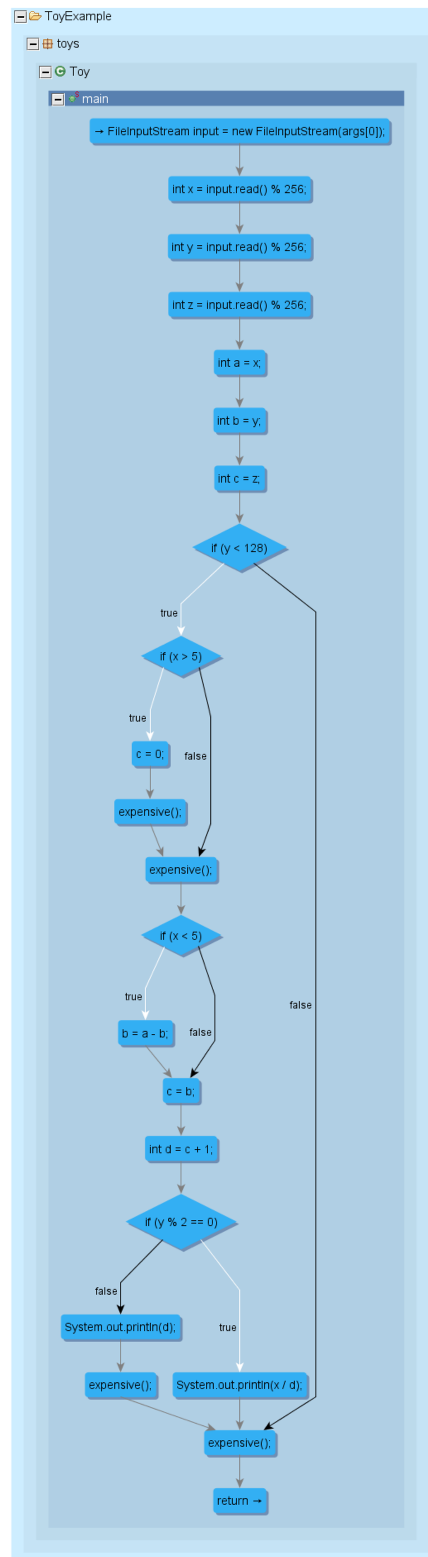



Figure 8.1    Toy Example - Control Flow Graph (CFG)

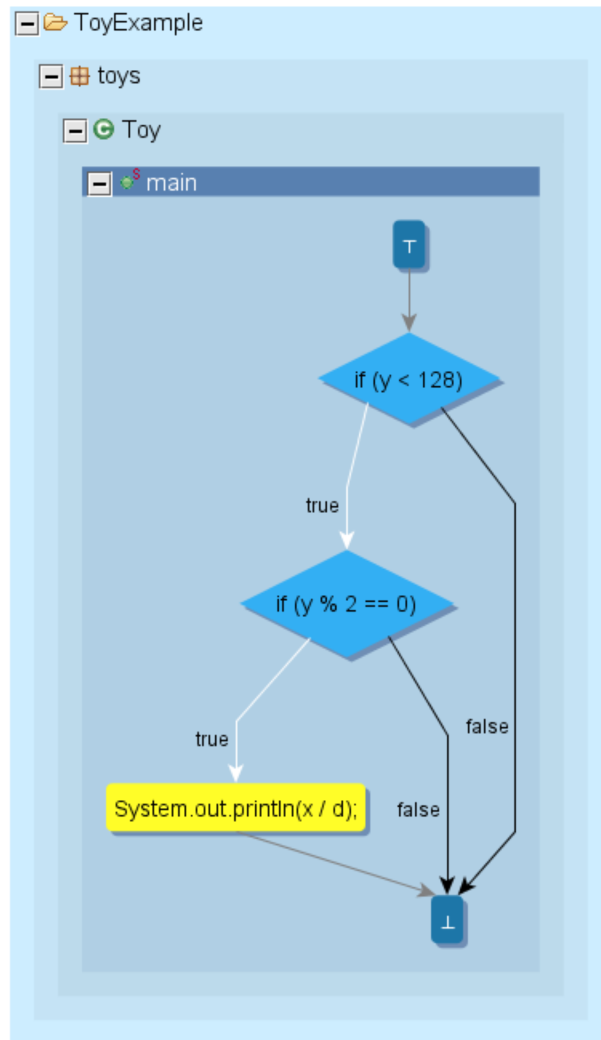



Figure 8.2 Toy Example - Projected Control Graph (PCG) of Division Operation

# CHAPTER 9. SIDIS FRAMEWORK

In this section we address our last research question: ***How can we compute homomorphic program invariants to aid in human program comprehension?***

To compute homomorphic invariants, we first statically compute homomorphic program behaviors and then automatically modify program binaries to restrict execution to homomorphic behaviors with respect to a set of control flow events (program statements of interest). We leverage the existing state-of-the-art in fuzzing (AFL [52]) and invariant detection tools (Daikon [47]). We employ our SIDIS framework, a culmination of our previous works described in Chapters 2, 3, 4, 5, 6, and 7, to produce a modified program binary that contains logic to abort execution for irrelevant behaviors. In this approach no modifications to AFL or Daikon are required. Our Mockingbird framework enables a targeted dynamic analysis amenable to integration with AFL. AFL drives input generation while an experiment coordinator sits at the center simulating successful program executions for irrelevant behaviors and redirecting relevant execution traces to the Daikon invariant detector. In this way, AFL is motivated to explore relevant program behaviors and only records inputs for relevant program crashes. This configuration also makes it so that Daikon only detects homomorphic invariants with respect to the control flow events of interest. The computed invariants could then be imported into the program graph as attributes for further static analysis, used to insert assertion statements in the program source or binary, or consumed directly by a human for improving program comprehension. Figure 9.1 shows a high level architecture diagram of the approach.

The remainder of this Chapter presents three techniques for automatically rewriting a program binary to (1) abort execution of irrelevant paths, (2) elide execution of irrelevant statements, and (3) inject fail early assertions. While only technique 1 is necessary to compute homomorphic invariants, all techniques increase the speed of program execution with respect to the program behaviors

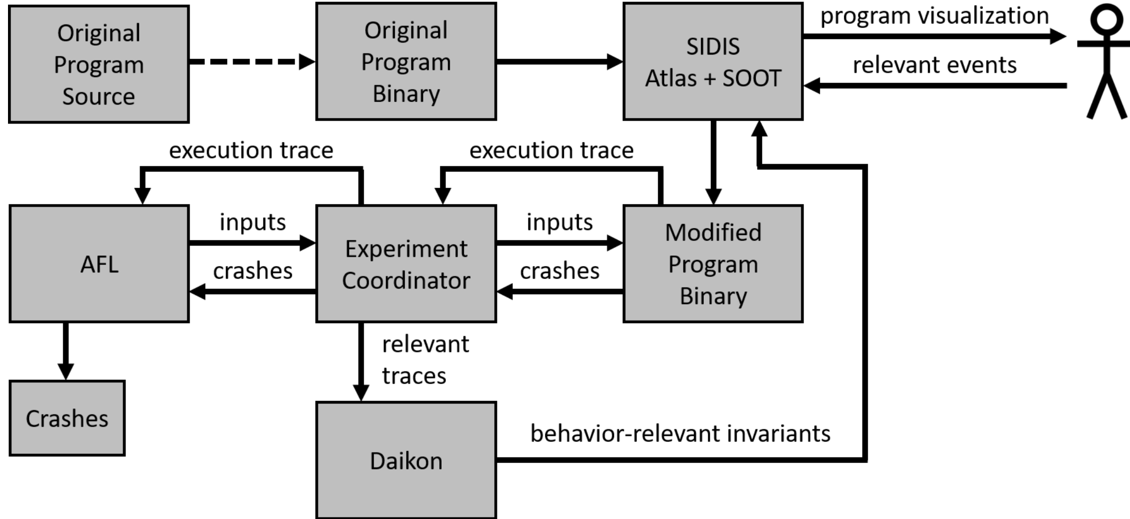



Figure 9.1 SIDIS High Level Architecture Diagram

of interest. Techniques 2 and 3 are presented for completeness and can safely increase program execution speed for intra-procedural analysis, but may introduce imprecision in inter-procedural analysis.

To illustrate each technique we refer to our toy example in Listing 8.1 as a motivating example to apply each modification technique, while Listing 9.1 represents the modified program. Our goal is to determine if a division by zero error could occur, which means that our control flow event of interest is the division operation on line 24. Recall that there are 7 paths through the CFG (shown in Figure 8.1), each representing a program behavior. Note that a division operation on line 24 can be executed irrespective of the branch decisions made on lines 13 and 18, however a division operation will only occur if the branch conditions at lines 12 and 22 are true, resulting in only 4 relevant paths that could perform a division operation.

## 9.1 Aborting Irrelevant Path Execution

Verifying the division by zero error in the motivating example or any program property for that matter poses two challenges: (1) computing the relevant behaviors out of all behaviors and, (2) verifying the control flow feasibility of relevant paths. The path feasibility problem is equivalent

to the satisfiability problem [54], but employing a fuzzer ensures that all results are computed on feasible paths. The large number of behaviors ($2^n$ behaviors for $n$ non-nested branches) make the problem of computing all behaviors computationally intractable, but it possible to efficiently group relevant and irrelevant behaviors [124, 59]. The result is a projection of a control flow graph called a *projected control graph* (PCG) [60], which groups behaviors so that each group corresponds to a unique relevant event defined by a sub-trace that retains only the relevant event statements and conditions. Since the PCG abstraction is parameterized by its input of relevant event statements it can be easily adapted to a wide variety of analysis tasks.

Figure 8.2 shows the PCG of the toy example for the relevant division operation. Any time spent executing program statements on paths that do not lead to the division operation is time wasted with respect to discovering a division by zero error. Significant execution time can be saved by inserting an abort signal before the first irrelevant statement on an irrelevant path. Specifically, an `abort-irrelevant` signal can be inserted before any statement in the CFG that is a successor of a branch reachable in a reverse step of the PCG from the $\perp$, omitting the relevant event statements, such that the successor is reachable from the same conditional edge value as the conditional edge value traversed in the PCG. Any execution trace that fires an `abort-irrelevant` signal can be discarded from invariant detection. Execution traces that are not aborted due to an `abort-irrelevant` signal are implicitly relevant. The `abort` signals can be implemented as a thrown exceptions, however compatible catch blocks should be modified to re-throw abort exceptions.

## 9.2 Eliding Irrelevant Statement Execution

The PCG in Figure 8.2 offers another opportunity to elide the execution of irrelevant statements that follow the division operation. As long as the relevant events are not contained in a loop, which would be indicated in the PCG, an `abort-relevant` signal can be injected immediately following any event statement reachable within one step of the PCG $\perp$. This abort signal however should be differentiated from an `abort-irrelevant` signal in that execution traces should not be discarded from invariant detection.

