Static analysis and computer security:
New techniques for software assurance

by

David A. Wagner

M.S. (University of California at Berkeley) 1999
A.B. (Princeton University) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Eric A. Brewer, Chair
Professor Alexander Aiken
Professor Hendrik Lenstra
Professor Christos H. Papadimitriou

Fall 2000

The dissertation of David A. Wagner is approved:

Chair   Date

Date

Date

Date

University of California at Berkeley

Fall 2000

Static analysis and computer security:
New techniques for software assurance

Abstract

Static analysis and computer security:
New techniques for software assurance

by

David A. Wagner
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Eric A. Brewer, Chair

One of the greatest challenges in computer security today is the software assurance problem: How do we deal with the fact that our most trusted software, even our security software itself, is often buggy? Our programming environments are flawed, which makes it easy to introduce bugs into security-sensitive applications. At the same time, a widespread reliance on legacy applications, developed in a more trusting age where security was less of a concern, raises the stakes. All of these factors suggest that the software assurance problem is likely to remain an important challenge for some time to come.

This dissertation argues that static analysis can be a powerful tool for software assurance, providing a new approach to the problem. To validate this methodology, we describe experience with two test applications. First, we describe new techniques for automated buffer overrun detection; a prototype implemenation has been used to find serious new vulnerabilities in a large, widely deployed software package (even though it had already been hand-audited). Second, we examine intrusion detection, showing how static analysis allows us to detect potential break-ins without raising any false alarms. In both cases, a key selling point of static analysis is that it allows us to eliminate or neutralize security bugs proactively before they are exploited.

These two success stories provide strong support for the methodology, and they

suggest that static analysis may prove a powerful tool for improving the assurance level of security-sensitive software in the future.

Professor Eric A. Brewer
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

This dissertation has benefitted tremendously from the discussion, wisdom, and advice of many individuals.

First and foremost, I am deeply indebted to Eric Brewer for his direction and advice. Eric's impact on my research has been tremendous and invariably positive. Eric has been not only an insightful research advisor but also a valued teacher, collaborator, colleague, and mentor, and I am incredibly fortunate to have had the opportunity to work closely with him.

I have also had the great fortune to benefit from the mentoring of several others: Steve Bellovin first introduced me to research in computer security during an internship at AT&T Bell Labs, and his guidance throughout my career has been invaluable; Bruce Schneier's insightful advice has guided me in my studies of applied cryptography and has helped me to join that research community; Alex Aiken has generously shared his wisdom in programming languages as I studied static analysis for this dissertation; and my officemates in 445 Soda—Ian Goldberg, Nikita Borisov, Steve Gribble, Armando Fox, Yatin Chawathe, and Mike Chen—have served as peer mentors and collaborators throughout my studies. I consider myself extraordinarily privileged to have the chance to interact with these individuals, and I am grateful for all that they have done to educate and enlighten me.

I am also grateful to a number of readers whose comments have substantially improved this dissertation. I would especially like to thank Alex Aiken, Steve Bellovin, Nikita Borisov, Eric Brewer, Yatin Chawathe, Crispin Cowan, Drew Dean, Manuel Fähndrich, Jeff Foster, David Gay, Ian Goldberg, Steve Gribble, Alan Hu, Gary McGraw, George Necula, Christos Papadimitriou, Adrian Perrig, Dawn Song, Thu Trinh, John Viega, and several anonymous reviewers for useful discussions about this work.

Special thanks are due to Alex Aiken, Eric Brewer, Drew Dean, and Jeff Foster, my collaboraters on some of this work.

I gratefully acknowledge the BANE research group for providing me with access to the BANE front-end (which was used to implement the buffer overrun detection tool) and David Gay for providing his `gcc`-based compiler (which was used to implement the intrusion detection system). Their program analysis infrastructure was an incredibly valuable resource in implementing the ideas described in this dissertation, and it saved me months of implementation work.

During my graduate studies I have been supported by the California Microelectronics Fellowship as well as funding from the National Science Foundation (NSF), Defense Advanced Research Projects Agency (DARPA), and industrial donors such as AT&T Bell Labs. Their support has made this work possible, and I am grateful for their generosity.

Finally, I would like to thank Simona; her unwavering support, patience, and encouragement throughout the often-stressful dissertation work has made it all worth it.

Chapter 1

Introduction

As we enter the so-called "information age" of global networks, ubiquitous computing devices, and electronic commerce, many businesses, consumers, and other users are becoming increasingly concerned about computer security. Yet the current state of computer security is lamentably poor in practice. One survey found that nearly 2/3 of Internet hosts are vulnerable to unsophisticated, well-known, easy-to-exploit attacks [57], and the number vulnerable to more clever or more recent attacks is presumably greater still. Many studies have shown that business losses are large and increasing [86]. Anecdotally, breakins are rampant, and hackers discover new vulnerabilities almost every day. There are several causes for these trends.

First, today's systems rely heavily on applications built in an age when security did not receive the same attention it does today. The Internet, once populated almost exclusively by cooperating researchers, and local-area networks, populated exclusively by local co-workers, spawned many legacy applications that were originally designed for use only in a friendly environment. This legacy code is unlikely to go away anytime soon, yet the threats it must defend against have changed dramatically: global networks now expose us to a much broader array of adverse interests, including hackers, vandals, competitors, and other untrustworthy entities. Nonetheless, we need some way to protect our data from a growing number of unfriendly and potentially malicious computer users even as we continue to rely on irreplaceable legacy applications.

This leaves us in a quandary with no obvious solution.

A second contributing factor is that building secure systems is fundamentally hard, and today's programming environments do not make the task any easier; in fact, they often make the task significantly harder.

- The operating system does not provide any way for applications to specify just the subset of privileges they actually need, and as a result when an application is compromised, the intruder typically obtains full access to the entire system.
- The Unix system-call interface does not provide any simple way to atomically query a system object for permission information before performing some operation on that object, and as a result exploitable race conditions are common.
- The language—almost invariably, C—does not assure memory safety, and as a result it may be easy to cause memory errors and force a security-critical application to crash.
- The standard C libraries provide string buffer management primitives that are unsafe, and as a result our programs fall prey to buffer overrun vulnerabilities.

In all cases, it is in principle possible to code the desired functionality in a secure way, but the secure way is often not the easiest, best-supported, most convenient, most portable, or standard way, and it is not always feasible to fix problematic aspects of the programming environment. Of course, when the standard primitives have security pitfalls, it becomes too easy for developers to inadvertently introduce security holes where these primitives are used. A symptom of this problem may be recognized in the frequency with which programs are found to contain the same mistakes again and again.

A third common cause of insecurity is that many of our security-critical applications are large and complicated. Complexity breeds subtle interactions, subtle bugs, and thus subtle security holes [30]. Moreover, sheer size can make the source code so unwieldy that it becomes very difficult to review the application for potential security errors, or even to understand exactly what it is doing. As a result, large, complex applications often go largely unscrutinized, despite their security-critical nature. One symptom of this phenomenom can be found in the growing number of vulnerabilities that have lain dormant in poorly reviewed source code for years before being discovered by the security community.

In essence, then, we have a software quality assurance problem with serious implications for computer security: How do we deal with the fact that our most trusted software, even our security software itself, is often buggy? Re-design and re-implementation of all security-critical code of questionable quality does not appear to be a viable option at the moment; it is simply too costly to be a general solution. As a consequence, we are stuck with an overwhelming amount of legacy and other code of questionable security, and we have no good way to check it for even the simplest, most common classes of coding errors.

At this point, some readers might suspect that the answer to these woes lies in careful, manual review of all security-critical code. Indeed, we would certainly agree that code inspection is essential to security. However, the root problem is that manual inspection is extremely time-consuming, and the amount of legacy code that would require auditing is enormous. As a result, code review is not applied as often or as thoroughly as it is needed. A second problem is that the sort of common, mindless mistakes that are easy for a programmer to make are also typically easy for a reviewer to overlook, and thus we might hope for an even higher level of assurance than manual code review can provide by itself. These comments suggest that we might do well to look for tools to help automate the process, to reduce the burden on the reviewer, to raise the assurance level, and in general to reduce the cost of the security quality assurance process.

1.1 Outline of this work

In this dissertation, we study a new class of automated techniques for getting a grip on the problem of software assurance for security-critical applications. For newly written applications, we show how to detect an important class of software vulnerabilities early in the software development cycle. Because these methods work by analyzing the program's source code, they may also be applied to analyzing the legacy applications long after their initial deployment. Additionally, we show a generic and powerful technique to detect intrusions that occur at runtime, even when these intrusion attempts use unanticipated and novel attack techniques. We demonstrate the power of the proposed methodology by applying our tools to real-world systems of considerable importance to computer security.

Our methodology revolves around the application of static analysis techniques from the programming languages literature. A key selling point of static analysis (i.e., compile-time analysis of source code) is that it permits *proactive* detection and prevention of security holes, so that vulnerabilities can be proactively eliminated or neutralized before they are exploited.

Static analysis allows us to build a model of the program that highlights the features that are of greatest importance to security. Ideally, such a model should fulfill several goals:

- The model should allow convenient *programmatic manipulation*, so that tools can work in a heavily automated fashion, with as little human assistance as possible.
- The model should be *faithful*, so that we can have a high level of assurance that what we have analyzed corresponds closely to what the application actually does.
- The model should permit *semantic global analysis* of whole-program properties, because many security bugs are global and semantic in nature.
- The model should be *lightweight*, so that it can be constructed and manipulated efficiently, even for large, complex programs, which are the ones at greatest risk for security weaknesses and thus the ones of greatest interest for deep analysis[1].

[1] We will see some evidence for the importance of treating large programs later, where we show several examples of bugs found by our analysis tool despite prior manual audits of the program source code, apparently because the source code was too unwieldy to permit easy inspection.

The burden on the programmer should also be light, and the importance of a migration path should not be forgotten. For example, developers should not be required to add extensive annotations to their code before they can see any benefit from the system.

- Finally, the process of deriving and analyzing this model should be *software lifecycle friendly*, so that it can be applied to new software without adversely impacting time to market or development costs, and, by extension, so that it can be applied to legacy code as well.

Static analysis meets these requirements admirably well. A powerful benefit of static analysis is that the field enjoys a rich literature of previous work, built up over decades of interest in program optimization and other problems, so that there are a wealth of techniques for efficient and scalable programmatic manipulation of analysis results. These powerful algorithms enable us to use property inference rather than program annotation, so the burden on the user is reduced. As a second important benefit, the field of static analysis has been put on firm mathematical foundations, so that we can—if we like—apply these results to gain a very high level of assurance that our model is faithful, a feature that is relatively rare in computer security. Finally, the compile-time nature of this approach allows programmers to apply our tools early and often, if they like, or in parallel with initial implementation, to avoid impacting time to market, or long after deployment, for legacy code. These properties make static analysis very well-suited to the software assurance problem.

This work contains a synthesis of ideas from several fields, including program analysis, theory, and systems security. As a consequence, we organize the technical presentation of the dissertation into two main parts: the systems issues, and the theoretical foundations.

The first half of this dissertation contains applied work on two test applications of particular interest to practical computer security, as a way to demonstrate the utility of the proposed methodology. Chapter 2 presents a new technique for automated detection of potential buffer overrun vulnerabilities in security-critical source code, showing a close connection to the integer range analysis problem. Then, Chapter 3 considers the intrusion detection problem, showing how static analysis may be used to detect even future, unforeseen attacks; one unique feature of our approach is that it allows one to eliminate all false alarms, so that every observed alarm is guaranteed to represent a true intrusion attempt. We also report on experience with prototype implementations of both these techniques. Experimental results are encouraging.

The second half of this dissertation presents theoretical work on the formal underpinnings of the applied work found in the first half, as well as providing supporting algorithmic results needed to build these tools. Chapter 4 presents a precise and efficient new algorithm for the integer range analysis problem, as needed by our buffer overrun detection tool. Next, Chapter 5 describes the design of a novel top-down algorithm for recognizing general context-free languages, as used in our intrusion detection system. Both of these chapters also present theoretical results that may be of broader interest to the static analysis, formal methods, parsing, and algorithms communities.

We summarize the main contributions of this dissertation as follows:

- We introduce a methodology for software assurance, based on using whole-program static analysis to efficiently discover semantic properties of security-sensitive systems. We present empirical evidence from two trials of this methodology, and both provide strong support for the approach. This application of techniques from the field of program analysis to computer security appears to be a promising new direction for future research.

- We show how the automated buffer overrun detection problem may be formulated as an integer range analysis problem. Our experience suggests that this approach provides an order of magnitude improvement over the state of the art in detecting buffer overrun vulnerabilities, a very important class of vulnerabilities which account for close to half of all security holes reported in recent years.

- We introduce a new approach to intrusion detection, showing how static analysis may be used to build a provably sound model of application behavior and in this way provide new techniques for handling one of the primary challenges in intrusion detection.

- We demonstrate the surprising result that useful intrusion detection without false alarms is achievable; indeed, it comes as a direct consequence of applying our program analysis-based methodology to the intrusion detection problem.

We also demonstrate several results that may be of independent interest:

- We introduce a new class of attacks on host-based intrusion detection systems, the *mimicry attack*. Mimicry attacks are based on the unavoidable threat (heretofore largely ignored in many previous works on the subject) that attackers are free to adapt their attacks to the form of whatever intrusion detection system has been deployed. We argue that intrusion detection systems should be evaluated according to their level of resistance to mimicry attacks, and we observe that our static approach to intrusion detection has the interesting feature that it appears to provide a high level of robustness against mimicry attacks and even against future, unforeseen attack techniques.

- We develop a general new technique for efficiently and precisely solving fixpoint equations over lattices of infinite height, when certain technical conditions are satisfied. As a special case, we obtain a precise and efficient algorithm for range analysis.

- We characterize the structure of a pushdown automaton's stack after it processes a specified input string, extending the work of [23, 27].

- Using this result, we develop a top-down recognizer for general context-free languages. Our algorithm, a generalization of Tomita parsing, illuminates the operation of Tomita's algorithm in a new way and extends Tomita's techniques to support top-down parsing for the first time.

Chapter 2

Automated detection of buffer overruns

We argued earlier that assurance for security software is an important unsolved problem, and that simple programming errors account for a surprisingly large proportion of security failures. Perhaps the best illustration of this phenomonem may be found in the *buffer overrun* vulnerability, a variant of array bounds violation error that forms one of the most prominent causes of insecurity in today's systems [40]. In this chapter we present an early warning system for buffer overruns: we develop a new technique for finding potential buffer overrun vulnerabilities in security-critical source code using static analysis, and we assess its utility with a prototype implementation.

2.1 Motivation

Buffer overrun vulnerabilities have plagued security architects for at least a decade. In November 1988, the infamous Internet worm infected thousands or tens of thousands of network-connected hosts and fragmented much of the known net [49]. One of the primary replication mechanisms was exploitation of a buffer overrun vulnerability in the `fingerd` daemon.

Since then, buffer overruns have been a serious, continuing menace to system

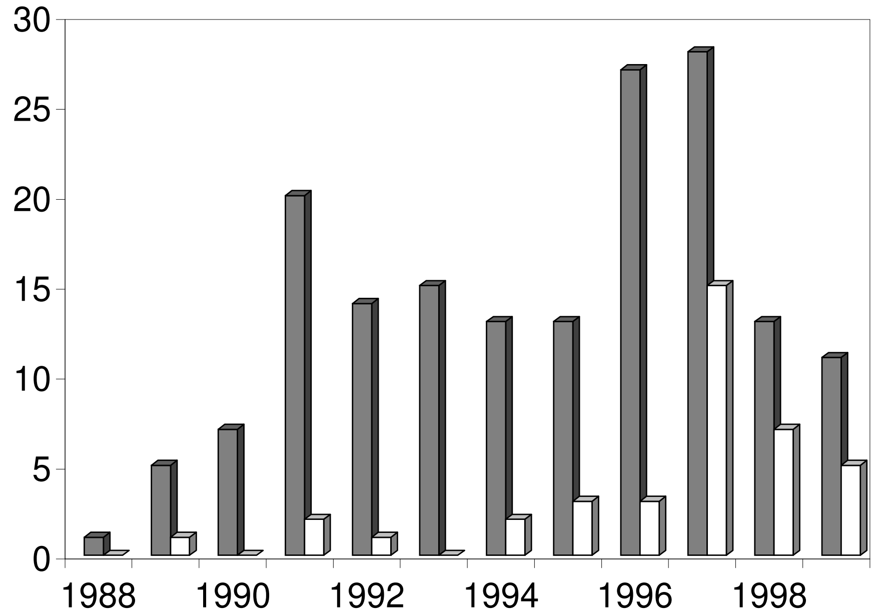

Figure 2.1: Quantifying the importance of buffer overrun vulnerabilities, using a classification of CERT advisories to count their frequency of occurrence. The chart shows, for each year, the total number of CERT-reported vulnerabilities (the darkly-shaded bars) and the number that can be blamed primarily on buffer overruns (the lightly-shaded bars).

security. If anything, the incidence of buffer overrun attacks has been increasing. See Figures 2.1–2.2 for data extracted from CERT advisories over the last decade. These figures show that buffer overruns account for up to 50% of today's vulnerabilities, and this ratio seems to be increasing over time. A partial examination of other sources yields roughly comparable estimates: buffer overruns account for 27% (55 of 207) of the entries in one vulnerability database [78] and for 23% (43 of 189) in another database [85]. Finally, a detailed examination of three months of the `bugtraq` archives (January to March, 1998) shows that 29% (34 of 117) of the vulnerabilities reported are due to buffer overrun bugs [24]. These estimates may even be too conservative: statistics gathered by other researchers blame buffer overruns for a substantially larger fraction of today's vulnerabilities [40].

Buffer overruns are so common because C is inherently unsafe. Array and pointer references are not automatically bounds-checked, so it is up to the programmer to do

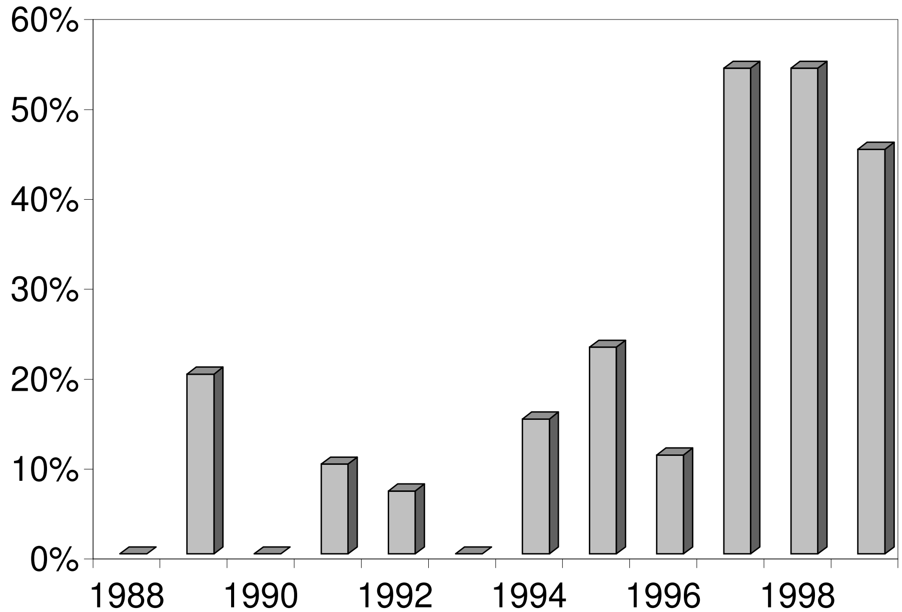

Figure 2.2: Quantifying the importance of buffer overrun vulnerabilities, using a classification of CERT advisories to count their frequency of occurrence. The chart shows, for each year, the percentage of CERT-reported vulnerabilities that were due to buffer overruns.

the checks herself. More importantly, many of the string operations supported by the standard C library—`strcpy()`, `strcat()`, `sprintf()`, `gets()`, and so on—are unsafe. The programmer is responsible for checking that these operations cannot overflow buffers, and programmers often get those checks wrong or omit them altogether.

As a result, we are left with many legacy applications that use the unsafe string primitives unsafely. Programs written today still use unsafe operations such as `strcpy()` because they are familiar. Even sophisticated programmers sometimes combine the unsafe primitives with some ad-hoc checks, or use unsafe primitives when they somehow "know" that the operation is safe or that the source string cannot come under adversarial control.

Unfortunately, programs that use just the "safe" subset of the C string API are not necessarily safe, because the "safe" string primitives have their own pitfalls [107]:

- The `strn*()` calls behave dissimilarly. As an example, `strncpy(dst, src,` `sizeof dst)` is correct but `strncat(dst, src, sizeof dst)` is wrong. Inconsistency makes it harder for the programmer to remember how to use the "safe" primitives safely.

- `strncpy()` may leave the target buffer unterminated. In contrast, `strncat()` and `snprintf()` always append a terminating `'\0'` byte, which is another example of dissimilarity.

- Using `strncpy()` has performance implications, because it zero-fills *all* the available space in the target buffer after the `'\0'` terminator. For example, a `strncpy()` of a 13-byte buffer into a 2048-byte buffer overwrites the entire 2048-byte buffer.

- `strncpy()` and `strncat()` encourage off-by-one bugs. To give an example, `strncat(dst, src, sizeof dst - strlen(dst) - 1)` is correct, but don't forget the `-1`!

- `snprintf()` is perhaps the best of the "safe" primitives: it has intuitive rules, and it is very general. However, until recently it was not available on many systems, so portable programs could not rely on it.

In all cases, the programmer must still keep track of buffer lengths accurately, which introduces another opportunity for mistakes.

In short, today's C environments make it easy to do the wrong thing, and, worse still, hard to do the right thing with buffers. This suggests that an automated tool to help detect this class of security-relevant coding errors may be of considerable benefit.

2.2 Our approach

The remainder of this chapter describes a tool we developed to detect buffer overruns in C source code. Though our implementation is only a first prototype, the results look promising. The tool has enabled us to find vulnerabilities others have missed: for example, the tool found several serious new vulnerabilities in one large security-critical application, even though the source code had been hand-audited previously. In general, our experience is that the tool makes auditing security software considerably less time-consuming than standard manual methods, and allows us to audit large programs that otherwise would have proven too complex to review by hand.

