The Dissertation Committee for Walter Chochen Chang
certifies that this is the approved version of the following dissertation:

Improving Dynamic Analysis with Data Flow Analysis

Committee:

Calvin Lin, Supervisor

Kathryn McKinley

James C. Browne

Sarfraz Khurshid

Andrew Myers

Improving Dynamic Analysis with Data Flow Analysis

by

Walter Chochen Chang, B.A.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2010

To my father, who encouraged me to start graduate school, and to my wife, who encouraged me to finish.

Acknowledgments

This thesis would not have been possible without the assistance of numerous colleagues who provided support, guidance, and advice at various stages.

For our dynamic data flow analysis work, I would like to thank Vitaly Shmatikov for his help on earlier drafts of the CCS'08 paper. His insights on positioning the work, advice on presentation to the security community, and general championing have been invaluable for getting this work out there.

For Bullseye and our software testing work, I am grateful for the valuable comments and feedback from Sarfraz Khurshid and Miryung Kim during the development and evaluation of Bullseye. Their insights have been valuable for familiarizing myself with the landscape of software testing and thinking through the place my work has in relation to the bigger picture. I would also like to thank James C. Browne, who always reminded me to keep my eye on the bigger scientific picture and deeper insights when I get too caught up in trivial details.

For the software infrastructure design and implementation, which must constitute the majority of the total hours spent on this thesis, I am grateful to have had assistance. I would like to thank Teck Bok Tok for enduring my numerous questions, insightful and inane, on the design and implementation of the Broadway framework. I would not have been able to implement the tracing algorithms for both dynamic data flow analysis and Bullseye without his guidance and example code. I would also like to thank Brandon Streiff, one of the finest students I've had the pleasure of teaching. In addition to helping with the test harnesses for dynamic data flow analysis, he supplied the bulk of the implementation for the symbolic execution and dynamic input generation component for Bullseye as part of his senior thesis. It is rare for teaching assistants to so directly benefit from their students as I have.

Finally, of course, I am indebted to my adviser, Calvin Lin, without whom none of this could have been possible.

Improving Dynamic Analysis with Data Flow Analysis

Publication No. ____________

Walter Chochen Chang, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Calvin Lin

Many challenges in software quality can be tackled with dynamic analysis. However, these techniques are often limited in their efficiency or scalability as they are often applied uniformly to an entire program. In this thesis, we show that dynamic program analysis can be made significantly more efficient and scalable by first performing a static data flow analysis so that the dynamic analysis can be selectively applied only to important parts of the program. We apply this general principle to the design and implementation of two different systems, one for runtime security policy enforcement and the other for software test input generation.

For runtime security policy enforcement, we enforce user-defined policies using a dynamic data flow analysis that is more general and flexible than previous systems. Our system uses the user-defined policy to drive a static data flow analysis that identifies and instruments only the statements that may be involved in a security vulnerability, often eliminating the need to track most objects and greatly reducing the overhead. For taint analysis on a set of five server programs, the slowdown is only 0.65%, two orders of magnitude lower than previous taint tracking systems. Our system also has negligible overhead on file disclosure vulnerabilities, a problem that taint tracking cannot handle.

For software test case generation, we introduce the idea of targeted testing, which focuses testing effort on select parts of the program instead of treating all program paths equally. Our "Bullseye" system uses a static analysis performed with respect to user-defined "interesting points" to steer the search down certain paths, thereby finding bugs faster. We also introduce a compiler transformation that allows symbolic execution to automatically perform boundary condition testing, revealing bugs that could be missed even if the correct path is tested. For our set of 9 benchmarks, Bullseye finds bugs an average of 2.5× faster than a conventional depth-first search and finds numerous bugs that DFS could not. In addition, our automated boundary condition testing transformation allows both Bullseye and depth-first search to find numerous bugs that they could not find before, even when all paths were explored.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software quality is important to today's computing environment. Buggy and unreliable programs have caused billions in losses annually [76], the loss of expensive space probes [75], and even the death of radiation therapy patients [64]. Moreover, flaws in common operating systems, web browsers, and servers have allowed malicious hackers to gain unauthorized access to numerous systems, enabling further attacks, fraud, and identity theft [65]. Thus, improving software quality is a critical part of improving computing today and will become even more important in our increasingly interconnected future.

To find and fix bugs and vulnerabilities, developers need powerful tools at their disposal. Many of the most powerful tools are based on *dynamic analysis*, where the program being debugged or fixed is augmented so that it can be analyzed as it runs. For example, Valgrind [78] analyzes every memory operation performed by a program and is used by millions of programmers to find and fix memory leaks and buffer overflows. Tools like Pin [67] enable programmers to write their own analyses. Dynamic taint tracking systems have been used to protect programs from attacks [92, 33, 80, 84, 99, 34]. Directed [44, 45] and concolic [87] test input generators automatically generate inputs to test programs.

Unfortunately, dynamic analyses often suffer from significant inefficiencies. Valgrind routinely experiences 10-100× overhead [78]. Software taint tracking systems also suffer from excessively high overheads. Test input generators can be slow to find bugs. To continue to make improvements in software security and quality, the efficiency and flexibility of these tools must be improved.

The cause of these inefficiencies is that the dynamic analysis performs many computations that are redundant or not actually needed for the task at hand. For example, dynamic taint tracking systems track taintedness on objects that are never involved in any attack, and directed test input generators generate numerous test cases that do not reveal any faults. If these inefficiencies could be reduced or eliminated, the dynamic analysis would be improved greatly.

In this thesis, we will show how dynamic analyses can be dramatically improved by first performing a static data flow analysis. By over-approximating or anticipating the desired dynamic behavior, static analysis can disable dynamic analysis on large swaths of the program, focusing analysis effort on the remainder and greatly improving the effectiveness of the system.

This thesis will focus on two specific areas within dynamic analysis: runtime security policy enforcement and automated test case generation.

1.1 Runtime Security Enforcement

The first major thrust of this thesis is runtime security enforcement. A major challenge today is to secure servers from attacks. Servers for businesses, governments, and even private individuals are attacked regularly, resulting in significant financial loss or serious breaches of national security. To make matters worse, many of these applications were written without security in mind or with simpler security models than today's needs demand. Securing these programs is an enormous but important undertaking. Thus, providing tools to help secure legacy code and defend servers against attacks is a critical part of protecting our electronic infrastructure.

Attacks on servers are made possible by flaws in the server program that allow an attacker to manipulate it into behaving in unintended ways. To prevent these attacks, we must ensure that the program cannot violate some security policy at runtime. One way to enforce policies is runtime monitoring, where additional code runs alongside the original program and monitors its execution, stepping in to prohibit actions that violate some given security policy.

One highly promising runtime monitoring technique is *dynamic taint tracking*. In dynamic taint tracking, input from untrustworthy sources is marked as tainted. The taint tracking system propagates taint information as data is copied and used throughout the program, using this taint information to ensure that tainted data is never used for certain sensitive operations. Taint tracking has been used to address a wide variety of security problems, including buffer overflows [92, 33, 80, 84, 99, 34] and web application vulnerabilities [95, 81, 99, 34].

Unfortunately, current taint tracking systems are limited in their capabilities. Most taint tracking systems, especially hardware-based systems [92, 33, 34], are not general and cannot be easily extended to handle more complex information flow problems without rewriting significant portions of the system or redesigning key data structures. As a result, current taint tracking systems cannot be used for other important issues like file disclosure attacks or enforcement of a labeled security model. Thus, to move beyond memory errors and to support higher-level policies, we need a more general system than taint tracking.

Moreover, software-based taint tracking systems are needlessly inefficient. Current state-of-the-art taint tracking systems [99, 84] have brought the overhead for I/O-bound applications from 37× [80] down to an average of only 5%, but for more computationally intensive applications, the average overhead is still 75% or higher. Much of this overhead exists because the system tracks taintedness for *all* memory used by the program. However, the number of instructions and the amount of data involved in any given attack is miniscule compared with the rest of the program [79]. To leverage this observation, the best current taint tracking systems reduce the amount of data that is tracked by applying conventional intraprocedural compiler optimizations. However, these optimizations miss opportunities because they do not reason about interprocedural flows and because they do not reason about the semantics of taint. For example, while a dependence analysis can eliminate tracking on scalar variables that can never be tainted [99], it cannot eliminate tracking on objects that are tainted but can never be misused. Eliminating such unnecessary tracking can therefore eliminate a significant amount of the overhead.

As part of this thesis, we have developed a system that addresses both of these shortcomings by coupling a rich declarative policy specification with a deep whole-program analysis that greatly reduces runtime overhead. We address generality by basing our system on data flow analysis [57]. Our system can handle any problem that can be conservatively modeled by typestate [91], which includes taint analysis as its simplest case. Users supply a problem definition and a security policy to our compiler, which then produces an enhanced version of the original source program that performs a dynamic data flow analysis [54] to enforce the specified policy.

Our use of dynamic data flow analysis as an enforcement mechanism also addresses the second issue of efficiency. Our compiler uses the user-supplied problem definition to perform a static whole-program data flow analysis on the source program to determine where dynamic analysis is actually required. In many cases, this is a very small fraction of the entire program, or none at all, if the program contains no vulnerabilities. By adding code to perform dynamic analysis only at places where it is actually required, our system adds a minimal amount of additional code and tracks significantly fewer objects than current taint tracking or dynamic data flow analysis systems.

We will discuss the details of our dynamic data flow analysis and its accompanying static analysis in Chapter 3. We describe our evaluation of the dynamic data flow analysis system in Chapter 4.

1.2 Efficient Software Testing

The second major thrust of this thesis is in software testing. Testing is the usual way that software functionality and correctness is verified. An effective testing system is one that helps find bugs quickly or provides higher assurance of the absence of bugs. A test case generator is a system that generates test inputs to a program to aid in software testing.

A major obstacle to testing programs of any significant size is the exponential number of possible paths through the program. It is simply not possible to test every path. Therefore, software test case generators adopt a variety of approaches to limit their scope. Most methods are based on the *small input hypothesis* [70], which assumes that for any given bug, there exists a small input that makes the bug manifest. Bounded exhaustive testing [18] systems find bugs by systematically generating all non-equivalent inputs up to a certain size or by exploring all paths up to a certain length. This exploration consumes considerable time and testing resources and does not readily find bugs that manifest only on larger inputs.

To address these shortcomings, *directed testing* [44, 87] has been proposed as an alternative means of exploring program behaviors. A directed test case generator starts with an initial path and proceeds to explore paths in the program in a depth-first manner by calculating branch conditions for the current path, altering the conditions, and generating a new input that forces execution to a different path. This path-oriented exploration allows the testing system to explore much longer paths than with bounded exhaustive testing. However, there is still no guarantee or assurance that the system will explore paths that lead to bug-inducing inputs. Assuming the *competent programmer hypothesis* [2], which states that a program written by a competent programmer will be mostly correct, directed testing systems will spend significant time generating inputs that do not reveal flaws, limiting their scalability and bug-finding efficiency.

Both of these current approaches share one thing in common: all paths are treated equally. These approaches carry with them inherent inefficiencies when applied to the task of fault identification in large programs. The number of paths in a larger program is enormous. Moreover, the overwhelming majority of possible paths reflect correct execution and will not reveal faults.

Lacking guidance as to where faults may appear in programs, test input generators spend most of their time generating inputs that produce correct behavior rather than incorrect behavior. Thus, efficient testing for bug-finding requires guidance about which paths are more likely to reveal bugs.

As part of this thesis, we have implemented and evaluated a system, Bullseye, that implements *targeted testing*, a new automated testing technique that focuses testing resources on only select parts of programs. Targeted testing utilizes testing resources more efficiently by focusing efforts on select areas of the program instead of treating the testing space uniformly, thereby increasing the efficiency and scalability of testing systems by increasing the relevance of generated inputs. Our key insight is that paths and statements in the program should not be treated equally. If it is possible to identify a set of paths or locations of greater importance, a test case generator can explore paths that are more relevant while ignoring the remainder of the program. By using static analysis to compute information flow relations in the program, Bullseye can guide test input generation towards paths that most affect or are most affected by points of interest in the program.

Bullseye starts from a given set of program statements and objects which we call *interesting points*. Bullseye then analyzes the flow of data through the program to determine both what the points of interest affect and what affects the points of interest. The directed test case generator then uses this information to guide testing along paths that are relevant to the points of interest, generating test cases that better target the points of interest. In our evaluation, we use Bullseye as though it were a part of a change management system, with program edits as points of interest. Other possible sources for interesting points include static error checkers, profilers, or even programmer hunches.

In addition, this thesis identifies a broader problem facing all test generators based on symbolic execution—the *path inadequacy problem*—which can prevent test input generators from finding faults even if they find the correct path. To address this problem, we introduce a method for performing automated boundary condition testing by encoding conditions in the control flow graph. This technique can be used in conjunction with targeted testing or independently as part of any other symbolic execution system. Our technique enables both directed and targeted testing to find numerous bugs that they could not find before.

We describe the details of our test input generator and static analysis in Chapter 5, as well as our boundary condition testing solution. We evaluate Bullseye against directed testing in Chapter 6.

1.3 Theme: Analyzing the Flow of Data for Selective Analysis

Although the areas of runtime monitoring and test case generation seem very different, the solutions that we have developed all rely on the same fundamental insights:

- Systems that perform a dynamic analysis often spend considerable time performing analysis on parts of the program that are not important to the goal; eliminating this unnecessary work by *selectively* performing the analysis is essential for highly efficient and scalable systems.

- *Analyzing the flow of data* in a program allows tools to determine what is important and thereby selectively apply the dynamic analysis.

In this thesis, we have applied these two principles to runtime monitoring and test case generation. For monitoring, we used data flow analysis to describe and enforce security policies, including existing taint-based policies. Static data flow analysis reduces the amount of dynamic data flow analysis to a small fraction of the program, resulting in a much more efficient dynamic system. For test case generation, our static analysis allows our directed test case generator to prioritize specific locations and objects in the program. As a result, Bullseye produces inputs that focus on this subset of the program rather than treating all program paths equally. In both cases, analyzing the flow of data allows the tool to model the problem and to narrow the focus to a small fraction of the space.

Although we have only explored these two areas in this thesis, we believe that these principles are broadly applicable. The basic idea of performing a heavyweight analysis in only select locations in a program has already been used successfully in a client-driven pointer analysis for error checking [50]. Our work builds on these ideas and this codebase and extends it to dynamic data flow analysis and software testing.

1.4 Contributions

This thesis makes the following contributions:

- We implement a system for performing general typestate-based dynamic data flow analysis on C language programs. This system uses static data flow analysis to determine where dynamic analysis may be required and eliminates unnecessary instrumentation everywhere else.

- We evaluate the effectiveness of our dynamic data flow analysis system against current taint tracking systems. Our evaluation shows that our system can match or improve upon the overhead of the best current taint tracking systems, sometimes by multiple orders of magnitude, thus demonstrating that dynamic data flow analysis is an efficient mechanism for implementing existing taint-based solutions.

- We demonstrate that our dynamic data flow analysis system is more general than taint tracking systems by applying it to a problem that taint tracking cannot handle.

- We introduce the idea of targeted testing, a new testing paradigm that complements existing coverage-based techniques and can be applied orthogonally to a wide range of existing test case generators.

- We introduce a new metric, *mutant kill speed*, that measures the speed at which a fault can be identified by a testing system. This new metric allows us to quantify the speed at which bugs can be found by a testing system, as opposed to the thoroughness of the test suite at the end.

- We implement a system, Bullseye, for targeted automated test case generation. Bullseye demonstrates how a standard concolic execution system can be extended to support targeted testing.

- We develop a technique that allows symbolic execution systems to perform automated boundary condition testing by encoding boundary conditions in the program's control flow graph. Our solution can be used in conjunction with targeted testing as well independently. Our boundary condition testing technique finds numerous bugs that prior symbolic execution systems could not find before except through chance.

- We evaluate the effectiveness of the Bullseye system, demonstrating that the original baseline concolic execution testing system shows significant improvements in fault identification speed when targeted testing techniques are used. We use Bullseye to reveal flaws in benchmarks and real-world example codes. We evaluate our boundary condition testing technique, measuring the improvement in fault coverage as well as the slowdowns incurred, and offer insights and explanations into these. Finally, we discuss new insights into test input generation and future directions for software testing.

The remainder of this thesis will proceed as follows. First, we will discuss related work in Chapter 2. Next, we will discuss the two projects that comprise this thesis. In Chapter 3, we discuss the design and implementation of our dynamic data flow analysis system and its accompanying static analysis. We evaluate this system in Chapter 4 on server programs, compute-intensive programs, and different security policies. Chapter 5 discusses the implementation of Bullseye, our test input generator, covering our additions to test input generation, the static analysis that guides Bullseye, and boundary condition testing. In Chapter 6, we evaluate the Bullseye system against directed testing, with and without boundary condition testing, as well as explore the effects of heuristic parameters and static analysis. Finally, we finish with thoughts on the future of dynamic analysis and other concluding remarks in Chapter 7.

Chapter 2

Related Work

In this chapter, we provide a brief overview of related work. As the fields of software security and testing are extremely broad, we cannot provide a comprehensive list of work in this area, but we cover a sampling of work that is most relevant thematically and technically to the design and implementation of our dynamic data flow analysis and software testing systems.

This chapter is organized around topics in related work. First, we discuss data flow analysis in Section 2.1, providing background in the data flow analysis that both our dynamic data flow analysis and Bullseye build upon. Next, we discuss static analysis and its applications to error checking and policy enforcement in Section 2.2. In Section 2.3 we discuss approaches to runtime monitoring for security. Next, in Section 2.4, we discuss dynamic taint tracking systems and compare and contrast them with our dynamic data flow analysis system. In Section 2.5 we discuss work in software testing evaluation with an emphasis on coverage criteria. In Section 2.6, we cover path-based test input generators, which we build off of to create the Bullseye test system. Finally, in Section 2.7, we discuss change impact management and its relation to Bullseye.

2.1 Data Flow Analysis

Both Bullseye and our dynamic data flow analysis are based on data flow analysis [57] and both are implemented on top of the Broadway static data flow analysis system [47]. Although their end uses are different, the static analysis components of each share a common iterative data flow analysis and data dependence analysis [73].

The static component of our dynamic data flow analysis system performs a sound but incomplete analysis; that is, if it indicates that there are no problems, then there are in fact no problems. However, it may report false positives. Moreover, while the other static program checkers discussed in Section 2.2 typically leave it to the user to fix any potential problems, our system inserts code to perform a dynamic analysis that guards against problems without requiring the developer to manually fix every reported vulnerability. Other systems that combine static and dynamic analysis require significant manual assistance [55], while ours is automated.

For software testing, our use of static data flow analysis is a key differentiator. Prior work in dynamic testing systems [62] focuses on control flow; while this may allow the test system to reason about paths towards a particular statement, it cannot reason about the effects of changes on variables. To capture the effects of changes to variables, a data dependence analysis is required. Prior research in thin slicing [89] suggests that data dependencies are more useful than control flow when reasoning about bugs and software faults. The static analysis component of Bullseye thus makes extensive use of data flow analysis to properly reason about the interaction between statements and variables.

2.2 Static Analysis

Static analysis refers to a family of techniques that analyzes a program (in either source or binary form) and attempts to derive information about the possible runtime behaviors of the program. Static analysis can be used for a wide variety of applications, including compiler optimizations and lightweight program verification. Static analyses differ widely in the properties that they can check for and in their soundness and completeness.

Static analysis has been widely used for error checking [88, 40, 6, 50, 96]. Systems range across the spectrum in configurability and depth of analysis. At the simple end of the spectrum, there are source code scanners that look for syntactic patterns common to programmer errors and coding style violations [40]. These systems typically rely on a simple set of hardcoded rules and look for syntactic patterns in source code. Rarely do they perform any rigorous analysis, and as such are necessarily unsound and incomplete, nor do they typically offer significant mechanisms for customization.

More sophisticated error checkers read user-defined analysis problems from annotation files and perform interprocedural analysis. The Metal system [6] uses user-provided state machines with machines associated with variables and state transitions driven by program syntax. It has successfully found errors in production code, including the Linux kernel [6, 12]. Type systems and type qualifiers [88, 42] provide a simple way for programmers to express analyses as in-line annotations, using type checking and type inference to find possible violations. The Broadway system [50] uses user-provided declarative problem specifications to perform a whole-program data flow analysis and has also found numerous bugs in real code. Our dynamic data flow analysis system builds directly off of Broadway and uses the same annotation language.

Static analysis can construct models of program behavior that can then be enforced dynamically. For example, control [61, 1] and data flow integrity [23] ensure that the program never deviates from statically computed models of control or data flow. That is, the program statically computes the control or data flows that are possibly legitimate and inserts extra guards to ensure that only these legitimate control or data flows are used at run time. However, these systems restrict dynamic program behavior to an *over-approximation* of possible legal behavior, which allows false negatives. Our dynamic data flow analysis over-approximates during static analysis but enforces the policy exactly during dynamic analysis, thereby avoiding both false positives and false negatives.

Finally, our techniques for vulnerability analysis and test generation bear some similarities with model checking [26]. Model checking proves program correctness by demonstrating that no incorrect path through the program is feasible. Model checking can reveal bugs by producing a counterexample when the constraints cannot be satisfied. However, model construction is difficult task, requiring significant time and effort by experts even for simple models, limiting widespread use [38]. Model checking can even miss simple bugs due to model simplification [38]. For dynamic data flow analysis, we use data flow analysis as a lightweight proxy for model checking, with any imprecision or ambiguity resolved by our dynamic data flow analysis at runtime. For security purposes, this combination of static and dynamic analysis yields similar guarantees to more rigorous systems while remaining scalable and easy to use, with the tradeoff of moving part of the verification problem to runtime.

2.3 Runtime Monitoring

Runtime monitoring refers to the class of solutions that improve the security and integrity of a system by monitoring the behavior of the program at runtime, stepping in to take action to prevent events that violate some security policy. These solutions differ significantly in how they model and enforce program behavior. For example, some systems only enforce memory safety, while others enforce policies based on control flow or taintedness. We now discuss representative examples of such solutions briefly.

Runtime monitoring is used in many current systems for preventing buffer overflow attacks and invalid memory accesses [7, 32, 31, 14, 9]. These techniques ensure that certain implicit language semantics (such as the integrity of the stack) and other low-level artifacts are maintained. They do not handle program behavior beyond such memory errors. While effective at stopping many attacks, these techniques do not allow for meaningful user configurability or detection of more subtle errors. Techniques that address only low-level memory errors are unnecessary in languages that already enforce memory safety, but these programs can and do contain errors and vulnerabilities. For example, information flow errors and simple bugs can and do occur in programs written in memory-safe languages such as Java.

There are several systems, most notably Inlined Reference Monitors [39] that check user-defined policies presented in the form of logical statements over events in the program [39, 27]. Unfortunately, their own designers admit that the system is difficult and cumbersome to use [39]. They are ill suited to problems that depend on the flow of information and dynamic allocation, including problems as simple as taint tracking, because they encapsulate everything in global state variables. [39].

Finally, language-based solutions for information flow and security have also been proposed [74, 86, 56]. These systems typically attempt to guarantee as much program safety as possible at compile time but include a small runtime monitoring component. While these systems can provide strong guarantees with respect to information flow non-interference or memory and type safety, they require significant developer effort to rewrite or redesign current programs. Our dynamic data flow analysis system is applicable to current code and does not require additional effort from the application developer.

2.4 Dynamic Taint Tracking

Our dynamic data flow analysis is a generalization of dynamic taint tracking [95, 80, 81, 92, 33, 24, 29, 99, 84, 63], which has been used to protect against buffer overflows, stack smashing, and format string attacks, and which covers attacks previously addressed separately by ad-hoc solutions [32, 31, 9, 30]. Dynamic taint tracking associates a bit of information with objects and memory addresses, encoding taintedness or nontaintedness. At runtime, taint tracking marks inputs and any derived copies as tainted and ensures that tainted data is never used in certain places. Much of the previous work in taint tracking has used specialized hardware [92, 33] or dynamic binary instrumentation frameworks [80, 29, 84].

Dynamic data flow analysis was first proposed for identifying uninitialized variables in 1979 [54]. Although static data flow analysis [57] has been widely used for decades by compilers to perform optimizations, work in dynamic data flow analysis has largely focused on uninitialized variables [54] or dynamic flow relations [62, 3, 77, 29]. Moreover, most existing systems for dynamic data flow analysis make little or no use of the corresponding static data flow analysis, resulting in excessive overhead [77]. We implement a general dynamic data flow analysis system that can handle typestate properties while using the corresponding static analysis to achieve low overhead.

The closest existing works to ours are Taint-Enhanced Policy Enforcement by Xu, *et al.* [99] and GIFT [63]. Xu, *et al.* use a compiler to insert taint tracking code into programs. By using highly tuned tag representations and Linux-specific memory management techniques, they are able to bring the overhead of taint tracking for I/O-bound server programs to an average of only 5%. However, their performance on compute-bound applications is considerably worse, with an average overhead of 75%. Their system is also designed specifically for taint tracking. In contrast, our dynamic data flow system can be applied to any typestate problem and is platform-independent.

GIFT [63] is a "General Information Flow Tracking" framework that uses a compiler to automatically add code to propagate and check tags associated with data. GIFT handles a wide range of dynamic data flow problems. However, GIFT policy specifications are provided in the form of code transformation patterns; the user essentially tells GIFT how to rewrite the existing program to incorporate calls to the GIFT runtime, much like an aspect-oriented programming system. GIFT then performs some program slicing in an attempt to reduce the added code, but these efforts are largely ineffective [63]. Our system uses declarative specifications that define an analysis to the compiler, with the policy framed in terms of predicates on analysis results. Thus, GIFT only tells the compiler *what* to add, not what it *means*. As a result, our system can eliminate many calls that GIFT could not, because our data-flow slicing algorithm can reason about the semantics of the runtime analysis.

Our dynamic data flow analysis system can also be used as a tool to weave cross-cutting security functionality into programs in the manner of aspect-oriented programming [59]. The cross-cutting nature of security in applications is well-known [94]. Most aspect-oriented programming systems allow the user to implement aspects as simple transformations and code insertion (termed *advice*) over syntactic elements such as method calls in the program (termed the *pointcut*). However, our system features a radically different pointcut and coding model that is based on declaratively specifying a dynamic data flow analysis. Although data flow pointcuts have been proposed as a means to make taint tracking possible in aspect-oriented programming [71] and aspect systems have been extended to expose compiler analyses to the pointcut definition [72, 5], both of these approaches retain the transformation-oriented development model. Programmers still implement aspects as additional code that is inserted at statically computed locations. In contrast, our approach is closer to declarative programming. Programmers using analysis-based aspects implement cross-cutting functionality by providing problem-specific information to a general analysis framework. Finally, current approaches do not perform an interprocedural flow-sensitive data flow and pointer analysis and are thus much less precise.