By considering control and data flow dependencies it is possible to efficiently compute a slice of relevant program statements [55, 25]. Figure 9.2 shows the CFG of the toy example with the set of statements not included in the program slice marked in red. Any time spent executing statements marked in red that are on a relevant program path is time wasted. By injecting goto's and labels the CFG can be modified to jump over execution of irrelevant statements. By eliding statements not in the program slice, the program is modified to behave as if its CFG were the PCG presented in Figure 9.2. In general, a goto optimization can be inserted for each edge in the PCG of relevant statements that does not exist in the CFG. Note that eliding irrelevant statements does not improve invariant detection, but does increase execution speed. In the case of inter-procedural analysis, automated slicing techniques such as [57] and even human-assisted slicing techniques such as [125] can introduce imprecision in the identification of relevant statements so extreme care must be used when eliding irrelevant statements.

## 9.3 Injecting Fail Early Assertions

Unless traces that produce a crash are separated from traces that do not produce a crash, the invariants detected on the modified toy example binary will be invariants that are relevant to the execution of a division operation not specifically a division by zero error. Since the division by zero error does cause a program crash, separating the execution traces is a simple task and one that will be performed automatically by most fuzzers including AFL.

In practice not all program behaviors of interest translate to program crashes. For example if we were interested in division by one events, a program crash would not occur. An assertion statement could be used to force a program crash by inserting assertion statements just before the events of interest. For example, in the division by zero problem we could make the crash explicit by adding an `assert(d != 0)` statement just before the division operation.

A keen reader would note that a similar opportunity exists to place an assertion of `assert-relevant(d==0)` earlier in the CFG just after $d$ is defined on line 22. Note that an `assert-relevant` statement would act like an `abort-irrelevant` signal if the assertion failed. By

placing an assertion on the value of $d$ earlier one relevant program statement can be elided without impacting the detected invariants.

By examining the conditions retained in the PCG and the reaching definitions of the variables used in the retained conditions, fail early data flow assertions can be inserted to elide additional relevant statements without impacting invariant detection. For the toy example, an `assert-relevant(y < 128 && y % 2 == 0)` can be inserted after line 5 and has the potential to elide up to nine relevant program statements. Since further eliding statements only serves to increase the speed of program execution this optimization is optional with respect to the goal of improving invariant detection. Similar to the second technique eliding irrelevant program statements, this technique is not required for invariant detection and can introduce imprecision in the presence of inter-procedural analysis challenges by preemptively eliding program statements.

Table 9.1 details the expected outcomes of the toy example. The program reads three successive bytes from an input file and stores each in the respective variables $x$, $y$, and $z$. If the file does not contain a byte to read then the variable is assigned a value of $-1$.

The program slice indicated that the value of $z$ is irrelevant to the division by zero problem. To reach the division operation, $y$ must be an even value less than 128. There are two distinct paths for $d$ to become zero. Working backwards from the division operation we see that since $d$ is assigned the value of $c + 1$, the value of $c$ must be -1. The reaching value of $c$ is the value of $b$ (at line 19 or 9). Let's follow the path that leads through line 19 first. Line 19 assigns $b$ the value of $a - b$. The reaching values of $a$ and $b$ are $x$ and $y$ respectively. For $c$ to be $-1$, $y$ should be one larger than $x$. However, there is a restriction of `x < 5` to reach the subtraction operation on line 19, leaving only $x = 1$, $y = 2$ and $x = 3$, $y = 4$ as possible inputs to cause a division by zero operation. The second path for $d$ to be assigned zero is when $x$ is 5 and $y$ is $-1$. If $x$ is 5 then neither `x > 5` nor `x < 5` are true and $c$ will be assigned the value of $b$ defined on line 9. Although $d$ is zero, the result of `-1 % 2` is $-1$, which does not satisfy the guarding condition to execute the division operation on line 23 making the crash impossible for $x = 5$.

The `expensive()` function in the toy example only contains logic to a sleep for 100 milliseconds ($1/10^{th}$ of a second). Since the callsites to the `expensive()` function are all on irrelevant paths or within irrelevant statements, we expect the fuzzer to be significantly faster on the modified program. The results in this report were computed in a single threaded environment, but the AFL fuzzer can be easily scaled up to utilize multiple CPU cores. Table 9.2 shows that the AFL fuzzer was 3.26 times faster at fuzzing the modified program and found one additional bug in the same time that it took to fuzz the original program.

Daikon instruments a program in two passes [47]. First the DynComp dynamic type inference executes the program and groups variables into comparability sets. Second the program is instrumented and executed with Daikon's Chickory frontend for Java, which leverages the DynComp comparability sets to detect program invariants. As a result of the two pass strategy, Daikon must execute a program with the test input twice each time paying an additional instrumentation overhead cost. Table 9.3 indicates that the instrumentation overhead slowed the program execution speed by roughly half and then doubled that time by executing the program twice. The experiment coordinator considered all execution traces in the unmodified program to be relevant, while only two traces corresponding to the two inputs that caused a division by zero error were relevant for the modified program.

Examining the detected invariants in Table 9.3 revealed that the homomorphic invariants made stricter assertions. Note that both invariants sets indicate that $x\ != 0$, which is not actually true. Since the fuzzer never tested an empty file, $x$ was always initialized to a positive value. The input constraints to cause the division by zero error are made clear in the first three homomorphic invariants: $x \in \{1, 3\}$, $y \in \{2, 4\}$, and `x <= y`.

Listing 9.1 Modified Toy Example

```
1  public class Toy {
2    public static void main(String[] args) throws Exception {
3      FileInputStream input = new FileInputStream(args[0]);
4      int x = input.read() % 256; // x = -1 to 255
5      int y = input.read() % 256; // y = -1 to 255
6      assert_relevant(y < 128 && y % 2 == 0);
7      int z = input.read() % 256; // z = -1 to 255
8
9      int a = x;
10     int b = y;
11     goto label_1;
12     int c = z;
13
14     label_1:
15     if(y < 128) {
16       goto label_2;
17       if(x > 5) {
18         c = 0;
19         expensive();
20       }
21       expensive();
22       label_2:
23       if(x < 5) {
24         b = a - b;
25       }
26       c = b;
27       int d = c + 1;
28       assert_relevant(d == 0);
29       if(y % 2 == 0) {
30         System.out.println(x / d);
31         abort_relevant();
32       } else {
33         abort_irrelevant();
34         System.out.println(d);
35         expensive();
36       }
37     }
38     abort_irrelevant();
39     expensive();
40   }
41 }
```

Table 9.1 Toy Example Expected Outcomes

| x | y | z | d | Division Executed | Outcome |
|---|---|---|---|---|---|
| 1 | 2 | * | 0 | Yes | Crash - Division by Zero |
| 3 | 4 | * | 0 | Yes | Crash - Division by Zero |
| 5 | 6 | * | 7 | Yes | No Crash - Failed Data Flow Constraint |
| 5 | - | - | 0 | No | No Crash - Failed Control Flow Constraint |
| - | - | - | 1 | Yes | No Crash - Failed Data Flow Constraint |

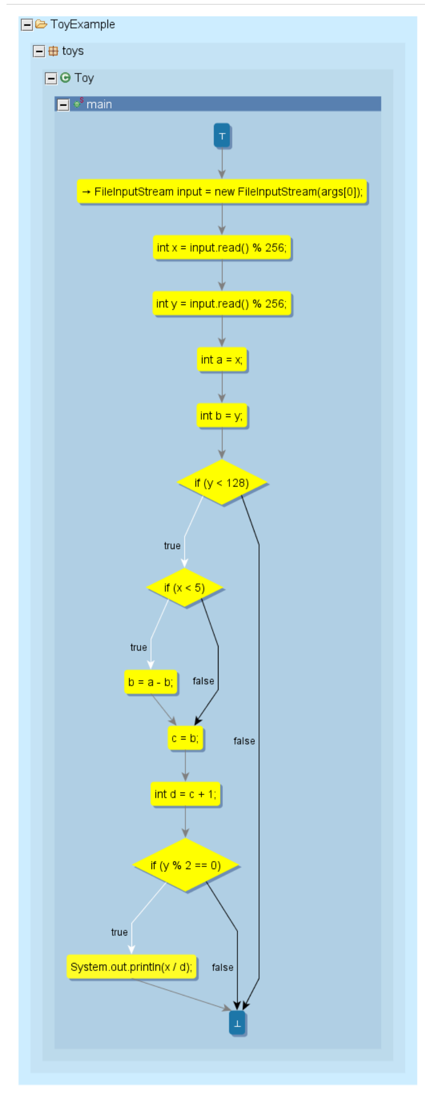
Note: The lack of a file byte is denoted with a "$-$". A wildcard value is a "$*$" symbol.

Table 9.2 Toy Example Fuzzing Speed

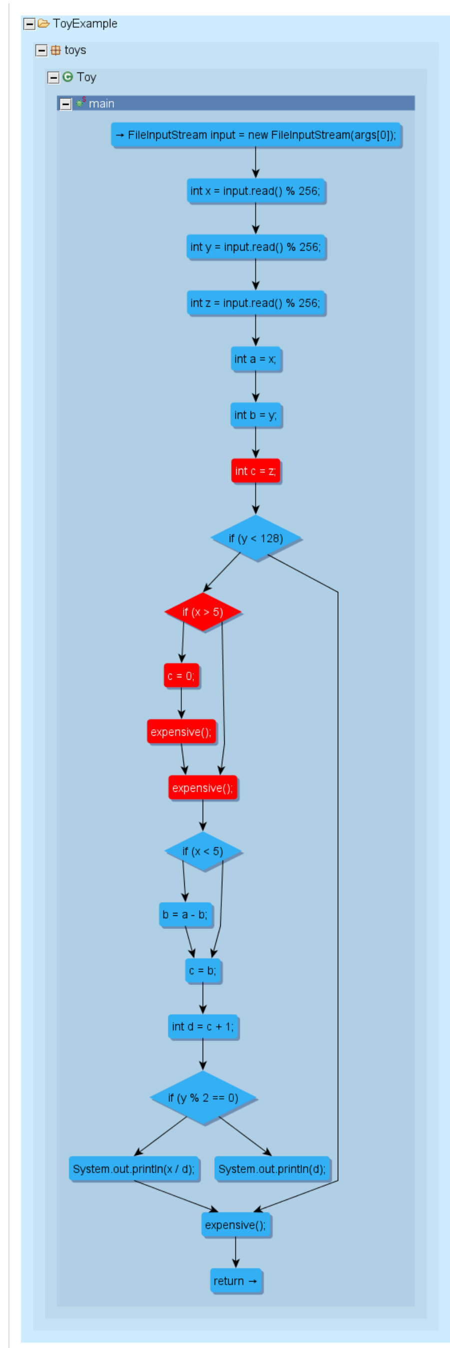
| Program | Fuzzing Time | Fuzzing Speed | Crashes |
|---|---|---|---|
| Original Program | 15 minutes | 2.93 executions/second | 1 |
| Modified Program | 15 minutes | 9.66 executions/second | 2 |

Table 9.3 Toy Example Invariant Detection

| **Program** | Original Program | Modified Program |
|---|---|---|
| **Restrictions** | None (all behaviors) | Homomorphic Behaviors |
| **Detection Time** | ~1 hour (2218 traces) | < 1 second (2 traces) |
| **Detected Invariants** | • `x >= 0`<br>• `y >= -1`<br>• `z >= -1`<br>• `x == a`<br>• `b != d` | • $x \in \{1, 3\}$<br>• $y \in \{2, 4\}$<br>• `x <= y`<br>• `x == a`<br>• `y >= b`<br>• `b == -1`<br>• `c == -1`<br>• `x >= d`<br>• `y >= d`<br>• `d == 0` |




PCG of Relevant Statements




CFG with Irrelevant Statements Marked

Figure 9.2 Toy Example - Eliding Irrelevant Program Statements

# CHAPTER 10. EVALUATION

In this chapter we use the tools and methodologies described in this work to evaluate three DARPA STAC challenge applications for vulnerabilities to algorithmic complexity attacks in space and time. Each evaluation includes a summary of the homomorphic program invariants that lead to a holistic understanding of the vulnerability that enabled the development of a *proof of concept* (PoC) exploit. The source code of each application is available online[1]. For each application we leverage human-in-the-loop reasoning and automated program analysis to statically identify likely candidates for algorithmic complexity vulnerabilities and then perform a *statically-informed dynamic* (SID) analysis to recover the homomorphic program invariants.