We formulate the buffer overrun detection problem as a *integer range analysis* problem. We use graph-theoretic techniques to construct an efficient algorithm for finding the range of values that each program variable may assume; we will defer any further discussion of this algorithm until Chapter 4. In this chapter, we will focus on describing how security knowledge is used to formulate heuristics that capture the class of security-relevant bugs that tend to occur in real programs.

Others have applied runtime code-testing techniques to the buffer overrun detection problem, using, e.g., black-box testing [105, 106] or software fault injection [67] to find buffer overruns in real-world applications. However, runtime testing seems likely to miss many vulnerabilities. Consider the following example:

```
if (strlen(src) > sizeof dst)
    break;
strcpy(dst, src);
```

This sample code contains a subtle buffer overrun. Note that off-by-one errors in buffer management, such as the one shown above, have been exploited in the past [91, 92]. The fundamental problem with dynamic testing is that the code paths of greatest interest to a security auditor—the ones that are never followed in ordinary operation—are also the ones that are the hardest to analyze with dynamic techniques. This explains our design decision to apply static analysis to the problem.

A recurring theme in this work is the trade-off between precision and scalability. If scalability is not addressed from the start, program analyses often have trouble handling large applications. Since we wish to analyze large programs, such as `sendmail` (tens of thousands of lines of code), we explicitly aim for scalability even if it comes at some cost in precision. This motivates our use of several heuristics that trade off

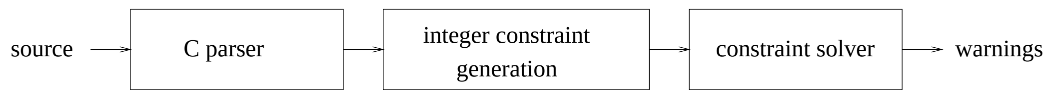

Figure 2.3: The architecture of our buffer overflow detection tool.

precision for scalability.

As a result of imprecision, our analysis may miss some vulnerabilities (*false negatives*) and produce many false alarms (*false positives*), but it is still a useful tool. In our experience, even though our relatively imprecise analysis generates many false alarms, it still reduces the number of unsafe string operations to be checked by hand by an order of magnitude or more; see Section 2.6. We expect that the human costs can be improved even further with more work.

We introduce two fundamental, new insights on automated buffer overrun detection:

1. **We treat C strings as an abstract data type.** In C, pointers are the bane of program analysis, and any code fragments that manipulate buffers using pointer operations are very difficult to analyze. However, most C buffer overruns are in string buffers, and most string operations use the standard C library functions. This suggests modelling C strings as an abstract data type with operations like `strcpy()` (copy strings), `strcat()` (concatenate strings), and so on. Any buffer overruns caused by manipulating strings using primitive pointer operations cannot be detected, but such code won't otherwise interfere with the analysis.

2. **We model buffers as pairs of integer ranges.** Rather than tracking the contents of each string variable directly, we summarize each string with two integer quantities: the number of bytes allocated for the string buffer (its *allocated size*), and the number of bytes currently in use (its *length*). The standard C library functions can be modelled by what they do to the allocated size and length of strings without regard to the strings' contents.

> We formulate the problem of detecting buffer overflows in terms of integer range tracking. Any algorithm for integer range analysis will work: we just check, for each string buffer, whether its inferred allocated size is at least as large as its inferred maximum length.

These two ideas provide a conceptual framework for buffer overrun analysis.

Our formulation of the problem suggests a natural division of labor for the implementation: a front end models string operations as an integer range analysis problem, while a back end solves this problem instance. We defer discussion of the back end until Chapter 4. See Figure 2.3 for a diagram of the system organization.

A secondary contribution of this research is a scalable and very fast integer range analysis. One novel feature of this analysis is the ability to handle cyclic data dependencies without loss of precision by invoking a fixpoint theorem. However, we could easily replace this algorithm with some other technique for integer range analysis.

The remainder of this chapter discusses the design, implementation, and evaluation of our tool. First, we describe how to translate string operations in the source code into a range analysis problem (see Section 2.3). Then, we shift our attention to an analysis of our approach, describing our initial experience with the prototype (Section 2.4).

2.3 Constraint generation

The first step is to parse the source code; we use the BANE toolkit [2]. Our analysis proceeds by traversing the parse tree for the input C source code and generating a range analysis problem that will characterize how the program operates on strings. With each string variable s we associate two variables, its allocated size (the number of bytes allocated for s), denoted alloc(s), and its length (the number of bytes currently in use), denoted len(s). We model each string operation in terms of its effect on these two quantities. In effect, we are introducing new implicit program variables, which we will then apply our range analysis to. Please note that len(s)

Original C code	The derived abstract model
`char s[n];`	`alloc(s) = n;`
`s[n] = '\0';`	`len(s) = min(len(s), n+1);`
`p = "foo";`	`len(p) = 4; alloc(p) = 4;`
`strlen(s)`	`len(s) - 1`
`gets(s);`	`len(s) = choose(1..∞);`
`fgets(s,n,...);`	`len(s) = choose(1..n);`
`strcpy(dst,src);`	`len(dst) = len(src);`
`strncpy(dst,src,n);`	`len(dst) = min(len(src), n);`
`p = strchr(s,c);`	`p = s + choose(0..len(s));`
`strcat(s,suffix);`	`len(s) += len(suffix) - 1;`
`strncat(s,suffix,n);`	`len(s) += min(len(suffix) - 1, n);`
`sprintf(dst,"%s",src);`	`len(dst) = len(src);`
`sprintf(dst,"%d",n);`	`len(dst) = choose(1..20);`
`snprintf(dst,n,"%s",src);`	`len(dst) = min(len(src), n);`
`p = malloc(n);`	`alloc(p) = n;`
`p = strdup(s);`	`len(p) = len(s);`
	`alloc(p) = len(s);`
`p = getenv(...);`	`int t = choose(1..∞);`
	`len(p) = t; alloc(p) = t;`
`h = gethostbyname(...);`	`int t = choose(1..∞);`
	`len(h->h_name) = t;`
	`alloc(h->h_name) = t;`
	`h->h_length = choose(-∞..∞);`

Table 2.1: Modelling the effects of string operations: some examples. The left column presents examples of C code that contains string operations; the right column shows a automatically generated model that abstracts away all information about the contents of those strings. We use the notation `choose(x..y)` to denote a non-deterministic choice of some integer $i \in \mathbb{Z}$ with $\mathtt{x} \leq i \leq \mathtt{y}$.

denotes a program variable, and not, e.g., a function application; do not be fooled by the syntax.

For convenience, the length of a string is defined to include the terminator `'\0'`. Thus, the safety property to be verified is

$$\mathsf{len}(\mathtt{s}) \leq \mathsf{alloc}(\mathtt{s}) \qquad \text{for all string variables } \mathtt{s}.$$

For each string operation in the input program, we pattern-match to determine how to model it in terms of integer assignments. Some sample translations are summarized in Table 2.1. The left column shows the C code for a string operation of interest, and the right column shows the generated integer model. For example, the fourth line says that the return value of the `strlen()` library call is the length of the string passed as its first argument, minus one (for the string terminator). The seventh line in the table says that the effect of the `strcpy()` operation is to overwrite the first argument with the second argument, and thus after the `strcpy()` the length of the first argument is equal to the length of the second argument. Note that although the `strncpy()` primitive may leave its target unterminated, for simplicity we have chosen not to model this behavior[1].

Because we abstract away the contents of all strings, our translation of string operations to integer operations may introduce non-determinism in some cases. We introduce the notation `choose(x..y)` to represent a non-deterministic choice of an integer $i \in \mathbb{Z}$ such that $\mathtt{x} \leq i \leq \mathtt{y}$.

At this point we could in principle analyze the transformed program using any integer range analysis available. However, in our implementation we have only experimented with a single range analysis implementation, described in detail in Chapter 4.

For scalability and simplicity of implementation, we use a *flow-insensitive* analysis, i.e., we ignore all control flow and disregard the order of statements. Flow-insensitive analyses sacrifice some precision for significant improvements in scalability, efficiency, and ease of implementation. We do not claim that flow-insensitive analysis is nec-

[1]This minor deficiency is not fundamental and could be easily repaired with only slight changes to the tool.

essarily the best approach for a production-quality buffer overrun tool; instead, we merely observe that its advantages (ease of implementation, scalability) mapped well to our initial goals (construction of a proof-of-concept prototype that can analyze large, real-world applications).

We note that the `strcat()` operation is difficult to model accurately in a flow-insensitive model, because we must assume that it can execute arbitrarily often (for instance, inside a loop). As a result, in a flow-insensitive analysis *every* non-trivial `strcat()` operation is flagged as a potential buffer overrun. This is a price we have to pay for the better performance of flow-insensitive analyses. Fortunately, most of the C library string operations are *idempotent*, which means that they do not present any intrinsic problems for a flow-insensitive analysis.

Finally, we model function calls *monomorphically*, i.e., we merge information for all call sites of the same function. See Chapter 4 for full details; for our purposes at the moment, the important consequence of a monomorphic analysis is that it is efficient (because we only examine each function once) but potentially imprecise (because information from all call sites is merged).

After the possible ranges of all variables are inferred, we may use the results of the range analysis to check the safety property for each string `s`. Suppose the analysis discovers that $a \le \mathsf{len}(\mathtt{s}) \le b$ and $c \le \mathsf{alloc}(\mathtt{s}) \le d$. There are three possibilities:

1. If $b \le c$, we may conclude that the string `s` never overflows its buffer.

2. If $a > d$, then a buffer overrun always occurs in any execution that uses `s`. (In practice, this case would more likely indicate either an error in our implementation or the presence of dead code, because bugs that are triggered in every application execution tend to be corrected early in the software lifecycle.)

3. If the two ranges overlap—i.e., $c \le b$—we will not be able to rule out the possibility of a violation of the safety property, so we conservatively conclude that there is the potential for a buffer overrun vulnerability in `s` and warn the user.

2.3.1 Handling pointers

Ideally, we would like the analysis algorithm to be *sound*: if the tool believes that $a \leq \mathtt{x} \leq b$, then it should be the case that there is no execution trace for which the value for the program variable `x` fails to satisfy $a \leq \mathtt{x} \leq b$. In practice, though, we may be willing to relax this condition to achieve lower false alarm rates or greater simplicity of implementation. Indeed, our current prototype implementation is unsafe in the presence of pointers or aliasing for reasons we discuss in detail next.

Table 2.1 is deliberately vague about pointer operations. This is because, in the simplistic model used in the prototype, pointer aliasing effects are largely ignored, and the rules for dealing with pointer expressions are highly ad-hoc. For example, in our current implementation, the statement `q = p+j;` is modelled with the two statements `alloc(q) = alloc(q)-j;` and `len(q) = len(q)+j;`. This interpretation is correct in the absence of writes to `*p` and `*q`, but due to the implicit aliasing of `p` and `q` a write to one string is not reflected when the other string is read in some subsequent program statement, which can in general lead to incorrectness. To illustrate this peril, Figure 2.4 gives an example of a code segment with a static buffer overrun that is undetected by our current implementation. Thus, ignoring pointer aliasing can cause the analysis to miss some vulnerabilities and, as we shall see later, can occasionally cause false alarms.

Doubly indirected pointers (e.g., `char **`) are hard to handle correctly with our heuristics and thus are ignored in our tool. Arrays of pointers present the same problems and are treated similarly. As an unfortunate result, command-line arguments (`char *argv[]`) are not treated in any systematic way.

Function pointers are currently ignored. We also ignore union types. These simplifications are all unsound in general, but still useful for a large number of real programs.

It seems that one can retain some benefits of static analysis despite (largely) ignoring pointers and aliasing. Nonetheless, in practice there is one related C idiom that cannot be ignored: use of C `struct`'s. Structures form essentially the only mechanism

```
char s[20], *p, t[10];
strcpy(s, "Hello");
p = s + 5;
strcpy(p, " world!");
strcpy(t, s);
```

Figure 2.4: A buffer overrun that the analysis would not find due to the pointer aliasing. In this example, a 13-byte string is copied into the 10-byte buffer `t`.

for abstraction or construction of data structures, so it is not surprising that they are widely used. Experience suggests that modelling structures properly is crucial to obtaining good results: an earlier analysis tool that ignored structures was mostly useless for understanding real programs of any non-trivial complexity. One aspect that complicates analysis of structures is that they are commonly used in conjunction with pointers (for example, we might want to know whether `p->left->right` refers to the same object as `q->right->right`), yet one of the goals of the prototype was to avoid the implementation complexity associated with a full treatment of pointers, if possible.

This seeming paradox is resolved with a simple trick for modelling structures: all structure objects with the same (or compatible) C type are assumed to be potentially aliased and are modelled with a single variable in the constraint system (see also [46]). In addition, structure field references are further disambiguated using lexical field names, so that `hp->h_length` is not considered the same memory location as `hp->h_addr`. This technique can introduce false alarms (but doesn't miss real vulnerabilities, unless the program uses casts in unusual ways), yet it seems to work well enough in practice, in lieu of a full pointer analysis.

2.4 Empirical evaluation of the prototype

This section details some early experience with the current version of the overrun detection tool. The experimental methodology was simple. The tool was applied to several popular software packages. The tool typically produced a number of warnings about potential buffer overruns, and we examined the source by hand to screen out the false alarms. Some sample output is shown in Figure 2.5.

The case studies can be grouped into two classes: (1) cases where the tool discovered a new, previously unknown security vulnerability in some interesting application[2]; and (2) cases where the tester already knew of the existence of buffer overruns in the application (but not any further details), and the tool was applied to see if the vulnerabilities could be re-discovered by automated means.

We applied the tool to about a dozen software packages. Due to lack of space, we omit the cases where the tool found nothing of interest.

2.4.1 The Linux net tools package

The best success story arose from an analysis of the Linux `nettools` package, which contains source for standard networking utilities such as `netstat`, `ifconfig`, `route`, and so on. The programs themselves total about 3.5k lines of code, with another 3.5k devoted to a support library[3].

This package had already been audited by hand once in 1996 after several buffer overruns were found in the code [81], so it came as somewhat of a surprise when the tool discovered several serious and completely new buffer overrun vulnerabilities. One library routine trusts DNS responses and blindly copies the result of a DNS lookup into a fixed-length buffer, trusting both the `hp->h_name` and `hp->h_length` values returned. In both cases, this trust is misplaced. Another routine blindly copies the result of a `getnetbyname()` lookup into a fixed-size buffer. At first glance, this may appear harmless; however, `getnetbyname()` may issue a NIS network query in some cases, and thus its response should not be trusted. Several other places also perform unchecked `strcpy()`'s into fixed-size buffers on the stack that can apparently be overrun by spoofing DNS or NIS results or by simply registering a host with an unexpectedly long name.

[2] All vulnerabilities and potential vulnerabilities have been reported to the appropriate parties.
[3] Throughout this thesis, we exclude comments and blank lines from our counts of code sizes.

```
Warning: function pointers; analysis is unsafe...
1.74user 0.07system 0:01.99elapsed 90%CPU
Probable buffer overflow in ‘dfname@collect()’:
  20..20 bytes allocated, -Infinity..257 bytes used.
  <- alloc(dfname@collect())
  <- len(dfname@collect()) <- len(@queuename_return)
Probable buffer overflow in ‘from@savemail()’:
  512..512 bytes allocated, -Infinity..+Infinity bytes used.
  <- alloc(from@savemail())
  <- len(from@savemail()) <- len((unnamed field q_paddr))
Slight chance of a buffer overflow in ‘action@errbody()’:
  7..36 bytes allocated, 7..36 bytes used.
  <- alloc(action@errbody())
  <- len(action@errbody())
...
```

Figure 2.5: Some example output from the analysis tool. This example is a small sample of some of the more interesting output from an analysis run of `sendmail` 8.9.3.

These vulnerabilities seem likely to be remotely exploitable[4]. It is worth stressing that these holes were previously unknown, despite an earlier manual audit of the code.

2.4.2 The `sendmail` 8.9.3 package

The latest version of `sendmail` (about 32k lines of C code) was one of the first programs analyzed. Some sample output is shown in Figure 2.5, which shows (for example) that solving the constraint system took less than two seconds; also, Section 2.6 presents a more detailed study of the warnings from the tool. `sendmail` makes an especially interesting test, because it has been extensively audited by hand for buffer overruns and other vulnerabilities. Also, we feel that it makes for a very thorough test of the applicability of the tool to large, complex applications.

The testing session did not uncover any security vulnerabilities in `sendmail`-8.9.3. A few small bugs were identified that could in theory lead to buffer overruns, but

[4] We haven't written exploit code to confirm this, but examination of the source suggests that standard attacks are likely to work.

they do not seem exploitable in practice because the relevant inputs are not under adversarial control. Nonetheless, the new bugs identified do demonstrate the potential to find subtle coding errors in real code using automated analysis techniques.

The most important bug identified by the tool was a complex off-by-one error in the management of string buffers. This bug is hinted at by the warning about '`dfname@collect()`': the tool discovered that 20 bytes were allocated for a buffer called `dfname` (defined in the `collect()` procedure), and that a string containing possibly as many as 257 bytes might be copied into the 20-byte buffer. This is a potential violation of the safety property. In this case, the tool suggests that the lengthy string came from the return value of `queuename()`, but was not able to identify any further dependencies of interest.

Upon further investigation, using other diagnostics from the tool, we found that a complex sequence of invocations can cause `queuename()` to return a 21-byte string (including the terminating `'\0'`). (The 257-byte figure is a result of imprecision in the analysis.) The troublesome sequence is: `orderq()` reads a file from the queue directory, and copies its filename (possibly as many as 21 bytes long, including the `'\0'`) into `d->d_name` and then into `w->w_name`; then `runqueue()` calls `dowork(w->w_name+2,...)`, and `dowork()` stores its first argument (which can be as long as 19 bytes) into `e->e_id`; next `queuename()` concatenates `"qf"` and `e->e_id`, returning the result, which is copied into `dfname`; but `queuename()`'s return value might be as long as $19 + 2 = 21$ bytes long (including the `'\0'`), which will overflow the 20-byte `dfname` buffer.

This minor bug is the result of a common off-by-one error: the programmer apparently forgot to include the string terminator `'\0'` when counting the number of bytes needed to store the return value from `queuename()`. The very complex calling pattern needed to trigger this bug illustrates why this type of vulnerability is so difficult for humans to find on their own and why automated tools are so well suited for this task.

We note that this coding error survived at least one manual audit (the bug predates version 8.7.5, and survived an extensive sweep of the code apparently inspired by

CERT advisory CA-96.20).

For completeness, we explain some of the other warning messages in Figure 2.5. The warning about '`from@savemail()`' is caused by imprecision in the analysis. The relevant code looks something like this:

```
if (sizeof from
    < strlen(e->e_from.q_paddr) + 1)
     break;
strcpy(from, e->e_from.q_paddr);
```

A human would realize that the `strcpy()` is not reached unless it is safe to execute. The tool does not find this proof of safety because the range analysis is flow-insensitive and thus blind to the `if` statement.

The warning about '`action@errbody()`' (another false alarm) is also instructive. The relevant section of code has the following form:

```
char *action;
if (bitset(QBADADDR, q->q_flags))
    action = "failed";
else if (bitset(QDELAYED, q->q_flags))
    action = "delayed";
```

We can readily see that $\mathsf{alloc}(\texttt{action}) = \mathsf{len}(\texttt{action})$ always holds for this code segment, so there is no safety problem. However, the "bounding box" range analysis is fundamentally unable to detect invariants describing the possible relationships between values of program variables—another form of imprecision—so it is unable to detect and exploit this invariant to prove the code safe.

In this case, the analysis can only assume that the string `action` may have as few as 7 bytes allocated for it but as many as 8 bytes copied into it. This happens fairly often: when a pointer `p` can refer to multiple strings of different lengths, the analysis usually reports that its size and length both have the same range $d \leq \mathsf{alloc}(\texttt{p}), \mathsf{len}(\texttt{p}) \leq e$, and when $e > d$ there is no way to rule out the possibility of a problem. We use several heuristics to try to detect this class of false alarms and prioritize all warnings: this class of violations of the safety property is labelled "Slight chance of a buffer overrun."

Improved analysis	False alarms that could be eliminated
flow-sensitive	$19/40 \approx 48\%$
flow-sens. with pointer analysis	$25/40 \approx 63\%$
flow- and context-sens., with linear invariants	$28/40 \approx 70\%$
flow- and context-sens., with pointer analysis and inv.	$38/40 \approx 95\%$

Table 2.2: Expected reduction in false alarms from several potential improvements to the analysis.

One aspect of this trial that is not apparent from Figure 2.5 is the large number of false alarms encountered (see Section 2.6). Weeding through the false alarms took a full day of staring at warning messages and source code. A developer already experienced in `sendmail` internals might have completed the task more quickly, but it would still undoubtedly be a time-consuming process.

2.4.3 The `sendmail` 8.7.5 package

Finding new security vulnerabilities is a compelling way to validate the effectiveness of the tool, but it requires considerable time with no guarantee of positive results. As a time-saving alternative, we applied the tool to old software known to contain serious vulnerabilities to see if the bugs could have been detected. `Sendmail` is one of the classic examples of an application that has been vulnerable to buffer overruns in the past. Since CERT reported several overruns in `sendmail` 8.7.5 (see CA-96.20), and since the next version was audited by hand to try to eliminate such bugs, we decided to use this as a test platform.

The tool found many potential security exposures in `sendmail` 8.7.5:

- An unchecked `sprintf()` from the results of a DNS lookup to a 200-byte stack-resident buffer; exploitable from remote hosts with long DNS records. (Fixed in `sendmail` 8.7.6.)

- An unchecked `sprintf()` into a 5-byte buffer from a command-line argument (indirectly, via several other variables); exploitable by local users with the invocation "`sendmail -h65534 ...`". (Fixed in 8.7.6.)