2.5 Software Testing and Coverage

Much software testing research has focused on coverage criteria. A suite of test inputs is said to cover a program if the inputs collectively satisfy a given coverage criteria. Commonly used coverage criteria [13] include branch coverage (where the goal is to ensure that every branch is tested) and statement coverage (where the goal is to ensure that every statement is executed at least once). Most evaluations of software testing techniques focus on measuring improvements in the coverage criteria, with the presumption that higher coverage indicates a higher quality test suite. The "gold standard" for coverage criteria is *all-paths* coverage, where the goal is to ensure that every single possible path through the program is tested. While all-paths is generally not achievable, the other coverage criteria can be thought of a subsets or approximations to all-paths.

In addition to the traditional coverage criteria, there is a significant body of work on data flow coverage criteria [85, 43]. The data flow coverage criteria expands on coverage to include criteria based on the uses and definitionss [73] in a program. For example, the "all-defs" criteria [85] seeks to cover every path between a def of a variable and some subsequent use, which can be further refined by distinguishing between computation uses (such as addition) and predicate uses (such as the conditional in a branch).

A somewhat different spin on measuring test suite quality can be found with mutation testing [2]. In mutation testing, mutants are generated by syntactically modifying the program. The quality of a test suite is then measured by measuring the percentage of the mutants that it can kill by distinguishing the difference between the original and the mutant; a test suite that can kill all of the mutants is said to be mutation adequate. While there are questions about whether automatically generated or hand-seeded mutants are representative of real-world bugs [4], mutation testing directly measures the ability of a test suite at bug-finding, in contrast to coverage criteria, which only attempt to measure the thoroughness of the test suite.

2.6 Path-Based Test Case Generation

Dynamic testing techniques [62, 46] are a family of techniques that involve executing (typically symbolically) the program in a restricted fashion to find the path constraints force execution to reach any given point in execution. By solving these constraints, the system can generate an input that forces execution to proceed down a particular path. Although the path constraints allow the testing system to target specific statements in a program, current dynamic testing systems have only been applied to coverage [46, 62], where the goal is to generate one input for every statement or branch in the program, thus achieving statement or branch coverage.

The closest work to Bullseye is Directed Automated Random Testing (DART) [44]. DART exhaustively explores the space of possible program behaviors (as opposed to program inputs) in a depth-first manner by generating inputs that cause the program to execute along different paths. A similar system, EXE [100, 21], explores in a breadth-first manner by forking concrete and symbolic execution at each branch. These systems can be extended to support pointer-based data structures as well [87]. All of these explore the space of program paths essentially blindly. In contrast, Bullseye uses static analysis to provide guidance to DART so that it preferentially explores paths that are more likely to contain bugs or are otherwise more "interesting" for the tester.

It is well-known that directed testing with a depth-first search can be inefficient. Recent work has focused on alternative search patterns for directed testing. Some simpler search patterns use random sampling and random restarts [17] to alleviate problems that can occur if the initial input is nowhere near a bug, but these solutions do not fundamentally alter the depth-first search aside from introducing more randomness. The bug-finding effectiveness of these techniques is not known as they were evaluated with the goal of achieving branch coverage.

White-box fuzz testing [45] uses a generational search pattern. Starting from the initial input, paths that differ by changing one branch form the first generation, paths that differ from the first generation by a branch form the second generation, and so on. This changes the search pattern to an expanding "beam" centered around the original input.

Heuristics are often used by existing systems to target common bugs such as overflows [21]. The test generator can use some hardcoded heuristics that can opportunistically attempt to solve for things like buffer overflows and integer overflows at every available opportunity. In fact, this technique is necessary to prevent breadth-first searches from exponentially exploding [21] by finding these errors early. While useful for finding these specific low-level errors in systems code, they do not help find higher-level bugs in the application logic because these bugs are necessarily program-dependent and domain-specific. The closest heuristic yet employed to our targeted testing approach is one evaluated in the Crest system [17], which uses the distance in the control flow graph to currently uncovered branches to guide the search. However, the evaluation of Crest only examines the effectiveness of heuristics at improving branch coverage, not fault identification.

In addition to simple idiom-finders, more flexible heuristics can be used. The Pex tool [98] uses fitness functions applied to paths to evaluate the ongoing search and uses path-searching techniques to seek paths that maximize the fitness function. This allows for considerably more flexibility in the search depending on which heuristic function is used. However, they do not investigate heuristics based on static analysis data, which is the key idea behind targeted testing and our bullseye system.

2.7 Change Impact Management and Data Flow

Recent advances in change management have focused on the impact of program changes on test suites. Orso, Shi, and Harrold have developed a technique for efficiently computing a subset of the regression test suite to rerun to test a program change [82], avoiding the need to rerun the entire test suite each time the program changes. Similar change impact analysis techniques can be used to recompute important metrics such as branch and statement coverage for the suite after a change without rerunning the entire test suite [25].

Another thread in recent work on symbolic test case generation focuses on explicitly solving for differences between program versions. This can be accomplished by first constructing a model of the program that captures which states and outputs are semantically equivalent, symbolically executing both versions of the program and tracking their state, and finally explicitly solving a formula that computes an input that forces the two versions into different final states, thus revealing a program deviation [20]. This technique has been used successfully to identify differences between different implementations of common protocols, but requires two complete versions of a program and significant user guidance.

Our work is complementary; Bullseye generates new test cases geared to expose differences between versions, while regression test selection techniques filter out existing test cases that do not show differences between versions. Aside from the program deviation work [20] there has been much less work on generating new inputs in response to changes versus selecting subsets of old inputs.

Chapter 3

Dynamic Data Flow Analysis

In this chapter, we describe an extensible compiler-based system that simultaneously advances the state of the art in improving both the overhead and the generality of dynamic taint tracking. The performance of our system is an order of magnitude better than any previous software taint tracking system while remaining portable and fully general to typestate problems.

Our system uses a combination of static and dynamic data flow analysis to enforce user-specified policies. The dynamic analysis tracks tag values that are used to enforce the policy. The static analysis identifies which statements in the program are necessary for computing the tag values at potential policy violations and eliminates tracking and instrumentation from statements and objects that are provably not involved in policy violations. This combination results in dramatically lower overheads than previous systems while maintaining generality.

3.1 Solution Overview

In this section, we will provide a high-level overview of our system. Our system is motivated by two design goals: the need for a more general model than simple dynamic taint tracking and the need to eliminate redundant and wasteful tracking to improve performance. After discussing these goals, we

Traditional Tainted Data Attacks
Format String Attacks
SQL Injection
Command Injection
Cross-Site Scripting
Privilege Escalation
Other Security Problems
File Disclosure Vulnerabilities
Labeled Security Enforcement
Role-Based Access Control
Mandatory Access Control
Accountable Information Flow

Table 3.1: A sampling of the kinds of problems that our system can handle. Taint tracking can only handle the top set of problems.

will provide a description of the architecture of our system before proceeding to more technical details.

3.1.1 Goal: Generalizing Taint Tracking

While dynamic taint tracking has been proven to be an effective mechanism for handling many basic types of attacks, it cannot handle more sophisticated problems in its basic form. Table 3.1 shows some of the attacks that taint tracking can handle. It also lists several more complex problems that taint tracking cannot handle below the line.

The general idea of taint tracking can be easily extended to more complex problems. In essence, taint tracking works by attaching symbolic *tags* to the concrete *data* in the program at runtime. These tags represent states or values for some *property* of the data. Actions in the program can tag values or propagate existing tags and security decisions are made by examining tag values at certain locations. These extensions give the system the power to handle all of the problems in Table 3.1. Such a more general model is a logical extension from taint tracking and has been termed General Information Flow Tracking (GIFT) [63] in prior work.

Our policies use a simple and flexible dynamic model similar to General Information Flow [63]. Our system associates symbolic tags with data objects at runtime, it updates the tags as the program executes, and it enforces policies based on the tag values. Unlike prior systems, our system is explicitly based on *data flow analysis* [57], a technique for computing facts about data by observing how it flows through the program. This design allows our system to both statically check for and dynamically guard against policy violations from the same specification. A static data flow analysis computes an approximate solution that holds over all possible executions of the program because a fully precise solution is undecidable [57]. In contrast, a dynamic data flow analysis [54] computes precise facts but only about the current execution. These complementary characteristics allow our system to use a static data flow analysis, discussed in Section 3.5, to compute a conservative solution at compile time and to refine the result at runtime to enforce a policy efficiently and precisely.

To perform the dynamic data flow analysis that actually enforces the policy, our compiler inserts into the source program calls to a small runtime library that manages tag information, as well as any required checks necessary to enforce the policy. Since nothing in our system is specific to taint tracking, our system and our optimizations apply to all general information flow tracking problems.

3.1.2 Goal: Eliminating Unnecessary Tracking

Performing a general dynamic data flow analysis by inserting calls naively into the program invariably leads to unacceptably high overheads. To understand why existing taint tracking systems are inefficient and have high overheads, consider the code in Figure 3.1. This code contains a format string vulnerability where a tainted buffer is printed. Assuming a policy that uses taint analysis to guard against format string attacks, current taint tracking systems, including those that perform some static analysis, would track taintedness on all buffers in this example, as well as anything that the `process` function touches and anything that those variables affect, leading to high overheads.

Fortunately, very little tracking is actually required. Our system can prove that tracking on `buf1` is not required because it is never passed to `printf` or any other sensitive function. Additionally, if tracking on `buf1` is not required, neither is tracking on `otherbuf` because `buf1` receives its value only from `otherbuf`. We also do not need to track anything in the call to `process(buf1)` because none of its results is used by `printf`. Moreover, we do not need to track the original `input` buffer because we know that it is always tainted; it is sufficient to simply mark `buf` as tainted at the call to `memcpy`. Finally, we do not need to track anything else that `process(buf)` can affect if none of the resulting values are misused.

The keys to removing unnecessary tracking are an interprocedural static analysis that leverages semantic information about the security policy, a sophisticated interprocedural pointer analysis to perform policy-specific optimizations, and a dependence analysis to reason about the scope of possible vulnerabilities. Without semantic information about the policy, our system

```
char input[1024];
char buf[1024];
char otherbuf[1024];
char buf1[1024];
...
read_from_network(input);
read_from_network(otherbuf);
...
memcpy(buf, input, 1024);
memcpy(buf1, otherbuf, 1024);
process(buf);
process(buf1);
...
printf(buf);
```

Figure 3.1: A simple example illustrating the benefits of our static analysis. Current systems must track all objects, while our static analysis can eliminate tracking on all except `buf`.

could not distinguish possible violations from safe events. Without a precise pointer analysis, our system could not account for flows between objects in an effective manner. Without a dependence analysis that builds on the pointer analysis and knowledge of the policy, our system could not determine which objects are involved in possible vulnerabilities. Moreover, all of these analyses must be interprocedural to eliminate flows among functions. These ideas drive the implementation of our solution.

3.1.3 Solution Architecture

Figure 3.2 shows the overall architecture of our system. The input is an untrusted program. The output is an enhanced program that enforces some specified security policy, which is selected by the end-user at compile time. Our implementation is built on the Broadway [49, 47] data flow analysis engine and the C-Breeze [48] source-to-source translator for C programs.

The policy itself is defined in an annotation file that describes the policy and the effects of standard library calls on the policy. Thus, the policy is entirely separate from the data flow tracking mechanism, so in addition to the existing security policies that we have already defined, new security policies can be specified without modifying either the compiler or the runtime system. In Section 3.2, we discuss how we use the Broadway annotation language to implement two different policies.

At runtime, our policies are enforced with the aid of a simple dynamic data flow analysis library, discussed in Section 3.3. This library is linked to the transformed code and provides mechanisms for tracking the propagation of the flow values specified in the policy at runtime. Calls to the library are inserted by our system according to a set of transformation rules, discussed in Section 3.4.

The key innovation of our system is the static analysis that we use to prevent the system from inserting instrumentation everywhere. The static analysis uses a lightly modified interprocedural data flow analysis combined with a client-driven pointer analysis to build data structures that our *data flow slicing* algorithm traverses. We discuss the specifics of our static analysis in Section 3.5.

The remainder of this chapter discusses the components of our system in more detail.

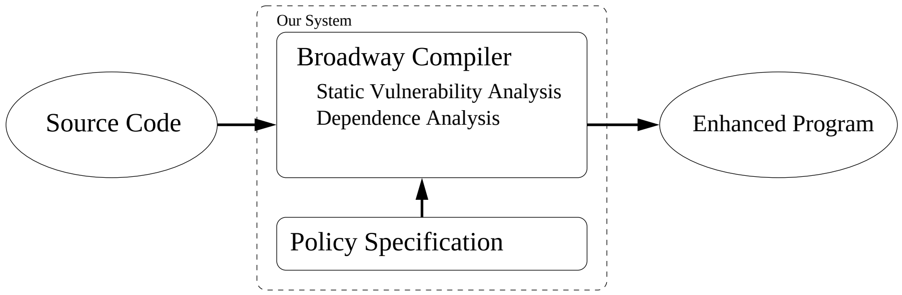

Figure 3.2: The overall structure of our system. The compiler takes a source program and a security policy and produces an enhanced version of the program that enforces the policy by performing dynamic data flow analysis.

3.2 Policy Specification

In most taint tracking systems, the semantics of taint analysis are hardcoded into the system. Because our system is designed to handle general data flow problems, our system instead factors out the semantics of the analysis and policy to an external file that contains annotations describing the property to analyze, the policy to enforce, and the effects of library procedures on the property. This file contains the *same information* that would have been hardcoded into a compiler-based taint tracking system, but it provides the capability to extend our system to other problems without changing the core analysis. Unlike in-lined annotations, our annotations define an analysis that is *independent* of the input program, enabling reuse across many programs. A typical user does not have to write any annotations to use an existing policy. The creation of new policy files is a careful activity that is only necessary when defining a new analysis or security policy.

Our system uses the Broadway declarative annotation language [49, 47], which has been previously used for static error checking [50] and library-level optimizations [51]. The annotation file tells the compiler how to perform a specific data flow analysis by supplying the specifics for the rules in Section 3.4. The rules fall into three categories:

- **Defining the Lattice.** The lattice for each typestate property must be defined. The tags used at runtime correspond to the flow values, while the lattice itself defines the meet function that specifies how flow values should be combined when used together in arithmetic and other operations.
- **Describing Effects of Library Calls.** The compiler also needs to know how the various library calls affect tag values. For each external function that affects the flow values, a brief summary annotation must be provided that describes how the function can affect the flow values of globals and arguments.
- **Defining Security Policies.** Lastly, the compiler needs to be given the definition of policy violations. Violations are defined as predicates over flow values that are checked at procedure boundaries, most commonly a check on the flow value of an argument. By default, violations trigger our default error handler, which logs the violation and blocks the operation, but the user can supply a custom error recovery function, which can be application-specific.

3.2.1 Example: Specifying Taintedness

To illustrate how the Broadway annotation language is used, we will now show how a well-known taint tracking problem, format string attacks, can be expressed.

```
property Taint : { Tainted, { Untainted } }
                   initially Untainted
```

Figure 3.3: Defining taintedness in the Broadway annotation language

The first step is to define taintedness as a property of data. To define the taint analysis with the Broadway annotation language, we use the specification in Figure 3.3. This specification defines a *lattice* (or property) called `Taint` with two *flow values* (or tags), `Tainted` and `Untainted`. It also defines a lattice order that puts `Untainted` higher than `Tainted`, so when tainted and untainted data are combined, the result is tainted. (In other words, $meet(Tainted, Untainted) = Tainted$.) The `initially` clause defines `Untainted` to be the default flow value. Note that in the original Broadway annotation language, the initial value is optional, but our dynamic data flow analysis *requires* that an initial value be specified as we assume that all freshly-allocated memory and any untracked variables default to this value.

Next, we must also provide summaries for library functions and system calls that affect taintedness. This defines how taintedness is introduced into the system. Figure 3.4 shows the specification that declares that data received from the network (via the `recv` library call) are always tainted.

Here, the `on_entry` keyword describes function parameters and gives a name, `buffer`, to *the object pointed to by* `buf`. This is important as it is not the *pointer* that is being tainted by `recv`, but rather the buffer that it points to. The Broadway annotation language uses the `-->` arrow notation to describe pointer relations among arguments, including multiple indirection, allowing us to precisely specify which object is which in a chain of pointer relations.

```
procedure recv(s, buf, len, flags)
{
  on_entry { buf --> buffer }
  analyze Taint { buffer <- Tainted }
}
```

Figure 3.4: Defining sources of taintedness in Broadway

The `analyze` keyword describes the effect of the function on flow values. Here, the previously-defined name `buffer` is used, and we specify that the object named `buffer` (which is the object pointed to by `buf`) becomes `Tainted`.

The compiler must also reason about the propagation of tainted data. For the source program, this flow is inferred automatically from the lattice, but for library functions for which the compiler does not have source code, this information can be provided by specifications. For example, the `strdup` function copies a string, and with it, whatever taintedness the string had. Figure 3.5 shows the specification for the `strdup` library function.

Here, the `on_exit` keyword is the counterpart to `on_entry`, describing pointer relations at function exit. This example specifies that `strdup` returns a pointer to `string_copy` and that `string_copy` should have whatever taintedness that `string` has.

Finally, to track the unsafe use of tainted data, we specify that certain functions such as `printf` should not accept tainted strings as their format strings, preventing format string attacks [30]. The specification for `printf` is shown in Figure 3.6. Here, the `error` keyword signals to the compiler that special action is required when the condition is met, invoking the default error handler if none is supplied.

```
procedure strdup(s)
{
  on_entry { s --> string }
  on_exit { return --> string_copy }
  analyze Taint { string_copy <- string }
}
```

Figure 3.5: Defining taint propagation via library calls in Broadway

```
procedure printf(format, args)
{
  on_entry { format --> format_string }
  error if (Taint: format_string could-be Tainted)
          "Error: tainted format string!"
}
```

Figure 3.6: Defining format string vulnerabilities in Broadway

3.2.2 Example: Specifying File Disclosure

Dynamic data flow analysis is not limited to taint-like problems. To illustrate the flexibility of our system, we also apply it to file disclosure vulnerabilities. File disclosure can occur when a remote user can connect and download the contents of arbitrary files, thus improperly revealing sensitive information. This vulnerability can be present when a program behaves unintentionally like an FTP server; that is, if the remote user can specify the name of a file whose contents are then sent over the network. Note that sending data from files not directly specified by the user is fine, as is sending responses constructed from user input. In essence, file disclosure is a simple privacy protection problem where the goal is to ensure that untrusted users cannot directly specify data to access. These attacks are not well-studied on C programs because overwrite attacks account for the majority of C vulnerabilities.

However, these vulnerabilities are common among web applications written in scripting languages such as PHP, Python, and Perl. Thus, our techniques remain relevant and applicable to safe languages.

File disclosure cannot be modeled with only taint tracking because taint tracking does not distinguish between the source of data and the trustedness of data. A taint tracking system could disallow the transmission of tainted data, but such a policy would also prevent legitimate echoes of network input. The taint tracker could also disallow transmission of any file data, but such a policy also eliminates legitimate transfers and would even prevent most query services from operating. To model file disclosure accurately, the system must track *both* the trustedness (whether the data is under attacker control) and the origin (whether the data comes from a file) of data within the system.

We will now show how file disclosure attacks can be prevented with a security policy in our system. To define file disclosure attacks with the Broadway annotation language, we must define both trust and origin of data, as shown in Figure 3.7. The trustedness of data is defined in a manner similar to taintedness and allows us to distinguish between trust levels corresponding to the program itself, local inputs, and remote inputs. The origin of data is determined by the kind of data source.

For the file disclosure problem, only *File* is relevant, but this definition allows reuse for other policies that wish to distinguish between other sources. The initial value of Program indicates that the data comes from within the program and not any external source.

In both cases, the lattice is flat, with elements merging to an implicit bottom element in the lattice.

```
property Trust : { Remote, External, Internal }
                 initially Internal

property Kind : { File, FileSystem, Client, Server,
                  Pipe, Command, StandardIO, Environment,
                  SystemInfo, NameServer, Program }
                initially Program
```

Figure 3.7: Defining the lattice for file disclosure vulnerabilities

The trustedness of a socket depends on whether it is a local Unix socket or an actual remote network socket. In this case, our system uses the actual concrete type of the socket as it is not always possible to know the precise type of a socket until runtime. To allow for this, the body of an `analyze` statement allows for simple conditions, as shown in Figure 3.8.

In many cases, it is possible to evaluate these conditions at compile time, as most calls to `socket` will use one of the constants like `AF_UNIX`. If it is not possible to determine this value at compile time, the value is determined dynamically at run time, with the appropriate additional code added by our system.

Functions that deal with I/O must mark their *Kind* and *Trust* appropriately. In Figure 3.9, when a file is opened with `fopen`, the file handle has *Kind File* and the *Trust* of the filename used to open it. Similarly, data read from files has the *Kind* and *Trust* of the file handle. The corresponding specifications are analogous to taint tracking, but with two distinct properties instead of one.

Finally, we define what a violation of the policy is in Figure 3.10. A file disclosure attack can occur when *File* data from a file with *Remote* trustedness

```
procedure socket(domain, type, protocol)
{
  analyze Kind { IOHandle <- Server }
  analyze Trust {
    if (domain == AF_UNIX)
      IOHandle <- External
    default
      IOHandle <- Remote
  }

  on_exit { return --> new IOHandle
            return --> null
          }
}
```

Figure 3.8: Trust and Kind assignments for socket open calls. Broadway allows flow values to be assigned from the concrete values of function parameters.

```
procedure fopen(path, mode)
{
  on_entry { path --> path_string
             mode --> mode_string  }
  access { path_string, mode_string }
  modify { Disk }
  on_exit { return --> new file_structure --> new IOHandle
            return --> null
          }
  analyze Kind      { IOHandle <- File }
  analyze Trust     { IOHandle <- path_string }
}
```

Figure 3.9: Trust and Kind assignments for fopen calls

```
procedure write(fd, buf_ptr, size)
{
  on_entry {
    fd --> IOHandle
    buf_ptr --> buffer
  }
  access { buffer }
  modify { IOHandle, Disk }

  error  if (Trust : buffer could-be Remote && \
             Kind : buffer could-be File && \
             Trust : IOHandle could-be Remote && \
             Kind : IOHandle could-be Server
           "ERROR file access violation detected"
}
```

Figure 3.10: Policy for preventing file disclosure vulnerabilities. Both the Trust and Kind of outbound data are checked before information is written to a network socket

is sent to a server socket with *Remote* trust (indicating that it was initiated by a remote user).

This policy can now be used to guard against file disclosure attacks. It is easily extendable to cover other forms of information disclosure. For example, the system can be prevented from transmitting information about the filesystem (such as the presence or absence of certain files) in addition to the file data itself by adding a few lines to the policy file.

3.2.3 Other Problems

Although we focus on the above two problems in this thesis, our system can be used to enforce a wide variety of problems. Lattices are a natural model for many security problems [10, 35, 16]. For example, multilevel security can be implemented with a lattice representing hierarchical levels, such as *Unclassified*, *Classified*, or *TopSecret*, along with properties representing categories, such as *Army*, *Navy*, etc. Library I/O functions would be annotated to call a user-provided helper function to read the appropriate label from the file, while the annotations for operations like string copy would remain essentially identical to those for taint tracking or file disclosure. For additional information on the Broadway language and its capabilities, please refer to prior work [49, 50, 47].

3.3 Dynamic Data Flow Analysis Runtime Library

At runtime, the security policy is enforced by means of additional compiler-inserted code in the program. This instrumented program makes use of dynamic typestate information to enforce the policy. To accomplish this, the system relies on a small library that provides facilities for tracking the flow values associated with bytes of memory at runtime.

At a high level, the library provides two basic functions. First, the program can request that a region of memory be tagged with a certain flow value. Second, the program can query the flow value associated with some byte of memory. All higher level operations, including any actual enforcement checks, are performed by the code inserted into the program.

At the implementation level, our runtime system is quite different from most existing taint tracking systems. Rather than maintaining a "giant array" to track flow values, we adopt a sparse representation. We will now discuss the particulars of our implementation.

3.3.1 Flow Values

Many existing taint tracking systems are designed specifically around taint and similar properties that only require one bit of information. These systems cannot be easily extended to handle more complex properties without serious invasive modifications and potentially serious memory consumption issues. In contrast, our system is designed to be fully general to typestate properties and does not place any restrictions on the size or number of flow values.

The flow values defined in the security policy are normally represented by simple integer values. The actual integer type is set at compile time. By default, the runtime uses 32-bit integers; however, it can be compiled to use integers of different sizes and even `char`s to save memory. The only requirement is that the flow value type be sufficient to represent all possible flow values for the property; if the property contains more than 255 flow values, `char`s cannot be used. The value -1 (for signed integers) or MAXINT (for unsigned integers) is reserved by the system can cannot be used to represent a value.

The runtime treats flow values as opaque numbers. Any operations that depend on the meanings of the flow values, such as *meet* operations or policy checks, are handled at a higher level by the inserted code, not the runtime library itself.

The concrete integer values for the flow values are assigned on a per-lattice basis, starting from 1. If multiple lattices are used by the analysis, elements from different lattices may have the same integer value. This is not a problem because the lattices themselves are kept in separate trees, so there can never be a value collision across lattices.

3.3.2 Memory Representation

Most software taint tracking systems employ large bit vectors to track taint information about memory regions. For example, one system [99] simply uses a 1gb address range that is otherwise unused in 32-bit Linux and uses simple arithmetic to convert from real addresses to locations in the bit vector. However, these methods, while simple, are extremely wasteful of memory, especially when the actual taint information is sparse. Moreover, reserving large amounts of memory is detrimental to performance; the previously-mentioned example [99] depends on an OS-specific memory convention to avoid the performance penalty of `malloc`'ing 1GB of memory at program start. Without this trick, the performance of their system is "unacceptable" [99].

For our dynamic data flow analysis runtime, we have developed a sparse representation for tracking tag information, loosely inspired by the tree-like structures previously used by Chilimbi for memory leak profiling [53]. Our tree is structured as follows:

- The root node of the tree represents the entire range of the address space (4GB in a 32-bit system). Every interior node in the tree has the same number of children, from 16 to 128 in powers of two. The default is a 16-ary tree. The interior nodes represent the corresponding fractions of the parent's address space. For example, if we are using a 16-ary tree, each child of the root node represents 1/16th of the entire address space (256mb in a 32-bit system).

- The leaf nodes represent fixed-size power-of-two address ranges, defined at compile time. They contain a pointer to a conventional bit vector that stores typestate information. For example, a leaf node representing a 16-byte range will contain a pointer to a bit vector containing 16 flow value entries. The size represented by the leaf nodes must consume whatever remains of the address space that is not represented by the interior nodes. For a 32-bit system, a 16-ary tree has 7 levels, with the last 4 of the 32 bits represented by a leaf node with 16 bit vector entries.

- Each interior node contains a slot for a stored flow value. If all memory addresses represented by the node have the same flow value, it is stored here and no children are created. Thus, large address ranges with the same flow value are stored high in the tree rather than duplicated across the leaf nodes.
- The tree is sparse; interior nodes are created on demand as needed. Most pointers to child nodes will actually be null because of this.
- The root node stores the default flow value. Queries to addresses not explicitly represented in the sparse tree are presumed to be in virgin territory and thus must contain the default flow value.