## 10.1 DARPA STAC Engagement Case Study I

This section describes a case study of how an algorithmic complexity (AC) vulnerability in a web application can be detected by this work. The analysis for the case study was done on the bytecode, but we include here some snippets of decompiled code for the reader's benefit. The *Blogger* challenge application is a web application whose Java web server implementation extends the NanoHTTPD[2] open source project. The application's functionality includes user sign in, creation of new blog posts and viewing of posts by other users. The application consists of 2320 lines of Jimple code, which translates to about 1800 lines of decompiled Java code.

The loops in the Blogger application, their nesting depths and the calling relationships between methods containing loops are precomputed by the static analysis described in [102, 84]. The analyst begins auditing with some domain knowledge about web server applications. A server application typically has one or more threads (server listener) dedicated to listening for client connections, and one or more threads (client request handler) for parsing, processing and responding to client

[1] https://github.com/Apogee-Research/STAC
[2] https://github.com/NanoHttpd/nanohttpd

requests. Starting with this domain knowledge, the analyst hypothesizes an AC vulnerability to be in the client request handler thread. By examining the code that initiates threads (callsites to constructors of `Runnable` subtypes), the analyst identifies that each client request is processed in a separate thread of type `ClientHandler`.

The analyst selects `ClientHandler.run` and invokes the LCG Smart View. As shown in Figure 10.1, the view shows a graph of methods containing loops and starting from `ClientHandler.run`. Nodes and edges of the LCG are color coded as described earlier (Section 4.1). For example, the edge from `ClientHandler.run` to `HTTPSession.execute` is yellow because the callsite to the latter is in a loop. As seen from Figure 10.1, the LCG shows 16 methods containing loops. Note that each method may contain multiple loops. The analyst formulates the following hypothesis:

> HYPOTHESIS. *An AC vulnerability can be caused in one or more loops in the 16 methods in the LCG.*

To narrow the hypothesis the analyst uses the dynamic analysis instrumentation included in the SIDIS toolbox. The analyst selects all methods containing loops in the LCG and uses the toolbox to instrument the loops in the selected methods with the *execution counter* probe to compare resource consumption of various loops. The SIDIS toolbox then compiles the instrumented bytecode for the server into an executable. The analyst starts the server (having replaced the application binary with the instrumented application binary), and triggers three simple HTTP request URLs to the server. Table 10.1 shows the loop iteration counts for loops exercised in the Blogger application when the URLs "/", "/test" and "/stac/example/Example" were requested from a web browser.

Table 10.1 shows that after the server processes the request URLs, two methods show an unusually large number of loop iterations – `HTTPSession.findHeaderEnd` and `URIVerifier.verify`. In particular, `URIVerifier.verify` has three loops, so the analyst clicks on the method in the LCG to go to its code (Listing 10.1). The method has three loops at lines 5, 9, 11 respectively. The `while` loop pushes to and pops from the list `tuples` in each iteration, and it is not clear whether the loop will always terminate. The analyst becomes suspicious about `URIVerifier.verify`, and decides to refine the hypothesis.

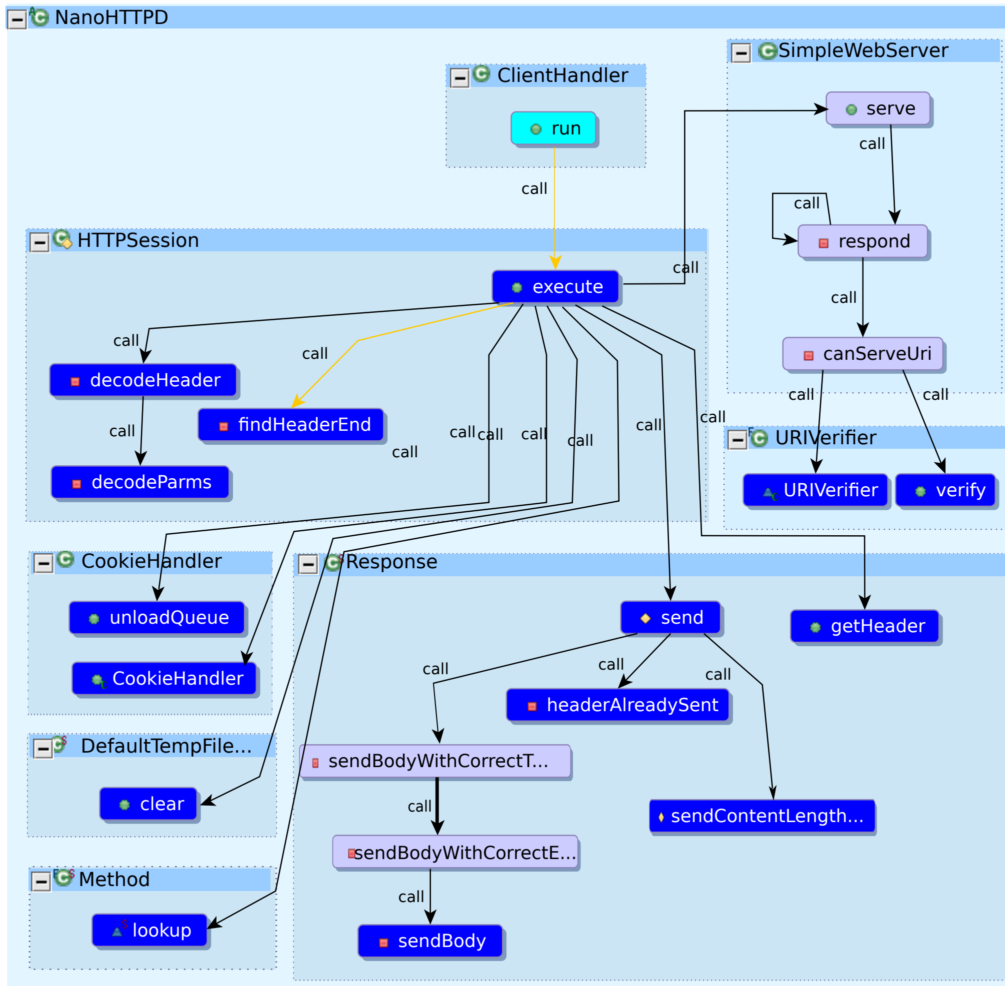



Figure 10.1 LCG for the Blogger Challenge Application

Listing 10.1 Decompiled code for `URIVerifier.verify`

```
1  public boolean verify(String s) {
2   Tuple peek;
3   LinkedList<Tuple> tuples = new LinkedList<Tuple>();
4   tuples.push(new Tuple<Integer, URIElement>(...));
5   while (!tuples.isEmpty() && (peek = (Tuple)tuples.pop()) != null) {
6     if (((URIElement)peek.second).isFinal &&
        ((Integer)peek.first).intValue()==s.length())
7       return true;
8     if (s.length() > (Integer)peek.first)
9       for (...)
10        tuples.push(new Tuple<Integer, URIElement>(...);
11    for (URIElement child : ((URIElement)peek.second).get(-1))
12      tuples.push(new Tuple(...));
13  }
14  return false;
15 }
```

Table 10.1 Loop Execution Counts for the Blogger Challenge Application

| **Loop Headers by Method** | **Iterations** |
|---|---|
| NanoHTTPD.HTTPSession.findHeaderEnd.label1 | 4341 |
| URIVerifier.verify.label5 | 2148 |
| URIVerifier.verify.label3 | 1188 |
| URIVerifier.verify.label1 | 1074 |
| URIVerifier.URIVerifier.label5 | 270 |
| URIVerifier.URIVerifier.label1 | 190 |
| NanoHTTPD.HTTPSession.decodeHeader.TrapRegion.label10 | 100 |
| NanoHTTPD.Response.headerAlreadySent.label1 | 24 |
| NanoHTTPD.CookieHandler.CookieHandler.label1 | 22 |
| NanoHTTPD.Response.sendBody.label3 | 22 |
| NanoHTTPD.HTTPSession.decodeParms.label2 | 16 |
| NanoHTTPD.ClientHandler.run.TrapRegion.label2 | 15 |
| NanoHTTPD.Response.send.TrapRegion.label6 | 12 |
| NanoHTTPD.CookieHandler.unloadQueue.label1 | 12 |
| NanoHTTPD.Response.getHeader.label1 | 12 |
| NanoHTTPD.HTTPSession.execute.TrapRegion.label6 | 11 |
| NanoHTTPD.DefaultTempFileManager.clear.label1 | 11 |
| NanoHTTPD.Method.lookup.label1 | 11 |

Refined Hypothesis. *An AC vulnerability can be caused by crafting an input that causes an infinite loop or long running loop in the* `URIVerifier.verify` *method.*

The concerning loops are contained entirely inside the `URIVerifier.verify` method so the problem is not inter-procedural, except that there is a data dependency on the initial state of the `URIVerifier` object. The `URIVerifier` object instance variables are always initialized the same by a private constructor method. The easiest way to exercise the `URIVerifier.verify` would be to mock the `URIVerifier` instance variables or to use Mockingbird to invoke the private constructor directly. For this case study we used the Mockingbird framework to call the private constructor and then exercise the `URIVerifier.verify` method with a given input string to verify. Since we are interested in a long running loop we select the loop header as the event of interest. The CFG and PCG for the `URIVerifier.verify` method are shown in Figure 10.2. Using the PCG, the SIDIS toolbox inserts an `abort-irrelevant` signal before the `return true;` at line 7 and the `return false;` at line 14.

Fuzzing the `URIVerifier.verify` method for seven minutes produces a string that exceeds a resource usage limit of 5 seconds. Note that the challenge resource usage limit was actually 300 seconds, but we set the fuzzer timeout to be 5 seconds as an approximation for expensive inputs since the `URIVerifier.verify` is supposed to be near instantaneous. Running the fuzzer for an additional three minutes produces 10 more exploit strings. The detected invariants indicate that the input strings are all 35 characters or longer and the result of `URIElement.get` could return more than one result (an array of size 0, 1 or 2), but do not give much more insight into why the method is vulnerable.