- An unchecked `strcpy()` to a 64-byte buffer when parsing stdin; locally exploitable by "`echo /canon aaaaa... | sendmail -bt`". (Fixed in 8.7.6)
- An unchecked copy into a 512-byte buffer from stdin; locally exploitable with "`echo /parse aaaaa... | sendmail -bt`". (Fixed in 8.8.6.)
- An unchecked `sprintf()` to a 257-byte buffer from a filename; probably not easily exploitable. (Fixed in 8.7.6.)
- A call to `bcopy()` could create an unterminated string, because the programmer forgot to explicitly add a `'\0'`; probably not exploitable. (Fixed by 8.8.6.)
- An unchecked `strcpy()` in a very frequently used utility function. (Fixed in 8.7.6.)
- An unchecked `strcpy()` to a (static) 514-byte buffer from a DNS lookup; possibly remotely exploitable with long DNS records, but the buffer doesn't live on the stack, so the simplest attacks probably wouldn't work. Also, there is at least one other place where the result of a DNS lookup is blindly copied into a static fixed-size buffer. (Fixed in 8.7.6.)
- Several places where the results of a NIS network query is blindly copied into a fixed-size buffer on the stack; probably remotely exploitable with long NIS records. (Fixed in 8.7.6 and 8.8.6.)

Most of these coding errors became a threat only because of subtle interactions between many pieces of the program, so the bugs would not be apparent from localized spot-checks of the source. This seems to be a good demonstration of the potential for finding real vulnerabilities in real software.

To our knowledge, none of the vulnerabilities found in `sendmail` 8.7.5 by our tool have been described publicly before.

In most cases, the vulnerabilities can be attributed to complex and unexpected data flows, where the programmer did not anticipate the possibility that adversarially controlled inputs could (perhaps indirectly) affect certain trusted program variables. This is exactly why an automated tool is so well suited for the task of detecting buffer overruns.

2.5 Performance

In our experience, the performance of the current implementation is sub-optimal but is usable. For example, the analysis of `sendmail` (about 32k lines of C code) took about 15 minutes of computation on a fast Pentium III workstation: a few minutes to parse the source, and the remainder for range analysis. (The range analysis runtime is dominated by the cost of constraint generation; solving the resulting constraint system took only a few seconds. See Chapter 4 for details on the architecture of our range analysis.)

The prototype generates extensive debugging output and has not been optimized, so we expect that the analysis time could be reduced with additional effort. On the other hand, the time required to examine all the warnings by hand currently dwarfs the CPU time needed by the tool, so better performance is not an immediate priority. For now, the most important property of the system is that it scales up readily to fairly large applications[5].

2.6 Limitations

The main limitation of the prototype is the large number of false alarms it produces, due to imprecision in the range analysis. As a consequence, a human must still devote significant time to checking each potential buffer overrun.

Our tool generates 44 warnings marked `Probable` for `sendmail` 8.9.3. Four of these were real off-by-one bugs, which leaves 40 false alarms. Despite the good success

[5] We have no experience with very large applications, e.g., programs with hundreds of thousands of lines of code, so it remains unknown how our techniques scale up to such program sizes.

rate (1 in 10 warnings indicated real bugs), eliminating the false alarms by hand still requires a non-negligible level of human effort.

One way to reduce the number of false alarms requiring human attention is to trade off time for precision in the integer analysis. For example, we could envision moving to a flow-sensitive or context-sensitive analysis. This obviously raises the question of which improvements are worth the effort and at what cost. To estimate the potential benefits of various possible improvements to the analysis, we classified—by hand—the causes of each false alarm in `sendmail` 8.9.3. See Table 2.2 for the results. (A *linear invariant* is a simple, linear relationship between program variables—such as $\mathtt{x} + \mathtt{y} < 5$ or $\mathsf{alloc}(\mathtt{buf}) \geq \mathtt{buflen}$—that holds in all program executions.)

These figures suggest that, in retrospect, it might have been better to use a more precise but slower analysis. We expect that standard analysis techniques (such as SSA form [41], Pratt's method [115] or Shostak's loop residues [132], and a points-to analysis) could be used to improve on our current prototype by an order of magnitude or more. However, significant engineering effort is probably required to get there.

Despite the unwieldy number of false alarms produced by our tool, our approach is still a substantial improvement over the alternative: in a typical code review, one would identify all the potentially unsafe string operations (perhaps using `grep`), trace back all execution paths leading to those unsafe operations, and manually verify that none of them lead to exploitable overruns. For comparison, there are about 695 call sites to potentially unsafe string operations in the `sendmail` 8.9.3 source that would need to be manually checked in a typical code audit—15$\times$ more than the number that must be examined with our tool—so we conclude that our tool is a significant step forward.

One important gap in our understanding of the prototype's limitations is that it is difficult to measure the false negative rate rigorously. As a first approximation, we may examine all the buffer overruns in `sendmail` that have been fixed in the three years since the release of version 8.7.5; any such bug not reported by the tool is a false negative. To our knowledge, the only publicly reported overrun in `sendmail` 8.7.5 is the `chfn` vulnerability [108], where a local user can overflow a 257-

byte buffer by changing their gecos field in `/etc/passwd`. Due to pointer aliasing and primitive pointer operations, our tool does not find the `chfn` bug, although a better pointer analysis would have revealed the problem. A detailed manual examination of the source code revision history shows that a number of other buffer overruns in `sendmail` 8.7.5 have been quietly fixed without any public announcements[6]. As far as we know, our tool finds all of those vulnerabilities (see Section 2.4.3 for examples). This evidence suggests that our tool's false negative rate is non-negligible but still acceptable.

A final problem with the tool is that it does not provide as much information about each potential buffer overrun as we might like. As can be seen from Figure 2.5, the output shows only which buffer overflowed, not which statement was at fault. This ambiguity is arguably an unfortunate consequence of the constraint-based formulation of our range analysis tool. To improve the situation somewhat, we extended the range analysis backend to report which variable(s) contributed to each violation of the safety property. This heuristic is not always reliable, but it does help.

[6]We do not know whether these bugs were known to the `sendmail` developers, or whether they were fortuitously eliminated by the more-defensive programming style initiated in versions 8.7.6 and 8.8.0.

Chapter 3

Intrusion detection with static analysis

In the previous chapter, we described a promising automated technique for early detection of an important class of software vulnerabilities. In principle, we could imagine expanding our coverage to look for a much broader set of potential security weaknesses. However, it is clear that vulnerability detection has one fundamental limitation: at best, static analysis tools can only detect those vulnerabilities that we know of in advance. In general, as attackers and defenders are in an arms race, deploying a system with strictly static (but incomplete) security measures is doomed to failure: this gives the attacker the last move, and therefore victory[1]. In this chapter, we develop *intrusion detection* systems that remedy this defect by providing an online auditing capability to alert the defender that something appears to be wrong.

3.1 Background

Early work on intrusion detection was due to Anderson [5] and Denning [44]. Since then, it has become a very active field. Most intrusion detection systems (IDS) are based on one of two methodologies: either they generate a model of a program's

[1] We are grateful to Drew Dean for this concise statement of the problem.

or system's behavior from observing its behavior on known inputs (e.g., [63]), or they require the generation of a rule base (e.g., [11]). In both cases, these systems then monitor execution of the deployed program or system and raise an alarm if the execution diverges from the model. The current model-based approaches all share one common problem: a truly robust intrusion detection system must solve a special case of the machine learning problem, a classic AI problem. That is, in order to prevent false alarms, the IDS must be able to infer, from statistical data, whether the current execution of the system is valid or not. The false alarm rate of present systems is a major problem in practice [7].

Ko *et al.*, and others have proposed a very natural solution to this problem: every program should come with a specification of its intended behavior [95, 93, 96, 127, 128]. This, of course, has been the dream of the formal methods community for 25 years, and is as yet unrealized. We believe it is likely to remain unrealized for some time to come. Although Ko *et al.*'s specification language is simple and admits relatively compact specifications, we believe that the need for manually written specifications will dramatically limit the impact of this work[2]. We philosophically agree with the direction of Ko *et al.*'s work, but we propose to side-step its main drawback by automatically deriving the specification from the program. In short, we bring program analysis to the world of intrusion detection.

One of the primary challenges in intrusion detection is modelling typical application behavior, so that we can recognize attacks by their atypical effects without raising too many false alarms. We show how static analysis may be used to automatically derive a model of application behavior. The result is a host-based intrusion detection system with three powerful advantages: a very high degree of automation, protection against future, unforeseen attacks, and the elimination of false alarms. We report on our experience with a prototype implementation of this technique.

We take a slightly unusual view of the role of a CPU and operating system: we consider the operating system to implement a virtual machine interpreter, whose

[2]However, one promising direction to remedy these limitations can be found in Ko's recent work on blending manual rule bases with automated specification generation [94].

language consists of the user mode instruction set of the CPU augmented with a set of more complex operations provided by the system call interface. This virtual machine is extremely efficient, as for most applications, 99+% of the instructions executed are implemented directly in hardware. But from a security viewpoint, the other <1% of instructions are generally the most interesting, because modern operating systems interpose their kernels between applications and the underlying system resources, thereby allowing the kernel to mediate access to these resources.

Our approach constrains the system call trace of a program's execution to be consistent with the program's source code. We assume that the program was written with benign intent. This approach deals with attacks (such as buffer overflows) that cause a program to behave in a manner inconsistent with its author's intent. These are the most prevalent security problems. Of course, some security problems are directly attributable to faulty application logic, such as programs that fail to check authentication information before proceeding, and one limitation of our intrusion detection system is that it does not detect attacks that exploit logic errors. Application logic bugs, however, are dwarfed in practice by buffer overflow problems and other vulnerabilities that allow for execution of arbitrary machine code of the attacker's choice [40, 145], and it is the latter type of vulnerability on which we focus.

In this chapter, we propose a new framework for intrusion detection (Section 3.2), investigate several possible models of program behavior (Section 3.3), report on unexpected implementation issues (Section 3.4), and evaluate our results empirically (Section 3.5).

3.2 The framework

We would like to detect the case where an application is penetrated and then exploited to harm other parts of the system. To this end, we define a specification of expected application behavior, and then we monitor the actual behavior to see if it ever deviates from the specification. We describe first how we monitor application behavior, and next we propose techniques for automated specification construction.

To reduce the potentially huge volume of trace data, we consider only the security-relevant behavior of the application of interest. The monitoring strategy should then ensure that a compromised application cannot compromise system integrity[3] while still evading detection. In general, it will always be possible for attackers to evade detection in our system if they do not cause any harm, but if they want to cause harm, they will need to interact with the operating system in a way that risks detection.

In many cases of practical interest, we may safely make the following convenient assumption [68]:

Assumption. *A compromised application cannot cause much harm unless it interacts with the underlying operating system, and those interactions may be readily monitored.*

If—as is typically the case[4]—the only way to interact with the operating system is via system calls, it suffices to monitor just the application's system call trace. Since monitoring system call traces is usually straightforward in practice, the bulk of the challenge will be to derive a specification of the application's expected interaction with the operating system.

We derive our specification of expected application behavior from the application source code, along with a fixed model of the operating system. We model the application as a transition system with some (possibly very large) set of states along with some admissible transitions. If we ever detect a system call trace that is incompatible with this transition system, we may conclude that the most likely explanation is that we are under attack: for instance, the adversary may have introduced malicious code of her own choosing and caused it to be executed, e.g., via a buffer overrun attack or format string attack. Therefore, to detect intrusions, our basic approach is to look for

[3]We do not consider denial of service attacks in this work.

[4]We do not claim that the assumption is always true. Some operating systems are starting to include partial exceptions to this rule (e.g., user-level networking, direct access to graphics framebuffers, and so on). However, few security-critical applications use these exceptional features, so we can simply forbid their use: the rare application that uses these features may introduce false alarms, but at least malicious code will not be able to exploit the special features in an attack. Consequently, we expect that our assumption will usually apply.

system call traces that could not have been generated by the underlying transition system.

One subtlety is that the adversary may adapt to our methods. Indeed, we later introduce a new type of attack, the *mimicry attack*, which applies to all intrusion detection systems and in some cases may allow the adversary to fool the intrusion detection system by camouflaging the malicious code so that it behaves much like the application would. We do not have a complete defense against mimicry attacks, but we make some progress towards quantifying resistance against this type of attacker tactic. See Section 3.5 for details.

Our intrusion detection system does not detect all attacks, but it does allow us to detect one of the most common effects of a penetration: execution of corrupted code. We observe that, in practice, once an attacker has compromised the target application, she will often download some 'exploit code' of her choosing into the application and use it to execute various operations with the application's privileges. Since this exploit code is not originally present in the application source code, if it is ever executed we expect to see behavior that is incompatible with the source code and thus to detect the attack.

One problem is that transition systems derived directly from the source are usually too complex to be useful. We could, of course, naively start a second 'slave' copy of the application running on the same inputs in an interpreter that simulates all interactions with the outside world, checking at every step whether we obtain the same system call trace from both the master and the slave. This naive replication strategy could probably be made to work, but it has two important disadvantages. First, replication may be hard to implement, because it is likely to be very difficult in practice to remove every last shred of non-determinism from the application (e.g., random number generators, process scheduling, timing channels, interaction with the outside environment, etc.) [97, 140]. Second, and most importantly, the slave is exposed to the same risks as the master: any set of inputs that tickles a security flaw in the master is likely to trigger the same flaw in the slave as well and thereby escape detection.

We tackle these problems by simplifying the transition system greatly, abstracting away unnecessary complexity. Since we care only about the sequence of system calls issued, we prune away all other aspects of the model, even to the point of disregarding the contents of local variables, data structures, and all other data flow. We then simulate the simplified transition system in an interpreter with correspondingly minimal operational semantics. This abstraction process has the potential to fix the problems of naive replication: it can be very fast, because most of the code has been pruned away; we can afford to deal with non-determinism, since the transition system has been drastically simplified (for instance, non-deterministic finite automata are not much more expensive to simulate than deterministic finite automata); and the minimal operational semantics may remove many of the pitfalls of C (e.g., buffer overrun attacks will not affect a model that ignores the contents of all buffers).

This simplification process could be formalized using the theory of *abstract interpretation*, a set of machinery from the programming languages literature that allows us to verify that a proposed abstraction correctly forms a conservative approximation of a given formal semantics [36]. However, we do not undertake such a project here.

To summarize our approach: We first pre-compute a model of expected application behavior, built statically from program source code; then, we monitor the program and check its system call trace for compliance to the model at runtime. The primary challenge is in automating model generation, which we discuss next.

3.3 Models

In this section, we propose a sequence of models that we use to specify expected application behavior: first, a trivial model to illustrate the main idea; then, the callgraph model; third, a refinement, the abstract stack model; and finally, the low-overhead digraph model.

Each model is intended to satisfy a common *soundness* property: false alarms should never occur. To achieve this goal, we must make a number of mild assumptions about our operating environment. We consider only portable C code that has no implementation-defined behavior: for example, we assume that there are no intentional array bounds violations, NULL-pointer dereferences, or other memory errors; we assume there is no function pointer arithmetic or type-casting between function pointers and other pointers; and we assume there is no application-defined runtime code generation. These assumptions are not critical: violations may introduce false alarms but will never cause us to miss attacks we otherwise would have detected. Nonetheless, in our experience the security-critical applications in widespread use do conform to these assumptions.

From a formal language viewpoint, all of our models involve recognizing a sentence in a regular or context-free language. However, this viewpoint is much less intuitive than dealing directly with automata and will not be discussed further. For ease of discussion, we will refer to terminating programs and finite or pushdown automata, as appropriate. All of our results directly extend to non-terminating programs.

3.3.1 A trivial model

We illustrate these ideas by describing a minimalist example of an intrusion detection system following this framework. Let S be the set of system calls that the application can ever make. The set of allowable system call traces—i.e., our model of expected behavior—will then be exactly the regular language S^*. If, at runtime, we ever observe the application issuing some system call not in S, we prevent the system call from executing, kill the application, and sound the alarm.

This model is easy to derive with automated source analysis tools. Because in practice system calls may be easily recognized in source code, the set S may be inferred easily by simply walking the parse tree and pattern-matching for system call invocations.

Such an approach is simple, easy to implement, sound, and efficient, but it will fail to detect many attacks. No attack that operates using just system calls from S will ever be detected, and in practice we can expect this failure mode to be common if S is too large. Another problem is that the approach is too coarse-grained, since

```
f(int x) {
  x ? getuid() : geteuid();
  x++;
}
g() {
  fd = open("foo", O_RDONLY);
  f(0); close(fd); f(1);
  exit(0);
}
```

Figure 3.1: A C program, which will be used as a running example to illustrate different models of program behavior.

many common system calls are too dangerous to allow without any restrictions. For example, if the `open()` system call is included in S, attackers will be free to modify any file whatsoever at any time without fear of detection; there will be no way to detect attacks that add a backdoor account to the password file, since such a backdoor may be added using code such as

```
fd = open("/etc/passwd", O_WRONLY | O_APPEND);
write(fd, "toor::0:0::/:/bin/sh\n", 21);
```

Furthermore, this naive approach scales poorly to large applications, which are exactly the ones at greatest risk for intrusions, because large applications yield large sets S. Consequently, this trivial approach is unlikely to be satisfactory in most scenarios. A more precise model is needed.

3.3.2 The callgraph model

The foremost problem with the naive model described above is that we have thrown away all information about the *ordering* of the possible system calls. A more precise characterization of the expected system call traces can be obtained if we introduce some notion of the ordering of the system calls. In this section we show how to retain some ordering information.

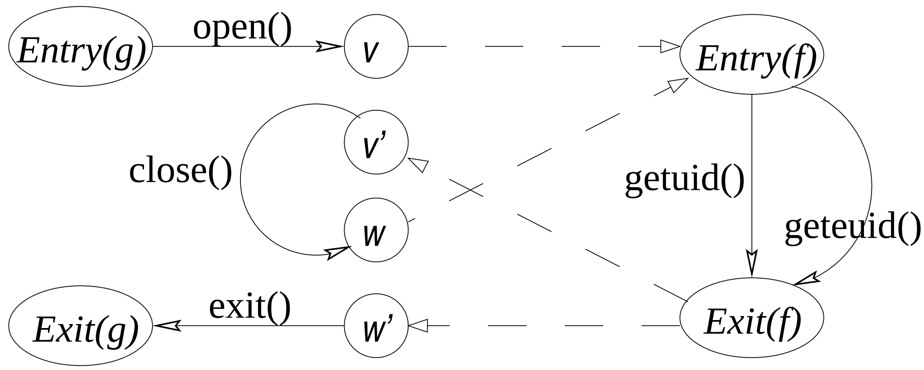

Figure 3.2: The callgraph model generated from our running example (see Figure 3.1). Transitions to Wrong are omitted to avoid cluttering the diagram. Dashed lines indicate interprocedural edges, which are represented as ϵ-transitions in the NDFA.

One clean way to represent information on the ordering of possible system calls is to express our model as a regular language over Σ, the set of system calls. For ease of model generation, it is convenient to use an equivalent representation of the model as a non-deterministic finite automaton (NDFA). We describe next how to use a NDFA to characterize the expected system call traces.

Building the model Deriving the model is a simple application of control-flow analysis. We first build the control-flow graph $G = \langle V, E \rangle$ associated with the program source code, like any optimizing compiler would do. We assume that each node of the control-flow graph executes at most one system call and that we can recognize where system calls occur. Then we note that the control-flow graph can naturally be viewed as a specification of a NDFA with statespace $V \cup \{\mathsf{Wrong}\}$, transitions induced by E, and alphabet Σ. Each edge $v \to w \in E$ of the control-flow graph induces a transition $v \xrightarrow{a} w$ of the automaton, if there is a system call a at node v, or the ϵ-transition $v \xrightarrow{\epsilon} w$ otherwise; we use the symbol ϵ to denote the empty string, so ϵ-transitions represent transfer of control where no system call is executed. Every *proper state* (i.e., each state $v \neq \mathsf{Wrong}$) is considered an accepting state. The special state Wrong is non-accepting and contains a self-loop $\mathsf{Wrong} \xrightarrow{a} \mathsf{Wrong}$ on every $a \in \Sigma$; when a node v contains no outgoing transitions on some symbol $a \in \Sigma$, we add an implicit transition $v \xrightarrow{a} \mathsf{Wrong}$. The resulting automaton is non-deterministic because in general we cannot statically predict, for example, which branch of an if-then-else expression will be taken at runtime. See Figures 3.1 and 3.2 for an example.

We use this automaton as our model of expected behavior, so that an observed trace is accepted only if it is accepted by the NDFA. We call this the *callgraph* model. Note that this model throws away a lot of information about the execution of the application: in particular, we ignore all of its internal state other than the program counter. Nonetheless, it preserves a soundness property:

Claim. *There are no false alarms when using the callgraph model.*

The claim follows from the observation that, by construction, every possible path of execution through the control-flow graph corresponds to an accepting path of the NDFA, and thus every dynamically possible execution trace will be accepted by the NDFA.

When implementing the callgraph model, we have found that several issues are important.

Monitoring algorithm When monitoring the application, we simulate the operation of the NDFA on the observed system call trace, resolving non-determinism by exploring all possible paths in breadth-first order. This requires $O(|V|)$ operations per observed system call. Note that more efficient techniques exist—for instance, the NDFA may be converted to a DFA, either ahead of time or on the fly, and caching may be used to speed up the simulation [82]—but we have not explored any of these alternatives. See Section 3.4 for more implementation details, and Section 3.5 for measurements of our implementation's performance and detection power.

Function calls One issue not mentioned so far is how to deal with function calls. Traditional compilers generate a separate control-flow graph for each function or procedure in the application, without connecting them together. In contrast, since we require a whole-program model, after we generate a control-flow graph for each procedure we connect them all together: we split each call site v into two nodes v, v' and add extra edges $v \to \text{Entry}(f)$ and $\text{Exit}(f) \to v'$ for each function f that could be called from v. See the dashed edges in Figure 3.2 for an example. Here $\text{Entry}(f)$ and $\text{Exit}(f)$ denote the unique entry and exit nodes for f, as might be expected. This so-called *monomorphic* (or context-insensitive) analysis produces a single large graph that may be analyzed as above.