Because our model explicitly assumes a default flow value, tree nodes for a memory region do not need to be created unless a non-default flow value is stored. When this occurs, the runtime creates the minimum number of nodes necessary to capture the information, invalidating flow values stored in interior nodes as necessary. For example, marking a 16KB page-aligned region might not require the creation of any leaf nodes, as the information can be stored in interior nodes, while marking a 1-byte region requires creating all nodes along the path to the corresponding leaf node, plus the bit vector for the leaf node marked up appropriately.

Each property used by the dynamic data flow analysis has its own separate tree. Thus, if there are two properties in play, there are two separate trees, one for each property. This engineering choice keeps the implementation simple and fast for the one-property case. However, for analysis problems with multiple properties that share common addresses, it results in duplicated bookkeeping information in the interior nodes. Although we have not found it to be necessary, multiple properties can be manually combined by the user into a single property, at the expense of having a more complex *meet* function.

3.3.3 Compiler API

The dynamic data flow runtime provides a simple API that allows instrumented programs to update and query flow values associated with memory ranges. All functions take a `property` argument to identify the property (lattice) that the call will be operating on; the remainder of the arguments are specific to the function. The functions are shown in Table 3.2.

Note that the runtime *does not* have any concept of the user-defined lattice, the *meet* function, or any other such niceties. These facilities are implemented with multiple calls to the runtime. For example, a *meet* typically results in two or more lookup queries, followed by the actual meet, followed by an insert call. This additional complexity is not a serious issue in practice. Unlike static data flow analysis, where lattice operations are used constantly to combine and merge information from multiple paths or contexts, a dynamic data flow analysis does not need to merge information. Therefore, the overwhelming majority of *meet*s in a static data flow analysis become simple `copy` operations in a dynamic data flow analysis. It is only when the lattice is used to represent hierarchical information that a dynamic *meet* is necessary, and this

Function	Description
`_insert(prop, base, size, val)`	Marks the address range starting at `base` of `size` bytes with the flow value `val`
`_lookup(prop, addr)`	Returns the flow value associated with the byte at address `addr`.
`_copy_flowval(prop, base, src, sz)`	Copies the flow values starting at address `src` for `sz` bytes to the corresponding region starting at `base`.
`_check_flowval(prop, addr, val)`	Checks if the flow value at `addr` is `val`.
`_error_handler(msg)`	Raises an error with message `msg`.

Table 3.2: Dynamic data flow analysis API calls. All functions take a property `prop` as the first argument and are prefixed by `ddfa`.

is handled by compiling down lattice operations during code transformation.

The `ddfa_error_handler` function calls the error handler with a message and indicates that a policy violation has occurred. We supply a default implementation that logs the violation and blocks the call; however, users can supply their own remediation call instead via the Broadway language's code replacement mechanism if different error handling is desired.

In addition, the API provides several convenience functions for handling C-style null-terminated strings, shown in Table 3.3. These convenience functions are thin wrappers around calls to the normal functions with a little extra code to derive size information from C strings. These string functions are meant to accompany calls to the C string functions that do not take length arguments; the functions that do take length/size arguments use the normal

Function	Description
`_insert_stringz(prop, str, val)`	Marks the address range of the null-terminated string starting at address `str` with flow value `val`
`_copy_stringz(prop, str, src)`	Copies flow values from source string `src` to destination string `str`.

Table 3.3: String-specific functions in the dynamic data flow runtime. These functions handle C-style null-terminated strings for which the length cannot be known statically.

API functions. For example, a `strcpy`'s actions on flow values can be updated by `ddfa_copy_stringz`. Note that these functions do preserve information even when the original C code overflows buffers. For example, when `strcpy` copies a large string into a string too small to contain it, `ddfa_copy_stringz` will ensure that the flow values of addresses past the end of the smaller string are correctly clobbered with the flow values of the larger string. This is essential for detecting C string bounds errors; otherwise, values that overrun buffers would be magically sanitized, which is highly undesirable.

One particularly common operation is translated down into a loop containing API calls. This operation is querying whether any element in a buffer contains a certain flow value. This must be translated as a loop that steps through the buffer byte by byte until it has found a byte that has the value or until it reaches the end of the buffer. The versions for fixed-size buffers and C strings differ only in their termination condition.

Finally, a few additional functions are present for initialization and memory performance statistics. The initialization call is a one-time call inserted at the beginning of the program, while the statistics calls return basic information on memory usage.

3.4 Code Instrumentation

To use the map to track flow values at runtime, the compiler instruments the original program with calls to functions that manage the map. This process is straightforward.

3.4.1 Elementary Transformations

Like most compilers, our system first transforms C to a simpler intermediate representation before performing analysis and transformations. At this level, the compiler only needs to consider assignments, basic operators, pointer dereferences, and function calls. Our transformation for inserting code is as follows:

- Constants are given the default flow value.
- Assignments transfer the flow value of the source to the target.
- Operators (such as arithmetic operators and array accesses) have the flow value of the *meet* of the operands. The *meet* operator in data flow analysis combines flow values based on their position in the lattice [57].
- Any address or pointer dereference that is used or assigned to acts on the corresponding entry in the map.
- In keeping with C's call-by-value semantics, function calls transfer flow values to the arguments in the function body, and function calls return any flow values through the return value.

Several examples of applications of these rules are shown in Table 3.4. In these cases, the instrumentation added is typically inserted after the original

Original Code	Instrumented Added
`x = constant;`	`ddfa_insert(prop, &x, sizeof(x), default);`
`x = y;`	`ddfa_copy_flowval(prop, &x, &y, sizeof(y));`

`x = a + b;`	`tmp1 = ddfa_lookup(prop, &a);`
	`tmp2 = ddfa_lookup(prop, &b);`
	`ddfa_insert(prop, &x, sizeof(x),`
	`meet(prop, tmp1, tmp2));`

`foo(arg);`	`ddfa_copy_flowval(prop, stackptr+off, &arg,`
	`sizeof(arg));`

`return b;`	`ddfa_copy_flowval(prop, ret_addr, &b, sizeof(b));`

Table 3.4: Example code transformation patterns for dynamic data flow analysis

line of code in question, except in the case of security checks, which must be inserted before.

Note that function call and return necessarily involve compiler-specific information. Because C functions are call-by-value and values are copied, the runtime must mark the addresses on the stack corresponding to the arguments with the appropriate flow values. That way, when the flow values of function arguments on the stack are queried inside the callee, the correct flow values will be associated with those addresses. These particulars may vary by compiler, architecture, and calling convention. However, in practice, this is almost never necessary after static analysis as virtually all policy annotations operate on the flow values of pointed-to objects rather than pointers themselves, so passing the flow values of the pointers or scalars is rarely needed.

These rules are analogous to the standard rules for applying data flow analysis [57] and remain the same for the wide variety of security problems that lattices naturally model [35]. When applied to taintedness, these rules are the same code insertion rules used by other compiler-based systems [99, 63]. However, our additional analysis and optimizations often allow us to remove considerable amounts of instrumentation. These rules track *explicit flows*, which are information flows that occur because of assignments or arithmetic operations. Like taint tracking systems, our system does not track implicit flows [92, 33, 80, 99, 84, 63]. Tracking implicit flows can lead to extremely high false positive rates, sometimes over 90% [60], rendering it impractical for deployed systems.

3.4.2 Policy-Specific Transformations

The basic transformations are sufficient for most operations on source code that simply move data around. However, most programs utilize library functions to manipulate data and any policy of interest will have policy checks on certain function calls. These actions and checks must also be added by code transformation.

Policy checks are perhaps the simplest to implement. The `error if` annotation contains a condition that is directly translated to the equivalent C code. References to named variables are directly translated to the appropriate lookup calls. For example, consider the `printf` function and its corresponding annotation for format string attacks shown in Figure 3.6. The corresponding code added to a `printf` call is shown in Figure 3.11

Because the Broadway annotation language does not provide a facility for denoting the size of null-terminated nature of arrays, our system uses a fixed set of rules to cover the Standard C Library. Since `printf` is known to take null-terminated format strings, a loop is inserted that checks every character in `format` that `printf` itself would read. The `could-be` check is translated to

```
char * tmp_fmt = format;
error_caught = 0;
while(*tmp_fmt) {
  if(ddfa_check_flowval(_PROP_TAINT, tmp_fmt, _TAINT_TAINTED)) {
    error_caught = 1;
    break;
  }
  tmp_fmt++;
}
if(! error_caught)
  printf(format);
else
  ddfa_error_handler("Error: tainted format string!");
```

Figure 3.11: Code added to guard against format string attacks.

use the `ddfa_check_flowval` call in the obvious manner. Finally, if an error is caught, an error handler is called; otherwise, the original `printf` call proceeds. For functions that use fixed-size buffers instead of C strings, the loop bound is the size of the buffer, but the remainder of the code is the same.

Our default error-handler logs the error and associated information for further analysis. If desired by the end user, this error handler can be replaced with remediation code by using the Broadway annotation language's code substitution annotation, allowing the user to substitute arbitrary C code at the callsite where a possible policy violation is found. The generated code remains the same except that the call to `ddfa_error_handler` is replaced by a block containing the user-supplied code.

Annotations for functions that manipulate flow values but contain no security checks are done similarly. In many of these cases, the string-handling convenience functions provided by the runtime library greatly simplify the code

```
dest = strdup(s);
ddfa_copy_stringz(_PROP_TAINT, dest, s);
```

Figure 3.12: Code added for `strdup` and similar string/buffer propagation operations

Broadway operator	Dynamic comparison
a is-exactly b	a==b
a is-atleast b	meet(a, b)==b
a is-atmost b	meet(a, b)==a
a could-be b	a==b

Table 3.5: Translation of Broadway lattice comparison operations to dynamic flow value comparisons

that must be inserted. For example, consider the annotation for `strdup` for format string attacks in Figure 3.5. The translation into code is quite simple and is shown in Figure 3.12.

Finally, we must note that the comparison operators for flow values are interpreted in a slightly different way in a dynamic setting than in a static setting. Table 3.5 illustrates how lattice element (flow value) comparisons are rendered in the instrumented code. In most cases, the translation is completely straightforward. Equality comparisons are translated to `ddfa_lookup` calls or their loop-based equivalents for buffers and strings. When the *meet* operator is required, we perform the *meet* operation by a simple series of if-else branches that test every combination of flow values in the lattice and store the result in a temporary.

Finally, Broadway includes a `could-be` operator that queries whether an object could have a certain flow value in cases where the flow value may not be definitively known or otherwise obscured by static approximations due to control flow merges. Since there are no approximations to flow values at runtime due to control flow merges, this annotation is equivalent to an equality test and is translated as such. In other words, the `could-be` operator was created to (statically) reason about possible ambiguity. However, because there is no ambiguity at runtime, it can be treated as a simple test.

3.5 Static Analysis

To avoid the cost of tracking all objects at runtime, our compiler statically performs an interprocedural data flow analysis that identifies program locations where policy violations might occur. Starting from these possible violations, a subsequent interprocedural analysis identifies statements in the program that affect the flow values—and therefore the policy decision—at these violations. Other statements do not require instrumentation because they cannot affect the relevant flow values and thus cannot affect policy enforcement decisions. This analysis is supported by a fast and precise pointer analysis, which is critical because a less precise pointer analysis would identify many more program locations as possibly violating the specified policy [50], leading to higher runtime overhead.

At a high level, the static analysis phase consists of two main steps. First, a static vulnerability analysis is performed, identifying possible violations of the security policy. This first phase uses the client-driven pointer analysis and error checker [50]. Second, the statements that can affect dynamic checks at these possible vulnerabilities are identified by a process we term *data flow slicing*.

We will now discuss these steps in detail in the following subsections.

3.5.1 Static Vulnerability Analysis

The first step is to statically check the program to identify all possible violations of the security policy as defined by the annotations [50]. If the compiler can prove that there are no such violations in the program, *no further analysis or code insertion is required.* However, in cases where the compiler identifies possible violations, additional analysis is needed to determine where instrumentation should be inserted.

To perform this first step, our system uses the Broadway static analysis system [47] to perform an iterative data flow analysis. At a high level, the following steps are performed:

- The policy specification file is read and the appropriate representations of the abstract properties and the analysis are instantiated within Broadway.
- The source code is read and lowered into a dismantled form.
- An initial flow- and context-insensitive pointer analysis is performed.
- The data flow analysis is performed to find possible violations.
- Based on the results of the data flow analysis, objects and contexts where loss of precision due to the pointer analysis may have affected results are selected for a higher-precision analysis. The pointer analysis is rerun with these objects and contexts analyzed flow- and context-sensitively.
- The data flow analysis is run again, producing a final list of possible violations.

Note that both the data flow analysis and pointer analysis are performed twice. This is a consequence of using the client-driven pointer analysis algorithm [50]. Although seemingly redundant, what the extra analysis phases give us is an analysis with a customized pointer precision policy, giving the static analysis the same effective precision as a very precise analysis but at only a fraction of the cost.

Our implementation utilizes Broadway's existing pointer analysis and data flow analysis framework largely as-is, with a few modifications that we will now discuss.

3.5.1.1 Iterative Data Flow Analysis

Broadway performs a standard iterative data flow analysis. The specific implementation details are discussed in prior work [47]. Our dynamic data flow analysis system does not modify Broadway's existing data flow analysis system except in the following two respects.

The first change allows us to perform a trace whenever a possible vulnerability is found. By default, Broadway only reports errors and performs no special actions. To hook into Broadway, we add a mechanism that allows the error checker to call a tracing function whenever a candidate error is encountered. This is implemented by adding an additional field pointing to an instance of the `Diagnostic` class to Broadway's enumerated properties (`enumProperty`). When the property analyzer finds a possible policy violation, it calls a `trace` function belonging to the `Diagnostic` instance associated with the property. This provides a convenient entry point for our data flow slicing algorithm as it is guaranteed to be invoked immediately on discovery of a possible policy violation.

The second change allows us to perform traces in terms of the definitions and uses of flow values rather than concrete values. By default, the available use-def information in Broadway is defined in terms of concrete variables, not flow values. However, our tracing algorithm, described in Section 3.5.2, is defined in terms of flow values. We modify Broadway's property analyzer to build use-def chains *in terms of flow values*. This is implemented by instrumenting the data flow analysis transfer functions. For example, when the transfer function evaluates the statement `a = b`, it assigns the flow value of `b` to `a`, merging values as necessary. Our extension also notes that there is a use of `b`'s flow value at this location and that it flows into `a`. This information is collected in a pair of complementary structures, `uses2defs` and `defs2uses`. Most importantly, these relations describe the flow of *flow values*, not concrete values. Thus, our trace bypasses cases where the concrete values change but the flow value does not, such as when a variable is incremented.

These extensions cause the data flow analysis to build use-def and def-use chains in terms of the data flow analysis, in addition to all other structures that it previously built. This collected information is used by the data flow slicing algorithm that is invoked on error detection.

3.5.1.2 Pointer Analysis

A significant obstacle to interprocedural program analysis is the use of pointers. To reason precisely about the flow of data, the compiler must know which objects a pointer could point to. The limited scalability of pointer analysis has stymied previous attempts to apply interprocedural analysis to dynamic taint tracking [63], so interprocedural analysis is not commonly used.

Our system uses a scalable and precise *client-driven pointer analysis* [50, 47]. The client-driven analysis is able to match the precision of a fully flow- and context-sensitive pointer analysis without requiring significantly more analysis time than a fast and imprecise flow- and context-insensitive analysis. Unlike most pointer analyses, the client-driven analysis cannot be used as a stand-alone pointer analysis. Instead, it requires a *client* that uses the results of the analysis, which in our system is the static data flow analysis that identifies possible policy violations.

At a high level, the client-driven pointer analysis operates as follows:

- A fast but imprecise (flow- and context-insensitive) pointer analysis is performed. Merge points where precision is lost are noted in a dependence graph.

- The client data flow analysis is performed. In our case, the analysis is the typestate analysis that determines if there are possible policy violations.

- The objects and statements involved in a possible violation are compared with the precision-monitoring dependence graph. This allows the compiler to determine which precision-losing actions may have affected the results of the data flow analysis. The variables whose flow-insensitivity and procedures whose context-insensitivity may have affected the data flow analysis are marked for increased precision.

- The pointer analysis is restarted with these marked objects and contexts analyzed flow- and context-sensitively, while the remainder of the variables and contexts are analyzed at the original precision.

- The data flow analysis is performed again using the results of the new mixed-precision pointer analysis.

By identifying locations where imprecision in the pointer analysis affects the precision of the client's results, the client-driven analysis is able to selectively increase precision for the pointer analysis in places where it will improve the results of the client analysis. Because the amount of extra precision is typically small [50], the client-driven analysis is able to avoid analyzing pointer relations that do not affect the client, dramatically improving scalability without sacrificing precision with respect to the client.

Our system is able to leverage a client-driven pointer analysis because our security policies are expressed in terms of a simple typestate data flow analysis. Similar extensible dynamic data flow analysis systems like GIFT [63] that define their policies procedurally lack a suitable data flow analysis client and therefore cannot use a client-driven analysis.

Finally, we note that the client-driven approach does not impact the soundness of the pointer analysis. Precise pointer analysis is an undecidable problem, so almost all pointer analyses, including Broadway's, compute a conservative over-approximation of the actual result. In particular, our pointer analysis is sound under the assumption that displacements between objects are undefined, a necessary assumption common to C pointer analyses [8].

3.5.2 Data Flow Slicing

The previous static vulnerability analysis identifies possible vulnerabilities by location and memory object. Our system must ensure that all the dynamic checks that are required to prevent possible vulnerabilities are performed correctly. We refer to the process of computing the statements that require instrumentation as *data flow slicing*, by analogy with program slicing [97].

The goal of data flow slicing is to determine the backwards slice of the program that can affect the data flow analysis at the point where slicing begins. Many concrete statements that would be included in a normal program slice can be excluded in a data flow slice. For example, incrementing a variable changes its concrete value but not its flow value. Thus, it must be included in a normal program slice, while it does not need to be included in a data flow slice as no flow values have changed. Similarly, conditionals are required in a normal program slice but not in a data flow slice because elementary conditions in the original program neither use nor affect any flow values even though subsequent statements on either arm of the conditional may contain statements that do affect flow values.

3.5.2.1 Data Flow Slicing: Definition

Abstractly, a data flow slice can be thought of as a program slice on an abstracted version of the program that includes only data flow operations. Such an abstracted program does not include assignments that do not affect flow values or any branch conditionals, while preserving the same control flow graph. A backwards slice through this "program," therefore, includes statements that update any flow values that may subsequently be used by the slicing start point, but not any control flow such as conditionals, which only use concrete values instead of flow values, nor any statements that affect flow values that do not contribute to the flow values relevant at the slicing start point.

More formally, we define a *data flow slice* with respect to some object o at some program location l to be the set S of statements and locations that affect a set O of objects, computed by a transitive closure as follows:

- l is in S and o is in O.
- If statement s' defines the flow value of some $v \in O$, then s' is in S.
- If statement $s \in S$ uses the flow value of some o', then o' is in O.

It is simple to see that the data flow slice S, by construction, captures any update to a flow value that could possibly affect the flow value of o at l. A statement can affect flow values only by defining them. If statement s affects the flow value of object o at location l, it is by definition in the data flow slice, and statements that affect o at l through intermediate assignments are also included because the data flow slice is a transitive closure. Our policies define security checks in terms of predicates on flow values at specific statements. Thus, the set of statements that can affect the typestate information used in the security check at a possible violation must be contained within the data flow slice.

This definition of data flow slices is similar to thin slices [89] except that producer statements, seeds, and direct uses are all defined in terms of flow values in the data flow analysis rather than concrete variables. The data flow slice can be thought of as a thin slice on an abstract program consisting only of operations on flow values.

3.5.2.2 Computing the Data Flow Slice

Our implementation leverages information computed during the static data flow analysis phase. Recall from Section 3.5.1.1 that whenever a data flow transfer function is executed, our system notes the flow of flow values from the right hand side to the left hand side. This information is used to aid our traversal.

When a possible policy violation is detected by Broadway, our tracing function is invoked. The tracing function begins computing the transitive closure of the data flow slice as follows:

- The error checker invokes `trace` with the location l involved in the possible violation and an initial set of objects O that the predicate at l uses.
- For each $o \in O$, our system find the corresponding use u of o's flow value.
- Next, our system looks up the corresponding def d that defines the flow value used by u by using the `uses2defs` map constructed during data flow analysis. The statement s where d is located is added to S.
- From d, which is the left-hand side of an assignment (either a true assignment or a ϕ-function in SSA form), we look up the corresponding right-hand side uses $\{u_0..u_n\}$ These are the uses of the flow values used in computing the flow value that d defines for s.
- For each of these uses, the process of looking up the corresponding def is repeated.

This is simply a standard traversal of the use-def chains constructed during static data flow analysis. The only difference is that the uses and defs are in terms of flow values, not concrete values.

3.5.2.3 Slice Truncation

Our data flow slicing implementation incorporates a small optimization that truncates slices when values are definitely known. Consider the example code in Figure 3.1. Starting from the possible vulnerability at the call to `printf`, we find the call to `memcpy` that copies `input` to `buf` as well as the call to `read_from_network` that taints `input`. Ordinarily, the runtime will mark `input` as tainted when `read_from_network` is called and will then copy `input`'s taintedness into `buf` at the call to `memcpy`. However, we can statically prove that at the call to `memcpy`, `input` is always tainted. There is no need to actually look up `input`'s taintedness during the `memcpy` as it is invariant. Therefore, there is no need to track the taintedness of `input` as it is never needed, and so we can eliminate any instrumentation on `read_from_network`. We may simply mark `buf` as tainted at the call to `memcpy` and still produce exactly the same behavior at `printf` as a fully-instrumented program.

We implement slice truncation by examining the cardinality of the possible flow values set for each object in question. Broadway, in addition to computing the flow value for a variable, also maintains a set of all *possible* flow values that variable could be at that point. For example, consider a lattice resembling the taxonomic hierarchy of animals, with the obvious meet function. Here, Vertebrates and Invertebrates would both be Animals. Mammals and Reptiles would both be Vertebrates, Cats and Dogs would both be Mammals, and so on. If Broadway must merge information about Cats and Dogs, their meet (and therefore the flow value) would be Mammal, but Broadway also notes that the possible values are Cats and Dogs, and not Koalas or Bears.

Because Broadway maintains this extra information during data flow analysis, we may easily determine if Broadway is absolutely certain of a flow value. If the size of the could-be set is one, then under all possible executions, the flow value computed must be the actual flow value. In other words, the dynamic flow value at that particular location is invariant. Because Broadway's data flow analysis is sound and conservative, we may safely truncate the trace at this point and simply insert a call that sets the flow value to the computed constant.

This optimization has an interesting side effect: At times, the tags of some addresses will not be up-to-date. For example, a buffer could be tainted, sanitized, tainted again, and then used. If the buffer is always tainted again, slice truncation would not include the original tainting and untainting. If the program were to query the taintedness of the buffer during the early stage, the runtime system may very well report that the buffer is in the default untainted state when it is in fact tainted. Although this behavior is curious, it cannot affect the result of any security checks as the static data flow analysis has already proven that there is no possible vulnerability involved during the initial taint/untaint phase. By the time execution reaches the possible violation, however, the taintedness of the buffer will have been updated to the correct tainted value, and the check will function correctly. Thus, our system is as secure as a system with full tracking, but avoids the performance penalty of keeping all tags up to date at all times.

3.6 Security Discussion

We now examine the security-related assumptions and advantages of our system.

3.6.1 Trusted Computing Base

As with other software taint tracking solutions, our system increases the size of the TCB, in our case adding the compiler to the TCB. Although there are security implications [93] to trusting the compiler, the additional trust required by our approach is mitigated by two factors. First, in typical modern environments, the compiler (usually `gcc` or some other widely used compiler) is *already trusted* to compile server programs. Second, our *source-to-source* translator relies on the user's already trusted compiler for generating binary code. The changes and modifications that our system makes to programs are thus transparent and human-readable, making it difficult to insert undetected malicious code. Thus, our system requires minimal additional trust beyond that which is already present in most deployed systems.

Like any system based on user-defined policies, the policies themselves are also a part of the trusted computing base. If the annotations that summarize the effects of external functions are incorrect or incomplete, the system may miss important data flow. Such an error is analogous to a bug or omission in a hardcoded taint tracking system. Fortunately, frequently-used external code resides in libraries like the C Standard Library that are relatively robust and whose semantics are well-understood, and we have found that providing accurate annotations for these functions is straightforward.

3.6.2 Attacks Detected

Our system is capable of detecting attacks that depend on the propagation of data through the system. More specifically, we can enforce any typestate policy, which includes traditional taint-based attacks as well as general information flow tracking [63]. These attacks include those that do not overwrite control data or violate data flow integrity and thus are problems even in safe languages.

In our evaluation, we enforce a taint-based policy that prevents format string attacks, similar to the format string policies used by existing taint tracking systems, such as TaintCheck [80], as well as interpreters with taint tracking modes [95, 81]. In addition, our system can enforce a policy that prevents attacker-controlled data leaks such as file disclosure vulnerabilities; this policy cannot be enforced precisely by an ordinary taint tracking system.

Our system only guarantees that violations of the specified policy do not occur. This situation is shared by all enforcement mechanisms—for example, a memory-safe database server can still be compromised by an SQL injection attack because such attacks do not violate memory safety. The soundness of our analysis prevents any attacks that violate the policy. However, if it is possible for the attacker to gain control through an attack that does not violate the policy, it may be possible to compromise the application.

3.6.3 Alternate Attack Channels

Like other taint tracking systems, we do not concern ourselves with *implicit flows*. Implicit flows occur when control flow influences the possible values of data. For example, information may be implicitly passed along branches of the form `if(x==0) y=1; else y=0;` which allows the user to influence the value of `y` by modifying the value of `x`. Taint tracking systems usually do not consider `y` tainted even if `x` is tainted. Although such cases result in implicit information flows that are theoretically exploitable, the majority of attacks depend on direct flow of data [33, 24], which our system does guard against.

Our system also does not defend against attacks that are not based on information flows in program code. For example, distributed denial of service attacks can harm systems without creating any individually anomalous information flows. Information can also be leaked via covert timing channels, which we also do not detect, although our requirement for source code limits the ability of malicious developers to introduce malicious code. Finally, our solution only defends against attacks, not arbitrary memory errors. A buggy program can still experience segmentation faults and other errors using only untainted data.

3.6.4 Defending the Enforcement Mechanism

The design of our system makes it difficult in practice for an attacker to subvert the enforcement mechanism itself. First, like other compiler-based systems [99, 63], the original program is written before the enforcement code is added, so the original program cannot directly access enforcement data. Moreover, unlike taint tracking systems that track taintedness using stack-allocated variables or fixed addresses [99], all of our structures are dynamically allocated on the heap and concealed behind function calls. Pointers to enforcement data never appear in application code, so the attacker cannot obtain a pointer to our enforcement data without sophisticated heap attacks. Thus, the attacker will not be able corrupt enforcement data without first hijacking the program by exploiting some vulnerability that the user's security policy does not guard against. Attacks that the user's policy do guard against are prevented.