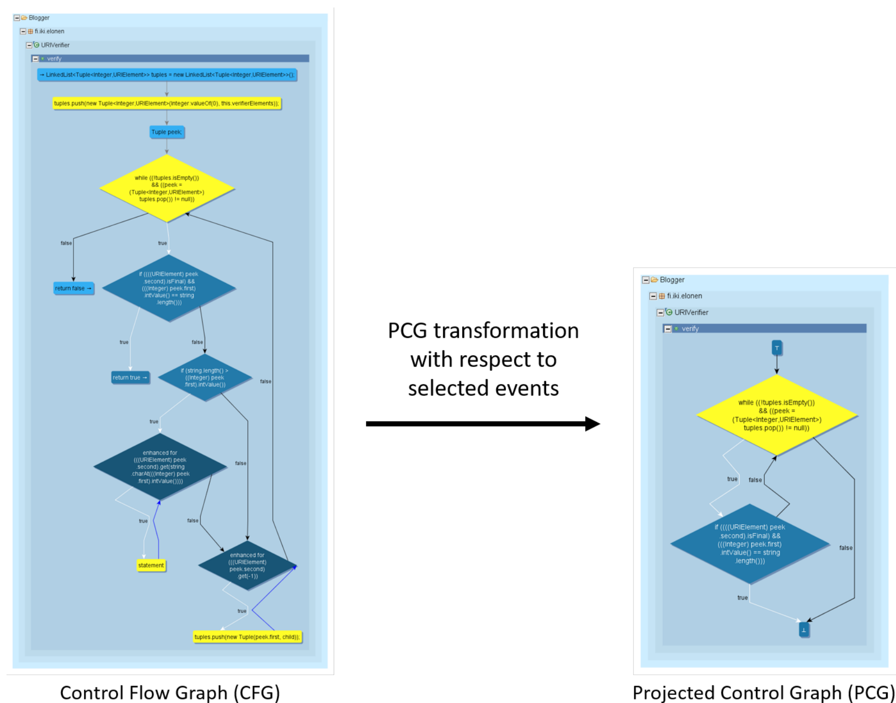



Figure 10.2 Blogger Challenge Application CFG to PCG Transformation

Finally, the analyst decides to instrument the `URIVerifier.verify` method and compute the execution counts of loop headers as a function of systematically generated string inputs of increasing length. Using the SIDIS toolbox, the analyst selects the iteration counter instrument for `URIVerifier.verify`. The toolboxes generates a default driver with a call to `URIVerifier.verify`. The analyst updates the driver to test the method with input strings of increasing length from 1 to

30 (Listing 10.2). The *time complexity analyzer* (TCA) runs the driver and plots the sum of iteration counts of the three loops (Figure 10.3) in the method. The plot shows that this count is clearly exponential in the length of the string parameter. The analyst then passes larger URL strings until a point when the server stops responding at URL length 35. Note that in the original challenge the analysis environment assumed that the application was run in Java's interpreted mode, which amplifies the impact of the vulnerability. In interpreted mode a string of length 35 is sufficient to exceed the resource usage vulnerability of 300 seconds, however with the JIT (just-in-time compilation) enabled a string of length 35 takes approximately 7 seconds to process. The exponential relationship reveals that the attack can be modified to work in JIT runtimes. Extending the input string length to 43 characters will exceed 300 seconds in a JIT optimized runtime. Our analysis for this work was performed with the JIT optimization enabled, which is the standard deployment of the Java JVM.

Listing 10.2 Driver with workload for `URIVerifier.verify`

```
1  public class CounterDriver {
2    private static final int TOTAL_WORK_TASKS = 30;
3    public static void main(String[] args) throws Exception {
4      for(int i=1; i<=TOTAL_WORK_TASKS; i++){
5        RULER_Counter.setSize(i);
6        URIVerifier verifier = new URIVerifier();
7        verifier.verify(getWorkload(i));
8      }
9      tca.TCA.plotRegression(
10       "URIVerifier.verify Workload Profile",
11       TOTAL_WORK_TASKS);
12   }
13   private static String getWorkload(int size){
14     String unit = "a";
15     StringBuilder result = new StringBuilder();
16     for(int i=0; i<size; i++){
17       result.append(unit);
18     }
19     return result.toString();
20   }
21 }
```

The analyst has thus identified the complex and hard-to-understand loops in `URIVerifier.verify` as the root cause of the AC vulnerability using SID analysis. The plot and table of measurements

produced by the TCA collectively serve as the evidence for the analyst to confirm his hypothesis that Blogger has an AC vulnerability triggered by passing long URL strings to the server.

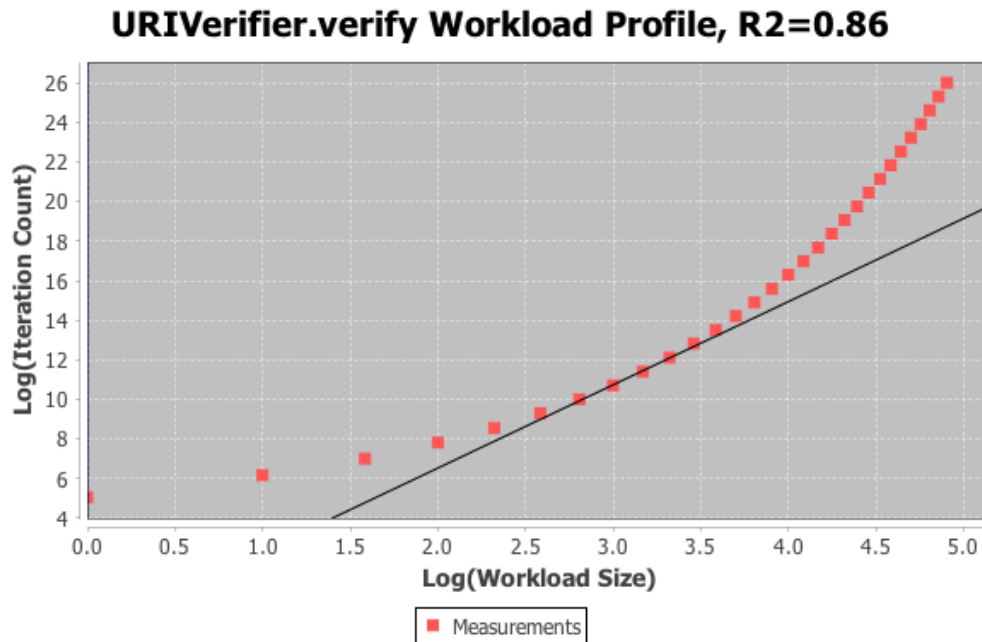



Figure 10.3 Log-Log Plot of Loop Execution Counts by Input Size

Since detecting AC vulnerabilities is an open-ended search through the application code, tools are important if they can reduce the search space effectively. Let us look at the results for this case study to understand how SID tools reduce the search space. The Blogger application consists of 200 Jimple methods, of which 31 contain loops. In hypothesis generation phase of the detection process, the LCG helped the analyst to narrow down the search to 16 of these 31 methods. These 16 methods are reachable from the entry point of interest (the code executed by the `ClientHandler` thread). The corresponding number of lines of Jimple code reduced from 2320 to 1197. Thus the LCG provided a significant reduction (over 48%) in the amount of code potentially containing an AC vulnerability that the analyst had to examine.

In the hypothesis verification phase of the detection process, the analyst was able to narrow down the search space for the vulnerability even further to 2 of the 16 methods (118 of the 1197 lines of code, an over 90% reduction) using the TCA. Using LCG and the TCA, the analyst only

needed to consider about 5% of the code in the application to find the AC vulnerability. In this challenge the homomorphic invariants were useful in the sense that they capture that the length of the input URL was related to the vulnerability. The loop of interest is implemented as a worklist style algorithm. The invariants also captured that the `URIElement.get` method could return multiple results, which upon closer inspection is the bounding condition for the loops that push new tasks onto the worklist. With a `pop` at the loop header being the only method to remove tasks from the worklist and two nested loops to add tasks to the worklist that each could result in multiple task additions this invarant does hint at exponential behaviors, but it's not immediately obvious. Since Daikon's default invariant detection did not include invariant patterns that related the loop header execution counts to the input URL length additional analysis was needed to gather evidence of how the two were correlated.

A post-mortem analysis of AFL's performance on this challenge showed that AFL was not particularly well suited to find this problem. By watching the inputs generated by AFL over time we see that after AFL initially explored the paths of the `URIVerifier.verify` method it had little incentive to produce longer inputs. In fact, AFL produces an aggressive culling strategy to reduce the size of inputs in an effort to increase execution speed. Replacing AFL's exploration strategy with a strategy more adept at searching for algorithmic complexity vulnerabilities (such as [126]) could improve this process.

## 10.2 DARPA STAC Engagement Case Study II

As a second case study we examine a DARPA STAC challenge application called *Image Processor*, which we analyze for algorithmic complexity vulnerabilities in space and time.

The case study demonstrates: (1) the use of Mockingbird framework to target a dynamic analysis, and (2) a follow-up experiment using Mockingbird's APIs to create a custom tool for computing program invariants that provide holistic understanding of the vulnerability. The follow-up experiment reveals not just one input but a class of inputs that cause the vulnerability.

DARPA's Image Processor is a command line application that is described as an image classification service that could be used to train and automatically apply appropriate tags to images.

Using the Security Toolbox an analyst statically identifies a concerning code segment, shown in Listing 10.3, that could be expensive for large inputs. Specifically, if the value of $n$ is large at line 4 the computation could be costly for the `factorial` method and `Math.pow` method.

Listing 10.3 Suspicious Code Segment in Image Processor Challenge Application

```
1   private static double factorial(double x) {
2    return Math.exp(Mathematics.lgamma(x + 1.0));
3  }
4  private static double exp(int x, int n) {
5    double exp_n = 0.0;
6    for (int i = 0; i < n; ++i) {
7      double aDouble = Math.pow((double)x * 0.00392156862745098, n);
8      double aDouble1 = 1.0 / Mathematics.factorial(n);
9      exp_n += aDouble * aDouble1;
10   }
11   return exp_n;
12 }
```

The LCG reveals that there is a direct call path from the application main method (which expects an input image file) to the `Mathematics.exp` method shown in the listing and that the `Mathematics.exp` method has a loop that is called within a loop from the `Mathematics.intensify` method, which itself is called with in a loop from the `Intensify.filter` method. With this understanding the analyst forms the following hypothesis:

Hypothesis. *An AC vulnerability can be caused by crafting an input image that causes expensive computation in the repeated calls to the* `Mathematics.exp` *method.*

The AFL adapter can easily be adapted to fuzzing this application's main method, which expects a file input of the image to process, however the input space is huge. This application accepts images up to 500 by 500 pixels, which creates a maximum of 250,000 pixels. Each pixel can have one of $256^3 \approx 16.7$ million colors, which gives a total of $4.19 * 10^{12}$ input valid images that could be generated. Forcing AFL to generate valid JPEG images is possible [127], but extremely expensive. After two hours of fuzzing we hadn't even generated a valid JPEG image. Using the Mockingbird framework we were able to directly target the `BufferedImage` pixel coordinate results, which guaranteed we generated valid images. Our execution speed was approximately three executions per second, but we notice that approximately 69% of the fuzzing time was spent in six other image processing filters when our interest was restricted to behaviors of the *intensify* filter. We refined our experiment one last time, leveraging our Mockingbird framework to directly invoke the `Mathematics.intensify` filter, which operated on a single pixel value. Since the `Mathematics.intensify` expected integer values for red, green, and blue values, we added a fuzzing constraint to Mockingbird to generate integers within the range of 0 and 255. Our execution speed increased to 9.62 executions per second. We set the fuzzer timeout to be 5 milliseconds (we found the average through random sample was $\approx 2$ milliseconds) and found five slow inputs in under three minutes.