Imprecision in the model One limitation of the callgraph model is that it includes impossible paths, due to the monomorphic treatment of function calls. In particular, consider two call sites v, w that both call the same function f; then the expanded control-flow graph will contain paths of the form $v \to \text{Entry}(f) \to \cdots \to \text{Exit}(f) \to w'$, corresponding to an execution that calls f from v and then returns back to a different call site w. See Figure 3.2 for an illustrated example. Such an impossible path cannot occur in any real execution, because function calls will always return to the site where they were called from. Unfortunately, a NDFA is unable to express this constraint, so we end up with impossible paths through the automaton.

Impossible paths in the callgraph model are a problem in practice. This imprecision causes our NDFA to be larger than necessary, and attacks that follow these impossible paths will remain undetected (even though we know that the impossible paths clearly could never be executed in normal operation). As a consequence, intrusion detection systems based on the callgraph model may in some cases be more permissive than we would like.

3.3.3 The abstract stack model

We next introduce the *abstract stack* model, which allows us to characterize more precisely the set of possible system call traces by eliminating impossible paths. The idea is to model not only the program counter but also the state of the call stack. We extend our model so that the set of possible system call traces is allowed to form a context-free language. It is then natural to represent this abstraction of the program as a non-deterministic pushdown automaton (NDPDA), or equivalently, a context-free grammar.

Building the NDPDA The pushdown automaton we construct will provide an intuitive model of program behavior. The state of the automaton will be an abstract summary of the state of the application. In particular, the automaton's stack will form an abstract version of the program call stack: each symbol on the automaton's stack will correspond to a single stack frame in the application's call stack, where everything but the return address has been abstracted away.

The construction is as follows. We assume that we are given an global control-flow graph $G = \langle V, E \rangle$ for the entire program that includes interprocedural call edges. We generate a NDPDA with stack alphabet $V \cup \Sigma$, input alphabet Σ, and transitions given as follows. Suppose first that there is a node $v \in V$ on the top of the stack. If v is a function call site referencing a procedure f, we pop v off the stack, push the corresponding return site v', and finally push $\text{Entry}(f)$ on to the stack. If v is a function exit node, we pop v. If v is a non-call node, we pop v, push s if v issues the system call $s \in \Sigma$ (otherwise, we do not push anything for nodes that do not make system calls), non-deterministically select some successor w of v with $v \to w \in E$, and finally push w. On the other hand, if $s \in \Sigma$ is at the top of the stack, we attempt to match s against the current input symbol s': if $s = s'$, we consume the current input symbol and pop s off the stack; otherwise, we enter the state Wrong and reject the input string. As in the callgraph model, all proper states are accepting states. See Figure 3.3 for an example.

This construction of the NDPDA ensures that every sequence of operations to the program call stack during a normal application execution will be among the set of paths explored during the simulation of NDPDA. Since the NDPDA is non-deterministic, other paths may also be explored, but we can be sure that the correct one will not be omitted. At the same time, the increased precision of the abstract stack model makes it less likely that real attacks will go undetected.

The context-free model In our implementation, the NDPDA is constructed directly. However, as the construction is rather detailed, it may be easier to consider building an (almost, as explained below) equivalent context-free grammar for the program, with non-terminals taken from V, terminals in Σ (the set of system calls), and rules given as follows[5]. If v is a function call site with corresponding return site v', we add the rule $v ::= \text{Entry}(f)\ v'$ for each function f that could be called from v. For each non-call node v and each successor w of v, we add the rule $v ::= a\ w$ if there is a system call $a \in \Sigma$ at v, or the rule $v ::= w$ otherwise. Finally, for each function f in the program, we add the ϵ-rule $\text{Exit}(f) ::= \epsilon$. This grammar is quite natural [122, 126, 35, 37, 4, 123, 53]. It is not too hard to see that the resulting context-free language conservatively summarizes all possible behavior of the application.

The simplification referred to above is that the NDPDA, by construction, also accepts all prefixes of sentences generated by the grammar above. The actual grammar would be more complicated if it took this into account.

The NDPDA described earlier may be obtained by applying the trivial top-down construction to the context-free grammar obtained above (similar to $LL(0)$ parsing, except that we keep the conflicts and thus obtain a non-deterministic automaton). This top-down construction is convenient because its operation corresponds closely to execution in procedural languages such as C. See Figure 3.3 for an example.

The construction of the NDPDA ensures that every sequence of operations to the program call stack during a normal application execution will be among the set of paths explored during the simulation of NDPDA. Since the NDPDA is non-deterministic, other paths may also be explored, but we can be sure that the correct one will not be omitted. Consequently, the NDPDA provides a safe approximation of normal application behavior.

Monitoring algorithm To detect attacks, we must monitor the system calls issued by the application and simulate the operation of the NDPDA on those inputs. It turns

[5]There are some complications with `setjmp()` and other non-standard forms of control flow; see Section 3.4.1 for extensions to handle them.

$$
\begin{array}{lcl}
\mathrm{Entry}(f) & ::= & \texttt{getuid}()\ \ \mathrm{Exit}(f) \\
 & | & \texttt{geteuid}()\ \ \mathrm{Exit}(f) \\
\mathrm{Exit}(f) & ::= & \epsilon \\
\mathrm{Entry}(g) & ::= & \texttt{open}()\ v \\
v & ::= & \mathrm{Entry}(f)\ \ v' \\
v' & ::= & \texttt{close}()\ \ w \\
w & ::= & \mathrm{Entry}(f)\ \ w' \\
w' & ::= & \texttt{exit}()\ \ \mathrm{Exit}(g) \\
\mathrm{Exit}(g) & ::= & \epsilon
\end{array}
$$

while (true)
 case $\textsf{pop}()$ of

$$
\begin{array}{lcl}
\mathrm{Entry}(f) & \Rightarrow & \textsf{push}(\mathrm{Exit}(f));\ \textsf{push}(\texttt{getuid}()) \\
\mathrm{Entry}(f) & \Rightarrow & \textsf{push}(\mathrm{Exit}(f));\ \textsf{push}(\texttt{geteuid}()) \\
\mathrm{Exit}(f) & \Rightarrow & \text{no-op} \\
\mathrm{Entry}(g) & \Rightarrow & \textsf{push}(v);\ \textsf{push}(\texttt{open}()) \\
v & \Rightarrow & \textsf{push}(v');\ \textsf{push}(\mathrm{Entry}(f)) \\
v' & \Rightarrow & \textsf{push}(w);\ \textsf{push}(\texttt{close}()) \\
w & \Rightarrow & \textsf{push}(w');\ \textsf{push}(\mathrm{Entry}(f)) \\
w' & \Rightarrow & \textsf{push}(\mathrm{Exit}(g));\ \textsf{push}(\texttt{exit}()) \\
\mathrm{Exit}(g) & \Rightarrow & \text{no-op} \\
a \in \Sigma & \Rightarrow & \text{read and consume } a \text{ from the input} \\
\text{otherwise} & \Rightarrow & \text{enter the error state, } \textsf{Wrong}
\end{array}
$$

Figure 3.3: The context-free grammar (left) and the abstract stack model (right) associated with our running example (see Figure 3.1).

out that efficient simulation of the NDPDA is a significant theoretical and engineering challenge, especially as we scale up to intrusion detection on very large applications.

The most naive approach is to exhaustively search through all possible non-deterministic choices of the NDPDA. In other words, at each time step, we maintain a list of all possible stack configurations of the NDPDA; when a new system call is observed, for each previously possible configuration we compute the set of new configurations the NDPDA might transition to, and update the list of possible stack configurations. However, in practice this approach is untenable for any but the simplest application, because these lists grow exponentially large in the length of the system call trace (in fact, even infinitely large, in the presence of left-recursion).

Less naively, we might hope that standard parsing algorithms might be applicable here. Of course, we cannot use standard parsers (such as `yacc`) because our NDPDA is non-deterministic, or equivalently, because our context-free grammar is usually not $LL(0)$. It is easy to see that, for every context-free grammar Γ, there is some program that generates Γ, and in practice, real applications produce grammars with considerable non-determinism and complexity. So, we need an efficient algorithm for online parsing of general context-free languages.

It is also important to have a top-down parsing routine. As described in Section 3.4, dealing with some of the special features of the Unix runtime requires us to occasionally step outside of the context-free framework and perform operations directly on the set of possible stack configurations. Real programs execute in a roughly top-down fashion—we start executing `main()` before executing any of its callees—so this seems to rule out bottom-up parsing. Unfortunately, much of the work in the literature on recognizing general context-free languages (e.g., the CYK, Earley, Tomita, and GLR techniques [149, 48, 69, 138]) uses bottom-up methods.

Consequently, we were forced to develop new techniques for efficient top-down parsing. A full description of our algorithm will be deferred until Chapter 5, but we list a few useful properties of the algorithm that make it well suited for our purposes:

- It supports online parsing: as each system call is observed, we can decide whether the resulting partial trace forms the prefix of a sentence in the context-free language, as required for real-time intrusion detection.

- It is relatively efficient: like other general context-free recognizers, its worst-case running time is cubic in the length of the system call trace. This is likely to be too slow for large applications, but is much better than exponential-time solutions. In practice, we encounter cubic-time behavior only occasionally.

- Most importantly, it supports real-time access to the set of possible top-down parse trees. The key data structure is a representation of the set of possible call stacks as a regular language over the alphabet of stack symbols. This lets us modify this data structure directly whenever we need to step outside of the context-free framework.

See Chapter 5 for more details on this algorithm.

A detailed assessment of the strengths and weaknesses of the abstract stack model will be deferred until Section 3.5.

3.3.4 The digraph model

We next introduce a very simple approach that combines some of the advantages of the callgraph model in a simpler formulation. The basic approach, first introduced in previous work on runtime intrusion detection [63], is to consider windows of consecutive system calls.

Our model will thus be a list of the possible k-sequences of consecutive system calls, starting at an arbitrary point during program execution. In our prototype implementation, we consider only the special case $k = 2$ for simplicity. Note that k-sequences of system calls with $k = 2$ are often referred to as *digraphs*, so we call this the digraph model. We consider here both the special case of digraphs and the general case.

Building the model We could derive the set of possible k-sequences from the control-flow graph in a straightforward fashion, but we observe that there is a more precise approach available if we use the context-free language of possible system call traces, $L(\Gamma)$, as introduced in Section 3.3.3. Recall that every possible system call trace is a sentence in $L(\Gamma)$.

There is a simple test to determine whether the sequence $s \in \Sigma^k$ can occur in a system call trace during normal application execution. Let $R_s = \Sigma^* s \Sigma^*$ (a reminder: Σ denotes the set of system calls) be the set of words containing s. Note that R_s is a regular language. It is well-known that the intersection of a context-free language and a regular language is another (efficiently computable) context-free language, and moreover we may efficiently check whether this language is empty [82, 122]. Therefore, we include the k-sequence s in our model of possible k-sequences if and only if $R_s \cap L(\Gamma) \neq \emptyset$.

Repeating this test for each $s \in \Sigma^k$ gives a general algorithm to build the desired model. Unfortunately, this precomputation has running time $\Theta(k^3 \times |E| \times |\Sigma|^k)$, which is exponential in k. In practice, it is slow enough that we have only experimented with the $k = 2$ case. We leave it as an interesting open problem to find a more efficient algorithm for larger values of k.

Monitoring algorithm Detecting attacks then becomes easy once we have performed the above precomputation to build a list of the allowed k-sequences. We keep a history of the last $k-1$ system calls, and when we see a new system call, we check whether the resulting k-sequence is allowed. Thus, the runtime monitoring algorithm is extremely efficient for this model; the trade-off is that the digraph model is less precise than the callgraph or abstract stack model, and thus can be expected to miss more attacks.

3.4 Implementation issues

We sketched above three theoretical frameworks for implementing intrusion detection using static analysis. In practice, though, there are a number of complications that arise when implementing these ideas. We discuss here some of the important implementation challenges and how to handle them.

3.4.1 Non-standard control flow

Implementations of control-flow analysis, when intended for optimization, often give up in the presence of non-local control flow (such as signals, `setjmp()`, and so on). However, we have found that, in practice, real applications of interest for intrusion detection often use these features. Therefore, we describe how to augment the modelling frameworks described above to incorporate these forms of non-standard control flow.

Function pointers To build the program call-graph in the presence of function pointers, it is crucial to be able to predict the possible targets of every indirect call through a function pointer[6]. Therefore, we use a points-to analysis to infer, for each

[6]Fortunately, we have never seen any application that invokes system calls indirectly, so it suffices to consider function pointers.

function pointer expression, the set of possible targets that the expression could refer to; each indirect call is then treated as a non-deterministic selection of a function from the points-to set followed by invocation of the selected function. Many sophisticated algorithms for pointer analysis are available in the literature [50, 135, 130], but in our implementation we simply assume that every pointer could refer to any function whose address has been taken. We find it interesting that, empirically, even this very crude technique seems to suffice for our purposes.

Signals Many operating systems allow applications to register a signal handler to be executed upon reception of a signal. This introduces extra possibilities for control flow that must be added to our model. It is straightforward to statically recognize signal handlers: we simply look for system calls of the form `signal(i,fp)`, which binds the handler `fp` to the signal `i` so that when this signal is received, the function referred to by the function pointer `fp` will be called. Consequently, the real challenge is to augment the model to represent these additional possibilities for control flow.

Naively, one might consider adding to the control-flow graph an extra edge from each node to each possible signal handler to represent this additional control flow. In particular, to every node $v \in V$ we could associate a fresh partner node v' that non-deterministically selects a signal handler and calls it (just like any other function call), and for every edge $u \to v$ we add the new edges $u \to v'$, $v' \to v'$, and $v' \to v$.

This naive solution would work, but it adds an enormous amount of extra non-determinism to the control-flow graph, so our analysis would become less precise: the intrusion detection system would become significantly slower (because we need to follow more possible paths in the control-flow graph) and poorer at recognizing intrusions (because real attacks might mimic unlikely paths through signal handlers and thereby avoid detection). Unfortunately, many real-world programs do use signal handlers, so we cannot avoid this problem. We would greatly prefer to model signals without incurring these costs.

Fortunately, there is a clean solution available. We exploit the presence of a runtime component in our system:

Principle 1. *If you can arrange to receive an extra event whenever some exceptional path (such as invocation of a signal handler) might be taken, you can often improve the precision of the model.*

In this case, we arrange to monitor not only the system calls the application makes but also the signals the application receives[7], and we ensure that all the extra paths in the control-flow graph are *pre-guarded* by an initial signal reception event. In many Unix operating systems, all signal handlers invoke the `sigreturn()` system call after they return, so we also add a *post-guard* to the end of each extra path, too.

It is straightforward to augment the control-flow graph to ensure that every execution of a signal handler will be bracketed by both a pre- and post-guard. The construction is as follows. We use $\Sigma := \mathsf{Syscalls} \cup \mathsf{Signals}$ as our alphabet of events. For each node $v \in V$ and each signal $i \in \mathsf{Signals}$, we add an extra node v'_i that represents raising of signal i, a second extra node v''_i that acts as a function call to the handler for signal i, and a third extra node v''' that invokes the `sigreturn()` system call. Then, for each edge $u \to v$ in the control-flow graph, we add the new edges $u \to v'_i$, $v'_i \to v''_i$, $v''_i \to v'''$, $v''' \to v'_i$, and $v''' \to v$.

These extra guarded paths in the control-flow graph will not be triggered unless the appropriate signal is received, and to save space they may be implicitly represented and only re-generated on demand, so they are effectively invisible except in the cases where they are necessary. These techniques provide a precise, efficient, and simple way to extend any of the models in Section 3.3 to reflect the semantics of signals.

The `setjmp()` primitive ANSI C provides a form of non-local control flow that is sometimes used to provide a crude form of exception handling or error recovery: the `setjmp()` primitive saves the stack pointer and other registers, and then `longjmp()` may be called by a subroutine to roll the registers, and hence the stack, back to its saved state. In the callgraph model, we may simply add an extra transition from each `longjmp()` to every possible `setjmp()`, but this will not work for the abstract stack

[7]The ability to monitor signals is conveniently already available with most existing mechanisms for process tracing, since it is used by some debuggers.

model because `longjmp()` modifies the call stack.

We do not know of a good static approach to call stack analysis in the presence of `setjmp()`, but fortunately, there is no need to solve this problem statically. Instead, we extend the runtime monitoring agent. Our monitor maintains a running list of all call stacks that were possible when some `setjmp()` call was visited earlier in this execution trace[8]. Each `longjmp()` call can be emulated by adding this accumulated list to the automaton's current set of states. Since sets of states are represented as regular languages in the abstract stack model (see Section 3.3.3), the union operations may be implemented efficiently.

As a future extension, we might also enforce the constraint that returning from a function activation invalidates any `setjmp()` it may have called. This would allow us to garbage-collect old `setjmp()` states (thereby reducing storage costs by some unknown amount) and to exclude impossible `longjmp()` targets (thereby improving precision and attack detection power). So far, though, we have not found the need. Our experience has been that `setjmp()` is typically used just often enough that it cannot be completely ignored but rarely enough that the burden of the above simulation techniques is minimal.

In any case, our experience with `setjmp()` suggests the following lesson for hybrid static-dynamic systems[9]:

Principle 2. *Some program properties that are difficult to infer statically may become easier to model satisfactorily when the burden is offloaded to a runtime agent, where*

[8] In other words, if we encounter a `setjmp()` call as we simulate the operation of the NDPDA, we make a copy of the current set of possible states of the pushdown automaton and store it in on the side. (As subsequent `setjmp()` calls are encountered, we take the running union of the corresponding sets of automaton states.)

[9] In the digraph model, neither Principle 1 nor 2 is much help, since no help is available from the runtime agent nor is there any convenient way to monitor `setjmp()` and `longjmp()` calls at runtime. Thus, we are forced to use more conservative techniques. Consider temporarily extending the alphabet with the symbols § and $\pounds$ to represent invocations of `setjmp()` and `longjmp()` (respectively). We infer that digraph $s_1 s_2$ is possible in some program execution only if (1) $s_1 s_2$ is a possible digraph in the original (unextended) language, or (2) both $s_1 \pounds$ is a possible digraph when the language is extended with $\pounds$ and $\S s_2$ is a possible digraph when the language is extended with § and $\pounds$.

available.

If we were able to receive an event on each `setjmp()` and `longjmp()` invocation, our treatment could be improved further by applying Principle 1 in combination with the above, but this seems to require modification of binary executables, so we have not investigated this direction.

The `exit()` system call Our runtime monitor enforces the constraint that no further system calls are allowed after a `exit()` call. As an optimization, in the abstract stack model our NDPDA simulator further exploits this property to trim the call stack if an `exit()` call occurs in the middle of a nested call stack, since we know that there is no need to maintain the set of callers who we might return to after the `exit()`.

The `fork()` system call When the runtime monitor sees a `fork()`, it makes a duplicate copy of the NDPDA simulator and associates it to the child process; then both simulators proceed in parallel.

3.4.2 Other modelling challenges

Libraries Our approach requires a model for each library function that might be called. Therefore, we use a modular analysis to build these models. In particular, we modified the `gcc` compiler to output intermediate analysis output files alongside each object file as it compiled, and we modified the linker to combine the intermediate files into a whole-program analysis. A side benefit was that we could analyze existing software packages by simply using the provided Makefiles to compile them.

We found that library code taxed the limits of our tool more thoroughly than most applications, and a disproportionate amount of our effort was spent on the C libraries. For instance, the GNU `stdio` implementation uses function pointers extensively to emulate an object-oriented programming style; with our naive pointer analysis, the inferred models were too imprecise, so we replaced our automated analysis results with hand-crafted models. In contrast, the database library `libdb` also uses function pointers extensively to parametrize database implementations, but in this case we were willing to accept the imprecision. As a third example, the GNU ELF libraries make heavy use of both `setjmp()` and function pointers to implement exception handling, so we resorted to refining the inferred model by hand in some places to improve its precision.

There are many disadvantages to hand-built models: they are time-consuming to construct; they are difficult to get right (and thus unsoundness and false alarms are a risk); and they make it unpleasant to keep up with changes to the code. Ideally, we would have preferred a more precise automatic analysis so that we could avoid these disadvantages, but in practice even our crude techniques were generally sufficient to get the job done without compromising our primary goals in the few cases where manual analysis was necessary.

Dynamic linking Dynamically linked libraries pose another challenge, because they force us to update the model at runtime. In our implementation, we predict in advance the set of libraries that might be linked in and build models for all of them from source. This can introduce false alarms if our prediction becomes out of date (when, e.g., a new version of the library becomes available), which means that everything must be updated whenever the underlying libraries are. This is not a fundamental limitation, and a more satisfying solution would be to build a model at runtime from object code, but we have not explored this direction because it has not been necessary in our experience. In any case, binding applications to libraries statically has substantial security benefits, because it prevents introduction of Trojan horses via dynamic linking attacks.

C oddities C contains many language constructs with undefined evaluation order or where the semantics is otherwise left up to the compiler writer. In contrast to typical compiler practice, rather than simply choosing an arbitrary evaluation order, we must incorporate all possible choices in our model. In practice, we add an edge to the control-flow graph for each legal implementation of any underdefined language construct.

Threads Threads pose yet another challenge, because the context-switching operation introduces another type of implicit control flow. If it were possible to reliably receive 'thread context-switch' events (see Principle 1), handling threads would be straightforward; this is no problem for kernel threads, but unfortunately, user threads pose a thorny challenge[10], and we know of no good general-case solution. A second issue is that threaded code may contain security vulnerabilities due to synchronization bugs, and we do not know how to detect this class of attacks. Because of these challenges, and because no security-critical application we examined used threads, our prototype implementation does not support threaded code.

3.4.3 Optimizations

Reducing the control-flow graph In practice, the control-flow graph often contains many irrelevant nodes, and these may be removed for better performance. We found it useful to shrink the control-flow graph by eliminating[11] nodes that do not perform system calls. This optimization is roughly analogous to removal of ϵ-transitions in NDFA's.