For additional protection, our mechanism can be easily combined with various defenses against memory errors. For example, address space randomization [15] or heap randomization [11] can be used to defend our system against corruption attacks.

Chapter 4

Evaluating Dynamic Data Flow Analysis

In this chapter, we will discuss our evaluation of our dynamic data flow analysis system. Our evaluation is guided by a desire to answer several important questions:

- Can our system enforce policies comparable to existing taint tracking systems?
- Can our system generalize to problems that existing taint tracking systems cannot handle?
- Does our dynamic data flow analysis system successfully prevent real attacks?
- What is the performance overhead of our system for typical security-sensitive server programs?
- Because server programs are often I/O-bound, what is the overhead of our system on compute-intensive benchmarks where overheads and latency cannot be masked by I/O?
- How difficult is it to annotate libraries for analysis?

We answer these questions in the remainder of this chapter. We demonstrate that our system is comparable to existing taint tracking systems by enforcing a standard taint-based security policy that guards against format string attacks. We show that our system is more general by also applying it to file disclosure vulnerabilities, a problem that cannot be solved with taint alone. We evaluate our system on both server programs and compute-intensive benchmarks, and verify that all known attacks are prevented. Finally, we discuss qualitatively the difficulty of annotating library calls.

4.1 Taint Analysis for Server Programs

We first evaluate our system against current taint tracking systems. Does the static analysis reduce the overhead of performing taint tracking on server applications? Are the absolute overheads sufficiently low to make deployment practical? Do the overheads scale with the size of the programs?

4.1.1 Benchmark Program Selection

For our evaluation of dynamic data flow analysis for taint tracking on server programs, we apply our system to five commonly-used open source server programs: `pfingerd`, `muh`, `wu-ftpd`, `BIND`, and `apache`. These programs are, respectively, a finger daemon, an IRC proxy, an FTP server, a name server, and a web server. Several are widely deployed and typically run in privileged mode, so their robustness and integrity are critical. These programs, the versions we used, and their size in terms of lines of preprocessed code, are shown in Table 4.1.

These programs were selected in part because our static data flow analysis identified potential vulnerabilities in them. Our test programs were selected from a suite of open-source server programs that were previously used for static program checking research [50]. For nine other programs in this suite,

Program	Version	LOC
`pfingerd`	0.7.8	30K
`muh`	2.05c	25K
`wu-ftpd`	2.6.0	64K
`bind`	4.9.4	84K
`apache`	1.3.24	67K

Table 4.1: Programs used to evaluate taint tracking.

our compiler analysis determines that there are no improper uses of tainted data and therefore no instrumentation whatsoever is required. These programs include `BlackHole`, `privoxy`, `sqlite`, and `pureftpd`, and indeed there are no known applicable tainted-data attacks against our tested versions in the CVE database. For these nine programs, our system does not modify the program and therefore exhibits 0% runtime overhead and 0% code expansion. *Only a system that performs a static interprocedural taint analysis can achieve these overheads.* We have chosen to exclude these nine programs from our results and to instead focus on those programs that have possible vulnerabilities, but these results nevertheless highlight an important advantage of our approach.

We use our system to produce a modified version of each program that contains additional code to perform dynamic taint tracking. In our tables, we refer to this version as `DDFA`. The actual analysis time, while not negligible, is no worse than four minutes for `apache`, our second-largest benchmark with nearly 67K lines of code, and thus does not pose a serious obstacle to deployment.

4.1.2 Policy and Environment

We evaluate our system with a taint checking policy that prevents the use of tainted format strings in exploitable functions. This strict policy is similar to that enforced in the TaintCheck system [80].

Since our system is a source-to-source translator, we compile the enhanced C programs using `gcc-3.3` on Linux with whatever the default compiler options and optimization levels were that were supplied by the original developers of the benchmark programs. The programs are then run on a 2.4 GHz Pentium 4 with 1GB of RAM, running Linux 2.6.17. For each benchmark, we use the program's documentation and examples to run the program with a reasonable configuration.

Since all of these applications are server applications, we must minimize the effects of the network and the benchmarking software when measuring performance. Because the workload generator itself can impose a noticeable burden on the system, the workload generator, which sends requests to the server application, is run on a separate computer. Both computers are located on the same 100mbps network to minimize the effects of network latency, and all experiments were conducted during periods of low activity.

Because these measurements are inherently subject to some degree of unpredictability, we report results averaged over a large number of runs. For our experiments, each workload is run 100 times and the result reported is the arithmetic mean of the 100 trials.

4.1.2.1 pfingerd

The pfingerd daemon is a finger daemon, allowing users to request simple information about other users. Our workload generator simply “fingers” a user repeatedly. Our metric for performance is response time, as measured in milliseconds.

4.1.2.2 muh

The muh program is an IRC proxy. Users can use muh to stay logged in to an IRC channel even when they are away. Our sample workload consists of a scripted login, a series of message retrieval operations, and a logoff. The metric for performance is the total elapsed time to complete this short sequence of operations.

The format string vulnerability present in muh is due to how muh stores and presents messages received while the user is away. A malicious attacker can send a specially crafted string as a message to the victim user. If the victim is not currently logged in, muh stores the message verbatim on local storage. Later, when the user logs in and retrieves his messages, muh displays the stored messages without appropriately sanitizing format characters, resulting in a compromise of muh.

4.1.2.3 wu-ftpd

The wu-ftpd server is one of the more popular FTP servers but has a long history of bugs and security vulnerabilities. In fact, the first format string attack discovered was the one present in the version of wu-ftpd that we evaluated.

As `wu-ftpd` is a file server, our primary performance metric is download throughput. Our server is set up to serve several files with sizes uniformly distributed among 4KB, 8KB, 16KB, and 512KB. The contents of these files are completely random. The benchmarking program connects to the server and begins downloading these files, one at a time, randomly, and the rate in MB/s is measured after a minute of sustained downloading.

4.1.2.4 bind

The `bind` (also called `named` for "name daemon") nameserver is one of the most widely used DNS nameserver programs in the world.

Since DNS requests are small, our performance metric is response time, measured in milliseconds. Our workload generator produces a stream of DNS requests for domains for which the test nameserver is authoritative. The workload generator does not request other names, as that can cause `bind` to contact other nameservers (who can then contact other nameservers) in an attempt to fulfill the request; this behavior can significantly pollute the performance results by introducing significant dependencies on the latency of networks and nameservers beyond our control.

The particular format string attack in `bind` is somewhat harder to exploit. The attacker needs to control a nameserver that is authoritative for some domain. This poisoned nameserver then returns bogus IP addresses for its requests and reports its own hostname as something containing format specifiers. When `bind` performs a lookup on this domain, it notes the bad nameserver's hostname (format specifiers and all), notices that the IP address returned is invalid, and logs the fact that a bad address was returned using `syslog` with an unsanitized format string, which is the bad nameserver's name.

For our security evaluation, we set up a separate nameserver on a third machine controlling a bogus domain, and then modified the victim copy of `bind` to direct requests to the bad nameserver.

4.1.2.5 apache

The `apache` webserver is perhaps the most commonly used web server in the world. Configuring `apache` can be complex, as it comes with many modules, most of which are optional, and has numerous other modules that can be installed independently.

Our configuration of `apache` is configured minimally, with only what is necessary to serve static files. All other settings are left unaltered from the default configuration file.

The workload generator for `apache` is nearly identical with `wu-ftpd`, using the same sets of files and the same measurement techniques, but using HTTP instead of FTP. Performance is based on throughput, in MB/sec.

4.1.3 Policy Annotation Burden

We now briefly evaluate the burden of providing the policy annotations that are required by our system. Our annotations can be thought of as consisting of three types: (1) *pointer annotations*, which describe pointer relations; (2) *analysis annotations*, which define a data flow analysis; and (3) *policy annotations*, which use the results of the data flow analysis to enforce a policy.

Pointer annotations are common to all policies because they describe the pointer relations of the arguments of each function, specifying what the function accesses and modifies; this information is used by the pointer analysis. Once a library has been annotated—in our case the Standard C library—pointer annotations need not be rewritten unless the library interface changes. For the Standard C library, there are pointer annotations for 116 procedures, with a median size of 3 lines and an average size of 4.68 lines.

Analysis and policy annotations can differ for different security policies. For the format string policy, there are 44 annotations of these types, with a median size of 6 lines each and an average size of 5.75 lines each. However, the vast majority of these are essentially duplicates. For example, the annotations for each member of the `printf` family of functions are essentially identical. When these "copy-and-paste" duplicates are eliminated, the total number is only 21. For the file disclosure policy, there are 65 such annotations, with a median size of 7 lines each and an average size of 6.52 lines. Again, the majority of these are essentially duplicates. When these are accounted for, there are only 36 unique annotations.

To understand the difference between analysis and policy annotations, we now discuss several different use cases. In the simplest case, the desired policy exists and there is no need to touch any annotation file. In other cases, a security expert may wish to modify an existing policy, for example, by calling a sanitization function when a violation is detected. Here, only the policy annotations require changes to account for the sanitization code. Finally, in the most invasive case, a new data flow analysis must be defined, in which case new analysis and policy annotations must be written.

The annotations themselves are not difficult to write. Our annotation files only use seven major constructs, so the language is easy to understand. All of these constructs are shown in the example in Section 3.2.1, and the

Program	Version	Exploit Ref	Detected
`pfingerd`	0.7.8	NISR16122002B	Yes
`muh`	2.05c	CAN-2000-0857	Yes
`wu-ftpd`	2.6.0	CVE-2000-0573	Yes
`bind`	4.9.4	CVE-2001-0013	Yes

Table 4.2: Evaluation of our system's ability to detect actual attacks. All attacks are detected successfully.

annotations shown are representative of the kind that must be written. In any case, the information provided by the annotations is required by any policy-enforcing system; in our system such information is specified by annotation files rather than being embedded in the code.

Furthermore, we believe that it is incorrect to conceive of the policy file as an extra burden that existing taint tracking systems do not have. A taint tracking system *must* know how taintedness propagates and where the taint sources and sinks are. In most systems, this information is hardcoded in the system. In contrast, rather than hardcoding this information, we move this information to an external policy file. Moreover, our system allows us to add new policies, which themselves need not be based on taint, far more easily than any system where the policy and analysis problem are hardcoded.

Conclusion Policy annotation files are not overly burdensome to create. Their reusability means that end users do not necessarily have to write their own policies.

4.1.4 Security Evaluation

We next evaluate our system's ability to detect attacks. Four of our benchmark programs contain known vulnerabilities that are exploitable. For example, `pfingerd` improperly trusts hostnames, while `muh` does not properly check format strings when reading or writing log files. The SITE EXEC format string vulnerability in `wu-ftpd` is actually the first discovered format string vulnerability [30]. `BIND` improperly writes requests to `syslog` when an authoritative nameserver is malicious. Our particular configuration of `apache` (core only without optional modules) does not contain any known format string vulnerabilities; it is included because our static analysis was not able to completely eliminate that possibility.

To test whether our system correctly detects the use of tainted data, we send malicious input to the instrumented programs. Table 4.2 shows the vulnerable programs, shows the vulnerability in question, and indicates that in each case our system successfully detects these attacks. In each case, it detects that tainted data is about to be used improperly and identifies the potentially malicious data.

The case of `muh` deserves special attention. The vulnerability exists because `muh` writes logged messages verbatim to disk. Later, when a user requests log information, `muh` reads the message back from disk and prints it directly using `printf`. Thus, if the original message contained dangerous format specifiers, `muh` could be compromised when the message is printed back. If the policy is to trust local files, then this attack will go undetected, which can be a serious problem in servers that cache data on disk. Several taint tracking systems trust local files by default [80, 84] and therefore miss this vulnerability; their performance when applying our more aggressive policy is

Program	Original	DDFA	Code Overhead
`pfingerd`	49655	49655	0%
`muh`	59880	60488	1.01%
`wu-ftpd`	205487	207997	1.22%
`bind`	215669	219765	1.90%
`apache`	552114	554514	0.43%
Average Code Expansion			0.91%

Table 4.3: The static code expansion required for dynamic taint tracking, as measured by compiled binary size (bytes).

unknown but likely to be worse due to the greater presence of tainted data. Our system can enforce this stronger policy without fear of incurring significant additional overhead because our interprocedural analysis can frequently prove that most uses of local file data are safe.

Conclusion Our dynamic data flow analysis system can effectively prevent real attacks on server programs. Our system can handle problems comparable to those handled by existing taint tracking systems.

4.1.5 Code Expansion

Because our system adds instrumentation to the source program, it introduces some static code expansion over unmodified code. Significant expansion of code size can have a negative impact on instruction cache performance, storage requirements, and more. Does our system result in unacceptably bloated applications?

We measure this expansion by comparing the sizes of the original and modified binary executables, with the same compiler options in the default makefiles for unmodified versions of the program, which results in the system libraries being dynamically linked. This measurement is inclusive of the dynamic data flow analysis runtime library, which is compiled with each application instead of being dynamically linked.

We use the binary code size because it provides a more accurate measure of code overhead than source code size. This is because the binary code size includes the effects of standard compiler optimizations. Measuring code expansion in terms of lines of code or similar metrics can be very deceptive as not all lines are equal (both in terms of complexity and impact on performance) and many lines of the instrumented code are subject to standard compiler optimizations and can be eliminated.

From Table 4.3, we see that the average code expansion for our benchmarks is less than 1%. In several cases, the compiled binary size does not actually increase because the added code falls in the padding that `gcc` adds. To place our results in context, LIFT with hot path optimizations can *at least double* the size of the code due to the need to maintain separate "fast" and "check" copies [84], while compiler-based systems like GIFT [63] report 30-60% increases in binary size.

Conclusion Our dynamic data flow analysis system increases code size in server programs by a minimal amount.

4.1.6 Runtime Overhead

The tracking of data flows incurs a runtime cost. In prior software taint tracking systems, this cost can be quite significant. If an even more complex software dynamic data flow analysis system is to be practical, the overhead must be kept to an acceptable level.

Program	Original	DDFA	Runtime Overhead
`pfingerd`	3.07s	3.19s	3.78%
`muh`	11.23ms	11.23ms	0.0%
`wu-ftpd`	2.745MB/s	2.742MB/s	0.10%
`bind`	3.580ms	3.566ms	-0.38%
`apache`	6.048MB/s	6.062MB/s	-0.24%
Average Overhead			0.65%

Table 4.4: Runtime overhead for performing dynamic taint tracking on server programs. This table shows the response time or throughput overhead for our `DDFA` system running on a 100mbps ethernet network.

For our set of server programs, we measure this cost by measuring server response time or throughput, as appropriate for the particular program. For example, for file servers, the end-user experience is most impacted by the throughput during file downloads. For information-providing services like finger or DNS, the response time is most critical.

As shown in Table 4.4, our solution has an average overhead of 0.65%. In all instances, the overhead is lost within the noise. In fact, in three instances, average server performance actually improves by small amounts when we perform taint tracking. This improvement may be due to differences in memory layout induced by our runtime system and the resulting effect on cache performance. As a point of comparison, the previous fastest compiler-based and dynamically optimized systems report server application overhead of 3-7% [99] and 6% [84], respectively—one order of magnitude higher.

Conclusion Our dynamic data flow analysis system adds negligible overhead to common server programs.

Program	Code Expansion	Overhead
`gzip`	0.0%	51.35%
`vpr`	0.0%	0.44%
`mcf`	0.0%	-0.32%
`crafty`	0.36%	0.25%
Average	0.09%	12.93%

Table 4.5: Runtime overhead for performing dynamic taint tracking on compute-bound programs. These versions of the SPECint benchmarks were modified to introduce a format string vulnerability.

4.2 Taint Analysis for Compute-Bound Applications

Measuring performance overhead on server programs only paints an incomplete picture of the true costs of dynamic data flow analysis. In many cases, the overhead is partially or completely masked by latencies in the network and other I/O, even when efforts are taken to minimize latency. Thus, the cost of performing a dynamic analysis on an application that is more compute-bound than I/O-bound may well be significantly higher.

We evaluate our system's performance on compute-bound applications by applying the same format string policy to four SPECint 2000 benchmarks, with all inputs marked as tainted. However, we found that the applications tested did not contain format string vulnerabilities. As a result, our static error detection phase found no possible policy violations and thus did not add any additional code. Therefore, *the true overhead of our system is 0%* for these benchmarks.

However, we still wished to measure the overhead of our system. We added additional buffers, spurious flows, and unchecked format string uses until our static analysis could no longer eliminate the "vulnerability" completely.

The four benchmarks that we use were chosen because it was possible to inject realistic format string vulnerabilities into them, a task that proved challenging for the other SPECint benchmarks. To ensure that these injected vulnerabilities are realistic and representative of real vulnerabilities, we use the following guidelines in selecting the locations for the artificial vulnerabilities.

- We choose locations where actual `printf`/`scanf` calls are being made, ensuring that our injected vulnerability appears at a location where it might be possible.
- We preferentially choose calls that operate on character data, eliminating unrealistic vulnerabilities, such as the use of integers as format strings.
- Finally, we check that our injected vulnerability is not eliminated by our static analysis.

Table 4.5 presents our results with the standard SPEC workloads. In all of the benchmarks, we demonstrate significant performance improvements over current software-based systems. The average overhead of 12.9% improves upon the best previously reported averages of 75-260% [99, 84]. Furthermore, in most cases, our system's overhead for compute-bound applications is essentially zero even when the application does contain vulnerabilities. Thus, our approach is less adversely affected by CPU-intensive programs than all current software-based techniques.

The `gzip` benchmark is a worst case for taint tracking systems [92, 99, 84, 34] due to its complex behavior and sensitivity to memory bandwidth. It operates on character data extensively and propagates tainted data everywhere, reducing the flows that our system can statically eliminate and negatively impacting performance. Nevertheless, our system's overhead of 51%

Program	Code Expansion	Response time
`pfingerd`	0%	0%
`muh`	2.67%	2.13%
`bind`	0.10%	-1.38%
Average	0.92%	0.25%

Table 4.6: Servers augmented by our system to guard against file disclosure vulnerabilities exhibit negligible overhead and code expansion.

represents a significant improvement over prior software systems, with overheads of 106% for a compiler-based system [99] to over 600% for dynamic instrumentation [84], and our result compares favorably with the 31% overhead for the most recent hardware-based solution [34].

Conclusion Although our performance is negatively impacted when the overheads are not masked by I/O, our average overhead is significantly better than previous systems, with our worst overheads being comparable to the previous best. Moreover, these overheads are only incurred when we forced non-existent vulnerabilities into our system that our static analysis could not completely eliminate, so our true overhead for these benchmarks is 0%.

4.3 File Disclosure Attacks

In addition to taint tracking, we evaluate our system's ability to prevent file disclosure attacks, as discussed in Section 3.2.2. Table 4.6 shows our results. For `pfingerd`, our static analysis was able to determine that it contained no FTP-like behaviors and therefore no instrumentation was required. For `muh` and `bind`, our system was unable to rule out this possibility and therefore had to insert a small amount of additional code. However, the delay in response time was so small as to not be consistently measurable. We omit the `wu-ftpd` and `apache` benchmarks as the behavior described in the file disclosure policy is entirely normal behavior for these two programs; thus, it would be nonsensical to enforce such a policy on these two programs.

These results highlight the advantages of our system. First, in some cases all instrumentation can be eliminated, giving 0% overhead. Second, in the cases where some tracking is required, our analysis is able to keep the additional code to a minimum, imposing only a small or negligible overhead. Finally, this example shows that without rewriting the compiler or its static analysis, our system can be applied to complex problems that taint tracking cannot directly handle.

Conclusion Our system can handle problems more complex than those that existing taint tracking systems can handle. Moreover, more complex policies do not necessarily result in higher overhead because the static analysis can prune away safe regions.

4.4 Discussion

We will now briefly discuss the key elements and insights that make our dynamic data flow analysis so successful.

4.4.1 Synergistic Dynamic and Static Analyses

One of the keys to success of our system is the fact that data flow analysis can be performed both statically and dynamically with minimal changes. In many other domains, a dynamic analysis does not always have a corresponding static analysis, and if a suitable static analysis does exist, it is often sufficiently different that both must be implemented and accounted for separately. In contrast, our system requires one to specify only one analysis—the dynamic analysis—and the corresponding static analysis is automatically constructed and performed.

We believe this is a significant advantage and is inherent to data flow analysis. A static data flow analysis is essentially the same as a dynamic data flow analysis except that it is performed at compile time and extra operations, known as phi-functions [73], must be inserted to handle control flow merges, calling contexts, and other ambiguous places. The dynamic analysis is simply the static analysis performed on a running program, but with flow values being associated with memory addresses rather than abstract objects and with no need at all for phi-functions because control flow is never ambiguous at runtime.

In contrast, a similar system that does not use data flow analysis to specify policies cannot leverage our techniques for high performance. Our dynamic data flow analysis could be implemented using the GIFT system [63]. However, GIFT requires the user to specify the policy in terms of code transformations that manipulate tag values. Because of this, it can be difficult or impossible to derive the corresponding conservative static analysis that approximates the dynamic analysis specified by the GIFT code transformations. As a result, their overheads are dramatically higher than ours. Thus, we believe we have demonstrated the value of specifying policies in terms of data flow analysis: it allows for both dynamic enforcement and static checking, and this particular combination realizes enormous performance improvements.

4.4.2 Data Flow and Sparsity

Another key factor to the success of our system is the fact that the scope of possible vulnerabilities is only a small fraction of the program. This observation has been made before in the literature [79] and our results confirm it. That is, for any given vulnerability, the chain of data flow events from the source of the malicious input to the location of the possible vulnerability is typically short and only involves a very small fraction of the program.

The sparsity of the source-sink chains also explains why our overhead is not necessarily related to the complexity of the policy. The file disclosure vulnerability involves two separate typestate problems and in the naive approach should incur twice the overhead of a taint-only system. However, our results show that the overhead for this more complex policy is actually *lower* than for the simpler taint-based policy. This is because a more complex policy does not always result in more statements being instrumented. In fact, it may result in fewer statements being instrumented because there are fewer vulnerabilities or fewer statements involved in vulnerabilities. In either case, our static analysis frees security professionals from considering the possible performance impact of additional policy complexity because we ensure that only the statements that can affect the security state *at a possible vulnerability* are instrumented. Thus, complex policies can be enforced without fear of introducing unnecessary overhead because the static analysis conservatively ensures that any instrumentation present is actually necessary.

4.5 Future Directions

We may broadly categorize future directions for this work in two categories: those that are amenable to modeling by typestate problems, and those that cannot be modeled by typestate but nevertheless may benefit from our general combined dynamic/static approach.

4.5.1 Extensions of Dynamic Data Flow

A clear extension of our work is to progress from problems that can be modeled precisely by typestate to problems that can be modeled *conservatively* by typestate. For example, consider SQL Injection Attacks (SQL-IA) [19, 69, 52]. Every legitimate query string will necessarily contain tainted user input or the query will not be very useful. Thus, our system will mark *every* database query as a potential vulnerability. However, SQL-IA requires that the specific tainted inputs match up with SQL keywords in the parse tree. Because we track data propagation on a byte granularity, we can identify the specific characters in the query string that are tainted and check to make sure that no SQL keywords are tainted. This policy cannot be modeled precisely with typestate because it requires knowledge of SQL syntax, but our mechanism can accommodate it by conservatively flagging all SQL queries as potential vulnerabilities and calling user code that checks the keywords within the query string. Our static analysis can still eliminate innocuous uses of user input that do not form part of the query string while also ensuring that any potentially problematic queries have any additional guards and checks inserted.

Another possible application is to use our tool for what we call *analysis-based* aspect-oriented programming. Like aspect-oriented programming [59], analysis-based aspects allow programmers to add cross-cutting functionality to existing code. Unlike conventional aspect-oriented programming systems, which implement aspects by defining simple syntactic code transformations [58], analysis-based aspects are implemented by defining a dynamic analysis. The weaver then generates the code required to implement the analysis. This extends our current system, which inserts *special instrumentation* to guard against *vulnerabilities*, to one that inserts extra *user code* to provide new *functionality*.

Analysis-based aspects have two advantages over conventional aspects. First, analysis-based aspects allow for more concise and easier-to-reason-about code. Many aspects that one would like to implement, such as access controls, information flow tracking, and privacy protection all pervasively cross-cut the entire program, share numerous common elements and mechanisms, and can be easily defined in terms of a dynamic analysis. By specifying these aspects directly as analyses, the programmer can focus on problem-specific concerns and avoid tedious reimplementation of common mechanisms. Second, by specifying the aspect as an analysis, the weaver can perform a conservative static analysis that determines where additional code is necessary, allowing the weaver to optimize away aspect code in a similar manner to how our dynamic data flow analysis system optimizes away unnecessary tracking.

Analysis-based aspects have clear applications in problems relating to information provenance, including privacy and information flow problems, because these problems are easily cast in terms of a typestate analysis. It is less clear if analysis-based aspects can be generalized to other problems. In the degenerate case, the “analysis” could be a null analysis and all join points are considered possible “vulnerabilities,” resulting in a system that is essentially a conventional aspect-oriented programming system with a declarative language for join points.

4.5.2 Combined Dynamic/Static Analysis

More generally, our philosophy is applicable to any dynamic analysis for which there is a conservative static counterpart. The conservative static analysis can identify which parts of the program are *not* involved in any policy violation, with the remainder of the program being instrumented as usual. For example, locks or shared variables [37] could be modeled conservatively, with deadlock-detection mechanisms applied only to the locks that could not be statically proven safe. Memory leak detection can be made more efficient by eliminating tracking on objects that provably do not leak. A profitable direction for future work would be to examine existing static analyses, finding dynamic analyses that attempt to enforce or study the same thing, and then integrating the two. Since there are many static and dynamic analyses, many fruitful combinations are possible.

Further gains may be found if the requirement for conservative over-approximation is relaxed. Coverity has found that in real-world software development, it is often beneficial or even necessary to sacrifice correctness and completeness in favor of reducing the false positive rate and improving error prioritization [12]; it does not matter that a tool conservatively finds all bugs if the false positive rate is so high that programmers ignore the reports. Similarly, it may be beneficial to use more relaxed static analyses that do not guarantee correctness for the corresponding dynamic analysis if tremendous improvements in overhead are possible. While such a policy is unacceptable for security applications, it may be quite practical for profiling, forensics, and debugging as long as the overall quality of the dynamic results is still high.

There will always exist dynamic analyses that do not benefit from our approach. In order to be effective, the static analysis must be able to reduce the scope of the dynamic analysis to be only a small fraction of the program. This is not always possible. For example, enforcing an information flow policy with implicit flows will be difficult with our approach because the false positive rate is too high [60] for the static analysis to be useful. Relaxing the policy can cause the system to miss real information leaks, which may not always be acceptable.

Chapter 5

Targeted Testing

In this chapter, we will describe the design and implementation of Bullseye, our system for targeted testing. Existing software testing systems are typically designed to thoroughly test programs and thus treat all parts of the program essentially equally. If instead the goal is to find bugs quickly, these systems are poorly designed as they spend significant time verifying correct program behavior rather than seeking out incorrect behavior. Our *targeted testing* approach steers execution to parts of the program that may be more likely to reveal bugs, thereby improving bug-finding speed by avoiding paths that serve only to verify correct program behavior.