Next we computed the homomorphic program invariants using the SIDIS framework. The events of interest are the callsites to `Mathematics.factorial` and `Math.exp`. Using the IPCG as a guide an `abort-irrelevant` signal is inserted before the return of `Mathematics.intensify`, which in this case is where we had scoped the targeted dynamic analysis.

Of the relevant invariants we learned that the value of $n$ was always 117, 207, or 1003 for the expensive inputs. The value of $n$ is selected from an array `accuracy` that consists of integers and the invariants revealed that the contents of array remained constant for all relevant execution traces. Plotting the contents of the array (see Figure 10.6) revealed that 1003 is the single worst case input that could come from the `accuracy` array. The index into the `accuracy` array is computed in the

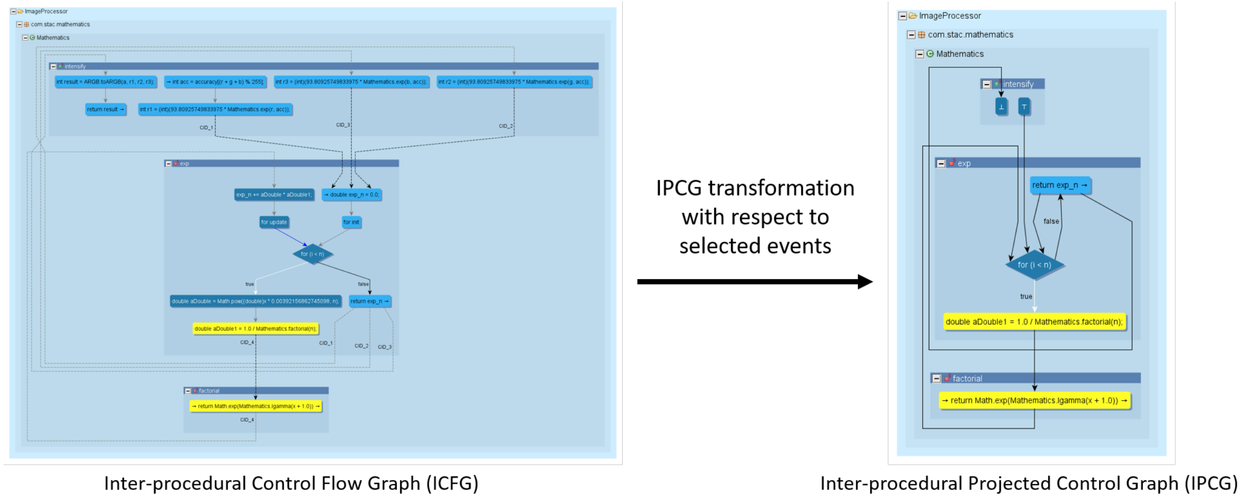



Figure 10.4 Image Processor Challenge Application ICFG to IPCG Transformation

`Mathematics.intensify` as a function of the red, green, blue pixel values. The invariants revealed that for the expensive inputs the index value was a value between 28 and 32, which corresponded to the highest values in the `accuracy` array (an index of 30 corresponded to accuracy value of 1003).

With the knowledge of these invariants, we re-examined the code and found that the index was computed as `int index = (r+g+b) % 255;`. We then computed the red, green, blue values that resulted in an index of 30 (solve for RGB colors where `(r+g+b) % 255 == 30`) and we were able to plot on a color wheel the most expensive pixel colors. Figure 10.6 show the regions colored black on the color wheel that correspond to pixel values with the maximum computation cost. With this information we are able to craft an image that contains a maximal number of expensive pixels to cause a denial of service attack in time. To maximize the number of pixels we choose a 500 by 500 size image that consists solely of the pixel colors identified as expensive. This analysis indicates that there is actually a family of exploits that could be crafted to exploit this vulnerability.

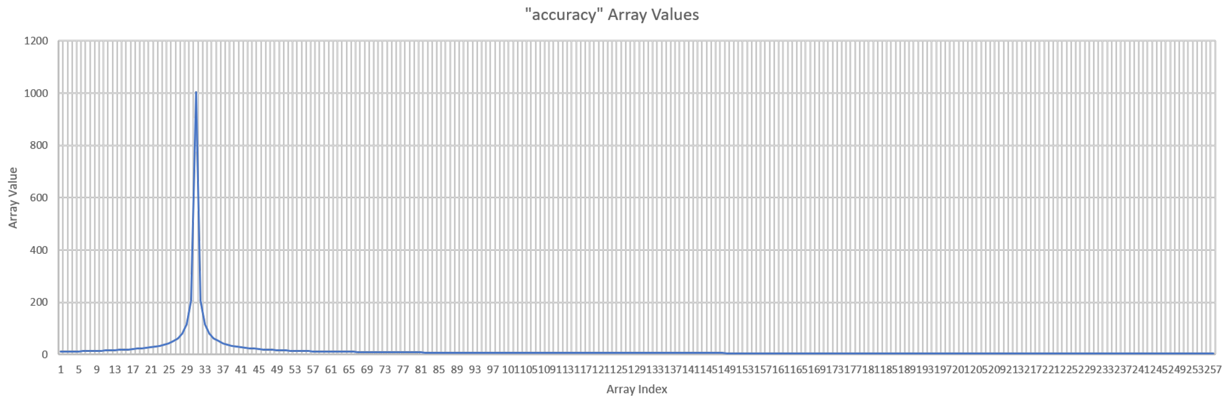



Figure 10.5 Image Processor Challenge Application `accuracy` Array Contents

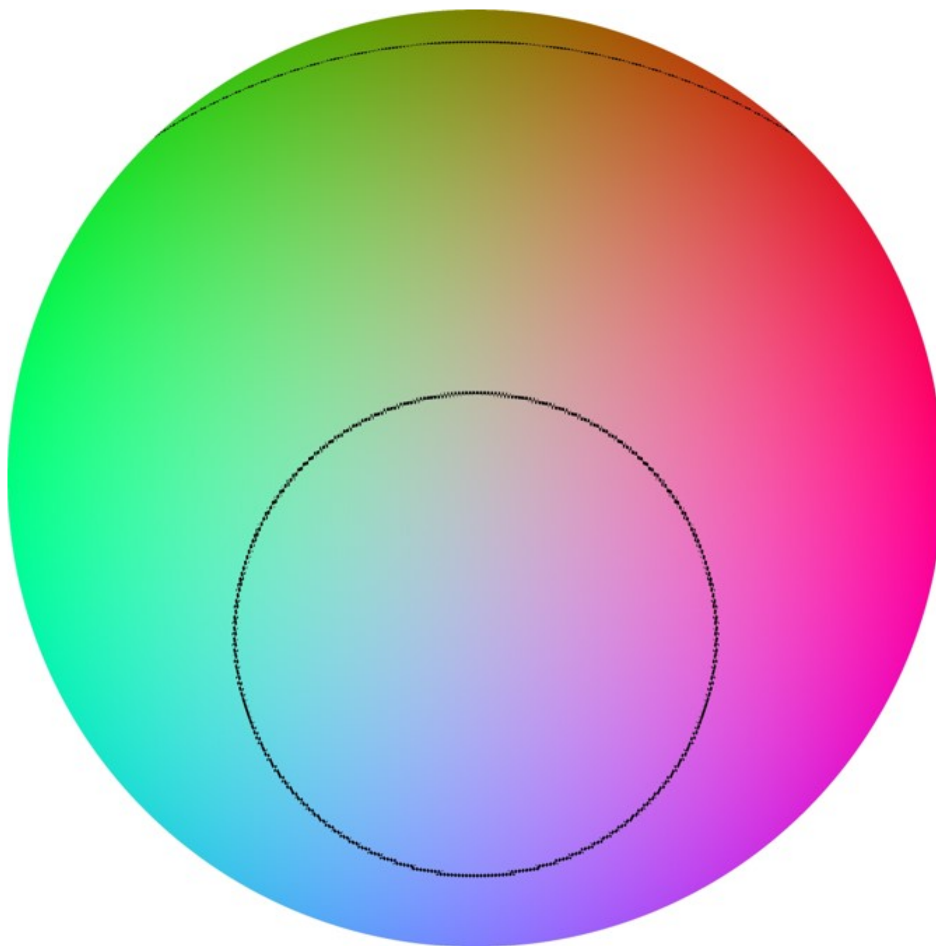


Figure 10.6 Image Processor Challenge Application Expensive Pixel Colors

## 10.3 DARPA STAC Engagement Case Study III

As a final case study we examine a DARPA STAC challenge application called *BraidIt* from the STAC program's final engagement phase, which we analyze for algorithmic complexity vulnerabilities in space and time.

The case study demonstrates: (1) the use of Mockingbird framework to efficiently discover an algorithmic complexity vulnerability, and (2) a follow-up experiment using Mockingbird's APIs to create a custom tool for computing program invariants that provide holistic understanding of the vulnerability. The follow-up experiment reveals not just one input but a class of inputs that cause the vulnerability; in particular, it reveals the smallest input that can cause the vulnerability.

DARPA's BraidIt application is described as a two player peer-to-peer game where players' attempt to recognize topologically equivalent braids (also known in mathematics as an Artin braid group [128]). For the purpose of demonstration, we have chosen a relatively small application with 5,844 lines of decompiled Java code (not including any of the 17 third-party library dependencies). Static analysis reveals that the application contains 51 loops, of which 29.4% of the loops are contained in a single class called `Plait`.

To find the relevant code for an AC time vulnerability, the loop analyzer [84] included in our Security Toolbox is used to locate loops with complex termination conditions. The set of loops is further refined by a reachability analyzer that retains only those loops whose termination conditions can be affected by external inputs. This leads to a loop in the `normalizeCompletely` method of the `Plait` class. The loop is shown in Listing 10.4.

Listing 10.4 Suspicious Loop in BraidIt Challenge Application

```
1 public void normalizeCompletely() {
2    this.freeNormalize();
3    while (!this.isReduced()) {
4       this.normalizeOnce();
5       this.freeNormalize();
6    }
7 }
```

The loop's termination condition depends on the result of the `isReduced` method. A data flow analysis of the `isReduced` method reveals that: (1) the termination condition consists of a complex series of string manipulations of a global variable called `intersections`, which `freeNormalize` and `normalizeOnce` both mutate in every loop iteration, and (2) the `intersections` variable is reachable from the outside.