Of course, call/return sites and function entry/exit nodes must not be removed from the graph when we eliminate uninteresting nodes.

Irrelevant system calls Up to now we have described an intrusion detection system that monitors all system calls the application invokes, and we originally expected this to be optimal. However, we found that ignoring, e.g., the `brk()` system call can

[10]With knowledge of the implementation of the user threads package, it may be possible to query the package for the current thread on each system call, to instrument the scheduler to generate extra 'thread-switch' events, or to watch for, e.g., `SIGALRM` signals (in the case of pre-emptive threading), but we have not investigated this direction.

[11]A node $v \in V$ is removed as follows: for each $u, w \in V$ with $u \to v \to w$, we add a new edge $u \to w$; then, we delete the node v and all incident transitions.

greatly improve performance by reducing the size and ambiguity of the model: in many programs, memory allocation can occur just about anywhere, so seeing a `brk()` system call gives us very little contextual information. This may cause us to miss denial-of-service attacks, but those are beyond the scope of this work.

In some cases, ignoring certain system calls can even improve the precision of the model. It may sound paradoxical that throwing away information can improve precision, but consider the digraph model: excluding very common system calls gives more context. It is useful to be able to enable this optimization on a per-application basis.

System call arguments The most important optimization is based on the observation that we can gain quite a bit of extra information about the application behavior by examining the arguments to each system call. Since we can often statically predict some system call arguments with little effort, we might as well check them at runtime. We recognize lexically constant system call arguments in our prototype and found that even this extremely crude technique provides noticeable improvements to both precision and performance; see the measurements in the next section.

3.5 Evaluation

In this section we measure the performance of our three approaches (the *abstract stack*, *callgraph*, and *digraph* models) on a number of typical security-critical applications that one might want to monitor for intrusions. For each model, we measure two variants: a basic implementation that ignores system call arguments, and then an improved implementation that checks all system call arguments that can be statically predicted. In each case, we focus on two key metrics: runtime overhead (*performance*), and robustness of detection against targeted attack (*precision*). As will become clear, our results indicate that there is a strong tradeoff between performance and precision.

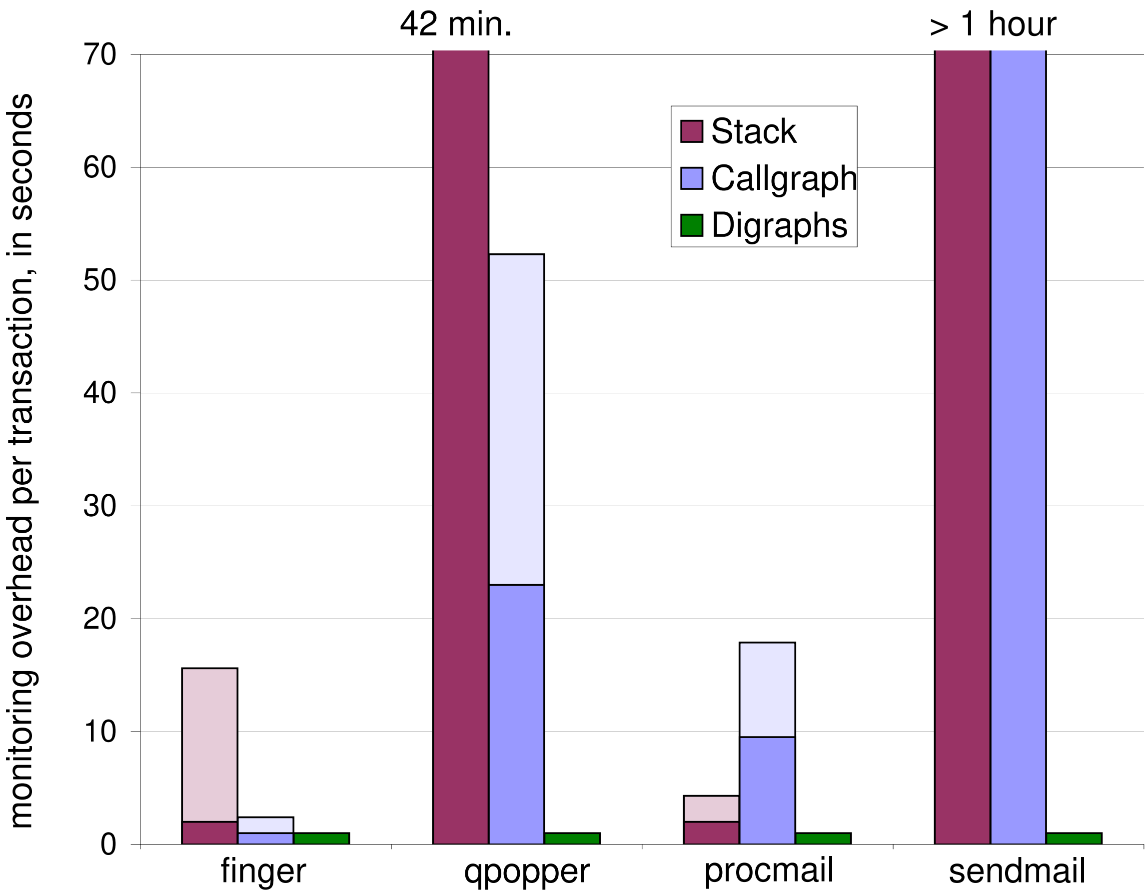

Figure 3.4: Overhead imposed by the runtime monitor for four representative applications, measured in seconds of extra computation per transaction.

Notes: For each application, we show measurements for three models using a cluster of three vertical bars: the abstract stack model (leftmost bar), the callgraph model (middle), and the digraph model (rightmost). Each vertical bar uses shading to represent two measurements: the shorter, solid-colored segment represents the case where arguments are checked; the total height of the bar (including both the solid-colored and lightly-shaded regions) shows the case when arguments are ignored.

Performance In Figure 3.4, we show the runtime overhead incurred by our system when applied to four representative applications with known security vulnerabilities, `finger`, `qpopper`, `procmail`, and `sendmail`. Of these, `finger` is the smallest (at 1K lines of code, excluding comments, blank lines, and libraries), and `sendmail` is the largest (at 32K lines); the other two are in the middle. The height of each bar in Figure 3.4 indicates the performance overhead of each model, measured in seconds of extra computation per transaction[12].

The figures use shading to show the effect of checking system call arguments. One might expect that checking arguments could improve performance by reducing ambiguity in the model and thus reducing the number of possible paths through the model that we need to explore at runtime. The measurements confirm this hypothesis, showing that—even though we implemented only an extremely crude data-flow analysis—the performance benefits are substantial.

We initially expected that, due to its complexity, the abstract stack model would be consistently slower than the callgraph model. This is partially confirmed by our experiments, but we were surprised to find many exceptions. For instance, in the case of `procmail`, it appears that the improved precision provided by the abstract stack model more than makes up for the complexity of this model. In general, moving to more precise models may reduce the degree of non-determinism and thereby reduce the number of possible paths explored at runtime.

Note that there is a wide variation in running times. The digraph models are consistently extremely fast (the overhead is too small to measure), but the other models are sometimes vastly slower. For `sendmail`, the callgraph and abstract stack models were so slow that we forcibly terminated the experiment after an hour of computation. Since our goal is for real-time intrusion detection, imposing more than a few seconds of latency onto any interactive application is absolutely unacceptable;

[12] We use the word transaction to denote a single interactive event, such as delivery of a piece of email. For interactive applications that are not compute-intensive, we believe the main goal is to avoid introducing more than a few seconds of latency per transaction, and so we measure absolute rather than relative overheads. All measurements were performed on a 450 MHz Pentium II running Java, using IBM's JIT for Linux.

an hour is clearly several orders of magnitude too much. Consequently, for some applications, only the digraph model is fast enough; for others, the more sophisticated callgraph or abstract stack models are also workable. We conclude that, in all cases, at least one of the approaches provides acceptable performance, but the type of model must be chosen on a per-application basis.

Our prototype implementation has known problems that make its performance sub-optimal. See Section 7.

Mimicry attacks To motivate the need for precise models, we introduce a new class of attacks against intrusion detection systems, the *mimicry attack*. Recall that one of our primary design goals is to detect not only the attacks that are common today, but also to detect the attacks of the future. Furthermore, our model of the application probably cannot be kept secret from attackers. Consequently, our models need to be precise enough that there is no way for an attacker to cause any harm without deviating from the model, even when the attacker can predict what model we are using. Otherwise, attackers will be free to develop malicious exploit code that mimics the operation of the application, staying within the confines of the model and thereby evading detection by our system despite its harmful effects.

In general, if the attacker somehow obtains control of the application when our intrusion detection automata is in the state s, and if some insecure state s' is reachable from s through any path in the automata, then the attacker will be able to bring the system to an insecure state without risk of detection by synthesizing the system calls that make up the path $s \to \cdots \to s'$. We call this a *mimicry attack*, and we expect that, as intrusion detection becomes more widely deployed, mimicry attacks are unavoidable [116].

Note that imprecise models contain impossible paths, which introduces a vulnerability to mimicry attacks if any of those paths can reach an insecure state. Consequently, the primary defense against mimicry attacks lies in high-precision models.

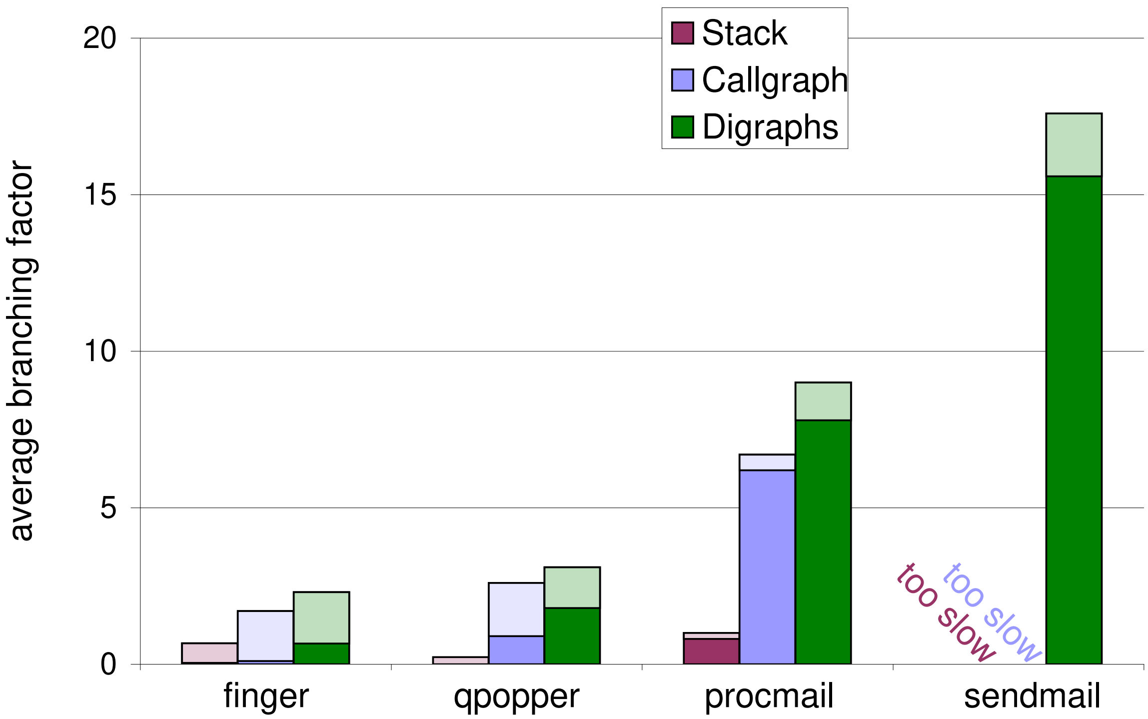

Figure 3.5: Precision of each of the models, as characterized by the average branching factor (see the text). Small numbers represent better precision.

Notes: For each application, we show measurements for three models using a cluster of three vertical bars: the abstract stack model (leftmost bar), the callgraph model (middle), and the digraph model (rightmost). Each vertical bar uses shading to represent two measurements: the shorter, solid-colored segment represents the case where arguments are checked; the total height of the bar (including both the solid-colored and lightly-shaded regions) shows the case when arguments are ignored.

Precision Unfortunately, we do not know the right way[13] to quantify an intrusion detection system's degree of robustness against mimicry attacks, so we do not have a complete characterization of the precision of our models. Nonetheless, we will attempt to give some intuition for the precision of our models by applying the following metric. Imagine freezing the intrusion detection system in the middle of some application execution trace. There is some set S of system calls that would be allowed to come next without setting off any alarms. We define the *branching factor* to be the size of S. A small branching factor means that the intruder has few choices about what to do next if she wishes to evade detection, and so we can expect that small branching factors leave the intruder most constrained and least able to cause harm. (In practice, not all system calls allow an intruder to cause harm, so we typically exclude the harmless ones (e.g., `read()` and `write()`) from the set S.) Finally, because we cannot predict at what point during execution the attacker might obtain control of the application, we suggest to measure the *average branching factor* over all normal execution traces.

We stress that the good values for the average branching factor metric do not guarantee security—a low average branching factor does not guarantee uniformly low branching factors (which is certainly important for security against a patient attacker), nor does it guarantee that all insecure states are unreachable—but this metric does seem to be provide a helpful starting point for comparison. We leave it as an open question to develop better metrics for quantifying robustness against mimicry attacks.

Figure 3.5 shows the precision of our models on our four sample applications, under the average branching factor metric. We can see that checking system call arguments provides substantial precision improvements, because it reduces the number of possible paths through the model, and because some system calls are harmless when their arguments are fixed in advance. For instance, an `open("/etc/motd", O_RDONLY)` call is harmless when its arguments are statically known, but otherwise could potentially be exploited by attackers to overwrite arbitrary files on the system. Our

[13] In practice, it may often be difficult even to identify just the set of insecure states of the system.

experience is that unchecked system call arguments greatly increase our exposure to mimicry attacks. Since checking arguments improves both performance and precision, we conclude that it should always be enabled.

We can also see that, when system call arguments are checked, the abstract stack model is much more precise than the callgraph model, which is itself more precise than the digraph model.

We have also examined the generated models by hand to evaluate how much harm a sophisticated attacker could cause using mimicry techniques. We are confident that all three of the `finger` models leave very little room for attack, due to the fact that the `finger` source code does little else but open a network connection and access world-readable files on the system. Results for the other applications, though, are mixed. The digraph model seems unlikely to resist a mimicry attack, and generally we feel it should not be relied upon for defense against malicious code specially tailored to fool our system. However, the abstract stack model seems to do fairly well: we believe it would successfully limit the harmful effects of any compromise in `qpopper` or `procmail`. On the other hand, for `sendmail`, the generated abstract stack model is too complex for us to make any determination.

Attacks detected We have tested our system on a number of known attacks from the past decade or so. For instance, each of the four applications discussed above has a known security vulnerability; we confirmed that we were able to detect the known attack on those applications.

Probably the most common class of attacks we detect are buffer overruns, which seem to account for perhaps half of all attacks in recent years [40, 145]. Because most existing exploit scripts grab full root privilege and take other distinctive actions (such as launching a shell under the attacker's control) immediately after exploiting the overrun vulnerability, detection is typically straightforward for our tool. Our tool may even be overkill for detecting misbehavior this blatant—many other systems will also detect these attacks, albeit with substantial false alarm rates—but an unusual feature of our tool is that it is also designed to detect some 'stealthy' mimicry attacks, too.

Our approach is also able to detect Trojan horses in trusted software. One current favorite of today's attackers is the `rootkit` toolkit, which replaces some system utilities with a version that contains a backdoor. We verified that our implementation was able to detect when some of these backdoors were exercised (which causes the behavior to deviate from that specified by the original source code).

The most interesting feature of our approach is that it can also detect more exotic attacks, even ones that the designers themselves did not know about. For instance, one extremely subtle attack exploited the ability to pass environment variables to `telnetd` to cause the dynamic linker to link with a shared library provided by the adversary; our system would have detected this attack, and any other dynamic-linking attack that might be discovered in the future, because our model is generated statically with the correct library. More recently, format string attacks have provided another unexpected way to introduce malicious code into vulnerable applications; since our detection mechanism makes no assumptions about how malicious code may be introduced, we can expect our system to apply to format string attacks, as well as to any other ways to take control of vulnerable applications that may be discovered in the future. We feel that these examples illustrate the importance of detecting unforeseen attacks.

Despite these successes, we feel strongly that our tool should not be used as the sole defense against any of these attacks, but instead should be used to complement other techniques. Prevention is often a more effective barrier, and intrusion detection systems are usually best viewed as a backup layer in case the main line of defense is breached.

Chapter 4

Efficient range analysis

We discussed earlier how a good algorithm for range analysis may be used to detect buffer overruns. Motivated by this observation, in this chapter we develop techniques for solving range analysis problem instances.

Recall that the *range analysis problem* is to infer, from a program's source code, the range of values that each integer program variable may assume. We describe an efficient new algorithm for integer range analysis. Our techniques are built on a constraint-based approach: we first derive a set of range constraints from the program source code through standard means, and then we solve those constraints using novel methods.

The main contribution of our work on range analysis comes in Section 4.4, where we describe an algorithm to solve systems of range constraints *precisely* and *efficiently*. We demonstrate experimentally that our algorithm scales to large, real-world programs.

It is noteworthy that our algorithm works without any loss of precision whatsoever. It is somewhat unusual to find cases where constraint systems over infinite-height domains may be solved precisely, and since the lattice of integer ranges has infinite height, one might not expect to be able to solve range constraints precisely. Nonetheless, our algorithm avoids the need for approximations. The key to success is a new technique, *precise widening around cycles*.

Precise widening around cycles forms a general framework for efficient and precise constraint solving over lattices of infinite height. The idea can be applied to solve constraints over a lattice $\langle L, \sqsubseteq \rangle$ of infinite height whenever certain technical conditions are satisfied:

1. All constraints must take the form $f(v_i) \sqsubseteq v_j$ (for some $f \in \mathsf{F}$, where F is a family of monotone, continuous functions with signature $L \to L$)[1];

2. F should be closed under composition (so that $f \circ g \in \mathsf{F}$ for all $f, g \in \mathsf{F}$); and

3. We require some way to solve for the least fixpoint of a single self-loop $f(v_i) \sqsubseteq v_i$ in isolation.

We expect that this approach to constraint solving may have applications elsewhere as well.

4.1 Motivation

The design of our range analysis is motivated by the following intuition. Suppose we are analyzing a program with k variables. Consider the statespace $\mathbb{Z}^k$ whose i-th component records the value of the i-th program variable. We may consider an execution of the program as a path through this statespace. With this perspective, our goal is to find a minimal bounding box that encloses all of the dynamically possible paths through the k-dimensional state space.

In this chapter, we give an efficient algorithm for finding a bounding box solution that encloses the set of all possible program executions. This is the *range analysis* problem.

Notice that our inference algorithm gives us bounds on the ranges of each program variable standing alone (and thus is a so-called *independent attributes* analysis), but it cannot give us any information on relationships that hold between multiple program

[1] However, in Section 4.5 we describe techniques to weaken this requirement, so that we may handle constraints of the form $f'(v_{i_1}, \ldots, v_{i_n}) \sqsubseteq v_j$ as well.

variables (as a *dependent attributes* analysis would). If we wanted to build a dependent attributes analysis, we could imagine computing a minimal convex polyhedron that encloses all the execution paths (using, e.g., the simplex method). This would return more precise results, because it would be able to infer program invariants that involve linear inequalities between multiple program variables. However, it would probably also scale up very poorly to the large problem instances encountered when analyzing real-world programs. For instance, our range analysis of `sendmail` (a medium- to large-sized security-critical application) used a constraint system with about 9k variables and 29k constraints. Consequently, we apply an independent attributes analysis so that we can handle large systems.

The simplification to bounding boxes is what allows our analysis algorithm to run very efficiently. It is this efficiency that explains our design choice to build an independent attributes analysis, despite its imprecision, rather than looking for a dependent attributes analysis, which would presumably be more precise but less efficient.

4.2 The constraint language

First we introduce a language of constraints, which we will use to model integer range operations.

Let $\mathbb{Z}$ denote the set of integers and write $\mathbb{Z}^\infty = \mathbb{Z} \cup \{-\infty, +\infty\}$ for the extended integers. The subsets of $\mathbb{Z}^\infty$ form a complete lattice with $\subseteq$ as the partial order. We restrict our attention to integer *ranges*. However, many of the comments here also apply more generally to arbitrary subsets of $\mathbb{Z}^\infty$.

A *range* is a closed interval $R \subseteq \mathbb{Z}^\infty$ of the form $[m, n] = \{i \in \mathbb{Z}^\infty : m \leq i \leq n\}$. When S is a subset of $\mathbb{Z}^\infty$, we write $\inf S$ and $\sup S$ for the infimum and supremum (with respect to $\leq$) of S; as a special case, for ranges we have $\inf [m, n] = m$ and $\sup [m, n] = n$. The *range closure* of any set $S \subseteq \mathbb{Z}^\infty$ is the minimal range R (with respect to $\subseteq$) containing S, i.e., $R = [\inf S, \sup S]$. For example, the set $S = \{-1, 0, 4\}$ has range closure $[-1, 4]$, since $\inf S = -1$ and $\sup S = 4$; note that the range notation $[-1, 4]$ is shorthand for the set $\{-1, 0, 1, 2, 3, 4\} \subseteq \mathbb{Z}^\infty$.

We extend the usual arithmetic operators to act on sets $S, T \subseteq \mathbb{Z}^\infty$ in the natural way:

$$\begin{aligned} S + T &= \{s + t : s \in S, t \in T\} \\ S - T &= \{s - t : s \in S, t \in T\} \\ S \times T &= \{s \times t : s \in S, t \in T\} \end{aligned}$$

For notational convenience we often write n as shorthand for the singleton set $\{n\}$, when $n \in \mathbb{Z}^\infty$. Thus, expressions such as $2T$ and $T + 7$ acquire their natural interpretation $2T = \{2\} \times T = \{2t : t \in T\}$ and $T + 7 = T + \{7\} = \{t + 7 : t \in T\}$.