Bullseye, our implementation of targeted testing, improves upon existing directed testing systems by using a static analysis to compute branch data with respect to a user-supplied set of interesting points and using this statically-computed data to guide execution towards program points that affect or are affected by the interesting points. By doing so, Bullseye can find faults in programs faster than existing directed testing systems and even finds bugs that directed testing could not because of state space explosion.

The Bullseye system also incorporates a new technique for performing automated boundary condition testing, enabling it to find faults that symbolic execution fundamentally cannot find unassisted. Our automated boundary condition testing solution is applicable to both targeted testing as well as

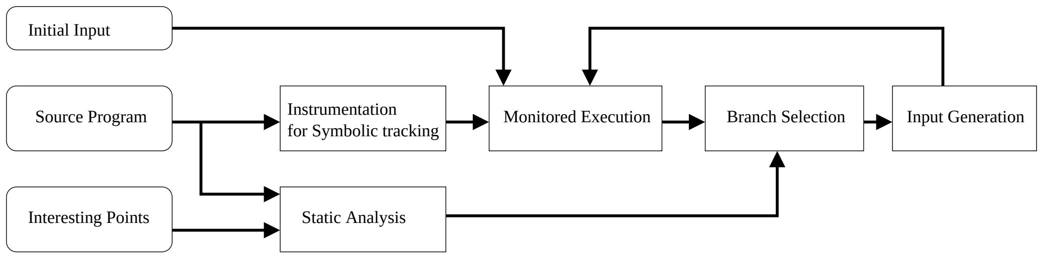

Figure 5.1: The design of our targeted testing system. The base system is a directed/concolic testing system where concrete executions are combined with symbolic evaluation to generate inputs. Bullseye-specific extensions are shown in the lower row of blocks, where we use static analysis to intelligently guide branch selection.

existing directed testing systems.

5.1 System Overview

The overall structure of our system is shown in Figure 5.1. At a high level, a programmer or tester would use our system as follows:

- Bullseye is given a list of statements and objects of interest in the program. The list can be derived from change logs, static error checkers, programmer hunches, or business priorities.
- Bullseye performs a static analysis to calculate the relevance and impact of other objects, locations, and branch conditions to the items of interest. This information is recorded in a separate file.
- Bullseye generates an initial input, or the tester provides one.
- The program under test is run with the input, and its behavior is monitored.

- Using information from the run and guided by the data computed by the static analysis, Bullseye generates a new candidate input that highly impacts or is highly impacted by items of interest.
- Input generation and execution is repeated until the tester is satisfied.

This workflow is similar to that of existing directed testing systems except for three steps. The key differences between Bullseye and directed testing are:

- Bullseye receives as input a set of program-specific interesting points, which are used to target testing. Directed testing does not use interesting points or any other program-specific method of prioritization.
- Bullseye performs a static analysis that computes the relative impact of various program points on the interesting points. Directed testing does not use static analysis.
- Bullseye uses the statically computed information to guide testing. While extensions to directed testing have explored simple hardcoded heuristics, none of them use statically computed program-specific information.

Bullseye also adds a fourth innovation that can be applied to both conventional directed testing systems as well as targeted testing. Full path coverage alone is insufficient for finding certain kinds of bugs. To address this problem, we modify our path-based testing system to perform boundary condition testing. Because the boundary conditions that expose bugs are typically not explicitly represented in the paths, we have developed a method for encoding conditions for automatic boundary condition testing directly in the control flow graph of the program, allowing any system based on symbolic execution to perform boundary condition testing. Our technique enables symbolic execution to find faults that previously could not be found because the conditions required to reveal them were not present in the program's paths.

Our discussion of the specifics of Bullseye will follow in three main parts. First, in Section 5.2, we will discuss the static analysis behind Bullseye, showing the process by which interesting points are used to generate the branch data file. Next, in Section 5.3, we will discuss our branch boundary condition transformation and how it may be used in conjunction with directed or targeted testing. Finally, in Section 5.4, we will discuss the implementation of our targeted testing engine itself.

5.2 Static Analysis for Targeted Testing

The Bullseye compiler takes the set of interesting points and performs data and control flow analysis to compute various metrics that allow the branch selector to focus testing effort on the interesting points. These metrics are associated with branches, and they are stored in the branch data file. In this section, we will show the process by which interesting points are used to compute the branch data file.

At a high level, the static analysis proceeds as follows:

- Bullseye reads in a set of *interesting points*, which are supplied via in-line annotations.
- Bullseye performs a standard dependence analysis on the program, computing def-use and use-def chains for every variable in the program.

- Bullseye uses the def-use and use-def chains to compute a set of *interesting locations* starting from the interesting points. The interesting locations capture the forward and backward data flows from the interesting points.

- For each interesting location, Bullseye identifies the *controlling conditionals* for that location, which are the conditionals that the interesting locations are control-dependent upon. This phase determines the branch weights, preferred branch directions, and control flow distances for each of the controlling conditionals.

- All branch data is collected and written to the final branch data file.

5.2.1 Encoding Interesting Points

Interesting points are how the user communicates priorities to a targeted testing system. Intuitively, an interesting point indicates that some particular statement, object, or operation is important to the testing goal at hand, and that test inputs should stress things that affect or are affected by the interesting point.

The Bullseye system maintains a generic notion of interesting points and is agnostic to their source. In our evaluation, we use program differences as our source of interesting points, as if Bullseye were part of a change impact management system. However, Bullseye can use many other sources for interesting points. For example, a static error checker can produce a list of possible bugs, or a programmer can manually designate certain "tricky" parts of code as interesting.

We now discuss how interesting points are communicated from the user to the Bullseye system. We support three different types of interesting points:

- *Interesting variables* indicate that some variable v is interesting at some location l.
- *Interesting statements* indicate that a specific statement s is interesting. The variables used or defined at that statement are not necessarily interesting, only the statement itself. Note that interesting statements can be considered a special case of interesting variables if a dummy variable is used.
- *Interesting operations* indicate that a specific comparison operator at some statement s is interesting.

These interesting points are provided to Bullseye in the form of inline annotations in the program source code, as shown in Table 5.1. These annotations have the syntax of function calls that wrap around variables or statements.

Textual appearances of a variable can be marked as interesting with the `check_interesting` annotation, which specifies that the object pointed to by `v` is an interesting variable. By using addresses instead of text variable names, the annotator does not have to worry about aliasing or namespace problems and names are not required; if the interesting object is a buffer, any pointer to the buffer will suffice, or if it is a scalar, the address of that scalar will suffice.

We treat interesting statements as a special case of interesting variables. In this case, the interesting "variable" is a dummy variable that has no uses or defs. Since nothing is ever done with the bogus variable, all that remains

Point type	Annotation
Variable	`check_interesting(v)`
Statement	`check_interesting(dummy)`
Operation	`interesting_op(v1, opcode, v2)`

Table 5.1: Annotations for the various types of interesting points.

that is interesting is the physical location. Because we use C function call syntax, we do not actually mark a specific statement as interesting; rather, we mark the space between statements (immediately before or after the interesting statement) as interesting.

Interesting operators require a little more annotation. The user must replace the operator with a synthetic function call. For example, if the user wants to mark the greater-than operator in `x > y` as interesting, the corresponding annotation is `interesting_op(x, __OP_GT, y)`. The Bullseye compiler will recognize this and replace it with the original code after it has noted the location and operator.

5.2.2 Dependence Analysis

Given a program that has been annotated with interesting points, the Bullseye compiler first performs a flow-sensitive interprocedural pointer analysis. The resulting aliasing information is then used to construct interprocedural def-use and use-def chains for all memory objects in the program. We use the existing mechanisms in the Broadway pointer analysis [47] to perform our dependence analysis.

Bullseye can be used with context-insensitive or context-sensitive pointer analyses. Our default is to enable context-sensitivity, but this option may be overridden by the user at analysis time. Again, our implementation simply uses the existing analyses in Broadway. Later, in Section 6.6, we will measure the impact of static context-sensitivity on the quality of the data it produces and the impact on fault-finding speed.

5.2.3 Interesting Locations

It is not sufficient to ensure that execution reaches an interesting point. What if the interesting point incorrectly modifies a variable that is used much later in the program? We would like to ensure that the subsequent use of this variable is reached. Similarly, if an earlier statement modifies a variable that is used at the interesting point, we would also like to ensure that it is tested. Previous test systems have already developed methods to guide execution to specific locations [62], but they do not address the issues of the effects of these locations on subsequent statements that can be far-removed. In order to properly reason about the effects of a statement, *data flow analysis* must be used.

To capture these causes and effects, we generalize the notion of interesting points to *interesting locations.* In this thesis, we use a data flow based definition of interesting locations. Intuitively, the interesting locations are simply the locations in the forwards and backwards thin slice [89] from each interesting point. In other words, the interesting locations are the locations that are on a chain of uses and defs (or defs and uses) that lead towards or away from an interesting point. The set of interesting locations captures data dependencies and is significantly smaller than a traditional executable program slice that includes all control dependencies, while still containing the program statements most useful to debugging [89].

The interesting locations are computed differently depending on their type because not all types of interesting points have data dependencies. We compute them as follows.

For interesting variables, the computation of the resulting interesting points is straightforward. We use the interprocedural def-use and use-def chains computed earlier to find the forward and backward data dependencies. Our implementation first performs a backwards traversal along the use-def chains starting from the use at the original interesting point. From there, we proceed to the corresponding def, and then from the def to any right hand side uses. Every location that is traversed is added to the set of interesting locations. After the backwards traversal is complete, a similar forwards traversal is performed along the def-use chains, treating the original interesting point as a def of the variable and then proceeding to any subsequent uses and their left hand side defs. The backwards trace is the same procedure used in computing a thin slice [89], while the forwards trace is the forwards analogue of the backwards thin slice.

For interesting statements, no such traversal is possible or necessary because there are no data dependencies. Thus, the interesting locations for an interesting statement is a set containing one element, the interesting statement itself.

Finally, interesting operators are treated in the same manner as interesting statements. An interesting operator means that the operator itself, not the operands, is what is interesting. Of course, there is nothing that prohibits the user from annotating an interesting operator that takes an interesting variable.

The interesting locations are computed separately for each interesting point in the program. When all of the interesting locations have been identified, they are combined into a single large set of interesting locations.

5.2.3.1 Data Flow Distances

We compute an additional metric while computing the interesting locations: the data flow distance. Intuitively, the *data flow distance* is the number of hops along a def-use or use-def chain required to reach some interesting location from the original interesting point. To compute the data flow distance, we modify the use-def and def-use traversal as follows:

- The original interesting variable v at location l has a data flow distance of 0.
- The def of v at l has a data flow distance of 1, as does the uses of the def of v at l.
- During the backwards traversal, if a use of some object on the right hand side of a statement has a data flow distance of d, the object on the left hand side has a data flow distance of $d + 1$.
- During the forward traversal, if a def of some object on the left hand side of a statement has a data flow distance of d, all objects on the right hand side have a data flow distance of $d + 1$.

For the interesting locations that are interesting statements or interesting operations, no data flow analysis is performed, so the default value of 0 is used as the data flow distance.

In some cases, it may be necessary to merge information while computing data flow distances. A merge can be required for two reasons: context-sensitivity and multiple interesting points. If a context-sensitive analysis is performed, Bullseye may compute different data flow distances for a location depending on the calling context of the containing function. Before this data can be written out to the branch data file, all the various contexts must be merged. Similarly, if there are multiple interesting points in the program, a location may have different data flow distances depending on which interesting point it was computed with respect to. In both cases, we handle merges by always selecting the smallest of all the candidate values.

5.2.4 Control Dependence Analysis

In order to guide execution to certain locations, we must assign weights to the branches in the program. To do this, we must identify the branches that "control" whether or not interesting locations are reached. The concept of *control dependence* is useful here [41], because if a node B is control-dependent on node A, A determines whether B is reached.

5.2.4.1 Controlling Conditionals: Definition

In the standard definition of control dependence [41], a node B is control-dependent on node A if for all paths P between A and B and all statements $s \in p$ on some path $p \in P$, B postdominates s but B does not postdominate A. In other words, A has an outbound edge that heads "towards" B and guarantees that B will be executed (modulo non-termination). However, A also has another outbound edge that does not necessarily guarantee that execution eventually arrives at B. Thus, A "controls" whether B is reached.

For a given interesting location l, we define the *direct controlling conditional* of l to be the conditional node c that l is control-dependent upon. The *preferred direction* of c is the edge $e : c \to s$ such that l postdominates s; intuitively, the preferred direction is the edge that heads "towards" l.

We extend the direct controlling conditionals to *controlling conditionals* by completing the transitive closure. The controlling conditionals for an interesting location l is a set consisting of l's direct controlling conditional c, c's controlling conditional c', c''s controlling conditional c'', and so on.

We compute controlling conditionals for an interesting location l by first finding the direct controlling conditional c by testing the basic blocks in the function that l is located in for control dependence. We then transitively compute the controlling conditionals by computing c's controlling conditional and so on. Our actual implementation uses a worklist to ensure that each new "generation" is evaluated only after all of its parents have been evaluated.

5.2.4.2 Interprocedural Controlling Conditionals

By adding one step, our definition of controlling conditionals can be easily extended to cover interprocedural controlling conditionals. If the location l is not in the `main` function, we compute the controlling conditionals of the callsites of the function that l is in. Thus, if location l is in a function `foo` which is called from line 32 of function `bar`, we first compute the controlling conditionals of l within `foo`. After that, we compute the controlling conditionals of line 32 of `bar`, `foo`'s callsite. If `foo` has any callers, we find the controlling conditionals of `foo`'s callsites after computing the controlling conditionals within `foo`.

In practice, interprocedural controlling conditionals are required for the concept to be useful in testing. If a location inside a function is interesting, we must ensure that the function itself is called, which requires computing controlling conditionals in the function's callers, and any of their callers in turn, as required. Without computing controlling conditionals interprocedurally, we would not be able to provide any guidance until execution stumbles upon the right function.

5.2.4.3 Control Flow Distances

In addition to finding the controlling conditionals themselves, we compute the control flow distance for each controlling conditional. Intuitively, the *control flow distance* is the number of other controlling conditionals you must pass through from the original interesting location to your current controlling conditional.

Control flow distances are computed by tracking an additional distance measure during the computation of the controlling conditionals. For an interesting location l, the computation of l's controlling conditionals is modified as follows:

- The location l itself has a control flow distance of 0 for the purposes of the analysis.
- The direct controlling conditionals of l have a control flow distance of 1.
- A branch that is the direct controlling conditional of another branch that has a control flow distance of d will in turn have a control flow distance of $d + 1$.

As with data flow distances, context-sensitivity or multiple interesting points may require data to be merged. As before, we resolve any such ambiguities by simply selecting the smallest value. All such merging is performed at the end, so the computations of individual interesting points are performed without any merges.

5.2.5 Branch Data

Once the controlling conditionals have been identified, the final branch weights are computed, along with a few other miscellaneous pieces of information. All the data gathered is then written out to the final branch data file.

5.2.5.1 Branch Weights

Having computed the interesting locations and their controlling conditionals, Bullseye may now compute the most important part of the branch data file: the branch weights. The branch weights consist of two numbers, which we call *trueness* and *falseness*. They indicate the weight on the true edge and false edge of the branch, respectively. The branch *direction* is simply the larger of the two and indicates which direction has higher weight.

The trueness and falseness of a branch is computed alongside the controlling conditionals. When Bullseye determines that c is a controlling conditional, it casts one "vote" for the preferred direction of c, incrementing one of either the trueness or falseness. Each interesting location thus grants exactly one vote to each of its controlling conditionals, which are allocated according to each controlling conditional's preferred direction.

A direction often receives several votes. For example, if a sequence of assignments and computations performed in the same basic block are all interesting locations, the controlling conditionals of that basic block will receive multiple votes on each of the preferred edges. Moreover, both the taken and not-taken edges of a branch can receive votes. For example, if both arms of a branch modify a variable that is used by a later interesting variable, then both arms of the branch will contain interesting locations. Thus, the branch will be traversed at least twice, corresponding to the two or more interesting locations, but with a different preferred edge each time.

Our method for computing branch weights has two useful properties. First, all branches that have branch weights are controlling conditionals for some interesting location. This is because votes for weights are assigned only during the computation of controlling conditionals *to* controlling conditionals. This means that any branch that lacks branch data was never a controlling conditional at any point and therefore does not directly influence the reachability of interesting locations. Second, the weight on each edge of a branch corresponds exactly to the number of interesting locations for which that edge is a preferred edge. This gives us a simple guide for the direction that would lead to the most interesting locations.

5.2.5.2 Miscellaneous Data

At this point, we have computed for each branch that is a controlling conditional of an interesting location the data flow distance, control flow distance, and branch weights. The final branch data file contains two more pieces of information.

First, we include a branch classification identifier. During the computation of control flow and data flow distances, we also track the type of interesting point that the metrics are being computed for. The branch classification identifier allows the dynamic system to determine whether the branch data for a given branch was computed for an interesting variable or an interesting operation. In the event that the same branch contains data computed for both an interesting operation and an interesting variable, we use the data associated with the interesting operation. This is consistent with our policy of always choosing the smaller metric; an interesting operation is an original interesting point and always has a data and control flow distance of 0, which is smaller than any control or data flow distance computed from any interesting variable. The branch classification identifier is encoded as an integer constant in the branch data file.

Second, we include an edge classification for each edge in the branch data file. The edge classification is computed using the standard depth-first spanning tree algorithm [73] and classifies edges as tree edges, cross edges, forward edges, and back edges. A classification for both the taken and not-taken edges of the branch are provided. As before, these are encoded in the file as integer constants. Note that edge classification is not provided for every edge or branch in the program, but only for those where other branch metrics are also computed, i.e., the controlling conditionals of the interesting locations.

5.2.5.3 Assembling the Branch Data File

The static analysis now has all of the information it needs to generate the branch data file. The file is a plain text file with one branch per line. The information is as follows:

- Branch identifier. A unique identifier associated with the branch and used internally by the runtime to distinguish between different branches.

Table 5.3 shows that evaluating a branch requires a branch ID argument. The branch identifier is the link between the branches during symbolic execution and the branch data file. The branch ID is encoded as an arbitrary whitespace-terminated string. Our implementation simply uses the textual representation of the address of the object representing the branch in the AST as the branch ID as it is guaranteed to be unique for each branch in the program with no possibility of hashing collisions. One side effect is that branch data cannot be reused if the program is recompiled by Bullseye, as the branch IDs will no longer be comparable.

- Branch classification. Indicates whether the branch information was computed for an interesting variable or an interesting operation. The classification is also encoded as an integer.

- Data flow distance. The data flow distance for a controlling conditional c that was computed starting from an interesting location l is the data flow distance of l.

- Control flow distance. We record the control flow distance of the branch as an integer.

- Branch weights. The weights are recorded as a pair of non-negative integers representing the trueness and falseness of the branch.

- Edge classification. The edge classification records whether the true or false edges are forward edges, cross edges, or back edges. The edge classification is encoded as a pair of integers representing the classification of the true and false edges.

The file by default has the same name as the output for the instrumented source code with `.weights` appended. The name can be changed by the user.

5.3 Branch Boundary Condition Transformation

This section discusses the path inadequacy problem and our solution, a technique for performing automated boundary condition testing within a testing system driven by symbolic execution.

5.3.1 Motivation: The Path Inadequacy Problem

During development and testing, we noticed a curious phenomenon where Bullseye would generate an input for every path in a small program and yet fail to kill that mutant. Upon further investigation, we found that it is quite possible to produce programs where every path can be explored without revealing any bugs. We term this the *path inadequacy problem* because all-paths coverage is inadequate for revealing all bugs.

To illustrate the path inadequacy problem, consider a simple function called `IsGreaterThanTen` with the obvious implementation. Suppose this function has a bug where instead of performing a greater-than comparison, a greater-than-or-equal comparison is used. There are exactly two paths through this program. Both paths can be covered with the inputs 5 and 15. However, the bug is not revealed because the input 10 is never tested. Thus, it is possible to test every path in a program and still not reveal a simple bug.

Although the path inadequacy problem in general is unsolvable because it reduces to the halting problem, many of its common manifesta-

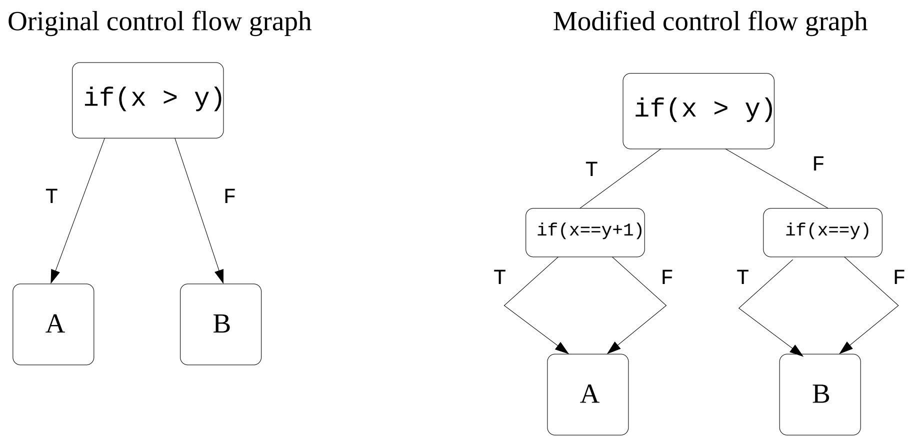

Figure 5.2: Transforming the control flow graph to include boundary conditions.

tions can be addressed by probing the correct boundary conditions. In the `IsGreaterThanTen` example, the bug can be found by probing the boundary condition on the branch that tests the input against ten. Many similar errors, such as bounds checks, off-by-one errors, and confusion between strict and nonstrict inequality can also be found by simply probing the boundary conditions.

We therefore address the path inadequacy problem by adding support for boundary condition testing. Normally, systems based on symbolic execution cannot perform boundary condition testing because the boundary conditions are not apparent from the paths. We address the problem by encoding the boundary conditions to be tested as additional paths in the program. Symbolic execution can then proceed as before, exploring the boundary conditions by exploring the paths that encode them.

5.3.2 Explicitly Representing Boundary Conditions as Paths

Rather than require changes to the constraint solver to support boundary conditions, our solution encodes the conditions into the tested program's control flow graph. Wherever there is a branch with a boundary condition to test, the compiler adds additional branches that test for the appropriate boundary condition on both sides of the original branch. By explicitly testing for the boundary conditions in the transformed program, symbolic execution follows the paths representing those conditions, thereby testing boundary cases.

Intuitively, the transformation is simple. Given a branch b, we make modifications to the true and false sides of the branch. On the true side, we insert an additional test that probes the value that makes the condition at b barely true. Similarly, on the false side, we probe the condition that makes it barely false. None of these branches should do anything in their consequents lest they change the behavior of the program, and everything should wind up back at whatever real code would have been executed in the original program. These extra tests simply add extra tests and edges to the control flow graph.

We now describe the transformation more formally. Suppose we have a branch b with comparison condition $c(x, y)$ that leads to target A if the branch condition evaluates true and B if it evaluates false. On the edge from b to A, which is the true edge, the compiler inserts a test between x and y for the case where $c(x, y)$ *barely* holds. For example, if c were the greater-than operator and x and y were integers, then the comparison to insert on this edge would be $x == y + 1$, because x being one greater than y is the smallest value for x that is still greater than y. Similarly, on the edge between b and B, where $c(x, y)$ does *not* hold, the compiler inserts a comparison where the condition between

Condition	True condition	False condition
$a > b$	$a = b + 1$	$a = b$
$a < b$	$a + 1 = b$	$a = b$
$a \leq b$	$a = b$	$a = b + 1$
$a \geq b$	$a = b$	$a + 1 = b$

Table 5.2: Additional conditions added to the true and false arms of the original condition for integer comparisons.

x and y barely does not hold. If the condition is $x > y$, the condition that barely does not hold is $x = y$, because it barely fails the strictly-greater-than test. The consequents for the newly inserted comparisons are empty, but they then proceed to their original respective targets of A and B. This example transformation on the control flow graph is illustrated in Figure 5.2.

For integers, the operators $>$, $<$, $\leq$, and $\geq$ have well-defined boundary conditions. Table 5.2 shows the additional boundary conditions that must be added to each arm of an original condition. The additional conditions correspond to the minimum or maximum value for which the condition still holds true on that arm. For example, if the original condition is $a > b$, the smallest value that a can be while still making the condition true is $b + 1$. Similarly, the largest value that a can be while still making the condition false is b. Thus, tests for these two values are added for the corresponding arms. The two original paths in the program, representing the conditions $a > b$ and $a \leq b$ now become four paths, representing the conditions $(a > b) \wedge (a \neq b+1)$, $a = b + 1$, $(a \leq b) \wedge (a \neq b)$, and $a = b$.

In this work, we apply our boundary condition transformation to the integer comparison operations `>`, `>=`, `<`, and `<=`, as shown in Table 5.2. Because we do not require the constraint solver to support more powerful theories such

```
temp = x > y;
...
...
if(temp) {
  true_branch();
} else {
  false_branch();
}
```

Figure 5.3: Boundary condition tests must recognize stored conditional results or incorrect boundary conditions may be generated.

as sets or strings, we do not implement other such boundary conditions.

It is not sufficient to simply apply the transformation to the branch conditionals found in the program. Many C programs store the results of comparisons in integer “booleans” for use in later conditions. If these stored booleans are not properly recognized and handled, vital boundary conditions will be missed. For example, consider the code in Figure 5.3. In the `if` statement, `temp` is compared (implicitly) to zero. Because we do not test for boundary conditions on equality comparisons, no boundary condition is tested. Even if we added tests that compare `temp` to `-1` or `1`, the correct boundary conditions involving `x` and `y` will not be tested. Stored conditions are used frequently in certain programs; `tcas` uses it almost excusively to avoid repeately writing out complex predicates.

To solve this problem, we perform an additional pass that scans the source code for these stored conditionals. The transformation is slightly more complex than the one for a simple conditional because the stored conditional need not be in the same basic block as the condition that subsequently uses it. Moreover, even if the corresponding conditional that uses it can be identified, we cannot simply inline the original test as values may have changed. Thus, the boundary conditions for a stored conditional must be evaluated on the spot. Immediately after the store to the "boolean," we add a branch that tests exactly the same condition that was stored. Thus, to continue the example in Figure 5.3, we would add another `if` after the line that tests for `x > y`. We then apply the exact same transformation as before on this new conditional. Note that in this case, we add four new paths where there was none before.

5.3.3 Generality and Limitations

This transformation is in principle applicable to any comparison for which a boundary value is well-defined, not just integers. For example, if c were set-inclusion between sets x and y, then the boundary case on the true side (where x is a subset of y) would be where x equals y, as the boundary case here is strict inclusion. Similarly, if c were string-prefix, the boundary case would be whether the string was a proper prefix.