Moreover, a static analyzer for locating code relevant for AC space vulnerabilities also reports the loop shown in Listing 10.4 because the loop children `normalizeOnce` and `freeNormalize` write to the file system during each iteration of the loop, which could cause an AC vulnerability in space. The inter-procedural loop analyzer reveals that the `normalizeCompletely` method calls `normalizeOnce` and `freeNormalize` in a loop, of which `freeNormalize` and `isReduced` also contain loops. Each of these methods operates on a global string variable called `intersections`. A taint analyzer deduces that the variable `intersections` is set by untrusted (attacker controlled) inputs. With incriminating evidence gathered from various static analyzers, the code shown in Listing 10.4 becomes a prime target for targeted dynamic analysis.

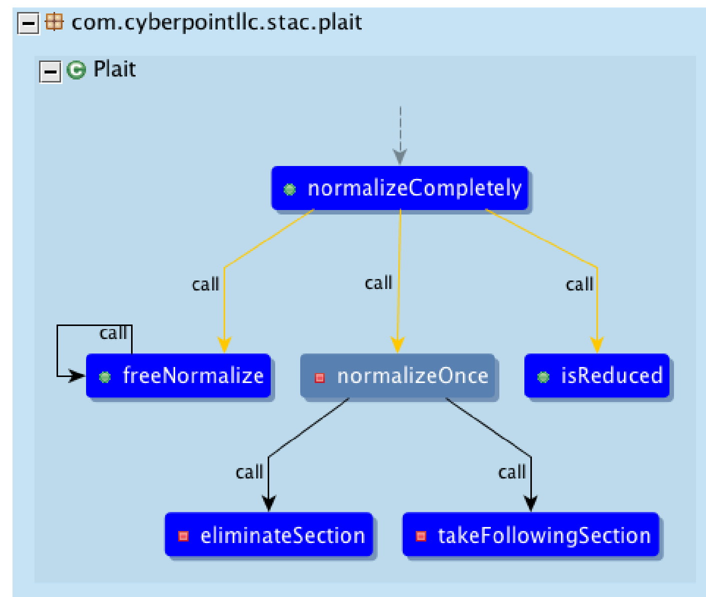



Figure 10.7 Screenshot of Loop Call Graph of `Plait.normalizeCompletely`

With the help of static analyzers, we have found the relevant code - it is small but extremely complex code. It is quite difficult to reason about this code manually or by using static analysis. In fact, between the two methods operating on the `intersections` variable there are 9 unique string operations in 62 locations, 20 of which are within loops as well as 8 unique character level operations in 60 locations, of which 27 locations are within loops. This is where targeted dynamic analysis can be immensely helpful. The `isReduced` method computes a non-trivial property on the `intersections` string, which is changing during each loop iteration. The analyst formulates the following hypothesis:

> HYPOTHESIS. *Can there be a string that has the property of being irreducible and therefore cause an infinite loop that writes to the file system?*

If the hypothesis is true, it is actually an algorithmic complexity vulnerability in time as well as space and thus an opportunity for the attacker to launch an extremely effective *denial of service* (DoS) attack.

Running the experiment using the Mockingbird framework yielded in 20 hours the first input that actually triggers the AC time vulnerability; an additional 22 inputs were found in 39 hours. The first input was "ËĎġčçęêďªã;" it hints that the vulnerability has to do with Unicode localization, but it does not explain the root cause of the vulnerability.

The motivating questions for the follow-up experiment are: *Why is the application is vulnerable to particular string inputs? Is there actually a class of inputs for the exploit?* Answers to such questions are important to gain holistic understanding of the vulnerability and modify the program to eliminate the root cause of the vulnerability.

To create a denial of service attack by causing an infinite loop the `isReduced` method should always return false. Therefore the relevant events of interest are the callsite to `isReduced` in `normalizedCompletely` and the two `return false;` statements in `isReduced`. The ICFG of the `normalizedCompletely` and `isReduced` methods is too large to show (55 nodes, 63 edges), but the IPCG is shown in Figure 10.8. From the IPCG we learn that there are three implicitly relevant branch conditions in `isReduced` and one branch condition (the loop header) in `normalizedCompletely`.

To discard irrelevant execution traces from invariant detection and abort execution on the start of irrelevant paths we insert an `abort-irrelevant` statement after the ICFG successor reachable on the `true` path of the branch condition in `normalizedCompletely` (which is just before the `normalizedCompletely` return statement).

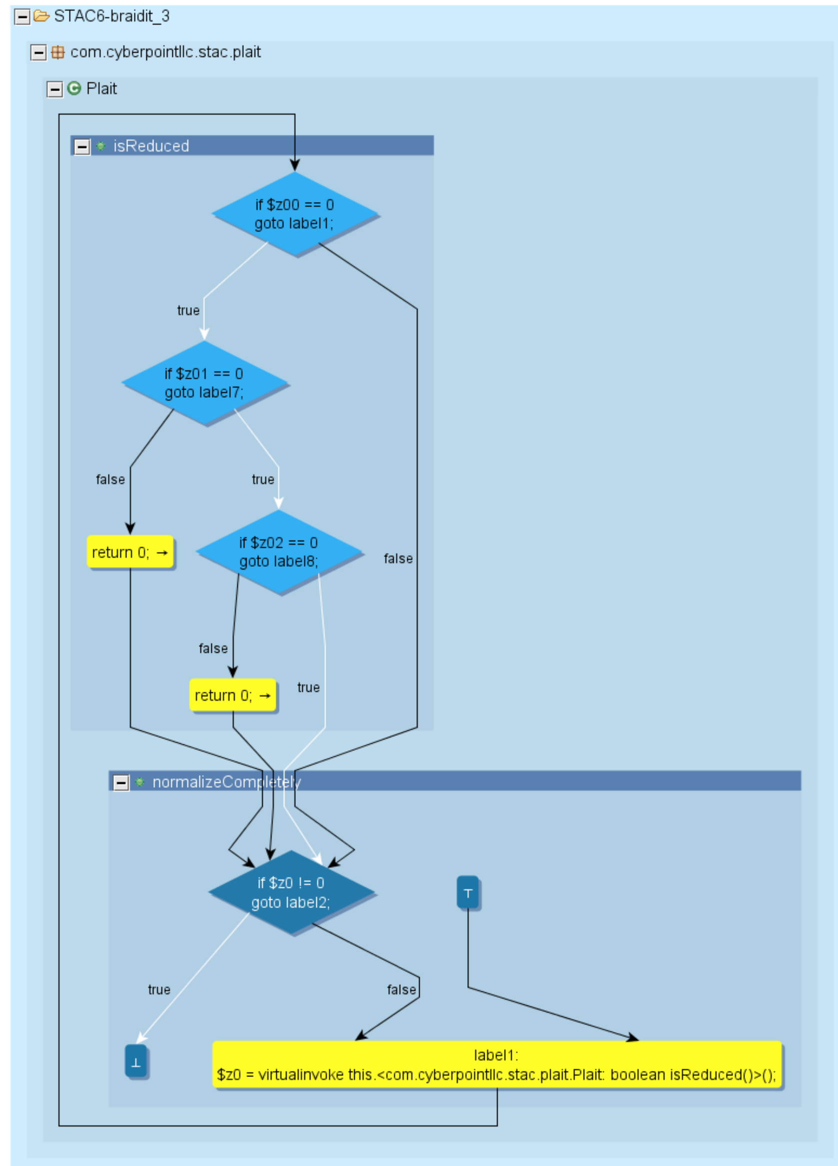



Figure 10.8 BraidIt Challenge Application Case Study IPCG

Using Mockingbird's APIs, we quickly built a custom tool that helps in answering the above questions. In this experiment, we targeted the same relevant code and retained the same mocking specifications, but replaced the AFL driver on the front-end with a brute-force search of the alphabet of characters accepted by the `Plait` constructor. Within 20 minutes the brute-force search revealed the string value "ªªª" as the first minimal string for the exploit as well as 13 other three character strings.

Each of the three character strings revealed in the follow up experiment produce execution traces of program behaviors that exhibit the infinite loop. The homomorphic program invariants computed for these execution traces reveal some useful insights into the program's behavior with respect to these malicious inputs.

As we expected, `isReduced` always returns `false`, but we learn that the `intersections` field is always a non-empty string before and after `normalizeOnce` and `freeNormalize` methods execute. Most importantly we learn that all exploit strings have at least one common character: "ª". With this information we learn that the "ª" character has some special property in the strings that are causing an infinite loop. Note that common string sequence invariants are not enabled by default in Daikon[3] and must be explicitly enabled to observe the invariant result.

Debugging the application with the minimal input and paying attention to the operations on the "ª" character reveals the heart of the vulnerability. We learn that the `freeNormalize` method removes a pair of matching characters (leaving a single 'ª' character remaining). Next the `isReduced` method returns false if the string contains an uppercase character of a lowercase character. This scheme is perfectly reasonable for ASCII characters, but there exists a small set of alphabetic Unicode characters such as 'ª' where uppercase('ª')==lowercase('ª'). Since a string of a single lowercase character that is identical to its uppercase is never reducible and the `freeNormalize` method will never affect a string of one character, the loop will never terminate.

After getting the holistic understanding of the vulnerability, we searched for previous studies of such vulnerabilities and found that these types of vulnerabilities are known to abound in codes

[3] https://plse.cs.washington.edu/daikon/download/doc/daikon/Daikon-output.html

for localization (which is exactly what the relevant code in the case study is for) and actually a patent is granted to the Honeywell Corporation for a method for detecting, analyzing, and mitigating similar vulnerabilities [129]. Finally, we used our knowledge to search for similar alphabetic Unicode characters whose uppercase is their lowercase and found there are 395 UTF-16 characters that could be used to craft exploits. Interestingly, this challenge program had been hardened and was thought not to be vulnerable. Yet, using the tools and methodologies described in this work we could discover an unintentional vulnerability that can be exploited for a denial of service attack.

The fuzzing result enabled by our targeted dynamic analysis framework computed in 20 hours is a valid exploit, but additional effort is required to understand the root cause of the vulnerability. The refined experiment to compute homomorphic program invariants revealed that that the exploit hinges on the property of a certain Unicode character in the tested inputs, which is key to understanding the vulnerability. The homomorphic invariants were computed in less than 30 seconds. For reference we tried computing the invariants that would be detected on the execution traces produced by the first 20 minutes of searching in the refined experiment without discarding the irrelevant traces and invariant detection took 48 hours before we stopped processing the remaining unprocessed execution traces. Of the invariants that were detected on the processed execution traces the key invariant of the common special character was not present, which would have been invalidated for execution traces with unique strings. It is conceivable that a human could understand the vulnerability without the aid of the homomorphic invariants, but the invariants make the task of program comprehension considerably easier.