When the result of an operation is not a range, we take its range closure. When this rule is followed, the extended arithmetical operators obey most of the usual algebraic laws. For instance, $S + T = T + S$, $S + 0 = S$, $S + S = 2S$, $S \times T = T \times S$, $0 \times S = 0$, and so on. However, the distributive rule does not hold (in general we only have $S \times (T + U) \subseteq S \times T + S \times U$; see [83]) and the rule for subtraction introduces a slightly ugly feature since in general $S - S \neq 0$.

In practice, it is useful to extend the constraint language to include min and max operators. We define their semantics as

$$\begin{aligned} \min(S, T) &= \{\min(s, t) : s \in S, t \in T\} \\ \max(S, T) &= \{\max(s, t) : s \in S, t \in T\} \end{aligned}$$

For example, when $S = \{1, 2, 3, 4\} = [1, 4]$ and $T = \{3, 4, 5, 6\} = [3, 6]$, we have $\inf T = 3$, $\min(S, T) = [1, 4]$, and $S - T = [-5, 1]$.

We define an *integer range expression* e as an expression generated by the following grammar:

$$\begin{aligned} e ::= v \mid n \mid n \times v \mid e + e \mid e - e \\ \mid \max(e, \ldots, e) \mid \min(e, \ldots, e). \end{aligned}$$

The terminals are integer constants, $n \in \mathbb{Z}^\infty$, and names of range variables, $v \in \text{Vars}$, a denumerable set of range variables. An *integer range constraint* has the form $e \subseteq v$. Notice we require the right-hand side to be a variable.

Note that equality constraints of the form $v+n = w$ fit within this framework, since they can be equivalently expressed as the pair of simultaneous constraints $v+n \subseteq w$, $w - n \subseteq v$. Equality constraints are useful for unifying variables that are discovered (during constraint generation) to refer to the same memory location.

An assignment $\alpha : v \mapsto \alpha(v) \subseteq \mathbb{Z}^\infty$ *satisfies* a system of constraints if all of the constraint assertions are true when the formal variable names v are replaced by the corresponding values $\alpha(v)$. For assignments α and β, we say that $\alpha \subseteq \beta$ if $\alpha(v) \subseteq \beta(v)$ holds for all variables v. The *least solution* to a constraint system is the smallest assignment α that satisfies the system, i.e., a satisfying assignment α such that any other satisfying assignment β obeys $\alpha \subseteq \beta$.

Theorem 1. *Every constraint system has a unique least solution.*

Proof. There is a natural intersection operator on assignments, defined by $(\alpha \cap \beta)(v) = \alpha(v) \cap \beta(v)$. The intersection of two satisfying assignments is also a satisfying assignment, since if $\alpha(e) \subseteq \alpha(v)$ and $\beta(e) \subseteq \beta(v)$, then $(\alpha \cap \beta)(e) \subseteq (\alpha \cap \beta)(v)$. This means that if a constraint system has a least solution, it is unique: supposing there are two minimal satisfying assignments $\alpha \neq \beta$, then $\alpha \cap \beta$ is another, smaller satisfying assignment, which contradicts the assumption of non-uniqueness. To show that a least solution exists, let α be the intersection of all satisfying assignments. This intersection is non-empty, since the trivial assignment $v \mapsto [-\infty, \infty]$ satisfies every constraint system. Clearly, if β satisfies the system, then $\alpha \subseteq \beta$. Therefore, α is a satisfying assignment, and it is the least such. $\square$

As we shall see later, a useful property of this formalism is that these constraint systems can usually be solved efficiently.

4.3 Constraint generation

The first phase of our range analysis algorithm generates a system of range constraints from the program source code. This system of range constraints is constructed to have a special property: if α is any satisfying assignment, then no program variable v can ever take on any value outside of $\alpha(v)$. In other words, the least solution is a sound abstraction of program execution.

Constraint generation becomes straightforward once we apply the technique of *abstract interpretation* [36]. We will not give all the details here—they are standard—but instead will sketch the form of the analysis to give the flavor of our constraint generation algorithm. We will assume that the program is given in some imperative language. We are primarily interested in C because, in practice, security programs are typically written in C; however, our range analysis techniques could be easily extended to other source languages as well. Program expressions will be written in a typewriter font (e.g., $\mathtt{v} := \mathtt{e}$). To each program variable $\mathtt{v}$, we associate a range variable $v \in \mathrm{Vars}$ for use in the constraint system; we write this judgement as $\mathtt{v} : v$, to indicate that the program expression $\mathtt{v}$ is represented in the constraint system by the range expression v.

Assignments are treated as follows. If the program source code contains an assignment between program variables of the form $\mathtt{v} := \mathtt{w}$, we may deduce that the value of $\mathtt{v}$ after execution of this statement might include any possible value of $\mathtt{w}$. Motivated by this reasoning, we introduce a range constraint $w \subseteq v$ to model this deduction, where the constraint variables v, w are intended to represent the program variables $\mathtt{v}, \mathtt{w}$. We write this rule formally as

$$\frac{\mathtt{v} : v, \quad \mathtt{w} : w}{\mathtt{v} := \mathtt{w} : w, \quad w \subseteq v}.$$

The predicate above the line is intended as an antecedent, and the judgement below the line is its consequence, so the rule says that if program variable $\mathtt{v}$ is represented by constraint variable v and if $\mathtt{w}$ is represented by w, then the program expression $\mathtt{v} := \mathtt{w}$ is represented by w and also we add the constraint $w \subseteq v$ to the current constraint system. This rule is conservative (it may introduce imprecision) but it will never err on the side of incorrectness (it is sound).

More general program expressions may be modelled using the following inference rules.

$$\frac{\mathtt{n} \in \mathbb{Z}^\infty}{\mathtt{n} : n}$$

$$\frac{\mathtt{n} \in \mathbb{Z}^\infty, \quad \mathtt{e} : v, \quad w \text{ fresh}}{\mathtt{n} \times \mathtt{e} : w, \quad n \times v \subseteq w}$$

$$\frac{\mathtt{e} : v, \quad \mathtt{e}' : v', \quad w \text{ fresh}}{\mathtt{e} + \mathtt{e}' : w, \quad v + v' \subseteq w}$$

$$\frac{\mathtt{e} : v, \quad \mathtt{e}' : v', \quad w \text{ fresh}}{\mathtt{e} - \mathtt{e}' : w, \quad v - v' \subseteq w}$$

$$\frac{\mathtt{v} : v, \quad \mathtt{e} : w}{\mathtt{v} := \mathtt{e} : w, \quad w \subseteq v}.$$

For example, from the assignment `i = i+j+17` we may infer the constraint $i+j+17 \subseteq i$ (simplifying constraints a bit).

We model functions *monomorphically*, i.e., we merge information for all call sites of the same function. Let `f` be a function defined as $\mathtt{T\ f(T'\ v)}$, so that it has a formal integer parameter `v` of type $\mathtt{T}'$. We add an implicit variable $\mathtt{r_f}$ of type `T` to hold the return value of `f`, and we treat each `return` statement in `f` as an assignment to $\mathtt{r_f}$. Each function call $\mathtt{b} := \mathtt{f(u)}$ is treated as an assignment of the actuals to the formals (i.e., $\mathtt{v} := \mathtt{u}$) followed by an assignment that carries the return value of `f` to `b` (i.e., $\mathtt{b} := \mathtt{r_f}$). Note that the body of each function is processed only once, so this strategy is simple and efficient, albeit imprecise.

We may also extend this technique to handle pointers. If we have deduced (through some other mechanism, e.g., a points-to analysis) the set of program locations that could be pointed to by each pointer, we may handle pointers analogously. Write $\mathtt{p} \mapsto \{\mathtt{v}_1, \ldots, \mathtt{v}_n\}$ if the pointer `p` can only contain a reference to one of the variables $\mathtt{v}_i$. Then we may introduce rules to handle pointer dereferencing (e.g., expressions such as $*\mathtt{p}$, where `p` is a pointer), as follows:

$$\frac{\mathtt{p} \mapsto \{\mathtt{v}_1, \ldots, \mathtt{v}_n\}, \quad \mathtt{v}_i : v_i, \quad w \text{ fresh}}{*\mathtt{p} : w, \quad v_1 \subseteq w, \ldots, v_n \subseteq w}$$

$$\frac{\mathtt{p} \mapsto \{\mathtt{v}_1, \ldots, \mathtt{v}_n\}, \quad \mathtt{v}_i : v_i, \quad \mathtt{e} : w}{*\mathtt{p} := \mathtt{e} : w, \quad w \subseteq v_1, \ldots, w \subseteq v_n}$$

In the above, we have relied on being able to generate *fresh* new constraint variables that have not been used before.

In the above we have described a *flow-insensitive* analysis, i.e., an analysis where we ignore all control flow and disregard the order of statements. Of course, the same ideas could also be used to derive a flow-sensitive analysis with only slightly more effort.

In our prototype implementation, we decided to use a flow-insensitive analysis for better scalability and simplicity of implementation. This approach does have costs, though: flow-insensitive analyses sacrifice some precision for significant improvements in scalability, efficiency, and ease of implementation. We do not claim that flow-insensitive analysis is necessarily the best approach for a production-quality buffer overrun tool. Instead, we merely observe that its advantages (ease of implementation, scalability) mapped well to our initial goals (construction of a proof-of-concept prototype that can analyze large, real-world applications).

4.4 Solving integer range constraints

In this section, we give an efficient algorithm for solving the system of constraints generated from the program source code. In practice, our algorithm scales linearly on our benchmarks, thanks to the simplification to a "bounding box" analysis.

Let $\mathsf{AF} := \{x \mapsto ax + b : a \in \mathbb{Z}, b \in \mathbb{Z}^\infty\}$ denote the set of affine integer functions, and extend each affine function $f : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$ to operate on ranges in the natural way, i.e., define $f(R) := \{f(r) : r \in R\}$.

We develop a constraint solver by beginning with the simplest case: we assume that arithmetic and min/max expressions are omitted, so that each constraint has the form $f(v_i) \subseteq v_j$ for some $f \in \mathsf{AF}$.

Then, we form the *dependence graph* associated with the constraint system to be solved. The dependence graph is a directed graph whose vertices are the variables

v_i mentioned in the constraint system. For each constraint $f(v_i) \subseteq v_j$ we add the labeled directed edge $v_i \xrightarrow{f} v_j$.

As the algorithm proceeds, we maintain an assignment α to the constraint variables. Each vertex v_i is marked with a range $\alpha(v_i)$ giving the current estimate of the solution. We use an *optimistic algorithm*, which means that the current assignment monotonically increases over time until we have found a satisfying assignment. Once we have found a satisfying assignment, we show that it will in fact form the least solution to the constraint system. The advantage of optimistic algorithms is that they are generally more precise and easier to combine with other analyses [33, 114]. The advantage of pessimistic algorithms is that they can be stopped at any time to get an upper bound on the solution.

More concretely, we begin by setting all ranges to the initial value $\alpha(v_i) := \emptyset$. Then, all constraints of the form $n \subseteq v$ are processed by setting $\alpha(v_i) :=$ $\textsc{Range-closure}(\alpha(v_i) \cup \{n\})$. After this initialization phase, the main constraint solving engine is invoked.

The solver works by propagating information in the dependence graph. We say that an edge $v_i \xrightarrow{f} v_j$ is *active* if $f(\alpha(v_i)) \not\subseteq \alpha(v_j)$. To propagate information along such an active edge (also known as *relaxation*), we update the assignment according to the rule

$$\alpha(v_j) := \textsc{Range-closure}(\alpha(v_j) \cup f(\alpha(v_i))).$$

An *augmenting path* is one containing only active edges. (This wording is in deliberate analogy to standard algorithms for shortest-paths and network flow problems.) The goal of the algorithm is to find augmenting paths and propagate information along them by relaxing the upper bounds on the solution.

If the resulting directed graph is acyclic, we can trivially solve the constraint system in linear time: we topologically sort the graph and propagate information along each edge in sorted order. Graphs with cycles are harder to handle.

The approach given above can be rephrased in the perhaps more familiar language of fixpoints over lattices. Each constraint $f(v_i) \subseteq v_j$ induces a continuous function F

on assignments given by

$$(F(\alpha))(v_k) = \begin{cases} \alpha(v_j) \cup f(\alpha(v_i)) & \text{if } j = k \\ \alpha(v_k) & \text{otherwise,} \end{cases}$$

and in this way the constraint system gives us a set of such functions $\{F\}$. Now note that a satisfying assignment for the constraint system forms a fixpoint for all the F's, and vice versa. Therefore, we are seeking the least fixpoint of the functions $\{F\}$, because it will be the least solution[2] to the constraint system.

We could search for the fixpoint using a standard worklist algorithm that visits all the augmenting paths in breadth-first order and propagates information along them by relaxation. However, the basic worklist algorithm would exhibit serious problems: in the presence of cycles, it might not terminate! For instance, consider the constraint system containing the two constraints $5 \subseteq v$ and $v + 1 \subseteq v$. A naive algorithm would loop forever, revising its initial estimate $\alpha(v) = [5, 5]$ to $[5, 6]$, $[5, 7]$, $[5, 8]$, etc. This "counting to infinity" behavior arises because the lattice of ranges has infinite ascending chains, and thus the monotonicity of $\{F\}$ is not enough to ensure termination.

At this point, we have three options for restoring termination.

1. We could restrict attention to those programs that induce acyclic constraint systems.

2. We could introduce a widening operator that raises variables involved in cycles to the trivial solution $[-\infty, \infty]$, as pioneered in [36] and [38]. This avoids infinite ascending chains.

3. We could directly solve the constraint subsystem associated with each cycle, using domain-specific information about the constraint language.

The first is not very attractive, because real programs often involve cycles, such as those created by loops and recursion. Even worse, cycles are almost unavoidable for a

[2] Since all the F are monotonic and even continuous, a least fixpoint exists.

flow-insensitive analysis: for example, the C assignment `i = i+1` will always induce a cycle in the form of a constraint $i+1 \subseteq i$. One disadvantage of the second option is that it introduces imprecision, i.e., it will only provide an approximate solution (an upper bound on the least satisfying assignment).

This paper follows the third option. We show how to avoid divergent behavior, without introducing any imprecision, by directly solving for the fixpoint of the constraint subsystem associated with each cycle.

A typical cycle looks like

$$f_1(v_1) \subseteq v_2,\ \ldots,\ f_{n-1}(v_{n-1}) \subseteq v_n,\ f_n(v_n) \subseteq v_1.$$

Transitively expanding this cycle, we find that $f(v_1) \subseteq v_1$ where $f = f_n \circ \cdots \circ f_1$. (We may view f loosely as Shostak's *loop residue* [132] for the cycle.) The composition of affine functions is affine, so f is affine. The observation is that we can precisely solve this cyclic constraint system without any divergence whatsoever, by using a simple fact on the fixpoints of affine functions.

Lemma 1. *Let $f(x) = ax + b$ be an affine function in AF with $a > 0$, let R be a range, and let $S \subseteq \mathbb{Z}^\infty$ be the minimal range satisfying $R \subseteq S$ and $f(S) \subseteq S$. Then (1) $\sup S = \infty$ if $\sup f(R) > \sup R$; also, (2) $\inf S = -\infty$ if $\inf f(R) < \inf R$. If neither clause (1) nor clause (2) applies, we have $S = R$. If both clauses apply, we have $S = [-\infty, \infty]$.*

Proof. Let $R = [d, e]$, so that $\sup f(R) = f(e)$ (since f is monotone and $a \geq 1$). If $f(e) > e$, then $f(x) > x$ for all $x \geq e$ (since $a \geq 1$), so that $f(f(e)) > f(e) > e$, etc., and (1) is proved by induction. (2) follows similarly. Finally, if neither clause holds, then $f(R) \subseteq R$, and by the minimality of S we have $S = R$. $\square$

Theorem 2. *We can solve the constraint subsystem associated with a cycle in linear time.*

Proof. It suffices to show that this is true for $a > 0$. (If $a = 0$, it is trivial; if $a < 0$, we apply the lemma to $f \circ f$ by traversing the cycle twice.) We show that it suffices to simply compute $f(\alpha(v_1))$ and compare the result with $\alpha(v_1)$. If $f(\alpha(v_1)) \subseteq \alpha(v_1)$, the least solution is $\alpha(v_1)$, and we can stop traversing the cycle. Otherwise, one or both of the clauses of the lemma apply. If both apply, we are done: set $\alpha(v_1) := [-\infty, \infty]$, and let the worklist algorithm trace out the implications for the v_j. If just one applies—WLOG suppose it is (1)—we simply apply the lemma (a second time) to $R' = [\inf R, \infty]$, and we will be done after this second application. Computing f requires time linear in the length of the cycle, and propagating the result of the analysis around the cycle also requires linear time, so the whole process runs in linear time. □

To restate the theorem intuitively: if we ever find an augmenting path that traverses an entire cycle, the theorem shows us how to immediately apply a widening operator *without any loss of precision whatsoever*. This provides a simple way to avoid the "counting to infinity" behavior that arises from traversing a cycle multiple times: we simply search for cycles and apply our precise widening technique to each cycle as soon as we encounter it. Thus, the real contribution of Theorem 2 is that it shows how to find the fixpoint of the system precisely and efficiently. Since we are working in a lattice with infinite ascending chains, standard techniques cannot provide this. We will use the phrase *precise widening around cycles* to describe the technique of looking for cycles and solving each one precisely in isolation, as introduced above.

Figure 4.1 presents an algorithm that uses precise widening around cycles to solve integer range constraints efficiently. We use a depth-first search. Each back-edge we encounter during the depth-first search is then extended to an active cycle in the dependence graph. The HANDLE-CYCLE procedure retraces the cycle discovered in the depth-first search using the predecessor pointers and then processes that cycle using the algorithm sketched in the proof of Theorem 2. Note that we specifically require the use of depth-first search (and not breadth-first search, or some other worklist-based algorithm) so that we can recover the edges participating in the cycle as soon as we see a back-edge.

In theory, this solution process could take $O((n + m)k)$ time in the worst case, where k counts the number of cycles in the graph. In practice, though, k seems to be

Constraint-solver

1. Set $\alpha(v_i) := \emptyset$ for all i, and set *done* := false.
2. For each constraint of the form $n \subseteq w$, do
3. Set $\alpha(w) :=$ Range-closure($\alpha(w) \cup \{n\}$).
4. While *done* $\neq$ true, call One-iteration.

One-iteration

1. Set $C(v_i) :=$ white for all i, set $q := |\mathrm{Vars}|$, and set *done* := true.
2. For each variable v, do
3. If $C(v) =$ white, do
4. Set prev(v) := null and call Visit(v).

Visit(v)

1. Set $C(v) :=$ gray.
2. For each constraint of the form $f(v) \subseteq w$, do
3. If $f(\alpha(v)) \not\subseteq \alpha(w)$, do
4. Set $\alpha(w) :=$ Range-closure($\alpha(w) \cup f(\alpha(v))$).
5. Set *done* := false.
6. If $C(w) =$ gray, call Handle-cycle(f,v,w,prev).
7. If $C(w) =$ white, do
8. Set prev(w) := v and call Visit(w).
9. Set $C(v) :=$ black and $r(v) := q$ and $q := q - 1$.

Range-closure(S)

1. Return the range $[\inf S, \sup S]$.

Figure 4.1: An algorithm that efficiently solves systems of integer range constraints.

small, and the algorithm usually runs in linear time, probably because of sparsity and locality in the constraint systems that arise during the analysis of typical programs.

We expect that the idea of precise widening around cycles may also be fruitful in more general settings. In particular, the algorithm shown in Figure 4.1 may be adapted to work over any lattice of infinite height, provided that there is some analogue of Theorem 2 to show how to solve for the least fixpoint of a cycle. Such an algorithm will always terminate if the precise widening operator may only be applied to an active cycle a finite number of times before there are no more active cycles to be handled. Note, though, that this technique is only applicable where all constraints take the form $f(v_i) \sqsubseteq v_j$. In particular, it is not applicable where we have a constraint $g(v_{i_1}, \ldots, v_{i_n}) \sqsubseteq v_j$ that mentions more than one variable on its left-hand-side.

This concludes our treatment of constraint solving for simple constraints. We have extended the algorithm to handle the full constraint language, including multi-variable expressions and min/max operators. The details may be found in the next section.

4.5 Extensions

In this section, we extend the basic algorithm presented in Section 4.4 to handle more general constraints. Let us first review how far we have come. We have an efficient algorithm that handles *simple constraints*, i.e., constraints of the form $av_i + b \subseteq v_j$. We have precise techniques for handling cycles. But the algorithms presented so far cannot handle arithmetic or min/max expressions on the left hand side of the constraint. Such constraints are relatively rare: for typical program analysis tasks, only about 2% of the constraints use complex arithmetical expressions, and less than 1% use min/max expressions. Nonetheless, they are still important enough that they cannot be ignored: consider, e.g., the C statement `sprintf(dst, "foo: %s %s", s, t)` to see why we need complex arithmetical expressions; also, modelling the standard library functions `strncpy()`, `snprintf()`, etc., clearly requires support for min/max expressions. We now describe how to extend the algorithm to handle these more general types of constraints.

Let C be a constraint system consisting of a system of simple constraints C' along with the complex constraint

$$a_1 v_1 + \cdots + a_n v_n + b \subseteq w. \tag{4.1}$$

We show how to construct a *reduced constraint system* $R_\alpha(\mathsf{C})$ containing only simple constraints, where the least solution to $R_\alpha(\mathsf{C})$ gives a useful lower bound on the solution to C. The idea is to note that, for each j, (4.1) implies $a_j v_j + b_j \subseteq w$, where the b_j's are given by

$$b_j = b + \sum_{i=1,\ldots,n;\, i \neq j} a_i \alpha(v_i)$$

and α is any lower bound on the least satisfying assignment to C. Thus we may take $R_\alpha(\mathsf{C}) = \mathsf{C}' \cup \{a_j v_j + b_j \subseteq w : j = 1, \ldots, n\} \cup \{\alpha(v_j) \subseteq v_j : j = 1, \ldots, n\}$, where the constants b_j are defined in terms of α as above. By construction, any satisfying assignment for C will then satisfy $R_\alpha(\mathsf{C})$.