Our transformation does not attempt to fully solve the path inadequacy problem, which easily reduces to the halting problem. Even ignoring issues of undecidability, our solution only addresses boundary conditions that appear in the original program. To understand this point, we first explain the root of the path inadequacy problem.

The path inadequacy problem is caused by a mismatch between the way that symbolic execution partitions the input space and the way that the formal specification of correctness partitions the space. Symbolic execution partitions the space by paths, such that each partition represents a path through the program. The test system then generates one input from each partition. If the program is buggy, there will exist "gaps" where the program path partitioning

```
/* Source program */
int GreaterThanTenExcept154(int x) {
  return (x > 10);
}

/* Correct program */
int GreaterThanTexExcept154(int x) {
  if(x == 154) return 0;
  return (x > 10);
}
```

Figure 5.4: A simple case where boundary condition testing on the source program cannot reveal the bug.

and the true partitioning differ. Our solution can address the cases where the gap overlaps with a branch that is *present in the program.* However, the path inadequacy problem can still arise if there are gaps that do not overlap a condition in the program, such as when the program fails to test for a condition entirely.

In general, the path inadequacy problem is unsolvable without a full formal specification of the program. Consider, for example, the function `IsGreaterThanTenExcept154` in Figure 5.4. If the function were implemented as shown, then even with full boundary condition testing, the bug would not be caught with input values 9, 10, and 11. The fact that 154 is special is not present anywhere in the original program, only in the specification. Thus, in the absence of a full formal specification, an automated testing system cannot guarantee that all boundary conditions are tested.

Finally, our solution only addresses errors that can be revealed by boundary values. For many operations, boundary values are not well-defined.

For example, if the operator is bitwise-and, it is not clear what the boundary condition tests should be. Our solution will not generate tests that are guaranteed to reveal these errors. However, the common corner cases of improper conditionals, such as less-than versus less-than-or-equal, off-by-one errors, and others are all handled by our solution.

5.3.4 Integration with Static Analysis

Unfortunately, our solution dramatically increases the number of paths that must be searched. For each eligible comparison operation with two paths, our transformation adds two additional paths to the control flow graph to represent the boundary conditions. In the case where the result of a comparison is stored in a variable, it adds a hammock containing four paths where there originally was none. Thus, this transformation can increase the number of paths from 2^n to 4^n or worse.

Because our boundary condition solution does not alter any control or data dependencies in the original program, our static analysis is performed *before* inserting boundary condition hammocks. When the new boundary condition branches are inserted, the compiler copies the branch data from the "parent" branch. The weights on the boundary condition branches are set to 0/0 because there is no preference for whether values are boundary values, only that they be tested. The classification field on the boundary condition branches is changed to indicate that the branch tests a boundary condition and is not actually present in the program, which allows the branch selector to discriminate accordingly.

5.4 Dynamic Test Case Generation Framework

Once the Bullseye compiler has transformed the input program and produced the Branch Data file, the resulting modified program can use our testing system to generate test inputs, as we now describe in this section.

At a high level, the dynamic test case generation framework in Bullseye has several components.

- Symbolic execution is used to record and collect the symbolic constraints along the path that was executed. The code to perform symbolic execution is added by the compiler. The system also records which branches were taken or not taken and in what order.
- Environmental modeling is required to handle I/O and other interactions with the system. We provide a simple mechanism for programs to read inputs from streams and to write outputs to streams.
- The branch selector takes the list of branches executed by the program and selects a branch to negate. The branch selector is the heart of the targeted testing system; it is the only component that differs fundamentally from directed testing and is the only component that directly uses the branch data file computed by static analysis.
- The constraint solver is used to generate a new input. The existing path constraints collected by symbolic execution plus the negated constraint corresponding to the branch chosen by the branch selector are used to generate a new input that behaves identically until it reaches the negated branch, at which point it proceeds down a different path.

- To determine whether the current input has found a fault, Bullseye calls a user-supplied oracle. The oracle must conform to our oracle interface.
- The history and statistics component maintains detailed information about past paths tried as well as the statistics used to generate our results.

In the remainder of this section, we will discuss each of these components in greater detail.

5.4.1 Symbolic Execution

As with directed test input generators [44, 87], the program under test must be modified so that it performs symbolic execution alongside its concrete execution. In our implementation, this transformation occurs after any static analysis or boundary condition encoding. Our compiler takes ANSI C source code as input and produces a modified C program as output, containing calls to our symbolic execution library. The resulting program is then compiled and linked with our testing system.

The modified C program itself is split into two components. The first is an automatically generated program-specific driver file. This driver is responsible for initializing the symbolic execution system and describing to the symbolic execution system all the C data types encountered. The second is the instrumented program itself, which contains the original program plus additional code that calls functions that perform symbolic execution.

5.4.1.1 Type and Variable Declarations

In order to properly generate inputs, the symbolic execution system needs to know about all of the types encountered in the program. Therefore, the driver file includes a long series of calls that defines the shapes of the various structures and unions used in the program, the names of their fields, the sizes of various data types, and so on. The internal representation of C types is a directed type graph, with primitive types as terminal nodes and other types at nonterminal nodes. For example, a struct with two integer fields will have a node for the struct that points to nodes for the fields, which in turn point to the terminal node representing primitive integers.

Variables must also be declared. At the beginning of each function, the compiler inserts a series of calls that symbolically declares all of the local variables. This allows Bullseye to associate names with addresses and to perform the proper lookups when symbolic constraints need to be solved. In addition, formal function parameters must also be declared; these behave slightly differently as the values must also be copied to properly simulate C's call-by-value semantics.

5.4.1.2 Symbolic Constraints

Symbolic execution is actually performed by calling a set of functions that handle symbolic assignments and evaluation. Each statement in the program under test is automatically transformed by the compiler to include the correct calls to perform symbolic execution. A sampling of the transformations is shown in Table 5.3.

The transformations themselves are fairly straightforward and follow the same pattern as prior work [87]. The `symbolic_assign` function as-

Source	Instrumentation
`x=y;`	`symbolic_assign((void*) &x, "y");`
`*x = y;`	`symbolic_assign((void*) x, "y");`
`x = *y;`	`symbolic_deref((void*) &x, "y");`
`*x = *y;`	`symbolic_deref((void*) x, "y");`
`x = y op z;`	`symbolic_eval((void*) &x, "y", OP_op, "z");`

`if(x op y) goto l;`	`comparison_result = x op y;`
	`evaluate_predicate("x", OP_op, "y",`
	`comparison_result, "branchID");`

Table 5.3: Source code and the corresponding symbolic instrumentation. The instrumentation is added before the original statement.

signs the symbolic value of the named variable to the specified address. The `symbolic_deref` function stores the symbolic address of the named variable in the specified address. The `symbolic_eval` function adds the appropriate symbolic constraints to values stored as the result of a computation. Finally, the `evaluate_predicate` generates the appropriate symbolic constraints for a branch condition and associates it with the branch identifier. The branch identifier tells Bullseye the key to use when looking up data for this branch in the branch data file, as discussed in Section 5.2.5.3.

Function call and return is handled with a stack. Several functions allow the system to symbolically push and pop variables onto a stack for function calls.

For additional information on our implementation of a directed testing system, see prior work [90].

Function	Purpose
`inputstream_init()`	Initializes stream
`inputstream_getchar()`	Returns char or EOF
`inputstream_getint()`	Returns int or EOF
`inputstream_scanint(ptr)`	Writes `int` to `ptr`
`inputstream_getinputstream()`	Returns pointer to array
`inputstream_getsize()`	Returns size of input stream
`inputstream_printstream_as_char()`	Prints contents of char stream
`inputstream_printstream_as_int()`	Prints contents of integer stream
`inputstream_outputint(i)`	Outputs i to output
`inputstream_getoutputstream()`	Returns copy of output array
`inputstream_getoutputsize()`	Returns size of output

Table 5.4: Functions for accessing input and output streams.

5.4.1.3 Entry Point

As with other test input generators, the compiler must be provided with a program entry point, which defines the input types that the system should generate. This entry point need not be (and often is not) the `main` function. For example, to test a list insert function, the user simply specifies that function as the program entry point, eliminating the need to write `main()` as a wrapper or test driver for that function. The inputs to the program are the arguments to the function and we use the input generation process described in previous work [87] to initialize any structures or complex pointer-based data types.

5.4.2 Environment Modeling

Many programs interact with the environment by reading from or writing to files and streams. Thus, in order to support these programs, interaction with streams must be modeled.

Our system supports limited interaction with input and output. We model a limited form of standard input and output. A program may read integer-type values from a single input stream (representing standard input) and output integer-type values to a single output stream. The API is shown in Table 5.4.

We implement streams by using arrays and by translating the appropriate calls to their corresponding array constraints. For example, the reading of the first character from the input stream into a variable `c` creates a constraint that specifies that `c` is equal to the first element of the input array and that the array index is now incremented. These array accesses are augmented to introduce constraints to simulate the behavior of the file input/output functions so that the program need not be aware that the input stream is in fact an array. For example, if an attempt is made to read data beyond the end of the "file," the appropriate EOF code is returned. Thus, programs employing the common `while(input != EOF)...` idiom can be tested without modifications to the program logic.

Because C does not allow for easy resizing of arrays, a maximum size must be chosen ahead of time. By default, our system limits standard input to ten inputs, but this value can be adjusted upward or downward by the user as needed.

Bullseye does not model directories or filesystems, like most testing systems [22].

The I/O interface is used by manually transforming I/O calls in the original program. Calls to various input functions such as `getchar` or `scanf` must be replaced with special calls to our engine. Note that our input stream is not exactly the same as standard input; a program can directly request an integer value instead of using `scanf` with format strings. This simplification removes unnecessary constraints relating integer inputs to their character representations and greatly reduces the complexity of the path search space.

5.4.3 Constraint Solver

The constraint solver is used to generate new program inputs. It receives a set of path constraints representing the path to follow plus a negated constraint. It returns either a solution to the path constraints plus the negated constraint or a "no solution" answer indicating that the path is infeasible. Our input generation methods, including those for pointer-based data structures, is identical with CUTE [87] and our implementation is based largely on the description in their paper.

Our implementation uses the CVC3 constraint solver to generate new program inputs. We generate linear constraints over integer and integer-like variables, including data structures with integer fields and arrays of integer-like values, including character arrays. In addition, we support complex pointer-based data structures using previous input generation techniques [87]. Floating point operations are not currently supported by our system, although that is not an inherent limitation. Non-linear constraints, such as those introduced by multiplication or division, are modeled by using the concrete value to reduce the expression to a linear constraint, as in concolic testing systems [87]. While this simplification does not guarantee that all paths will be explored, it can in most cases allow execution to continue.

5.4.4 Branch Selectors

The branch selector is the component responsible for choosing the branch to negate in order to produce the next input. Bullseye provides an interface for multiple branch selectors. A branch selector has access to data collected during the previous execution, such as path constraints and branch history. It returns a prioritized list of branches to negate; the first branch returned is the branch selector's top pick, and so on. The testing engine can reject picks from the branch selector if they are duplicates, and it defaults to depth-first search if the branch selector does not indicate a preference for any branches.

Our branch selectors are implemented using priority queues. The branch selector examines all branches from the current execution and adds to the priority queue any that it wishes to alter. When the testing engine requests a candidate branch to invert, the lead element of the priority queue is popped off and returned.

5.4.4.1 Simulating Directed Testing

To simulate the depth-first search behavior of systems like DART, we implement a simple branch selector that uses the branch index as the priority. Thus, the first branch in the program has priority 1, the second has priority 2, etc, so the last branch will have the highest priority. In Section 5.4.6.1 we discuss how Bullseye records all previously explored paths and subtrees. The Bullseye engine will reject previously explored paths and any proposed branch selections that would cause it to re-explore already-explored space. By using the path history to prevent repeats, the resulting behavior is identical to a simple depth-first search. This branch selector allows us to directly compare the performance of our system against prior work.

5.4.4.2 Targeted Branch Selection

At runtime, the *only* conceptual difference between Bullseye and directed test input generators is that the Bullseye branch selector uses priorities computed by static analysis. These branch priorities are computed as follows:

- The default priority for a branch is the branch index, which results in a default strict depth-first search. If there is no data for the branch, the priority remains as the index.
- If the branch tests for boundary conditions and has never been selected before, the priority is increased by 10000. This ensures that new boundary conditions are explored immediately upon being exposed.
- If the branch goes in a direction that does not match the branch weights, then the priority is adjusted upward depending on the degree of the mismatch. If the branch weights indicate an absolute (X/0 or 0/X) preference for a direction and the current branch does not match, the priority is increased by 1000. If the branch is a mismatch and the ratio between the weights in that direction exceeds a *threshold*, the priority is increased by the square of the ratio, capped at 400 (20^2). The final weight is multiplied against a *scale* factor. We use a default threshold corresponding to a 1.3:1 ratio and a scale of 1.0.

We evaluate our system with just this simple priority function, whose values were empirically derived. Although it is surprisingly effective, there is still significant work that can be done in exploring the best heuristics for efficient targeted testing. In future work, we plan to study the relative importance and sensitivity of the various parameters across a variety of benchmarks. In addition, automated machine learning techniques such as genetic algorithms [83] can be used to learn coefficients for the priority function.

Later, in Section 6.7, we will examine the effects of varying the scale and threshold parameters and determine how sensitive targeted testing is to the heuristic function.

5.4.5 Test Oracle Interface

To know whether a given execution was correct, the system calls a test oracle supplied by the user. The compiler automatically generates an `oracle.h`, which contains the signature of an `oracle` function. The oracle accepts as parameters the following:

- Return value. The return value, if present, of the original function under test. This parameter is not available if the function returns `void`.
- Function arguments. The oracle has access to the arguments passed to the function under test. Note that these arguments are from after the execution of the function under test. If the arguments point to mutable data structures, the oracle will see through these parameters the data structures after the function under test has been executed.
- Function argument copies. The oracle has access to copies of the arguments to the function under test. These copies are not touched or perturbed by the function under test in any way; if they point to mutable data structures, the oracle will be able to examine an identical

(except for memory addresses) copy of the data structure before the function under test was executed.

The oracle returns a nonzero value to indicate that the case “passes” and zero to indicate failure. If the user does not wish to provide an oracle or does not wish to use an oracle, a simple function that always returns 1 will suffice; Bullseye will then consider every input to have passed.

For programs that use the input and output stream interfaces, the oracle writer also has access to copies of both the input and output stream. The input stream is given as an array of integers and a corresponding size indicating the number of items in the stream. The output stream is given similarly. It is the responsibility of the oracle writer to ensure that the oracle reads from the input stream in a manner semantically compatible with the original program, and to perform any necessary and appropriate comparisons between the output stream and any expected output.

5.4.6 History and Statistics

The Bullseye system also contains components for tracking path history and various statistics. The path history component allows Bullseye to track which paths have been explored as well as which subtrees have been fully explored. The statistics component monitors Bullseye as it runs and produces statistics that we use in our evaluation.

5.4.6.1 Path History

Unlike prior directed testing systems, Bullseye needs to keep track of explored path information. Bullseye must track such information because it can (and often will) explore branches in an out-of-order manner. If branches are explored depth-first, only minimal history needs to be maintained, but targeted branch selectors have the freedom to explore branches in arbitrary order.

The path history tracker logs information about paths that have already been explored. Whenever an input is generated and executed that causes execution to proceed down a previously unexplored path, the path is noted and logged. By tracking information about actual and attempted paths, the path history tracker can provide several essential services to the engine and branch selector:

- Duplicate verification. The path history tracker can determine if a candidate path (or just-executed path) is a duplicate of an already-explored path. Duplicate verification allows us to log and measure the "redundancy" caused by out-of-order exploration.

- Proposed path checking. Given an existing path (branch vector) and a proposed branch flip, the path history tracker can determine if all paths in that direction have already been explored. If so, then flipping the branch is guaranteed to result in a duplicate path. The history mechanism allows Bullseye to avoid the duplicate selection and choose another branch.

- Full subtree exploration. The history tracker can verify whether all paths with a given prefix have been explored, allowing Bullseye to systematically explore execution subtrees. With a null prefix, it is equivalent to exploring all paths. The engine uses full subtree exploration from the root as a termination condition.

Support for path feasibility is integrated into the path history tracker. When the constraint solver is invoked to determine the feasibility of a path (via an existing path prefix from a known feasible branch vector plus a proposed branch flip) and determines that the proposed flip is infeasible, the entire subtree representing the infeasible flip is marked completely explored.

The path history tracker is implemented as a lazily-constructed splay tree with information stored along paths from the root to leaves. Each node in the tree represents a possible branch decision (true or false). A path that has been explored is represented as a path through the tree, with the particular decisions exactly reflecting the corresponding branch vector. All nodes in the tree are created on-demand; nothing is added until the path that requires those nodes is added to the tree. The last branch in the branch vector represents the end of the program and corresponds to leaf nodes in the tree; they are marked as having no children.

Additionally, the interior nodes store completion information representing existential and universal quantification for all paths with the same prefix as the node. The path-explored bit on each branch represents whether there exists a path with the same prefix as the current node, followed by the relevant true/false decision. The all-paths-explored bit records whether all paths with the same prefix as the current node followed by the relevant choice have been explored. For example, if node n represents the path prefix TTF, the outgoing F edge will have the path-explored bit marked if there exists an already-explored path with the prefix TTFF, and will have the all-paths-explored bit marked if all paths with the TTFF prefix have been explored.

Moreover, due to the nature of concolic execution, there are concrete conditions that cannot be flipped. The flip side of a concrete condition is considered infeasible and therefore completely explored. Information about concrete branches is added whenever a new branch vector is added.

Because the execution tree can be potentially enormous, the implementation aggressively deletes nodes after they are no longer required. If a node's true and false subtrees are fully explored, the the appropriate side on the parent is now fully explored and the node (and all of its children) can be deallocated. If Bullseye is allowed to run until all paths are explored, eventually all nodes except the root and the very last node will be deallocated.

The path history tracker incorporates a small amount of instrumentation to collect memory usage information, recording the current size of the tree as well as the maximum size attained thus far. All measurements are in terms of the number of nodes.

5.4.6.2 Logging and Statistics

The logging and statistics facility provides a means to gather experimental data for mutant kill speed and other relevant metrics. The statistics package reports numerous things:

- Number of iterations. The total number of iterations (inputs generated) thus far.
- Number of unique paths. The number of unique paths explored thus far.
- Unsatisfiable paths attempted. The number of candidate paths rejected by the constraint solver as unsatisfiable.
- Duplicate paths. The number of duplicate paths added. Duplicates can result from flipping branches out of order.

- Errors found. The number of errors found thus far, according to the test oracle.
- First error found. The iteration on which the first error was found. The iteration number also measures mutant kill speed.
- Passed branches. A complete list of branch flips that resulted in a passing path according to the oracle.
- Failed branches. A complete list of branch flips that resulted in a failing path according to the oracle.
- Branch flip histogram. A histogram of branches and the number of times each was flipped.
- Path length (unique). The average, min, and max path lengths, as calculated over unique paths.
- Path length (all). The average, min, and max path lengths, as calculated over all paths (which includes duplicates).

The statistics package offers a number of public methods for getting, setting, and incrementing various elements of logged data. These are called from the relevant points within the engine. These values can be reported and printed at any time; currently the statistics are simply dumped at the end. In our experimental evaluation, these results and statistics are fed into spreadsheets for further processing and analysis. Not all of the information is used in our evaluation; we have found that the branch lists and histograms are not generally useful or interesting outside of debugging, and the path length numbers are similarly unenlightening.

As a practical matter, the fact that the statistics and logging package keeps full histories and information about every path executed limits the number of iterations that Bullseye can run before it runs out of memory. For typical programs on a 32-bit system, Bullseye typically runs out of memory after a few hundred thousand inputs. Disabling the statistics and history package would improve this number substantially, but would inhibit the data collection needed to perform studies and experiments.

Chapter 6

Evaluating Bullseye

In this chapter, we will evaluate Bullseye, our targeted testing system, as well as our automated boundary condition testing transformation. Our evaluation is guided by a desire to answer the following questions:

- Does targeted testing find bugs faster than directed testing?
- Does targeted testing find more bugs than directed testing?
- Does automated boundary condition testing find more bugs when added to targeted or directed testing?
- Does automated boundary condition testing slow down bug finding?
- When does boundary condition testing perform well and why?
- How sensitive is targeted testing to the precision of the static analysis?
- How sensitive is targeted testing to the parameters in the heuristic function?
- What insights have we gained and what future directions are suggested by our results?

6.1 Metrics

To answer our main questions, we need a metric to measure how fast a testing system finds bugs. Because dynamic test input generators produce a continuous stream of candidate inputs rather than a single fixed test suite, we cannot use traditional measurements like the size of the test suite. Prior evaluations of directed testing systems have focused largely on the number of inputs generated, percentage of statements covered, or the number of bugs found. While valuable for demonstrating high coverage, these metrics are inadequate for evaluating targeted testing because it does not directly measure bug-finding speed. Instead, our metric is *mutant kill speed* (MKS), which is the number of inputs that the system generates before revealing the first fault. Mutant kill speed directly measures how fast an input generator finds bugs while remaining independent of hard-to-control variables like processor type and system memory that might pollute measurements such as bugs per second. Lower numbers for mutant kill speed represent faster fault identification.

We also wish to find whether targeted testing can result in repeated or duplicate test inputs. Targeted testing can generate duplicate inputs while a fixed search pattern will not. This is because targeted testing can invert early branches before it has finished exploring all the later ones. For example, suppose the initial random input sends the program down some path P. The targeted selector decides to flip branch 3 rather than explore the paths near the initial input. Later, if the targeted selector flips 3 back to the original direction, there is a chance that the constraint solver will generate an input that happens to follow the same path P. If this happens, the input is a duplicate. In our results, we measure the *efficiency* of our system, which is the proportion of total inputs generated that are not duplicates. By this metric,

Name	Purpose	Versions	LOC
`triangle`	Triangle classification	9	42
`tictactoe`	Tic tac toe checker	4	70
`tcas`	Aircraft collision detection	41	173
`schedule`	Process scheduler	8	412
`printtokens`	String tokenizer	6	569
`polymorph`	Filename converter	12	716
`sjeng-core`	Chess (move checker)	9	3425
`eqntott`	Truth table calculator	9	4363
`tr`	Unix text translator	5	7683

Table 6.1: List of benchmarks, including the number of mutants (versions) evaluated and the program size in terms of source code.

directed testing has 100% efficiency.

6.2 Test Benchmarks

We evaluate our system using nine benchmarks, shown in Table 6.1. From the Software-artifact Infrastructure Repository [36] we use three programs from the `siemens` test suite: `tcas`, `schedule`, and `printtokens`. From the PEST suite [68] we use the classic `triangle` classification program. From the BugBench [66] suite, we use the GNU `polymorph` utility. From SPEC CPU2006 we use the `sjeng` benchmark, and from SPECint92 the `eqntott` benchmark. From the GNU Coreutils package, we test the standard Unix utility `tr`. Finally, we add one synthetic benchmark of our own that evaluates tic-tac-toe boards, `tictactoe`.

In order to provide a more complete picture of targeted testing, we chose our programs to maximize qualitative diversity. If only system utilities or only numeric applications or only state machines were tested, we would not be able to generalize our results to other types of programs. We selected these benchmarks because they represent a wide range of program *characteristics*, differing greatly in the data structures used, the operations performed, and the overall program structure.

Our study includes several tiny benchmarks, which admit a detailed study of program and branch selector behavior, allowing us to understand why certain test strategies succeed or fail. The smallest programs contain no loops and allow the test generators to explore the entire path space, which is particularly important for our study of boundary condition testing.

We will now discuss the benchmarks in more detail. For the benchmarks from the SIR suite, slight modifications had to be made because the SIR programs include their own test driver. The SIR suite relies on I/O differencing instead of a native-language oracle and therefore requires a driver that reads input from standard input and parses it before passing the input to the function that actually performs the task. Since Bullseye can directly call functions and programs under test, this driver is not needed and Bullseye either calls the appropriate function directly or calls a wrapper driver that we provide. Other modifications and details are discussed in the section for each benchmark.

6.2.1 triangle

The `triangle` benchmark is a short triangle-classification program that classifies triangles, given by the lengths of their sides, as equilateral, isosceles, and so on. The program has a very small state space, is wholly contained within a single short procedure, and is a standard benchmark for evaluating software testing systems.

6.2.2 tictactoe

The `tictactoe` benchmark was written by the author and contains a single procedure. It takes as input a tic-tac-toe board and determines if any player has won the game and if so, returns the player. The program essentially consists of a long series of short-circuited comparison operations that checks the rows, columns, and diagonals and was intended to be superficially similar to the `sjeng-core` benchmark on a much smaller scale. Virtually every path in the program is feasible and represents a different board configuration. As a result, the program has a far larger state space than its small size would indicate.

This program was also deliberately written to serve as a worst-case scenario for our automated boundary condition testing technique. The board representation employed uses negative integers for player X, positive integers for player O, and zero for empty squares. Thus, the program consists almost entirely of greater-than and less-than comparisons. However, there are no boundary conditions whatsoever in `tictactoe`. Therefore, boundary condition testing is completely useless while also guaranteeing worst-case exponential blowup.

6.2.3 tcas

The `tcas` benchmark is extracted from an aircraft collision warning system and was originally from the `siemens` software testing suite. The program contains no loops and evaluates a very complex predicate (the collision warning predicate) in several steps, using numerous short functions as shorthand for certain formulae and arithmetic expressions. The program heavily stresses numerical predicates but does not use complex data structures; all variables are integers and no data structures are created dynamically.

6.2.4 schedule

The `schedule` benchmark simulates the operation of a job scheduler. The program accepts some initial values for the number of jobs as well as a stream of scheduler commands such as “upgrade priority” and “flush.” It produces a list containing the order in which processes exit the scheduler as output.

In addition to the core process scheduling logic, a significant portion of the program is devoted to the implementation of priority queues. The program itself is structured as a single main loop that reads commands and handles them in the various arms of a very large `switch` statement.

Our test driver treats the initial process settings as input parameters and uses the I/O library to simulate new scheduler commands. Differences in the process exit order are considered fault-revealing deviations.

6.2.5 printtokens

The `printtokens` benchmark parses an input stream and produces tokens along with a classification of the token type, such as integer constant, string constant, identifier, and so on. The program makes extensive use of standard string operations.

Our test driver uses the I/O library to provide character inputs to the tokenizer and considers differences in the resulting token classification to be fault-revealing.

6.2.6 polymorph

The `polymorph` benchmark is a small Unix utility that converts DOS-style filenames into Unix-style file names. The program stresses character-level string operations but not string classification.

We replace the `main` function of the original program with a test driver that calls the function that actually converts filenames. The few remaining other parts of the program exist simply to make system calls to perform the actual file renaming and are therefore not relevant to our test harness as we do not simulate the underlying file system. Thus, we only check that it performs the filename conversions correctly. Our test driver uses the I/O library to provide arbitrary-length input strings to serve as filenames. Any deviation in the output string is considered fault-revealing.