# CHAPTER 11. CONCLUSIONS

In this work, we reviewed some of the fundamental challenges of program analysis that appear in modern languages by studying the properties of a toy language called *Elemental* that was created for this work. We then reviewed contemporary approaches to program analysis and identified the tradeoffs to each approach. Our primary objective of this work was to build practical program analysis tools and methodologies to detect software security anomalies. We began addressing our research question *RQ1* by reasoning that a human-in-the-loop approach augmented with machine automation could be used as an alternative approach to deal with intractable problems in program analysis. We demonstrated this approach had merit by routinely outperforming competing RD teams over a period of 7 years on two high profile DARPA programs – APAC [61] and STAC [77]. Through the work for these two research awards, we developed a multitude of static analysis research that addressed research question *RQ2* and enabled us to quickly come to suspicious code segments. During the APAC program, the work performed for *RQ1* and *RQ2* was often enough to locate the malicious code, however during the STAC program, which focused on finding vulnerable code segments with respect to an attacker budget, we found it was necessary to develop tools to dynamically verify a hypotheses about suspicious code segments. Through this work, we began to develop a methodology for performing a statically-informed dynamic analysis which addressed our third research question *RQ3*. Because regions of software can be very tightly coupled with less interesting code segments we found it was necessary to address *RQ4* in order to more effectively perform a targeted dynamic analysis. Some related approaches by [130, 49] attempt to perform a targeted analysis using either automated symbolic execution, concolic testing, and program slicing. Recent work by Ferles [131] applied Hoare logic to compute necessary preconditions to produce a target failure and then introduced a technique called program trimming to produced a modified binary with a reduced number of program paths. In order to scale the analysis, the analysis computes

modular but local function summaries and does not consider inter-procedural analysis globally. Our work focuses on a methodology for transitioning from human-centric static analysis to a dynamic analysis and the framework that is necessary to support a smooth transition. In our static analysis phase, we automatically compute the equivalence classes of relevant control flow paths across all functions in the program and then, similar to program trimming, we inject assertions to restrict program execution to the relevant program paths during dynamic analysis. In our dynamic analysis we mine invariants that hold true on just the remaining program paths that are relevant to the given concern, which we call homomorphic program invariants, to aid in program comprehension.

Our work on research question *RQ5* deals with the path explosion problem by leveraging human reasoning to identify control flow events of interest with respect to an analysis task. In this work, we present a tool to compute a *projected control graph* (PCG) an extend it to include inter-procedural analysis results. The PCG allows us to reason about equivalence classes of intra-procedural and inter-procedural control flows. Our primary contribution in this work is provided in our answer for research question *RQ7*. *RQ7* extends work by Ernst [69], which computes program invariants. A program invariant is defined as "a property that is true at a particular program point or points" [69] and is relevant if it is useful to a programming activity. Since program invariants are often undocumented assertions made by a programmer that hold the key to both a human and a machine reasoning correctly about a software verification task, we sought to improve the quality of program invariants. One way to improve the quality of program invariants is by improving the quality of immutability analyses. Our work on research question *RQ6* evaluates the current state-of-the-art immutability analyses and finds that there is no clear winner. Between the two competing approaches, one either pays a high computational cost for increased precision or accepts some inaccuracies to reduce the computational overhead and scale the analysis. *RQ7* takes a different approach and makes the observation that computing program invariants that hold true on equivalent classes of control flow can produce relevant program invariants that are not included in dynamically detected invariants across all program paths. As our primary contribution we propose

*homomorphic program invariants*, which are invariants that are computed only on relevant program paths with respect to an analysis task.

In Chapter 9, we present our SIDIS framework, which is a culmination of all prior work and off the shelf components (AFL [52] and Daikon [47]) to compute homomorphic program invariants. In our approach, the homomorphic program invariants are not the final product of the analysis, but instead one of many analysis outputs that may aid in human program comprehension. In Chapter 10, we present three case studies to show how our tools and methodologies can be used to detect algorithmic complexity vulnerabilities in three DARPA STAC applications that were hardended against static and dynamic analysis techniques. In the first case study the human understanding of vulnerability is quickly observed, although the homomorphic program invariants detected by Daikon were less useful and the case study instead relied on analysis provided in our prior work to uncover the underlying invariants. The extensibility of Daikon does present room for future work to include the instrumentation probes and time complexity analysis used to recover the relevant invariant in the first case study. In the second case study we leveraged our Mockingbird framework to perform a targeted dynamic analysis of an image processing routine that would otherwise be difficult to decouple from the supporting image rendering libraries. With the ability to execute the suspicious code segment with little overhead, we were able to mine the homomorphic program invariants that revealed that vulnerability would cause excessive computation for a very precise set of images. In some cases it is possible to use targeted dynamic analysis to scope the analysis in a way that only produces relevant execution traces (as was done in Case Study II), however in the general case, and in order to solve the toy example in Chapter 8, a path specific analysis is required to differentiate relevant and irrelevant execution traces. In the final case study, we managed to detect an unintentionally exploitable vulnerability that caused a denial of service attack in both space and time. The vulnerability of Case Study III was complex and difficult to understand even when given a proof of concept exploit. The homomorphic program invariants pointed to a property of the exploit that was necessary to understanding the root cause of the vulnerability and revealed that there was an entire family of exploits that could be used to exploit the vulnerability.

# BIBLIOGRAPHY

[1] L. F. Menabrea and A. Lovelace, “Sketch of the analytical engine invented by charles babbage,” 1842.

[2] N. History and H. Command, “NH 96566-KN The First "Computer Bug",” https://www.history.navy.mil/our-collections/photography/numerical-list-of-images/nhhc-series/nh-series/NH-96000/NH-96566-KN.html, 2018.

[3] K. Gödel, *On formally undecidable propositions of Principia Mathematica and related systems*. Courier Corporation, 1992.

[4] A. M. Turing, “On computable numbers, with an application to the entscheidungsproblem,” *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[5] MITRE, “Mitre common vulnerabilities and exposures,” https://cve.mitre.org, 2018.

[6] “CVE-2014-0160.” Available from MITRE, CVE-ID CVE-2014-0160., Dec. 03 2013. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[7] “CVE-2014-1266.” Available from MITRE, CVE-ID CVE-2014-1266., Jan. 08 2014. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266

[8] “CVE-2014-6271.” Available from MITRE, CVE-ID CVE-2014-6271., Sep. 09 2014. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271

[9] K. S. Holland, Benjamin, “A Bug or Malware? Catastrophic consequences either way.” Derbycon 4.0, 2014.

[10] Anonymous, “Internet census 2012: Port scanning/0 using insecure embedded devices,” *http://census2012.sourceforge.net/paper.html*, 2013.

[11] T. Koppel, *Lights out: a cyberattack, a nation unprepared, surviving the aftermath*. Broadway Books, 2015.

[12] E. Jackson, “Dawn of the code war: America’s battle against russia, china, and the rising global cyber threat,” 2018.

[13] Wikipedia, “Brainfuck — Wikipedia, the free encyclopedia,” https://en.wikipedia.org/wiki/Brainfuck, 1993, [Online; accessed 1-November-2018].

[14] C. Böhm, “On a family of turing machines and the related programming language,” *ICC Bull*, vol. 3, no. 3, pp. 187–194, 1964.

[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman, “Introduction to automata theory, languages, and computation,” *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001.

[16] T. Hornby, "The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program. Tweet." https://twitter.com/defusesec/status/523139275542368256, 2014.

[17] ——, "When you sort a deck of cards, you're moving them around, but it's the numbers on the cards that are telling you where to move them. Tweet." https://twitter.com/DefuseSec/status/523139973604597760, 2014.

[18] O. Aleph, "Smashing the stack for fun and profit," *http://www. shmoo. com/phrack/Phrack49/p49-14*, 1996.

[19] R. Langner, "To kill a centrifuge: A technical analysis of what stuxnetâĂŹs creators tried to achieve," *The Langner Group*, 2013.

[20] "CVE-2012-3015." Available from MITRE, CVE-ID CVE-2012-3015., May 30 2013. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3015

[21] F. E. Allen, "Control flow analysis," in *Proceedings of a symposium on Compiler optimization.* New York, NY, USA: ACM, 1970, pp. 1–19.

[22] "Linux 4.8 Drivers wiretest.c," https://elixir.bootlin.com/linux/v4.8/source/drivers/staging/lustre/lustre/ptlrpc/wiretest.c#L44, 2018.

[23] B. A. Nejmeh, "Npath: a measure of execution path complexity and its applications," *Communications of the ACM*, vol. 31, no. 2, pp. 188–200, 1988.

[24] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 1973, pp. 194–206.

[25] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[26] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88*, ser. POPL '88. New York, NY, USA: ACM, 1988, pp. 12–27.

[27] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," *Ecoop'95*, vol. 952, no. December, pp. 77–101, 1995.

[28] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *ACM SIGPLAN Notices*, ser. OOPSLA '96, vol. 31, no. 10. New York, NY, USA: ACM, 1996, pp. 324–341.

[29] F. Tip and J. Palsberg, *Scalable propagation-based call graph construction algorithms.* ACM, 2000, vol. 35, no. 10.

[30] Y. Smaragdakis, G. Balatsouras *et al.*, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[31] L. O. Andersen, "Program Analysis and Specialization for the C Programming Language," Ph.D. dissertation, University of Cophenhagen, 1994.

[32] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 1996, pp. 32–41.

[33] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, 1992.

[34] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1467–1471, 1994.

[35] T. Reps, "Undecidability of context-sensitive data-dependence analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 1, pp. 162–186, 2000.

[36] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *ACM SIGPLAN Notices*, vol. 39, no. 6. ACM, 2004, p. 131.

[37] O. Lhoták, *Program analysis using binary decision diagrams*, 2006, vol. 68, no. 03.

[38] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA 09*, vol. 44, no. 10, p. 243, 2009.

[39] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java," in *Drops-Idn/6116*, vol. 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016, pp. 1–26.

[40] P. Cousot and R. Cousot, "Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Principles of Progamming Languages.* ACM, 1977, pp. 238–252.

[41] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT—a formal system for testing and debugging programs by symbolic execution," in *ACM SIGPLAN Notices*, vol. 10, no. 6. New York, NY, USA: ACM, 1975, pp. 234–245.

[42] L. A. Clarke, "A Program Testing System," in *Proceedings of the 1976 Annual Conference.* ACM, 1976, pp. 488–491.

[43] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[44] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, vol. 8, 2008, pp. 209–224.

[45] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, p. 20, mar 2012.

[46] P. Godefroid, N. Klarlund, and K. Sen, "Dart," *ACM SIGPLAN Notices*, vol. 40, no. 6, p. 213, 2005.

[47] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[48] M. D. Ernst, "Static and dynamic analysis: synergy and duality," in *WODA 2003 ICSE Workshop on Dynamic Analysis*. New Mexico State University Portland, OR, 2003, pp. 24–27.

[49] W. Le, "Segmented symbolic analysis," in *Proceedings - International Conference on Software Engineering*. IEEE Press, 2013, pp. 212–221.