In principle, this immediately yields an algorithm for solving a constraint system C containing complex arithmetical expressions: compute the least solution β to $R_\alpha(C)$ (using the algorithm in Figure 4.1) and set $\alpha := \alpha \cup \beta$, repeating these two steps iteratively until convergence. Termination is guaranteed since a cycle in C will induce a cycle in $R_\alpha(\mathsf{C})$ and thus will be processed efficiently by the Handle-cycle procedure.

In practice, our implementation exploits a more efficient approach, where we update the reduced system $R_\alpha(C)$ in place as α is updated. In the optimized algorithm, each change to $\alpha(v_i)$ in the algorithm of Figure 4.1 immediately forces an update to $R_\alpha(C)$ whenever v_i participates in the left-hand side of some complex constraint. This technique seems to work very well for our purposes, probably because complex constraints are relatively rare.

The approach used to handle to min/max constraints is currently very simplistic: the current implementation propagates information through min/max constraints but does not attempt to handle cycles containing min/max constraints. In principle, this could introduce "counting to infinity", but we have yet to encounter this behavior. This simplification reflects implementation considerations more than any fundamental difficulty with handling this type of constraints. If we ever encounter cycles containing min or max operations, we will implement the following extension of Lemma 1 to min/max constraints:

Lemma 2. *Let $f(x) = \min\{g_1(x), \ldots, g_n(x), c\}$ for $g_1, \ldots, g_n \in \mathsf{AF}$ and $c \in \mathbb{Z}^\infty$, where each g_j is of the form $g_j(x) = a_j x + b_j$ for $a_j > 0$. Let R be a range, and let $S \subseteq \mathbb{Z}^\infty$ be the minimal range satisfying $R \subseteq S$ and $f(S) \subseteq S$. Then (1) $\inf S = -\infty$ if $\inf f(R) < \inf R$; also, (2) $\sup S = c$ if $\sup f(R) > \sup R$. If neither clause (1) nor clause (2) applies, we have $S = R$. If both clauses apply, we have $S = [-\infty, \infty]$.*

Proof. Clause (1) is an immediate consequence of Lemma 1: if $\inf g_j(R) < \inf R$, then $-\infty \in S$, since $\inf f(S) \leq \inf g_j(S)$ for all S. To prove clause (2), note that $\sup S \leq c$, so it suffices to prove that $\sup S \geq c$. Suppose not, i.e., that $\sup S < c$. Let $e = \sup R$. Since $\sup f(R) > \sup R$, we have $g_j(e) > e$ for all j. Also, by Lemma 1, $g_j(x) > x$ for all $x \geq e$ and for each j. Since $f(S) \subseteq S$, we must have $\min\{g_1(S), \ldots, g_n(S), c\} \leq \sup S$, and so $g_j(\sup S) \leq \sup S$ for all j. At the same time, $R \subseteq S$ implies $\sup S \geq e$, so $g_j(\sup S) > \sup S$, a contradiction, which establishes clause (2). Finally, if neither clause applies, then $f(R) \subseteq R$, and by the minimality of S we have $S = R$. $\square$

The algorithm could be further improved with slightly more sophisticated techniques. For example, we could compute the acyclic component graph (where each strongly connected component is shrunk down to one vertex) and then iteratively process each strongly connected component in topologically sorted order, using a depth-first search to discover the cycles within each strongly connected component. However, we have not explored these possibilities for optimization, because the existing solver is already much faster than necessary.

Chapter 5

A general context-free recognizer

In this chapter we describe the design of a novel top-down algorithm for recognizing general context-free languages. This algorithm was developed for use in our intrusion detection system (see Chapter 3); we use it to verify that execution traces comply with an abstract stack model (see Section 3.3.3). In essence, we build a top-down parser for context-free languages.

Our intrusion detection system also introduces two special requirements. First, the algorithm must support online parsing, where the top-down parse tree—or, more generally, the set of possible parse trees (in the case of ambiguities, there may be many possible parse trees)—is updated incrementally as each input symbol is read. Second, the algorithm must support external access to the set of parse trees and must operate correctly even if this set is modified by the intrusion detection system between processing of input symbols. We show how to support these needs economically, without loss of efficiency.

The main theoretical tool is a characterization of the reachable configurations of a pushdown automaton. We prove that the set of reachable stack configurations of a pushdown automaton may be concisely represented as a regular language. Moreover, we show how we may efficiently update this representation on the fly as we simulate the operation of the pushdown automaton.

This allows us to build an efficient simulator for arbitrary pushdown automata.

By applying this idea to the pushdown automaton associated with top-down parsing of the specified context-free language, we obtain a general top-down algorithm for recognizing general context-free languages. Moreover, because our simulation works on the set of parse stacks represented in a concise symbolic form, we obtain a recognizer with competitive runtime complexity.

This insight yields a recognizer that may be viewed as a generalization of Tomita's algorithm [138], and we suggest that this connection may be of independent interest. See Section 5.5 for a discussion.

5.1 Notation and conventions

We adopt a special formalization of a non-deterministic pushdown automaton (NDPDA). For convenience, we assume that the NDPDA's finite-state control has exactly one state. This assumption comes at no serious loss of generality, because any NDPDA can transformed so that it takes on this form.

In our formalization, the pushdown automaton comes equipped with an input alphabet Σ_I, a stack alphabet $\Theta = \Sigma_I \cup \Sigma_S$ (where Σ_S is disjoint from Σ_I), and an initial stack symbol $\jmath_0 \in \Theta$.

For convenience, we assume that the transitions of the NDPDA have a special form: the only transitions that consult the input tape are ones that match the next input symbol against the symbol on the top of the stack; assuming a match is found, the symbol is consumed from the tape and popped from the stack. The automaton may also contain ϵ-transitions, which do not consult the input tape. In fact, this is the only degree of freedom accorded to the NDPDA designer, and the ϵ-transitions are specified by a map $\tau : \Sigma_S \to 2^{\Theta^*}$. The intended interpretation is that, whenever the stack symbol $a \in \Sigma_S$ is on the top of the stack, the automaton may non-deterministically select some $t \in \tau(a)$, pop a, and push the string t (all without moving the input head). The only case where the automaton is allowed to move its input head is where the current input symbol a matches the stack symbol on the top of the stack, and in this case a is popped from its stack and the input head moves to consume the input symbol a. Every NDPDA can be transformed into one that takes this form.

Finally, we assume that $\tau(a) \subseteq \{\epsilon\} \cup \Theta \cup \Theta^2$ for all a. As always, this assumptions incurs no loss of generality, because every NDPDA can be modified to take this form.

We express every stack configuration as a string of stack symbols, with the top of the stack at the rightmost position. Thus, stack configurations may be treated as elements of Θ^*, and sets of reachable stack configurations may be represented as subsets of Θ^*.

It will also be useful to establish some additional notation. For a pair of stack configurations $t, t' \in \Theta^*$ and an input string $s \in \Sigma_I^*$, we write $t \overset{s}{\Rightarrow} t'$ to show that the pushdown automaton may transition from stack configuration t to stack configuration t' while consuming the input string s. Define $\mathsf{post}^*[s] : 2^{\Theta^*} \to 2^{\Theta^*}$ by

$$\mathsf{post}^*[s](L) := \{t' \in \Theta^* : t \overset{s}{\Rightarrow} t' \text{ for some } t \in L\};$$

the notation $\mathsf{post}^*[s](L)$ is intended to represent the set of stack configurations that may be reached from one of the starting configurations in $L \subseteq \Theta^*$ by consuming the input string $s \in \Sigma_I^*$. Finally, let

$$L/a := \{t \in \Theta^* : ta \in L\}$$

denote the right-quotient of the language L with the symbol $a \in \Theta$. In our algorithm, we will exploit the well-known fact that L/a is regular when L is. Note that the empty set $\emptyset$ is a regular language, so L/a is regular even in the special case where L does not contain any strings ending in the symbol a.

5.2 Observations

Our results are as follows. (See also [23, 27, 1, 62, 146] for special cases and related results.)

Lemma 3. $\mathsf{post}^*[\epsilon](L)$ *is a regular language when* L *is.*

Lemma 4. *If $a \in \Sigma_I$, then* $\mathsf{post}^*[a](L) = \mathsf{post}^*[\epsilon](\mathsf{post}^*[\epsilon](L)/a)$.

Theorem 3. *Fix a pushdown automaton and an input string $s \in \Sigma_I^*$. Then the set of stack configurations reachable after reading s is a regular language, and moreover may be recognized by an effectively computable finite automaton with $O(|s|)$ states.*

Proof. Since L/a is regular when L is, by induction on the length of s and repeated application of Lemmas 3 and 4, $\mathsf{post}^*[s](L)$ is regular when L is. But now $\{\jmath_0\} \subseteq \Theta^*$ is trivially regular, and $\mathsf{post}^*[s](\{\jmath_0\})$ represents exactly the set of stack configurations reachable after reading the input string s, so the set of stack configurations is indeed regular. The number of states needed comes from a close examination of Algorithm A (see below). □

All of the heavy lifting is done in Algorithm A, which shows how to compute $\mathsf{post}^*[\epsilon](L)$ from L. See Figure 5.1. (Lemma 3 follows immediately from a proof of correctness for this algorithm; see below for details.) We represent each regular language as a NDFA. As a notational device, in Figure 5.1 and elsewhere we write $q \overset{a}{\mapsto} q'$ if there is a transition from state q to state q' with label a in the NDFA under consideration.

Algorithm A takes a NDFA for L and extends it to a NDFA for $\mathsf{post}^*[\epsilon](L)$ by adding states and transitions. We never create duplicate transitions; if a transition to be added is already present in the NDFA, we do nothing. Note that adding new transitions may trigger the addition in Step 6 of further transitions, and so on, but the process will always terminate, because the set of transitions grows monotonically and the total number of possible transitions is finite. This fixpoint computation may be implemented with a standard worklist algorithm or with an application of a HornSAT solver [47].

Algorithm A provides the heart of a simple algorithm for recognizing general context-free languages. To check whether the input string $s \in \Sigma_I^*$ is accepted by a given NDPDA, we simply compute $\mathsf{post}^*[s](\{\jmath_0\})$ using repeated application of Algorithm A and then check whether $\epsilon \in \mathsf{post}^*[s](\{\jmath_0\})$. Each application of Algorithm A requires at most $O(n^2)$ steps of computation when its input NDFA has n states; since

Algorithm A.

Input: A NDFA accepting $L \subseteq \Theta^*$.

Output: A NDFA accepting $\mathsf{post}^*[\epsilon](L)$.

1. Add a fresh accepting state q_A to the NDFA.
2. For each accepting state $q \neq q_A$ of the NDFA, do
3. Add a new transition $q \overset{\epsilon}{\mapsto} q_A$ to the NDFA, and make q non-accepting.
4. For each $a \in \Theta$,
5. Add a fresh state δ_a and a new transition $\delta_a \overset{a}{\mapsto} q_A$ to the NDFA.
6. Repeat the following rules as many times as they are applicable:

   (Rule 0) If $q \overset{\epsilon}{\mapsto} q_A$ and $p \overset{t}{\mapsto} q$, add a new transition $p \overset{t}{\mapsto} q_A$.

   (Rule 1) If $q \overset{a}{\mapsto} q_A$ and $a \in \Sigma_S$ and $t \in \tau(a)$ for some $t \in \{\epsilon\} \cup \Theta$,
   add a new transition $q \overset{t}{\mapsto} q_A$.

   (Rule 2) If $q \overset{a}{\mapsto} q_A$ and $a \in \Sigma_S$ and $bc \in \tau(a)$ for some $b, c \in \Theta$,
   add a new transition $q \overset{b}{\mapsto} \delta_c$.
7. Return the modified NDFA.

Figure 5.1: The heart of our context-free recognizer: An algorithm for computing the ϵ-closure of a set of configurations of a NDPDA.

$n \leq (|\Sigma_I|+1) \times |s|$ at all times, our context-free recognizer has a worst-case running time of $O(|s|^3)$ (where the big-O notation hides a multiplicative factor of $|\Sigma_I|^3$).

The worst-case running time of Algorithm A is cubic in the length of the input, so the asymptotic running time of our recognizer is comparable to that of other well-known algorithms. However, in practice, our algorithm has three key advantages:

- It is well-suited to online parsing, where we update the set of possible stack configurations immediately as each input symbol becomes available;
- It allows for outside inspection and modification of the set of stacks, so the context-free framework can be extended as needed (see, e.g., Section 3.4.1 for examples); and,
- Unlike Tomita or CYK parsing, it allows for *top-down* parsing of general context-free languages, as needed for our intrusion detection application.

5.3 Proofs

Proof of Lemma 3. Run Algorithm A on a NDFA for L, and let L' denote the (regular) language accepted by the output of the algorithm. We show that $L' = \mathsf{post}^*[\epsilon](L)$.

First note that we have $\mathsf{post}^*[\epsilon](L) \subseteq L'$ by construction: if the stack configuration $u \in L$ may ϵ-transition to another configuration u' in a single step, then any accepting path for u through the L-NDFA may be readily extended to an accepting path for u' through the L'-NDFA, so by induction on the number of ϵ-transitions we have $\mathsf{post}^*[\epsilon](L) \subseteq L'$.

Finally we show that $L' \subseteq \mathsf{post}^*[\epsilon](L)$. Note that, after adding all transitions, q_A still has no outgoing transitions, and all successors of each δ_t are in the set $\{\delta_u : u \in \Theta\} \cup \{q_A\}$, and $L \subseteq L'$ (since we only add transitions and never delete them). Let $\beta(q)$ denote the set of strings accepted by the L'-NDFA if we start at state q. We show that, as we add each transition according to the three rules above, we always have the invariants $L \subseteq L' \subseteq \mathsf{post}^*[\epsilon](L)$ and $\beta(\delta_t) \subseteq \mathsf{post}^*[\epsilon](\{t\})$ for all $t \in \Theta$.

Clearly this is true before we add any transitions. Since rule 0 has no effect on L' or $\beta(\delta_t)$, it does not destroy the invariants. Rule 1 maintains the invariants, too: if we consider any path $q_0 \overset{s}{\rightsquigarrow} q \overset{t}{\mapsto} q_A$ enabled for the first time by an application of rule 1, we see that $sa \in \mathsf{post}^*[\epsilon](L)$ (since $q_0 \overset{s}{\rightsquigarrow} q \overset{a}{\mapsto} q_A$ was an accepting path before this application of rule 1); but now $t \in \tau(a)$ so $sa \overset{\epsilon}{\Rightarrow} st$ and thus $st \in \mathsf{post}^*[\epsilon](L)$, which means that the first invariant remains satisfied. A similar argument shows that rule 2 also respects the first invariant: if $q_0 \overset{s}{\rightsquigarrow} q \overset{b}{\mapsto} \delta_c \overset{w}{\rightsquigarrow} q_A$ is a path whose $q \overset{b}{\mapsto} \delta_c$ link is newly created by rule 2, then $q_0 \overset{s}{\rightsquigarrow} q \overset{a}{\mapsto} q_A$ is an accepting path so $sa \in \mathsf{post}^*[\epsilon](L)$, and also $a \overset{\epsilon}{\Rightarrow} bc \overset{\epsilon}{\Rightarrow} bw$ since $w \in \beta(q_c)$, hence $sbw \in \mathsf{post}^*[\epsilon](L)$ and the first invariant remains satisfied. The second invariant is handled similarly. This shows that $L' \subseteq \mathsf{post}^*[\epsilon](L)$, and the lemma follows. $\square$

Proof of Lemma 4. First, we show that $\mathsf{post}^*[a](L) \subseteq \mathsf{post}^*[\epsilon](\mathsf{post}^*[\epsilon](L)/a)$. Suppose $w \in \mathsf{post}^*[a](L)$, so that for some $t \in L$ we have $t \overset{a}{\Rightarrow} w$. Then there is some $u, v \in \Theta^*$ so that $t \overset{\epsilon}{\Rightarrow} u \overset{a}{\Rightarrow} v \overset{\epsilon}{\Rightarrow} w$ where u transitions under a to v in a single step. But then $u \in \mathsf{post}^*[\epsilon](L)$, and $u = va$, so that $v \in \mathsf{post}^*[\epsilon](L)/a$, whence $w \in \mathsf{post}^*[\epsilon](\mathsf{post}^*[\epsilon](L)/a)$, as desired.

The other direction is justified similarly. Fix any $w \in \mathsf{post}^*[\epsilon](\mathsf{post}^*[\epsilon](L)/a)$. This means that there must be $t, u, v \in \Theta^*$ so that $t \in L$ and $t \overset{\epsilon}{\Rightarrow} u$ and $u = va$ and $v \overset{\epsilon}{\Rightarrow} w$. But then u can transition under a to v in a single step, so $t \overset{a}{\Rightarrow} w$, whence $w \in \mathsf{post}^*[a](L)$. This shows that $\mathsf{post}^*[a](L) \supseteq \mathsf{post}^*[\epsilon](\mathsf{post}^*[\epsilon](L)/a)$, completing the proof. $\square$

5.4 Extensions

As a further curiousity, we may extend $\mathsf{post}^*[\cdot](\cdot)$ in a natural way to operate on sets of input strings by defining

$$\mathsf{post}^*[S](L) := \bigcup_{s \in S} \mathsf{post}^*[s](L).$$

Then one may readily extend our techniques above to show the following result.

Theorem 4. *The language* $\mathsf{post}^*[R](L)$ *is regular when* R *and* L *are. Moreover, when* R, L *are represented by finite automata with* s_R, s_L *states,* $\mathsf{post}^*[R](L)$ *may be represented by an effectively computed finite automaton with* $O(s_R \cdot s_L)$ *states.*

The algorithm is simply a straightforward extension of Algorithm A; for each state p_i of the NDFA for R, we introduce a state q_A^i to represent (i.e., accept) sets of parse-stacks possible when the NDFA for R is in state p_i, and for each $a \in \Theta$ and each i we introduce a state δ_a^i with a transition $\delta_a^i \overset{a}{\mapsto} q_A^i$. We presented only the easier result in full detail above, for clarity of exposition.

5.5 Relation to Tomita parsing

The techniques sketched in this chapter yield a recognizer that may be viewed as a generalization of Tomita's general context-free parser [138]: Tomita's algorithm used bottom-up LR-parsing, but we eliminate the restriction to LR-parsers and build techniques that may be applied to the simulation of any pushdown automaton. For instance, we can build a top-down parsing automaton, apply our generalization of Tomita's methods, and in this way obtain a top-down parser for general context-free languages.

We recall here the basic idea of Tomita parsing. One naive way to simulate a pushdown automaton is to resolve non-determinism by simply cloning the automaton at each non-deterministic choice point and apply breadth-first search, running each of the cloned automata in parallel. However, this naive method can easily lead to exponential behavior, and in fact may even fail to terminate in the presence of cycles in the grammar.

Tomita's technique is based on the insight that multiple stack configurations with the same symbol at the top of the stack will all behave identically until this symbol is popped off the stack. If we identify the topmost elements of configurations when they share the same value, we can take advantage of this insight to speed up our simulation of the pushdown automaton; applying this idea recursively yields a graph-like data structure to represent the set of stack configurations currently under exploration. Consequently, to avoid exponential behavior, Tomita's algorithm compresses the internal data structure using a graph-structured stack to concisely represent the set of reachable stack configurations. Tomita's algorithm uses a complex set of rules, which are specialized to LR-parsing, to update the graph-structured stack.

In comparison, we use a regular language to concisely represent the set of stacks configurations that are explored. Note that the NDFA representation of this regular language corresponds closely to Tomita's graph-structured stack: if we simply view our NDFA as a graph with vertices and edges given by the states and transitions of the automaton, we obtain a graph-structured stack a la Tomita. In other words, every NDFA induces a graph-structured stack[1]. Nonetheless, despite this close connection, there are a few beneficial features to our formulation, as outlined below.

First, in contrast to Tomita's parser, which simulates the operation of a LR-style automaton, our construction can handle arbitrary pushdown automata. An important consequence is that Tomita's inherently bottom-up techniques can be extended to support top-down recognition of general context-free languages.

As a secondary benefit, our proof techniques provide a simple explanation of why and how Tomita's algorithm works. The key insight may be justified in one-sentence slogan form as

"The reachable configurations of a NDPDA form a regular language."

This slogan implies that the set of reachable states may be represented as a regular language, and we may manipulate this set using extremely well-understood techniques for operating on finite-state automata. In comparison, Tomita's parsing algorithm was fairly complex: in essence, it was forced to re-derive standard algorithms for manipulating finite-state automata, and moreover, this simple explanation of the operation of Tomita's parser was disguised by the fact that the algorithm was prematurely specialized to the case of simulating a LR parsing automaton. We note

[1] However, the converse is not true in general: NDFA's may contain loops, but Tomita's graph-structured stack cannot, at least as outlined in the original algorithm [138].

that previous proofs of correctness for Tomita's parser, which were fairly complex, can be substantially simplified by working at the higher level of abstraction provided by our approach. We feel that, once one recognizes that Tomita parsing is essentially a slightly obscured simulation of our Algorithm A, the intuition about how and why Tomita parsing works becomes clearer.

Note also that Tomita's original presentation of his parsing algorithm [138] had a bug: it did not work correctly in the presence of cycles in the context-free grammar [110]. This bug may be attributed to omitting one case in the ad-hoc proof of correctness for the parsing algorithm. It was later shown that this bug can be fixed [110] with a special treatment of loops in the graph-structured stack, but this example illustrates the complexity of Tomita's specialized technique and of the proofs of correctness for it. In contrast, cycles in the grammar are already handled naturally in the NDFA formulation and require no extra effort to treat correctly.

Our experience with implementing a general context-free recognizer using our formulation has been unexpectedly pleasant. We were able to build a general context-free recognizer over an extended weekend (two days for implementation, and a day for debugging, after which we have not encountered any further bugs despite considerable use). When this is compared to the usual error-prone nature of parser implementation, it seems to speak favorably for the conceptual cleanliness of the NDFA formulation.