6.2.7 sjeng-core

The `sjeng-core` benchmark is adapted from Sjeng 11.2, a chess playing program. The original program is a sophisticated chess-playing program that uses, among other things, a heuristic search combined with a database of opening moves. We test the portion of Sjeng that evaluates the legality of proposed chess moves, omitting the portions that handle the move database and search heuristics. The size of the full Sjeng program is over 17,000 lines of code; the number we report in Table 6.1 is only the portion directly involved in checking move legality.

Our test driver also contains some code that validates chess boards before passing them to the move checker. Sjeng assumes that the board always represents a valid chess board and will behave incorrectly if the board is invalid. For example, placing six kings on the board or removing the edges of the board will cause the program to return nonsensical results. To eliminate these spurious errors, our test driver enforces board validity as a precondition before transferring control to the Sjeng move checker.

6.2.8 eqntott

The `eqntott` benchmark, which was formerly a part of the SPEC suite, converts boolean equations into truth tables. The program stresses sorting and iterative computation.

The program itself first parses the input strings and builds representations of the equations. It then builds several data structures representing the constraints in the equations and repeatedly "grinds" over this structure until it converges on an answer, which is the final truth table.

Our test driver bypasses the parser and tokenizer and directly passes a well-formed equation to `eqntott`. The equation itself is obtained by reading symbols and operations via our I/O interface in postfix order; this minimizes the amount of time the testing system is required to spend on parsing. Any deviation in the output truth table is considered fault-revealing.

6.2.9 tr

The `tr` benchmark is the Unix `tr` utility, from the GNU Coreutils package, and is installed on virtually every Linux system today. The program is commonly used by shell scripts to perform text substitution and replacement tasks. It takes as input two specification strings and translates an input stream by replacing characters in the first specification string with their counterpart in the second specification string.

Our test driver simulates invocation of `tr` from the command line with default options, with two strings up to length 5 for the specification strings and an input stream as simulated by our I/O interface. Faults are revealed if the program and the test oracle produce different output strings.

6.3 Test Methodology

We evaluate our system as though it were a part of a change management system, and we use a form of mutation testing [2] to simulate a step in program evolution. The original program represents the previous program version, the mutants represent new versions derived by changing the original, and the test goal is to evaluate Bullseye's ability to identify bugs introduced by the changed program, both in terms of the number of mutants it could kill and the number of inputs generated before each mutant was killed.

6.3.1 Mutant Selection

Unlike some evaluations using mutation testing, we do not use randomly generated mutants as the difficulty of killing randomly generated mutants may not be representative of real bugs when compared with hand-seeded bugs. In fact, prior work has shown that hand-seeded faults are significantly harder for testing systems to find than randomly generated mutants and in some cases can be harder than even real-world bugs [4]. Thus, for all of our experiments, we rely on hand-seeded faults.

For the SIR [36] and PEST [68] benchmarks, we use the mutants supplied by the maintainers. In cases where the mutation is in the test driver, we discard the mutant. The `polymorph`, `sjeng-core`, `tr`, and `eqntott` programs do not come with standard mutants or different program versions, so we generated hand-seeded mutants, emphasizing the same types of mutations used by SIR. Finally, since `tictactoe` is our own program, we hand-seeded our own faults.

6.3.2 Test Oracle

Because our testing system requires an oracle to determine if a fault has been found, we supply test oracles for all of our benchmarks. We create these oracles by creating a complete second copy of the original version of the program under test, with all function names and global variables name-mangled to prevent namespace collisions during compilation. The `oracle` function itself performs any actions necessary to prepare the inputs for the copy and to compare the results after the copy returns. If the program uses our I/O interface, the oracle mimics stream access with access to the underlying input stream arrays, essentially the reverse of the original process of simulating streams with backing arrays.

6.3.3 Test Parameters

In general, it is impossible to completely test a program, especially if a program contains loops, because the number of paths may be extremely high or infinite. We must therefore choose an upper limit on the number of inputs generated before the testing system is deemed to have failed to find a fault. For our experiments, this limit is set at ten thousand inputs; if no faults are found within this limit, the test system is considered to have failed to find any faults in that mutant.

In addition, several of the programs read from potentially unbounded streams. In order to prevent the program from simply testing longer and longer streams and to limit the likely state explosion from long input streams, we must bound the size of any input streams. For all of our experiments, we bound standard input to ten inputs. This is not the same as ten characters; a program can directly read ten integers without any parsing from our input stream, which would have a textual representation of much more than ten characters. There is no bound on the data structures that are passed directly as function parameters.

We must also control for randomness because the constraint solver can randomly generate satisfying inputs which can affect the mutant kill speed. For each program version, we execute the program seven times and use the *median* result, based on the iteration on which the fault was identified. This approach reduces the noise that is possible due to luck on the part of the constraint solver. The Bullseye system itself uses no random numbers, so the constraint solver is the sole source of non-determinism.

6.4 Evaluating Targeted Testing

Our first task is to evaluate the effectiveness of Bullseye against directed testing on the baseline program versions. Table 6.2 shows a comparison between Bullseye (labeled TGT) and directed testing (labeled DFS).

6.4.1 Faults Found

The most important goal is finding faults in programs. As shown in our results, Bullseye finds 71/103 faults while directed testing finds only 65/103. There are no cases in which directed testing finds a fault that Bullseye does not. Most notable, directed testing was unable to find any faults among the six versions of `printtokens` within ten thousand iterations, while Bullseye was

Benchmark	Bugs Found		Iter Found		Imp	Paths	Eff.
	DFS	TGT	DFS	TGT			
`triangle`	5/9	5/9	8.2	3.8	2.04×	15.89	0.9248
`tictactoe`	4/4	4/4	15.5	15.5	1.0×	1645.6	0.9756
`tcas`	30/41	30/41	19.62	7.97	2.76×	196.65	0.9324
`schedule`	0/8	0/8	NA	NA	NA	10000	0.9612
`printtokens`	0/6	5/6	NA	26.2	NA	10000	0.9888
`polymorph`	12/12	12/12	7.83	4.83	1.58×	10000	0.8414
`sjeng-core`	6/9	6/9	21.67	14	2.28×	10000	0.9100
`eqntott`	8/9	9/9	971.13	318	5.48×	10000	0.9512
`tr`	0/5	1/5	NA	78	NA	10000	0.9035
Average/Total	65/103	71/103	175.72	58.54	2.52×		0.9321

Table 6.2: Comparison of depth-first search (DFS) vs. targeted testing (TGT). Iteration found is the average iteration found across all versions of that program. The relative improvement is an average of the ratio of the iteration found over each individual version and thus differs from a simple ratio of the average iteration found. Efficiency is TGT's proportion of paths explored that are not duplicates.

able to find 5/6 after an average of only 26.2 iterations. Moreover, Bullseye could find all of the faults in `eqntott` while directed testing misses one case, and Bullseye finds one fault in `tr` while directed testing is unable to find any at all. Thus, targeted testing is able to find more bugs than ordinary directed testing.

6.4.2 Mutant Kill Speed

Let us now look at mutant kill speed, measured here in terms of the number of iterations required to find a fault. The *Iter Found* columns in Table 6.2 show the average iteration number on which the fault was found for each of the benchmarks. The *Imp* column shows the relative improvement in mutant kill speed from directed to targeted testing. This improvement is not the same as the ratio of the average iteration found for each of the benchmarks (which is a ratio of averages); rather, it is computed by first computing the ratio for each of the mutants and then finding the average of the ratios (an average of ratios).

We see here that in the cases where both Bullseye and directed testing are capable of finding the fault, Bullseye finds the fault 2.52× faster on average. That is, Bullseye only generates one-half to one-third the number of inputs as directed testing before it can find a fault. For some programs, the ratio can be even greater, up to 5.48× in the case of `eqntott`.

Moreover, this speedup figure understates Bullseye's advantage over directed testing because this ratio can only be computed for the mutants where both systems find the fault. If only Bullseye finds the fault, it is not credited with what is arguably an infinite speedup.

6.4.3 Input Generation Efficiency

The last column in Table 6.2 gives the efficiency of the Bullseye system as a ratio between the unique paths tested and the total number of paths tested. There is significant variation between the benchmarks, from a high of almost 99% for `printtokens` and a low of around 84% for `polymorph`, with an average efficiency across all benchmarks of around 93%. Thus, up to 16% of the inputs Bullseye generates are redundant and duplicate prior effort. As long as the efficiency does not unduly affect mutant kill speed, it is largely meaningless as the goal of Bullseye is to find faults quickly, not to generate lots of inputs quickly. Nonetheless, we believe that the average efficiency remains quite acceptable even if one is aiming to simply generate many inputs.

Fortunately, we find that the somewhat lower efficiency of Bullseye does not hamper its power to find faults as measured by mutant kill speed. To understand this better, we enabled detailed debugging output for the benchmarks and observed the running efficiency ratio while Bullseye generated inputs. We found that efficiency actually remains very high until some number of fault-revealing inputs are generated and that the efficiency worsens around the middle or end of the run. What this means is that the overwhelming majority of these duplicate inputs are being generated after the fault has already been found and therefore does not affect mutant kill speed. This is not surprising, as duplicates are only generated when Bullseye flips back a branch that it flipped earlier. As long as branch flips take Bullseye towards faults, there should be minimal duplicates until the fault is found.

6.4.4 Additional Observations

The results for the two smallest programs are quite remarkable. In fact, it was these results that motivated us to develop our automated boundary condition testing technique in the first place. We see that in 11 out of 41 cases in `tcas` and in 4 out of 9 cases in `triangle`, neither targeted nor directed testing is able to identify the faults, despite the fact that the programs contain no loops and *all paths are explored*. Upon close inspection, the affected versions suffer from the path inadequacy problem. The constraint solver generates inputs that traverse every path, but it does not generate the exact input required to reveal the fault. The frequency with which boundary condition errors occur shows that the path inadequacy problem is surprisingly common. We emphasize that the issue is not with the specific search strategies used; any automated testing systems based on symbolic execution can similarly fail regardless of the sophistication of the search strategy.

Neither targeted testing nor depth-first search find any faults in any version of the `schedule` benchmark. Depth-first search fails for two reasons. First, the search space is enormous and the initial random input never executes a path close to one that would reveal a fault. Second, human inspection shows that these faults can typically be revealed only through a highly specific set of operations. For example, one version introduces a simple fault in the priority queue code that would be easy to find via careful unit testing. However, because the priority queue code is evaluated in the context of the scheduler that uses it, the fault is missed unless a specific and non-obvious schedule is used. Bullseye is unable to help for a third reason: All operations are performed on a global data structure. Because every operation uses the priority queue, Bullseye's static analysis determines that every branch in the program affects every other, resulting in useless branch data.

Conclusion Bullseye is able to find more faults than ordinary directed testing. In the cases where both are able to find a fault, Bullseye finds it an average of 2.5× faster. Although Bullseye is slightly less efficient than directed testing when it comes to generating duplicate inputs, this does not affect mutant kill speed because the duplicates are generated after the fault has already been found.

6.5 Evaluating Boundary Condition Testing

We now evaluate our technique for automated boundary condition testing. We apply our transformation to our benchmarks and examine how boundary condition testing affects both Bullseye and directed testing.

Benchmark	Bugs Found		Iter Found		Imp	Paths	Eff.
	DFS	TGT	DFS	TGT			
`triangle`	9/9	9/9	22.56	15.22	1.46×	86.33	0.9248
`tictactoe`	4/4	4/4	89.5	92.75	0.97×	10000	0.9763
`tcas`	38/41	38/41	195.7	139.51	6.11×	5473	0.9759
`schedule`	0/8	0/8	NA	NA	NA	10000	0.9115
`printtokens`	0/6	5/6	NA	23.8	NA	10000	0.9868
`polymorph`	12/12	12/12	69.58	8.67	4.68×	10000	0.9076
`sjeng-core`	8/9	8/9	23.75	15.38	2.35×	10000	0.8866
`eqntott`	9/9	9/9	776.67	227.56	7.18×	10000	0.9539
`tr`	0/5	3/5	NA	100.33	NA	10000	0.9131
Average/Total	76/103	81/103	196.29	77.9	3.79×		0.9374

Table 6.3: Comparison of depth-first search versus targeted testing with boundary condition testing. The columns are the same as in Table 6.2.

Our main results are presented in three tables to better show the effects of boundary condition testing on directed and targeted testing. Table 6.3 shows the performance of both directed and targeted testing with boundary condition testing side-by-side. Table 6.4 shows directed testing with and without boundary condition testing side-by-side, allowing us to examine the effect of boundary condition testing on directed testing alone. Finally, Table 6.5 shows targeted testing with and without boundary condition testing side-by-side, showing the effects of boundary condition testing on targeted testing.

6.5.1 Faults Found

We first examine boundary condition testing's effect on fault identification. In Table 6.3, the *Bugs Found* column shows the number of faults identified for each of the benchmarks. We find that boundary condition testing identifies significantly more faults, finding a total of 88/103 of the seeded

Benchmark	Bugs Found		Iteration Found		Slowdown
	Base	BCT	Base	BCT	
`triangle`	5/9	9/9	8.2	22.56	2.28×
`tictactoe`	4/4	4/4	15.5	89.5	6.74×
`tcas`	30/41	38/41	19.62	195.7	4.75×
`schedule`	0/8	0/8	NA	NA	NA
`printtokens`	0/6	0/6	NA	NA	NA
`polymorph`	12/12	12/12	7.83	69.58	6.33×
`sjeng-core`	6/9	8/9	21.67	23.75	1.09×
`eqntott`	8/9	9/9	971.13	776.67	0.95×
`tr`	0/5	0/5	NA	NA	NA
Average/Total	65/103	80/103	175.72	196.29	3.69×

Table 6.4: Comparison of boundary condition testing, using depth-first search. Base represents directed testing without boundary conditions, while BCT represents directed testing with boundary condition testing. Slowdown shows the average of the ratio of the iteration found over each individual version. Path increase is calculated for the two programs that have a finite number of paths.

faults when used in combination with targeted testing, compared with only 65/103 for directed testing without boundary condition testing. This represents a 35% improvement in fault coverage, a very significant improvement.

Moreover, this improvement is due in large part to the effectiveness of boundary condition testing at tackling the path inadequacy problem. The `triangle` and `tcas` benchmarks are small benchmarks for which it is possible to explore all paths in the program. Therefore, any faults not found after a complete exploration of the path space are due to the path inadequacy problem and thus *cannot* be found by any symbolic execution system except through chance. Even without directed testing, Table 6.4 shows that our boundary condition testing technique reliably improves the fault coverage of directed testing for `triangle` from 5/9 to all 9/9 versions, and improves `tcas` from

Benchmark	Bugs Found		Iteration Found		Slowdown
	Base	BCT	Base	BCT	
`triangle`	5/9	9/9	3.8	15.22	3.55×
`tictactoe`	4/4	4/4	15.5	92.75	6.95×
`tcas`	30/41	38/41	7.97	139.51	2.67×
`schedule`	0/8	0/8	NA	NA	NA
`printtokens`	5/6	5/6	26.2	23.8	0.91×
`polymorph`	12/12	12/12	4.83	8.67	1.98×
`sjeng-core`	6/9	8/9	14	15/38	0.93×
`eqntott`	9/9	9/9	318	227.57	1.15×
`tr`	1/5	3/5	78	100.33	3.78×
Average/Total	71/103	88/103	58.54	77.9	2.74×

Table 6.5: Comparison of boundary condition testing, using targeted testing. The columns are the same as Table 6.4.

30/41 to 38/41 versions. Because these programs can be completely explored, no amount of cleverness in targeted testing could have delivered these gains; boundary condition testing is required to expose the conditions that reveal the faults.

Boundary condition testing also shows benefits in larger programs as well. Boundary condition testing enables ordinary directed testing to find two additional faults in `sjeng-core`, bringing the total from 6/9 to 8/9, while also finding the last fault in `eqntott`. For targeted testing, these additional benefits are also realized. As shown in Table 6.5, targeted testing gains the same improvement in fault coverage for `sjeng-core`. Boundary condition testing is not required for `eqntott` because targeted testing already finds all 9/9 faults. Moreover, coverage for `tr` improves from a mere 1/5 faults to a much more respectable 3/5 faults. In these cases, both targeted testing and boundary condition testing were required; directed testing with boundary condition testing still does not reveal any faults in `tr`. This is because targeted testing and boundary condition testing were designed to serve complementary goals: the former brings testing closer to the fault, while the latter makes it possible to reveal faults masked by the path inadequacy problem. In other words, targeted testing helps to find the correct path, and boundary condition testing helps ensure that the fault actually manifests once the correct path is found.

Conclusion Boundary condition testing can reveal numerous bugs that symbolic testing systems previously could not reveal except through chance alone. Both ordinary directed testing and targeted testing benefit greatly and can find more faults than before. Moreover, a combination of both targeted testing and boundary condition testing can be required to find faults that are both difficult to reach and that are affected by path inadequacy.

6.5.2 Impact on Mutant Kill Speed

Automated boundary condition testing can cause exponential blowup in the path space of programs. How great is the impact on our benchmarks?

First, let us consider boundary condition testing with ordinary directed testing. The *Iteration Found* columns in Table 6.4 show the average iteration on which the fault was found for directed testing (Base) and for directed testing plus boundary condition testing (BCT). The slowdown is the ratio between boundary condition testing and ordinary directed testing. Like the improvement figures in Table 6.2, the slowdowns are computed as an average of the ratios. We find that on average, adding boundary condition testing causes the test system to take 3.69× longer to find a fault in the cases where both vanilla directed testing and directed testing plus boundary condition testing find faults. This slowdown is quite significant, but is hardly exponential.

A similar pattern holds for boundary condition applied to targeted testing. In this case, however, the average slowdown is only 2.74×, which while still significant, is not as high as the 3.69× slowdown that directed testing experiences. This holds not just for the average but also on a benchmark-by-benchmark basis. Table 6.3 shows a head-to-head comparison of directed and targeted testing with boundary condition testing enabled for both. When computed on a mutant-by-mutant basis and averaged across all of the benchmarks, targeted testing finds faults 3.79× faster than directed testing. This is significantly higher than the 2.52× difference when boundary condition testing is disabled. This means that directed testing is far more negatively impacted by boundary condition testing than targeted testing, or equivalently, that targeted testing mitigates some of the high overhead of boundary condition testing.

The case of `tictactoe` differs from the other small benchmarks. Even without boundary condition testing, `tictactoe` has vastly more paths than the other benchmarks, with an average of 1,645.6 paths across the versions. The larger and seemingly more complex `tcas` has only an average of 196.65 paths. When boundary condition testing is applied, the number of paths grows exponentially and `tictactoe` no longer finishes within ten thousand inputs. In fact, we ran `tictactoe` until Bullseye ran out of memory after over 200,000 iterations and it still had not explored every path, so we do not know how many paths in total were added by boundary condition testing. This is not surprising as `tictactoe` was written by the author to serve as a worst case for boundary condition testing; virtually every branch will be expanded by our technique and none of them will help in fault identification because the program does not suffer from the path inadequacy problem. Even so, the slowdown in mutant kill speed is only a factor of 6.74-6.95×, which although far higher than the slowdown for the other benchmarks, is hardly exponential or catastrophic.

These results show that although boundary condition testing imposes a penalty on the mutant kill speed, it does not exponentially worsen mutant kill speed even though it may exponentially increase the search space. We find this result remarkable because we fully expected the slowdown to be punishingly high due to the exponential increase in the path space. We were surprised to find such reasonable slowdowns for targeted testing and even directed testing, which should have been impacted severely by the extra paths introduced by boundary condition testing.

Moreover, three of the benchmarks are particularly striking for their near-total lack of overhead. The `sjeng-core` and `eqntott` benchmarks are not negatively impacted by boundary condition testing for either directed or targeted testing. In fact, boundary condition testing improves the mutant kill speed of `eqntott` for directed testing and `sjeng-core` for targeted testing. Moreover, in the `printtokens` benchmark, in which only targeted testing found faults, boundary condition testing improved the mutant kill speed as well.

Table 6.6 compares the static proportion of branches changed to the relative slowdowns in directed and targeted testing. We instrumented the Bullseye compiler to report the total number of branches in the intermediate representation of the program as well as the number of branches or statement locations actually changed. The *Changed* and *Total* columns show the average number of branches changed by boundary condition testing versus the total number of branches. The *Ratio* column shows the average ratio between the changed branches and the total branches. The *Slowdown* columns repeat the relative slowdowns from Table 6.4 and Table 6.5.

These results show a relationship between the proportion of the branches changed and the slowdown. The `printtokens` benchmark has the smallest proportion of changed branches at around 8.6% and also has the lowest relative slowdown at 0.91×, which in this case is actually a speedup. The `sjeng-core` and `eqntott` benchmarks have changed branch proportions of around 22-31% and still have essentially no slowdown. After that, the slowdowns increase dramatically, shooting up to 2.67-4.75× for `tcas` at only 34.5% branches changed.

This pattern is interesting. For our set of benchmarks, as long as the proportion of branches changed remains below 1/3, the slowdown is minimal or negligible. Once the proportion rises above 1/3, performance falls off a cliff and the slowdown increases dramatically. However, higher changed branch proportions above 1/3 do not necessarily result in higher slowdown. While it is the case that the benchmark with the highest proportion of changed branches (`tictactoe` with 100% of branches changed) also has the highest slowdown, there is no correlation between the proportion of changed branches and higher slowdown once the 1/3 threshold is crossed.

The case of `tr` would seem to be an exception as only around 20% of the branches are changed but it still experiences a relative overhead of 3.78×. However, we can consider this benchmark an outlier. The `tr` slowdown is based on a single data point as targeted testing without boundary condition testing was only able to find a single fault. With boundary condition testing, targeted testing was able to find an additional two faults, but these slowdown ratios of course cannot be calculated.

Benchmark	Branches			BCT Slowdown	
	Changed	Total	Ratio	DFS	TGT
`triangle`	5.89	9.67	0.6175	2.28×	3.55×
`tictactoe`	25.5	25.5	1.0000	6.74×	6.95×
`tcas`	12.05	34.85	0.3456	4.75×	2.67×
`schedule`	18	41	0.4391	NA	NA
`printtokens`	7	81.5	0.0859	NA	0.91×
`polymorph`	7	18	0.3889	6.33×	1.98×
`sjeng-core`	60	275	0.2183	1.09×	0.93×
`eqntott`	68.22	219.67	0.3106	0.95×	1.15×
`tr`	43	207	0.2077	NA	3.78×
Average			0.4015	7.11×	2.74×

Table 6.6: The proportion of branches changed by boundary condition testing compared to the relative slowdowns.

For `schedule`, boundary condition testing does not help. Boundary condition testing does not help when the largest obstacle is simply getting the program on the right path; it can only help when the constraint solver fails to produce a fault-revealing input when it is already on the right path.

Conclusion Contrary to initial expectations, boundary condition testing need not seriously hamper mutant kill speed. If the proportion of changed branches is below 1/3, the slowdown is not significant. Above 1/3, the slowdown is higher but still far from exponential.

6.6 Context-Sensitivity

Another question we wish to answer is whether highly precise static analysis is necessary for achieving high mutant kill speeds. We answer this by disabling context-sensitivity in our static analysis for both pointers and for

Benchmark	Iteration Found				Efficiency		
	CI	TGT	DFS	Diff	CI	TGT	Diff
`tcas`	10.37	7.9	19.53	1.46×	0.9457	0.9324	1.0200
`schedule`	NA	NA	NA	NA	0.9164	0.9612	0.9552
`printtokens`	25.2	26.2	NA	0.96×	0.9919	0.9888	1.0032
`polymorph`	6.08	4.83	7.83	1.35×	0.8696	0.8414	1.0477
`sjeng-core`	16.83	14	21.67	1.41×	0.9286	0.9100	1.0229
`eqntott`	483.56	318	971.13	2.96×	0.9496	0.9512	0.9999
`tr`	252	78	NA	3.23×	0.8644	0.9035	0.9482
Average				1.90×			0.9996

Table 6.7: Targeted testing with context-insensitive static analysis compared with the default context-sensitive analysis.

Benchmark	Iteration Found				Efficiency		
	CI	TGT	DFS	Diff	CI	TGT	Diff
`tcas`	161.95	136.11	192.74	3.23×	0.9672	0.9759	0.9940
`schedule`	NA	NA	NA	NA	0.9225	0.9115	1.0138
`printtokens`	27	23.8	NA	1.13×	0.9890	0.9868	1.0022
`polymorph`	25.58	8.67	69.58	2.36×	0.9031	0.9076	0.9972
`sjeng-core`	15.38	15.38	23.75	1×	0.8866	0.8866	1.0000
`eqntott`	498	227.56	776.67	5.4×	0.9579	0.9539	1.0041
`tr`	308	100.33	NA	1.04×	0.8554	0.9131	0.9332
Average				2.36×			0.9921

Table 6.8: Targeted testing with boundary condition testing with context-insensitive static analysis compared with the default context-sensitive analysis with boundary condition testing.

our computation of interesting locations. The results are shown in Table 6.7. Note that two benchmarks, `triangle` and `tictactoe`, are omitted because both of these are single-procedure programs and therefore are not affected by context-sensitivity in the static analysis. Naturally, these results are for targeted testing, as directed testing does not use the static analysis results anyway.

The *Iteration Found* columns in Table 6.7 show the average iteration on which the fault is found for each of the benchmarks. The CI column shows the result for context-insensitive static analysis, while the TGT and DFS columns repeat the context-sensitive targeted testing and the directed testing results, respectively, for convenience. The Diff column is the average ratio between the mutant kill speeds of the context-insensitive and the context-sensitive versions of the static analysis.

Our results show that context sensitivity in the static analysis improves mutant kill speed. The average improvement for enabling context-sensitivity (or equivalently, the average slowdown for performing a context-insensitive analysis) is 1.90× across all benchmarks. The actual factor varies with the benchmarks, with 1.35-1.46× for the `tcas`, `polymorph`, and `sjeng-core` benchmarks, to a significantly higher 2.96× and 3.23× for the larger `eqntott` and `tr` benchmarks.

We also consider what happens when we enable boundary condition testing. These results are shown in Table 6.8. The average slowdown caused by context-insensitivity increases to 2.36×. However, the situation is slightly more complex as there is also greater variation between the benchmarks. For example, `tr` was significantly affected by context-insensitivity when boundary condition testing was not used, with an average slowdown of 3.23×. However, when boundary condition testing was enabled, these differences vanished. Similarly, although less impacted by context-insensitivity, `sjeng-core`'s 1.41× slowdown completely vanished when boundary condition testing was enabled. The slower average is caused by the other benchmarks, which saw their slowdowns go up with context-insensitivity.

While these slowdowns are non-trivial, they are also not catastrophic. Although often significantly slower than targeted testing with a context-sensitive analysis, they also perform better than directed testing, so benefits can be realized even without context-sensitive static analysis. The variation between the benchmarks suggests that the decision of whether or not to perform a context-sensitive analysis depends not only on the tradeoff between analysis time and performance, but also on the particular benchmarks themselves, as not all programs benefit greatly from context sensitivity.