[50] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings 2016 Network and Distributed System Security Symposium*, vol. 16, 2016, pp. 1–16.

[51] Y. Shoshitaishvili, "25 Years of Program Analysis." DEFCON 25, 2017.

[52] M. Zalewski, "American Fuzzy Lop (2.52B)," http://lcamtuf.coredump.cx/afl/, 2018.

[53] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the Art of War: Offensive Techniques in Binary Analysis," in *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, 2016, pp. 138–157.

[54] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM, 1971, pp. 151–158.

[55] M. Weiser, "Program Slicing," in *Proc. ICSE '81*. IEEE Press, 1981, pp. 439–449.

[56] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 121–141, jan 1979.

[57] S. Horwitz, T. Reps, and D. Binkley, *Interprocedural slicing using dependence graphs*. ACM, 2004, vol. 39, no. 4.

[58] A. Tamrawi and S. Kothari, "Projected control graph for computing relevant program behaviors," *Science of Computer Programming*, vol. 163, pp. 93–114, 2018.

[59] ——, "Projected control graph for accurate and efficient analysis of safety and security vulnerabilities," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. IEEE, 2017, pp. 113–120.

[60] B. Holland, P. Awadhutkar, S. Kothari, A. Tamrawi, and J. Mathews, "Comb: Computing relevant program behaviors," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 109–112. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183476

[61] DARPA, "Automated Program Analysis for Cybersecurity (APAC)," https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-11-63/listing.html, 2011.

[62] R. A. De Millo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," *Communications of the ACM*, vol. 22, no. 5, pp. 271–280, 1979.

[63] T. Murray and P. van Oorschot, "Bp: Formal proofs, the fine print and side effects."

[64] DARPA, "Explainable Artificial Intelligence (XAI)," https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-16-53/listing.html, 2016.

[65] ——, "Computers and Humans Exploring Software Security (CHESS)," https://www.fbo.gov/spg/ODA/DARPA/CMO/HR001118S0040/listing.html, 2018.

[66] B. Schneier, *Beyond fear: Thinking sensibly about security in an uncertain world.* Springer Science & Business Media, 2006.

[67] J. Boyd, *Destruction and creation.* US Army Comand and General Staff College, 1987.

[68] F. P. Brooks, "The computer scientist as toolsmith II," *Communications of the ACM*, vol. 39, no. 3, pp. 61–68, 1996.

[69] M. Ernst, *Dynamically discovering likely program invariants.* University of Washington, 2000.

[70] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium.* ACM, 2000, pp. 449–458.

[71] C. Pacheco and M. D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," in *European Conference on Object-Oriented Programming.* Springer, 2005, pp. 504–527.

[72] H. Fouladgar, B. Minaei-Bidgoli, and H. Parvin, "On possibility of conditional invariant detection," in *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems.* Springer, 2011, pp. 214–224.

[73] S. Wei, "A survey and categorization of program comprehension techniques," Ph.D. dissertation, Concordia University, 2002.

[74] A. Von Mayrhauser and A. M. Vans, "Program understanding behavior during adaptation of large scale software," in *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on.* IEEE, 1998, pp. 164–172.

[75] R. Langner, "To Kill a Centrifuge," The Langner Group, http://www.langner.com/en/wp-content/uploads/2013/11/To-kill-a-centrifuge.pdf, Tech. Rep. November, nov 2013.

[76] "Heartbleed conspiracy theories." [Online]. Available: http://www.businesscomputingworld.co.uk/openssl-heartbleed-criminal-and-government-conspiracy-theories/

[77] DARPA, "Space / Time Analysis for Cybersecurity (STAC)," https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-14-60/listing.html, 2014.

[78] T. Deering, S. Kothari, J. Sauceda, and J. Mathews, "Atlas: A New Way to Explore Software, Build Analysis Tools," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 588–591.

[79] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout," in *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, ser. CCS '12. New York, NY, USA: ACM, 2012, p. 217.

[80] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 1101–1118.

[81] B. Holland, T. Deering, S. Kothari, J. Mathews, and N. Ranade, "Security toolbox for detecting novel and sophisticated android malware," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 733–736.

[82] G. R. Santhanam, B. Holland, S. Kothari, and J. Mathews, "Interactive visualization toolbox to detect sophisticated android malware," in *Visualization for Cyber Security (VizSec), 2017 IEEE Symposium on*. IEEE, 2017, pp. 1–8.

[83] G. R. Santhanam, B. Holland, S. Kothari, and N. Ranade, "Human-on-the-Loop Automation for Detecting Software Side-Channel Vulnerabilities," in *International Conference on Information Systems Security*. Springer, 2017, pp. 209–230.

[84] P. Awadhutkar, G. R. Santhanam, B. Holland, and S. Kothari, "Intelligence Amplifying Loop Characterizations for Detecting Algorithmic Complexity Vulnerabilities," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 00, 2017, pp. 249–258.

[85] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks." in *Usenix Security*, vol. 2, 2003.

[86] E. Adi, Z. A. Baig, P. Hingston, and C.-P. Lam, "Distributed denial-of-service attacks against http/2 services," *Cluster Computing*, pp. 1–8, 2016.

[87] "XML denial of service attacks and defenses," https://msdn.microsoft.com/en-us/magazine/ee335713.aspx.

[88] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[89] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Costa: Design and implementation of a cost and termination analyzer for Java bytecode," in *Formal Methods for Components and Objects*. Springer, 2008, pp. 113–132.

[90] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl, "Alternating runtime and size complexity analysis of integer programs," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 140–155.

[91] J. Ouaknine and J. Worrell, "On linear recurrence sequences and loop termination," *ACM SIGLOG News*, vol. 2, no. 2, pp. 4–13, 2015.

[92] T. Wei, J. Mao, W. Zou, and Y. Chen, "A New Algorithm for Identifying Loops in Decompilation," in *Static Analysis*, no. 2006. Springer, 2007, pp. 170 – 183.

[93] "TimSort," http://bugs.python.org/file4451/timsort.txt.

[94] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *Foundations of Software Engineering*. ACM, 2007, pp. 395–404.

[95] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a Java bytecode optimization framework," in *Conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[96] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," *IEEE Transactions on Software Engineering*, vol. 40, no. 12, pp. 1185–1205, 2014.

[97] A. M. Ben-Amram, S. Genaim, and A. N. Masud, "On the termination of integer loops," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2012, pp. 72–87.

[98] O. Olivo, I. Dillig, and C. Lin, "Static detection of asymptotic performance bugs in collection traversals," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 369–378.

[99] S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: precise and efficient static estimation of program computational complexity," in *ACM SIGPLAN Notices*, vol. 44, no. 1. ACM, 2009, pp. 127–139.

[100] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in *Programming Language Design and Implementation*. ACM, 2012, pp. 67–76.

[101] M. Abliz, "Internet Denial of Service Attacks and Defense Mechanisms," *University of Pittsburg*, no. March, p. 50, 2011.

[102] B. Holland, G. R. Santhanam, P. Awadhutkar, and S. Kothari, "Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities," in *Proceedings - 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation, SCAM 2016*. IEEE, 2016, pp. 79–84.

[103] B. Holland, G. R. Santhanam, and S. Kothari, "Transferring State-of-the-Art Immutability Analyses: Experimentation Toolbox and Accuracy Benchmark," in *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*. IEEE, 2017, pp. 484–491.

[104] R. Kersten, M. Field, K. Luckow, and C. S. PĂČsĂČreanu, "POSTER : AFL-based Fuzzing for Java with Kelinci âĹŮ," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2511–2513.

[105] C. Ashbacher, *Growing Object-Oriented Software, Guided by Tests*. Pearson Education, 2010, vol. 9, no. 3.

[106] E. Corp, "Modeling Lessons From Verifying Large Software Systems for Safety and Security," in *Proceedings of the 2017 Winter Simulation Conference*, no. LDV, 2017, pp. 1431–1442.

[107] ByteBuddy.net, "Byte Buddy," http://bytebuddy.net, 2018.

[108] E. Kuleshov, "Using the ASM Framework to Implement Common Java Bytecode Transformation Patterns," *Proc. of the AOSD - Intl. Conf. on Aspect-Oriented Software Development*, pp. 1–7, 2007.

[109] N. Voss, "afl-unicorn: Fuzzing Arbitrary Binary Code," https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf, 2017.

[110] P. Godefroid, "Micro execution," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 539–549.

[111] T. Kaczanowski, *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski, 2012.

[112] P. Hell and J. Nesetril, *Graphs and homomorphisms*. Oxford University Press, 2004.

[113] R. E. Tarjan, "A unified approach to path problems," *Journal of the ACM (JACM)*, vol. 28, no. 3, pp. 577–593, 1981.

[114] A. Sălcianu and M. Rinard, "Purity and side effect analysis for java programs," in *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, 2005, pp. 199–215.

[115] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, "Verifiable functional purity in java," in *Proc. of the Conference on Computer and Communications Security*. ACM, 2008, pp. 161–174.

[116] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proc. of the International Conference on Software Engineering*, 2013, pp. 422–431.

[117] B. Holland, "Immutable objects in github projects," http://boa.cs.iastate.edu/boa/?q=boa/job/public/43284, July 2016.

[118] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, "Reim & reiminfer: Checking and inference of reference immutability and method purity," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 879–896.

[119] W. Huang and A. Milanova, "Reiminfer: method purity inference for java," in *Proc. of the International Symposium on the Foundations of Software Engineering*, 2012, p. 38.

[120] A. Milanova and W. Huang, "Dataflow and type-based formulations for reference immutability," in *International Workshop on Foundations of Object-Oriented Languages*, 2012, p. 89.

[121] A. Milanova and Y. Dong, "Inference and checking of object immutability," in *Proc. of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM, 2016, pp. 6:1–6:12.

[122] O. Lhoták and L. Hendren, "Context-sensitive points-to analysis: is it worth it?" in *International Conference on Compiler Construction*. Springer, 2006, pp. 47–64.

[123] "java.lang.runtimeexception when processing class files with array accesses on null arrays #527." [Online]. Available: https://github.com/Sable/soot/issues/527

[124] A. Y. Tamrawi, "Evidence-enabled verification for the Linux kernel," Ph.D. dissertation, Iowa State University, 2016.

[125] C. Steindl, "Static Analysis of Object Oriented Programs." in *ECOOP Workshop for PhD Students in OO Systems*, 1999, pp. 112–117.

[126] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2155–2168.

[127] M. Zalewski, "Pulling JPEGs out of thin air," https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html, 2014.

[128] "Artin group," https://en.wikipedia.org/wiki/Artin_group, Spet. 2018.

[129] D. B. Kirk Schloegel, "Method for software vulnerability flow analysis, generation of vulnerability-covering code, and multi-generation of functionally-equivalent code," Patent US8 407 800B2, mar, 2013.

[130] Z. Cui, W. Le, M. L. Soffa, L. Wang, and X. Li, "Magic: Path-guided concolic testing," 2011.

[131] K. Ferles, V. Wüstholz, M. Christakis, and I. Dillig, "Failure-directed program trimming," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 174–185.