Finally, we also compute the effect of context-insensitivity on the efficiency of Bullseye. The average ratio of the efficiency between context-insensitive and context-sensitive analysis across all benchmarks without boundary condition testing is 0.9996, which means that overall, context-sensitivity or lack thereof has no meaningful effect on the efficiency of Bullseye. When boundary condition testing is enabled, the average efficiency ratio becomes 0.9921, essentially unchanged. We may conclude that context sensitivity does not affect efficiency.

Conclusion Context sensitivity in the static analysis has a noticeable effect on mutant kill speed but lack of context sensitivity is not necessarily a dealbreaker. There is significant variation between the benchmarks, indicating that the benefits of performing a context-sensitive analysis are likely to be

Benchmark	Norm. Iter Found ($t =$)				Efficiency ($t =$)			
	0.1	0.3	0.5	1.0	0.1	0.3	0.5	1.0
`triangle`	1.76	1	1.3	1.32	0.9746	0.9248	0.9746	0.9808
`tictactoe`	1	1	1	1	0.9757	0.9756	0.9756	0.9756
`tcas`	1.02	1	1.17	1.01	0.9314	0.9324	0.9248	0.9314
`schedule`	NA	NA	NA	NA	0.9378	0.9612	0.9848	0.9956
`printtokens`	1.11	1	1.15	1.13	0.9787	0.9888	0.9888	0.9888
`polymorph`	1.71	1	2.06	2.97	0.7614	0.8414	0.8159	0.8157
`sjeng-core`	1	1	1	1	0.9100	0.9100	0.9100	0.9100
`eqntott`	2.22	1	2.74	2.75	0.9548	0.9512	0.9547	0.9556
`tr`	1	1	1	1	0.8379	0.9035	0.8552	0.8527
Average	1.35	1	1.43	1.52	0.9180	0.9321	0.9316	0.9340

Table 6.9: The effects of varying the threshold parameter on mutant kill speed and efficiency. The MKS values are normalized to the default threshold of 0.3.

program-dependent.

6.7 Heuristic Parameters: Threshold and Scale

We also wish to examine the sensitivity of our results to the constants used in our heuristic function. To answer this question, we varied the two main parameters in our heuristic function separately.

The *threshold* parameter determines which branch weights should be ignored. In some cases, the static analysis may produce branch weights that are very close, for example, something like 28/27. In these cases it may be better to ignore the branch direction completely as the static analysis does not indicate a strong preference for either side and because needlessly flipping branches decreases the efficiency of the system and may harm mutant kill speed. The threshold parameter controls which weights are ignored. If the ratio between the larger and the smaller weight exceeds $1+t$ for some threshold

Benchmark	Norm. Iter Found ($s =$)				Efficiency ($s =$)			
	1.0	2.0	5.0	10.0	1.0	2.0	5.0	10.0
`triangle`	1	1.46	1.22	1.04	0.9248	0.9746	0.9746	0.9746
`tictactoe`	1	1	1	1	0.9756	0.9756	0.9756	0.9756
`tcas`	1	0.99	1.05	1.04	0.9324	0.9316	0.9319	0.9325
`schedule`	NA	NA	NA	NA	0.9612	0.9956	0.9956	0.9955
`printtokens`	1	1.15	1.16	1.17	0.9888	0.9888	0.9888	0.9888
`polymorph`	1	1.86	1.85	2.25	0.8414	0.8328	0.8344	0.8050
`sjeng-core`	1	1	1	1	0.9100	0.9100	0.9100	0.9100
`eqntott`	1	2.65	2.64	2.85	0.9512	0.9477	0.9523	0.9431
`tr`	1	0.5	0.53	0.53	0.9035	0.8331	0.8247	0.7922
Average	1	1.33	1.31	1.36	0.9321	0.9322	0.9320	0.9241

Table 6.10: The effects of varying the scale parameter on mutant kill speed and efficiency. The MKS values are normalized to the default of 1.0

value t, then Bullseye will use the branch data; otherwise the branch is ignored. In other words, the threshold factor determines whether Bullseye should act on a particular mismatched branch.

For our main experiments, we use a default threshold value of $t = 0.3$, corresponding to a ratio of 1.3:1. This value was empirically determined by informal experimentation on our benchmarks and was used as it seemed to perform better than higher or lower values.

Table 6.9 shows the effect that varying the threshold parameter on mutant kill speed. We test the values 0.1, 0.5, and 1.0 against the default of 0.3. The mutant kill speed is normalized in the table to that of the default of 0.3. The results show that across all benchmarks, $t = 0.3$ is indeed at a "sweet spot" as lower or higher values degrade mutant kill speed by 35-52%.

This result is not surprising, both because the value $t = 0.3$ was already experimentally determined to be good and because we expect a U-shaped curve for intuitive reasons. Intuitively, we would expect that very low values for threshold would result in poor performance, as Bullseye would spend time flipping branches for which the weights were almost even, reducing efficiency and possibly hurting mutant kill speed. In fact, this is exactly the case, as Table 6.9 shows that mutant kill speed is affected and the efficiency is lowered. We also expect high threshold values to perform poorly as they would ignore useful branch data. With an infinite threshold, all branch data is ignored and Bullseye degenerates into a depth-first search, so we would expect efficiency to at least be unaffected as we increase threshold beyond the default. This is also the case; mutant kill speed is harmed as threshold increases and the efficiency remains essentially flat.

The `scale` parameter controls how aggressively Bullseye promotes branch flips that deal with mismatched directions. If the ratio exceeds the threshold defined earlier and the branch does not go in the same direction as the branch data indicates it should, Bullseye computes a weight that roughly indicates how many branches forward it should promote the mismatched branch for branch selection. The scale parameter is multiplied against this to further increase the priority of the mismatched branch. In other words, the scale factor determines how strongly Bullseye should act on a branch once it decides that it may need to be altered.

We developed our heuristic formula with a default scale factor $s = 1.0$. Values below 1.0 can actually result in weakly mismatched branches being given lower priority than branches with no data at all and is thus nonsensical. Higher values for scale greatly increase the priority of mismatched branches; if our default heuristic increases the priority of a branch by 2, a scale factor of 10.0 would increase the priority by 20.

Table 6.10 shows the effect of varying the scale parameter on mutant kill speed and efficiency. Surprisingly, the initial value of 1.0 for scale turned out to be the best overall value. This means that in the majority of cases, increasing Bullseye's aggressiveness about flipping earlier branches is actually counterproductive.

This effect does not hold for all of the benchmarks, however. Most strikingly, the `tr` benchmark benefits strongly from a higher scale value. Increasing scale from $s = 1.0$ to $s = 2.0$ causes it to find the fault twice as quickly. The `tcas` and `sjeng-core` benchmarks are not meaningfully affected by scale at all.

Overall, there is a clear relationship between sensitivity to threshold and sensitivity for scale; benchmarks that are sensitive to one are also sensitive to the other. The `tictactoe`, `tcas`, `printtokens`, and `sjeng-core` benchmarks are not strongly affected by either threshold or scale. The `triangle`, `polymorph`, and especially `eqntott` are strongly affected by both threshold and scale. The `tr` benchmark is the only exception to this pattern; it is completely unaffected by threshold and strongly affected by scale. However, this result is based on a single data point because Bullseye could only find 1/5 faults in `tr`, so this single data point may not be meaningful.

6.8 Observations and Insights

By examining detailed debugging output, we have gained insights about the behavior of our system and about techniques for effective fault identification.

6.8.1 Targeted Testing and Variance

Although we did not gather data for all trials, only the median, we observe anecdotally that Bullseye is significantly more predictable than depth-first search with respect to mutant kill speed. While depth-first search will often have outliers both good and bad, Bullseye's grouping tends to be more clustered around the median.

This difference is because depth-first search's performance is dependent on luck; it performs well if the initial input is near a faulty path and poorly if the initial input is far from any faulty path. Bullseye reduces the impact of chance by backing out rapidly when the initial input is unfavorable, leading to greater consistency in mutant kill speed. We believe that in larger testing scenarios, this greater consistency in mutant kill speed is beneficial, on top of the strong raw improvements in mutant kill speed afforded by targeted testing.

6.8.2 Exponential Explosion and Mutant Kill Speed

One of the most surprising lessons learned from our work is that exponential techniques may be far more viable than previously believed. We believed initially that the boundary condition transformation would have punishingly high slowdowns and should only be used as a matter of last resort when faults are being missed due to the path inadequacy problem. However, our results show that the effect on mutant kill speed, while nontrivial, is far from an exponential slowdown. In fact, as long as the proportion of changed branches stays below 1/3 of the total branches, the slowdown is quite minimal.

The difference between exponential explosion in the space of paths and exponential explosion in mutant kill speed is easier to distinguish in hindsight. While intuitively one might expect any technique that exponentially increases the number of paths to affect mutant kill speed by a proportional amount, this does not have to be the case. Even in the `tictactoe` benchmark, which was specifically designed to be a worst-case with 100% of the branches being changed, the slowdown in mutant kill speed was less than 7× while the increase in the number of paths was well over 100× (the exact amount could not be determined as it was too large for Bullseye to explore). We did not anticipate this dichotomy because prior work focuses heavily on achieving high coverage, which of course would be negatively impacted by our technique in exactly the expected way. However, the goal of targeted testing is fault identification, not coverage, and so boundary condition testing becomes much more viable.

One possible avenue to pursue for future work would be an investigation into other techniques in software testing or model checking that also exponentially expand the space of paths. While such techniques obviously still remain impractical if the goal is to explore the entire state space, our results suggest that these exponential techniques may be much more viable if the goal is rapid fault identification.

6.8.3 Program-Library Interaction

One significant difficulty encountered is in dealing with the interaction between library-like code and the application code. Bugs in the library code can be easily masked by the way in which the application uses it. For example, the `schedule` benchmark contains a small library for managing priority queues in addition to the core scheduling code. We can introduce obvious errors into the priority queue code that could be easily unmasked with trivial unit testing. However, if the code is being used in the context of the scheduler, finding the fault becomes harder. We not only have to find a way for the scheduler to use the priority queues in a way to trigger the bug, but also to ensure that the effects of the bug result in different final output. For example, the priority queue may make mistakes in assigning priorities, but as long as the final order of the jobs does not change, no fault is exposed. This difficulty can mask serious bugs that would be exposed with even simple unit testing. Thus, any whole-program testing system must consider adopting some unit-testing techniques to ensure that all bugs are found.

6.8.4 Library Code and Symbolic Execution

Because most constraint solvers do not handle a theory of strings, directed testing systems support string operations indirectly by including an implementation of the string functions with the main program. Upon close observation, we find that this *severely* limits the ability of a testing system to choose new paths. For example, symbolically executing an implementation of `strcmp` can prevent the testing system from rapidly altering the string. Because the path through `strcmp` is now part of the path through the whole application, changing the result of a string compare requires changing a significant portion of the path across many branches and comparisons. Similar problems occur if the testing system is reasoning about paths through a red-black tree implementation rather than the set operations that they implement.

Future work on this problem can proceed in at least two ways. The first is to simply move to more powerful theories in constraint solvers. This also has the added benefit of eliminating many cases where the results are semantically equivalent but take different paths through the implementation. The second would be to investigate mechanisms for selectively ignoring certain portions of the path. For example, we can construct an annotation system that would tell

Bullseye to not track constraints or branches within certain functions. This would prevent symbolic execution from being bound to the concrete paths in library-like code, but could also lead to unexpected path divergence. The design tradeoffs and empirical experiences would make for interesting future study.

6.8.5 Global Data Structures and Static Analysis

The failure of Bullseye and Bullseye with boundary condition testing to find any of the faults in the `schedule` benchmark raises questions about the fundamental limits of our technique. One key difficulty are program design patterns that place the program in a "command loop" where each of the commands operates on some global data structure. Because every part of the program operates on the global data structure, if the data structure is considered interesting, Bullseye (or any other static analysis) can only determine that all parts of the program are interesting. This largely eliminates the benefit of static analysis as it is no longer able to properly prioritize among branches and statements.

6.8.6 The "Data Inadequacy Problem"

We encountered in the `sjeng-core` and especially the `tr` benchmarks an extremely nasty variant of the path inadequacy problem that we can call the data inadequacy problem. This problem arises when semantically different decisions can be made using data while not being reflected in the paths the program takes.

The example of `tr` is the most illustrative of this problem. The last part of `tr` is a simple loop that reads one character at a time from the input stream. This character is used as an index into a table that was constructed in the earlier part of the program and is used to find the character that it translates to. For any input of size n, this loop will run n times and will be the only control flow in this part of the program. In order to expose a fault, the input stream must contain characters that index into the affected parts of the translation table. Unfortunately, there is absolutely no difference in the path the program takes when it reads a fault-revealing character or when it reads an unaffected character. Because this last loop is the only time there is any connection between the input stream and control flow, there is simply no way for any symbolic execution system, even with boundary condition testing, to explore different characters in the input stream as they all lead to equivalent paths.

One possible but undesirable solution would be to recognize this idiom and translate these table lookups into a massive n-way switch statement. This makes each lookup in the table follow a different path, making these states accessible to symbolic execution. However, such a technique is certain to be costly and also does not allow for methods for static analysis to distinguish between the new synthetic paths. In effect, it forces symbolic execution to simply iterate blindly over the space of inputs that result in equivalent paths.

Chapter 7

Conclusion

In this thesis, we have implemented and evaluated techniques for improving dynamic analyses by utilizing static data flow analysis. We have applied our philosophy and approach to two widely separate areas: runtime security policy enforcement and software test input generation.

The central theme behind this thesis is that dynamic analyses, defined broadly, contain inherent inefficiencies due to an inability to properly see and reason about the problem space at hand. This handicap causes the dynamic analysis to waste time on tasks that are ultimately irrelevant to the goal but are performed because the dynamic analysis does not know that it is irrelevant.

Our key insight and contribution is that these inefficiencies can be addressed by first performing a static data flow analysis that computes information that is specific to both the problem at hand and the program that it is being applied to. In our two problem domains, the specific information is information about the *future behavior of the program*, which is not available to dynamic analysis but can be attacked by static analysis. In taint tracking, the future information required to eliminate inefficiencies is whether or not some piece of data will be used at a specific place in the future. In testing, the information is whether or not a particular candidate path is likely to reveal a fault. In both cases, static analysis can supply this information, with absolute or probabilistic assurance, allowing the dynamic system to avoid needless work.

We will now briefly recap some high-level points and lessons learned from our two systems. Additional commentary and thoughts on future work specific to the systems can be found earlier in the discussion portion of our evaluations for each of the systems; insights and future directions specific to dynamic data flow analysis are discussed in Section 4.4, while insights and observations specific to test input generation can be found in Section 6.8.

7.1 Summary: Dynamic Data Flow Analysis

In our dynamic data flow analysis work, we were able to build a system for enforcing user-defined security policies at runtime. This system is not only far more general than existing taint tracking systems, but is also one to two orders of magnitude more efficient, with overhead that is essentially negligible.

In the case of dynamic data flow analysis, the inefficiencies result from the fact that dynamic taint analysis (and other similar analyses) track a large amount of information about the program, but only a vanishingly small fraction of it is ever used. In a server program, much of the data the program manipulates will wind up being tainted, which would necessitate extensive tracking by a taint tracking system to ensure that no tainted items are missed. Although almost none of this tainted data is used unsafely, a purely dynamic analysis cannot "see into the future" and know that it is computing facts that will never be used. Static analysis is able to help because it can see into an approximation of the future; this is used to rule out possibilities that it can prove are irrelevant. Rather than tracking information for every object in memory, our system only tracks those that are involved in a policy violation.

Our use of static analysis to address the inefficiency resulting from dynamic taint analysis's inability to reason about future actions leads to great improvements in performance. We were pleasantly surprised to find that our overhead was so low. While we naturally expected significant improvement over taint tracking systems, our average overhead of 0.65% for server programs is stunningly low, essentially within measurement noise. When testing compute-bound programs, we were even more surprised: our static analysis can completely eliminate the possibility of vulnerabilities, leading to an overhead of 0%. In order to evaluate our system on compute-bound applications, we had to insert synthetic vulnerabilities that our static analysis could not eliminate. Even so, our overhead remains quite low, with our worst overhead being better than the best overhead of previous comparable systems.

As an added bonus, our system is also far more general than standard taint tracking systems, matching the expressive power of General Information Flow Tracking [63]. As discussed in Section 4.4, our generality comes directly from our use of data flow analysis as opposed to other possible models for specifying policies. Using dynamic data flow analysis for the policy allows us to easily perform static data flow analysis and easily ensure that the static analysis is an appropriate conservative over-approximation. Moreover, we are pleased to note that using a more complex policy does not necessarily result in higher overhead. The file disclosure vulnerability we evaluate in Section 4.3 would have around twice the space and time overhead of taint tracking if implemented naively because it tracks two separate properties instead of one. However, the actual overhead is only 0.25% on average, lower even than taint tracking. This is because our static analysis could identify the paths involved in any vulnerabilities, which were even sparser for this more complex example.

Decoupling complexity and overhead is an unique advantage of our hybrid analysis approach.

7.2 Summary: Targeted Test Input Generation

For test input generation, we implement and evaluate a system, called Bullseye, that uses static analysis to guide a dynamic search of program paths for bugs. We find that Bullseye finds more bugs than directed testing and in the cases where directed testing was able to find a bug, Bullseye finds it significantly faster, on average 2.5× faster. We also develop and evaluate an orthogonal technique that addresses the inadequacy of paths in revealing faults. This boundary condition testing technique allows any system based on symbolic execution to communicate boundary conditions to the constraint solver with no additional changes by simply encoding the boundary conditions in the control flow graph. We show that boundary condition testing allows both directed and targeted testing to find numerous bugs that they could not before, and that the slowdowns incurred by introducing an exponential number of new paths is actually quite reasonable. We have also investigated other factors, such as parameters for the search heuristic function and the role of precision on static analysis on fault-finding speed.

The inefficiency in testing stems from a rather different source than in dynamic data flow analysis. Where the inefficiency in dynamic data flow analysis was due to tracking information that will never be needed, the inefficiency in testing is in running test inputs that reveal no faults. To find faults faster, the search must ignore or defer the many paths that do not reveal faults in favor of those that are likely to reveal faults. Bullseye enables this with the concept of "interesting points" that are then used to perform a static analysis that allows Bullseye to generate inputs that are highly relevant to the interesting points. As long as the interesting points are useful in fault identification, Bullseye can find faults much faster by steering the search away from irrelevant paths.

Again, data flow analysis is important to Bullseye. Previous efforts in dynamic test case generation have focused on control flow, with the goal of forcing execution to reach a particular point. Unfortunately, control flow alone does a poor job of capturing what affects what in a program. If a value is incorrectly computed and then used much later in the program, control flow analysis will not be able to see the connection. In Bullseye, we first perform a data flow analysis so that effects can be fully seen, followed by a control flow analysis to actually drive execution to those points.

7.3 The Future of Dynamic Analysis

As fixing bugs and security vulnerabilities become increasingly high priorities for developers, we expect a continual increase in the availability and power of dynamic tools that help programmers find and fix these errors. This thesis has shown how static analysis can be used to greatly improve the performance of dynamic analyses by giving the dynamic analysis a chance to avoid doing work that can be shown to be unnecessary in the future. While we have had great success in the two areas of runtime security policy enforcement and test input generation, we believe that this pattern can be fruitfully extended to other dynamic analyses as well.

Consider, for example, memory profiling and analysis tools such as Valgrind [78]. While it is an indispensable tool for programmers debugging memory errors and leaks, it also has hideously high overheads. These overheads come from the need to track every memory operation and every byte of memory, even though most of these are not involved in leaks or overruns. With static analysis, much of this instrumentation could be avoided, leading to lower overhead. Moreover, there are tasks like performance profiling and other such studies where absolute correctness is not required as it is with security. In these cases, static analysis may help with intelligent sampling approaches and other such approximation techniques.

Another important category of tools are systems designed to find concurrency errors. There are numerous dynamic tools for finding races and deadlocks, but all must deal with the immense number of possible orderings, most of which do not reveal any errors. Recent work uses partial order reduction to eliminate equivalent interleavings among threads [28], reducing the search space and improving the ability to find errors caused by interleavings. Indeed, there are many other such inefficiencies in concurrency testing that can be eliminated by a variety of techniques that exploit some form of reasoning about the future to avoid inefficiency.

Finally, we believe the complementary nature of static and dynamic analysis should be and will be increasingly exploited in the future. Static analysis can give approximate information across the entire space of possible program executions, while dynamic analysis delivers the complement—highly precise information but only for the current execution. By marrying the two, static analysis can be used to reduce the overhead of dynamic analysis or dynamic analysis can be used to address the imprecision in static analysis. Current test generation techniques, for both single-threaded [18] and concurrent programs [28], draw inspiration from model checking techniques. Language-based approaches to security rely on a mix of static verification and dynamic enforcement [74]. Moving forward, we believe that static and dynamic analysis will be recognized and treated as two sides of the same coin, instead of as different fields entirely.

Bibliography

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the ACM Conference on Computer and Communication Security*, 2005.

[2] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, 1979.

[3] Hiralal Agarwal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[4] James. H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, pages 402–411, 2005.

[5] Tomoyuki Aotani and Hidehiko Masuhara. SCoPE: An AspectJ compiler for supporting user-defined analysis-based pointcuts. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 161–172, 2007.

[6] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[7] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[8] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering*, pages 332–341, 2005.

[9] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 251–262, 2000.

[10] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, MITRE, March 1973.

[11] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabalistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

[12] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, February 2010.

[13] Boris Bezier. *Software Testing Techniques.* International Thomson Computer Press, second edition, 1990.

[14] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[15] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, 2005.

[16] Ken J. Biba. Integrity considerations for secure computer systems. Technical Report ES-TR-76-372, Electronic Systems Division, Hanscom Air Force Base, April 1977.

[17] Jacob Birnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd IEEE/ACM Conference on Automated Software Engineering (ASE)*, pages 443–446, 2008.

[18] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[19] Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, 2004.

[20] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th USENIX Security Symposium*, 2007.

[21] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. Technical Report CSTR-2005-04, Computer Systems Laboratory, Stanford University, 2005.

[22] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson Engler. EXE: Automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communication Security*, 2006.

[23] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[24] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 378–387, 2005.

[25] Pavan Kumar Chittimalli and Mary Jean Harrold. Recomputing coverage information to assist regression testing. In *Proceedings of the International Conference on Software Maintenance*, pages 164–173, 2007.

[26] Edmund Mellon Clarke, Oma Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[27] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, 2000.

[28] Katherine E. Coons, Sebastian Burckhardt, and Madanal Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *Proceedings of the 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programing*, 2010.

[29] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rwostron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. *ACM SIGOPS Operating System Review*, 39(5), December 2005.

[30] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[31] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, 2003.

[32] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[33] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 221–232, 2004.

[34] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th International Symposium on Computer Architecture*, 2007.

[35] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[36] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[37] Dawson Engler and Ken Ashcraft. RacerX: Effective static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 237–252, 2003.

[38] Dawson Engler and Madanlal Musuvathi. Static analysis versus model checking for bug finding. In *Proceedings of 5th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 191–210, 2004.

[39] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, 1999.

[40] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[41] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[42] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, 2002.

[43] Phyllis Frankl and Elaine Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[44] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[45] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[46] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Generating test data for branch coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 219–227, 2000.

[47] Samuel Z. Guyer. *Incorporating Domain-Specific Information into the Compilation Process.* PhD thesis, The University of Texas at Austin, Austin, Texas, 2003.

[48] Samuel Z. Guyer, Daniel A. Jimenez, and Calvin Lin. Using C-Breeze, 2002.

[49] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 39–52, 1999.

[50] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *10th Annual Static Analysis Symposium*, June 2003.

[51] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE, Special issue on program generation, optimization and adaptation*, 93(2):342–357, January-February 2005.

[52] William G. J. Halfond and Alessandro Orso. Amnesia: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183, 2005.

[53] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

[54] J. C. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, SE-5(3):226–236, May 1979.

[55] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Securing web application code by static analysis and

runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, pages 40–52, 2004.

[56] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, 2002.

[57] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–176, January 1976.

[58] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001.

[59] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, 1997.

[60] Dave King, Boniface Hicks, Trent Jaeger, and Michael Hicks. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[61] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Porceedings of the 11th Annual USENIX Security Symposium*, pages 191–206, 2002.

[62] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[63] Lap Cheung Lam and Tzi-Cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006.

[64] Nancy Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[65] Steven Levy and Brad Stone. Grand theft identity. *Newsweek*, September 5 2005.

[66] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for evaluating bug detection tools. In *Bugs 2005 (Workshop on the Evaluation of Software Defect Detection Tools) in Programming Languages Design and Implementation*, 2005.

[67] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[68] James R. Lyle, Mary T. Laamanen, and Neva M. Carlson. PEST: Programs to evaluate software testing tools and techniques. National Institute of Standards and Technology, Information Technology Laboratory.

[69] Ofer Maor and Amichai Shulman. Sql injection signatures evasion. *Imperva white paper*, April 2004.

[70] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering*, 2001.

[71] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the 1st Asian Symposium on Programming Languages and Systems*, 2003.

[72] Sean McDirmid and Wilson C. Hsieh. Splice: Aspects that analyze programs. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, pages 19–38, 2004.

[73] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, first edition, 1997.

[74] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[75] National Aeronautics and Space Administration. Mars Climate Orbiter Mishap Investigation Board Phase I Report, November 1999.

[76] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. *Planning Report*, 02(3), May 2002.

[77] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[78] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the 3rd Workshop on Runtime Verification*, 2003.

[79] James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the Network and Distributed Security Symposium*, 2006.

[80] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed Security Symposium*, 2005.

[81] Anh Nguyen-Tong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, 2005.

[82] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 241–252, 2004.

[83] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test data generation using genetic algorithms. *Software Testing, Verification, and Reliability*, 9(4):263–282, 1999.

[84] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead information flow tracking sys-

tem for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM Symposium on Microarchitecture*, pages 135–148, 2006.

[85] Sandra Rapps and Elaine Weyuker. Selecting software test data using dataflow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[86] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[87] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 263–272, 2005.

[88] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the USENIX Security Symposium*, pages 201–218, 2001.

[89] Manu Sridharan, Stephen J. Fink, and Ratislav Bodik. Thin slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122, 2007.

[90] Brandon Streiff. Bullseye: A system for targeted test generation. Undergraduate Honors Thesis, The University of Texas at Austin, Austin, Texas, May 2008.

[91] Robert Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[92] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[93] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[94] John Viega, J. T. Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2), February 2001.

[95] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, Sebastopol, California, United States, third edition, 2000.

[96] Gary Wasserman and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[97] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[98] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 39th International Conference on Dependable Systems and Networks*, 2009.

[99] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[100] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 243–257, 2006.

Vita

Walter Chang was born in Chicago, Illinois, in 1981. He grew up in Bloomington, Indiana and graduated from Bloomington High School South with additional coursework at Indiana University. He graduated from Cornell University in 2002 with a Bachelors in Computer Science and a minor in Science and Technology Studies and has also interned with IBM Research in Almaden and IBM Server Group in Endicott. He spent 2002-2010 at the University of Texas at Austin pursuing a Ph.D. in Computer Science and will start a job at Google in the fall of 2010.

Permanent address: 707 Ramble Lane Unit B
Austin, Texas 78745

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.