Chapter 6

Related work

6.1 Automated vulnerability detection

Early static analysis work The seminal PA (Program Analysis) project [84] examined the vulnerability detection problem and developed informal tools for identifying suspicious source code patterns that might indicate the presence of race conditions, validation errors, or data leaks ("residuals"). Their approach is similar to ours: they focus on identifying patterns that serve as a heuristic to identify potential errors, building tools to help find those patterns in source code, and flagging potentially problematic regions for human analysis.

However, they suffered under one serious constraint that is no longer an issue today: at the time, their methods required significant human effort, essentially to simulate by hand certain algorithmic analysis procedures that would today probably be automated. The Program Analysis project, arguably, suffered only from being too far ahead of its time.

Note also that the threat model has changed substantially since the 1970's. For example, buffer overruns were almost unheard of in 1978. This suggests that it may be useful to revisit the list of vulnerabilities that they suggested would be well-suited to automated detection.

Lexical tools Many practitioners have noted that `grep` can be a useful if crude test for finding certain kinds of simple coding errors. One paper took this a step further, using a `perl` script to detect race conditions in file access; most notably, the authors used the tool to find several new vulnerabilities in `sendmail` [12]. However, the purely lexical—rather than semantic—nature of the race detection tool is a severe limitation (for instance, the `perl` script only recognizes two expressions as equivalent when they are syntactically identical, and interprocedural analysis is impossible).

Due to these issues, the authors suggested to look for a more precise alternative, such as dynamic race detection. In comparison, this thesis has provided initial evidence that an alternate solution may lie in borrowing sophisticated techniques from the program analysis community, which has developed tools for inferring semantic—not merely syntactic—properties of programs.

Program slicing Two groups have advocated program slicing as a tool for understanding and auditing security-critical code. First, Fink, Bishop, and Levitt used a combination of static and dynamic analysis to re-discover an already-known but subtle logic error in a medium-sized FTP daemon [60, 59, 61], and Lo, Levitt, and Olsson developed general techniques for assisting manual code review of security software using program slicing [100]. Also, Guha and Mukherjee have used slicing as a technique for finding vulnerabilities in network stacks [70].

It seems that program slicing improves manual code review by focusing attention the reviewer's attention on only the code that is relevant to the property under consideration. It has been suggested that one easy way to improve the capability of human source-code auditors is to increase the size of the screen (so that one may examine more code simultaneously) [43], and program slicing effectively provides the same benefits, by eliminating irrelevant code. The use of program slicing does require significant human interaction, but it appears to be a powerful technique to augment human strengths at finding subtle logic errors.

Type systems One work has recently developed techniques for static enforcement of Java's stack-based privilege management policy using a novel type system [133]. Others have shown how to check control-flow properties of Java programs using finite-state verification [88]. Volpano *et al.* have developed theoretical techniques for checking information flow policies using type systems [144, 141, 142, 143]. These projects all focus primarily for proving the absence of bugs (e.g., showing that security checks won't go wrong at runtime), but presumably they might also be useful for finding bugs.

Other static techniques In recent work, Gavrila has developed a tool for finding information flows in ANSI C programs [65]. Given a source variable and a sink variable, this tool is able to show all paths across which values may flow from the source to sink, and Gavrila shows how to use this to help audit code for various information flow and correctness properties.

Others have suggested using model checking to detect vulnerabilities that arise due to unexpected interactions between components of a system [119, 120] (see also [124]). In this work the models were derived by hand, although the authors suggest that it might be possible to build models from source code using program analysis.

Runtime testing Miller *et al.* have applied a black-box testing tool, `fuzz`, to find bugs in system software by checking whether the software crashes on random inputs [105, 106]. Their work showed that a surprisingly large fraction of stock Unix utilities could be crashed with random inputs. Some of the bugs they discovered are in fact due to exploitable buffer overruns, which demonstrates that black-box testing can detect at least some security vulnerabilities.

More recently, Ghost *et al.* have explored applying more sophisticated techniques from software engineering to find buffer overruns. For example, software fault injection has been used to rediscover serious vulnerabilities in several widely deployed network daemons [66, 67]; one disadvantage is that, like our buffer overrun detection tool, their technique produces many false alarms that must be checked by hand.

As discussed in Section 2.2, another disadvantage is that runtime testing seems likely to miss many vulnerabilities. The problem is that the code paths of greatest interest to a security auditor—the ones that are never followed in ordinary operation—are also the ones that are the hardest to analyze with dynamic techniques. We believe that static analysis can serve to complement dynamic techniques by detecting some vulnerabilities that dynamic techniques miss.

6.2 Program verification

There has been a great deal of work on using theorem proving and program verification, and proving the correctness of security-sensitive applications has been one of the notable motivations for this line of work [58, 28, 104].

One recent system of interest is ESC, an automated program checker for Modula-3 and Java that catches many programming errors at compile-time, using partial program verification [45]. ESC was not developed for checking security properties, but it certainly could be used for this purpose. One disadvantage of ESC is that it requires coders to annotate module interfaces with information about expected pre- and post-conditions, but it can use this information to find a very large class of potential bugs. Implementations are available for Modula-3 and Java, but not for C (but see the Eau Claire project, an effort to adapt the ESC technology to the purpose of finding security holes in C programs [29]). In general, we can expect full program verification for real-world programming languages to be more costly than ESC.

The major barrier to widespread adoption of full program verification is that it comes at a high cost in development and training time, and one typically cannot obtain partial benefits from a partial investment. Our work tries to achieve some of the benefits of program verification at a much lower cost, by targeting specific program properties of interest. This restriction allows us to develop analysis tools that require very little effort from the developer and can be applied at any point in the software lifecycle.

To improve the likelihood of being successful in practice, we believe that analysis systems should give returns proportional to the effort invested by the developer rather than espousing an all-or-nothing philosophy. Thus, we expect that our approach will complement formal verification nicely.

6.3 Static detection of non-security bugs

Lint-like tools Several commonly used tools [89, 56, 55] use static analysis and some heuristics to detect common programming errors (such as type errors, abstraction violations, and memory management bugs), but these tools don't detect buffer overruns or other semantic, whole-program properties.

Practitioners have sometimes used `grep` to find buffer overruns by searching for all invocations of unsafe string primitives; however, a substantial time investment is often required to deal with the very large number of false alarms. Our results demonstrate an 15× improvement over `grep` for the case of sendmail 8.9.3 (see Section 2.6). In general, lexical techniques are often poorly suited to capturing properties at a higher level of abstraction.

Checking control-flow properties Many program correctness properties can be expressed as constraints on the sequencing of various primitive operations. Olender and Osterweil developed a tool, Cesar, that checks Fortran programs against properties specified as a generalized regular expression over the alphabet of primitive operations [112]. More recently, Engler *et al.* have developed tools for checking C programs against sequencing properties specified as a finite automaton [52, 31, 51]. Engler *et al.* have shown how this basic approach can be used to check many important properties and have found an astonishing number of real bugs in this way.

Both tools may be viewed as model checkers for a simple temporal logic: Engler's specifications have this flavor because NDFA's are of course a simple way of specifying a subset of temporal properties (namely, the safety properties), and Cesar's specifications may be viewed more directly as predicates in a temporal logic with both universal and existential quantifiers. One may also find recent work that directly focuses on model checking of programs as an explicit goal in its own right; see, e.g., the Bebop system [9, 8].

Other approaches The PREfix tool is able to find many common bugs in C and C++ code by randomly selecting many straight-line paths through the source code, symbolically executing the program on these paths, and checking for errors [26]. Aspect is a tool that allows programmers to specify intended dependences between program variables and then statically checks the source code against the specification [87]; Jackson shows that this detects an interesting class of bugs that complements traditional type checking techniques.

6.4 Guarding against buffer overruns

Stackguard Stackguard is a runtime tool that detects buffer overruns on the stack before they cause harm [39]. Stackguard imposes very little performance overhead and has been applied to large suites of applications, including an entire Linux distribution. Stackguard is a powerful tool that can serve as a strong deterrent against many existing buffer overrun attacks; however, it does not stop all overrun attacks, and thus should not be relied upon as the only line of defense.

Array bounds checking One way to avoid buffer overruns is to use runtime array bounds checks. There are several implementations of array bounds checking for C, including SCC [6], gcc extensions [90], Purify [118], and BoundsChecker [111]. However, many of these tools impose a large performance overhead (instrumented programs are typically 2–3× slower than the original versions [6, 90, 32, 71]). As a result, the tools are usually used only for debugging, not for production systems.

To reduce the high cost of runtime bounds checking, several researchers have studied optimization techniques for eliminating redundant checks [71, 102, 134]. However, they typically focus on moving bounds checks to less frequently executed locations, rather than on eliminating all bounds checks. For example, hoisting bounds checks out of loops using loop invariants greatly reduces the performance impact of the bounds checks but cannot reduce the number of checks in the program's source code. Therefore, these optimization techniques are not well suited for finding buffer overruns proactively.

Other works have concentrated on eliminating all bounds checks for some type-safe languages. For example, Necula and Lee develop a certifying compiler for a type-safe subset of C that eliminates most bounds checks using Shostak's loop residues [109]. Also, Xi and Pfenning propose a method to eliminate runtime array bounds checking for ML with the help of some assertions added by the programmer to capture certain program invariants [147, 148]. Of course, none of these tools can eliminate buffer overruns in large legacy applications written in C.

6.5 The range analysis problem

Range analysis Our approach to range analysis builds on much prior work in the literature, including early work on abstract interpretation [36] and symbolic range propagation [76], as well as more mature work on systems for static debugging [21, 22], generalized constant propagation [139], branch prediction [113], parallelization [16, 15], and other applications [125]; however, our emphasis on analysis of large C programs spurred us to develop new techniques with better scaling behavior.

Parallelizing compilers One important application for array reference analysis is in discovering implicit parallelism in sequential Fortran programs [103, 10, 117, 98]; however, those techniques do not seem to help with general range analysis, or with the buffer overrun problem, because they focus too narrowly on the special case of loop optimization.

Linear programming Many papers have suggested using linear programming to discover program invariants, including the simplex method, Fourier-Motzkin variable elimination [129], the Omega method [117], the SUP-INF method [14, 131], Shostak's loop residues [132], and algorithms for special classes of linear systems [80, 34, 101]. Typically, one combines linear programming with *abstract interpretation* over some simple domain (convex polyhedra, octagons, etc.) [36, 38, 72, 74, 75, 73, 121]. In this context, linear programming algorithms provide a tool for manipulating subsets of $\mathbb{Z}^k$, with operations such as $\cup$, $\cap$, projection, widening, and testing for feasibility. See especially [38] for an early example of a tool that infers linear invariants of small programs using abstract interpretation and the simplex method. Although the simplex-based techniques offer more precision than our range analysis, it is not clear how well they scale (but see [137] for an implementation where it is recommended to avoid programs with more than 20 variables). There has also been recent work on improving efficiency using constraint solving techniques [17]; we speculate that these techniques might be useful for reducing false alarms in buffer overrun detection.

Constraint-based analyses Philosophically, our algorithm for range analysis may be viewed as a constraint-based analysis [3]; however, it is unusual to incorporate arithmetic expressions in the set constraint language and solver (but see [77] for an important partial exception).

Note also that techniques for solving integer constraint systems may be found in the artificial intelligence [42, 83, 99, 79] or user interfaces [18] literature; however, those algorithms typically stress generality for small problems ("hundreds of nodes and constraints" [42]) over scalability and thus are not directly applicable here.

6.6 Intrusion detection

There has been a great deal of previous work on intrusion detection [5, 44]. However, previous work has relied on policies that have been obtained either by manual specification of known attack signatures or else by machine learning applied to normal system executions. This dissertation is the first work we are aware of that builds specifications through automated static analysis of source code.

We will not attempt to provide an exhaustive survey of work in intrusion detection, but one class of works is notable for its relation to our proposal. Ko *et al.* proposed a specification language, based on a generalization of context-free grammars, for expressing hand-written intrusion detection policies, and their work provided "hyper-parsing" algorithms for checking that system executions conform to the policy [95, 93, 96]; however, they apparently did not anticipate automated static generation of policies.

Chapter 7

Future work

Our work on static analysis in intrusion detection opens up many new avenues for future research in intrusion detection. We could apply the vast literature on static program analysis to make our automata more precise. Adding dataflow analysis would help both by giving us more argument information to system calls (recall that we presently only capture constant arguments), and also by improving our control flow analysis. More sophisticated points-to analysis would improve our results on programs that make non-trivial use of function pointers. As we have seen before, shrinking the automata could vastly improve performance.

Our prototype implementation has been driven by ease of prototyping rather than performance. We pay for this in two areas: the entire runtime monitoring software (including automaton interpretation) is written in Java, and it runs in user mode. We believe a highly optimized C or assembly language automaton would be much faster. In addition, for each system call in the monitored application, we are performing a context switch to the runtime monitoring software and a second user/kernel barrier crossing. A production implementation of our approach could live entirely in the operating system kernel. Furthermore, for the callgraph model, it would be interesting to explore determinizing (via the subset construction) the automaton, and then applying standard DFA minimization techniques.

Now that we have an automatically derived specification of a program, we could consider other uses to make of it. We could change our view from the specification being a "golden master" that defines correct behavior to a specification under test, as in traditional verification efforts. By defining a list of "best practices" for secure programs, we could then use a model checker to see whether the program we're considering obeys the accumulated wisdom of the community [13, 64]. For example, we might check whether a program that uses elevated privileges opens a finite number of filesbefore dropping its privilege level back to normal, thus following the principle of least privilege. We note that our callgraph model is a finite automaton that appears nearly ideal for a model checker. Our stack model will be more challenging to model check, but there has been theoretical work in this area [25, 62, 136, 146, 53, 20, 19, 54]. This approach will lead to a completely automatic way of ensuring that security-critical programs follow good programming practices.

Note that we have no specification–implementation gap, as our specification is mechanically extracted from the implementation. But this raises the question as to the correctness of the specification. Model checking gives us the ability to do some sanity checks on the specification automatically. While this is not a complete solution, there is no good general answer to ensuring that a formal specification satisfies an informal requirements statement.

Also, our work on static detection of buffer overrun vulnerabilities suggests many interesting directions for future work. The most obvious direction lies in improving the precision of our analysis. Reducing the number of false alarms would make our tool more palatable to use routinely, and in Section 2.6 we have suggested several improvements to the analysis algorithm that look promising in this respect.

This success with buffer overrun detection suggests that static analysis may be a useful technique for detecting other classes of vulnerabilities as well. We think it would be interesting to build tools to find race conditions, unchecked parameters, data leaks, format string bugs, improperly nested privileged sections, tractorbeaming attacks, memory management errors, and many other well-known and common classes of vulnerabilities. Also, we suggest that, in addition to enforcing the absence of known-bad behaviors, it may be useful to build tools to ensure the presence of known-good code patterns, e.g., idioms for ensuring compliance with the principle of least privilege. A combination of static analysis and model checking may be a useful foundation for both goals.

The example applications found in this dissertation should help evaluate static analysis as a tool for advancing software assurance, and we feel that their successes provide evidence that this methodology holds great promise for the future.

Chapter 8

Conclusion

This dissertation demonstrates that program analysis can be a powerful and versatile tool for improving the assurance level of security software. These results suggest that there is likely to be a promising future for static analysis in the security community.

Using buffer overrun bugs as a test case, we showed that, with the application of domain-specific security knowledge, static analysis can be used to build a lightweight, principled, effective tool for detecting vulnerabilities in large, real-world codebases. One notable benefit of static analysis is that it allows us to eliminate or neutralize security bugs proactively before they are exploited. Consequently, we suspect that this methodology may in the future prove to be an attractive tool for improving software assurance by detecting many important classes of coding errors.

Next, we showed that static analysis can be used to build a radically different type of intrusion detection system, with a number of unusual properties: automated derivation of very sophisticated program models; the ability to detect novel attacks not foreseen by the designer; and an utter lack of false alarms. This contribution demonstrates that static analysis may be profitably combined with more conventional runtime techniques.

We also developed some important algorithmic scaffolding to support these applications. We first showed that one may efficiently and precisely discover the range of values that each integer-valued program variable can take on, and we showed how our techniques for solving this problem may be used to detect buffer overrun vulnerabilities. Next, we showed that efficient top-down parsing of general context-free languages is possible. Our parser exploits a concise representation for the states of a pushdown automaton to achieve a cubic worst-case running time, and these ideas were used to build an especially precise intrusion detection system. We expect that some of these theoretical techniques may find applications elsewhere.

Bibliography

[1] Foto Afrati and Christos Papadimitriou. The parallel complexity of simple logic programs. *Journal of the ACM*, 40(4):891–916, September 1993.

[2] A. Aiken, M. Fähndrich, J.S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *TIC'98: Types in Compilation*, pages 78–96. Springer-Verlag, 1998.

[3] Alexander Aiken. Set constraints: results, applications, and future directions. In *PPCP'94: Principles and Practice of Constraint Programming*, pages 326–335. Springer-Verlag, 1994.

[4] Glen Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI'97*. ACM Press, 1997.

[5] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[6] T.M. Austin, S.E. Breach, and G.S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI'94*. ACM Press, 1994.

[7] Stefan Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, 1999.

[8] Thomas Ball and Sriram Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN '00: Model Checking of Software*, 2000.

[9] Thomas Ball and Sriram Rajamani. Checking temporal properties of software with boolean programs. In *Proceedings of the Workshop on Advances in Verification*, July 2000.

[10] U. Banerjee. *Dependence analysis for supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1998.

[11] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Operating system enhancements to prevent the misue of system calls. In *Proc. of the 7th ACM Conference on Computer and Communications Security*, pages 174–183, Athens, Greece, November 2000.

[12] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[13] Matt Bishop. How to write a setuid program. *;login: (The USENIX Association Newsletter)*, 12(1):5–11, Jan./Feb. 1987.

[14] W.W. Bledsoe. The SUP-INF method in Presburger arithmetic. Technical Report Memo ATP-18, Math Dept., U. Texas Austin, December 1974.

[15] William Blume and Rudolf Eigenmann. Demand-driven, symbolic range propagation. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[16] William Blume and Rudolf Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium.* IEEE, April 1995.

[17] Ratislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *PLDI 2000*, 2000.

[18] A. Borning, R. Anderson, and B. Freeman-Benson. Indigo: a local propagation algorithm for inequality constraints. In *Proc. 1996 ACM Symp. on User Interface Software and Technology*, pages 87–96, 1996.

[19] Ahmed Bouajjani, Javier Esparza, Alain Finkel, Oded Maler, Peter Rossmanith, Bernard Willems, and Pierre Wolper. An efficient automata approach to some problems on context-free grammars.

[20] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *12th Computer Aided Verification*. Springer-Verlag, 2000.

[21] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *Proc. PLIP '90*, LNCS 456, pages 302–323. Springer, 1990.

[22] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI'93*. ACM Press, 1993.

[23] J. R. Büchi. Regular canonical systems. *Arch. Math. Logik Grundlag*, 6:91–111, 1964.

[24] The `bugtraq` mailing list, `http://www.securityfocus.com/`.

[25] Olaf Burkart. *Automatic verification of sequential infinite-state processes*, volume 1354 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[26] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30:775–802, 2000.

[27] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106:61–86, 1992.

[28] Maureen H. Cheheyl, Morrie Gasser, George A. Huff, and Jonathan K. Millen. Verifying security. *Computing Surveys*, 13(3):279–339, 1981.

[29] Brian Chess. Eau Claire. See `http://sctest.cse.ucsc.edu/chess/EauClaire/`.

[30] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[31] Andy Chou, Benjamin Chelf, Dawson Engler, and Mark Heinrich. Using meta-level compilation to check FLASH protocol code. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 59–70, November 2000.

[32] F. Chow. A portable machine-independent global optimizer—design and measurements. Technical Report Tech. report 83-254, Computer Systems Lab, Stanford Univ., 1983. Ph.D. thesis.

[33] C. Click and K.D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Programming Lang. and Systems*, 17(2):181–196, March 1995.

[34] E. Cohen and N. Megiddo. Improved algorithms for linear inequalities with two variables per inequality. *SIAM J. Computing*, 23(6):1313–1347, December 1994.

[35] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*. ACM, 2000.

[36] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Intl. Symp. on Programming*, April 1976.

[37] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*. ACM, 2000.

[38] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM POPL*, pages 84–97, 1978.

[39] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conf.*, January 1998.

[40] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proc. 2000 DARPA Information Survivability Conf. and Exp. (DISCEX '00)*, pages 154–163. IEEE Comp. Soc., 1999.

[41] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. An efficient method of computing static single assignment form. In *POPL'89*, 1989.

[42] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, July 1987.

[43] Theo de Raadt. Personal communication, November, 2000.

[44] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2), February 1987.

[45] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. Technical Report 159, Compaq SRC, 1998.

[46] A. Diwan, K. McKinley, and E. Moss. Type-based alias analysis. In *PLDI'98*, 1998.

[47] W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formula. *Journal of Logic Programming*, 3:267–284, 1984.

[48] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13:94–102, 1970.

[49] M.W. Eichin and J.A. Rochlis. With microscope and tweezers: an analysis of the Internet virus of Nov. 1988. In *Proc. IEEE Symp. Security and Privacy*, 1989.

[50] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Lanugage Design and Implementation*, pages 242–256. ACM SIGPLAN, 1994.

[51] Dawson Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, May/June 1999.

[52] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[53] Javier Esparza and Jens Knop. An automata-theoretic approach to interprocedural data-flow analysis. In *Proceedings of FoSSaCS'99*, LNCS 1578, pages 14–30. Springer, 1999.

[54] Javier Esparza and Andreas Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *ACM International Conference on Principles of Programming Languages (POPL 2000)*, 2000.

[55] D. Evans. Static detection of dynamic memory errors. In *PLDI'96*, 1996.

[56] D. Evans, J. Guttag, J. Horning, and Y.M. Tan. LCLint: a tool for using specifications to check code. In *SIGSOFT Symp. on Foundations of Software Engineering*, December 1994.

[57] D. Farmer. Abstract: Shall we dust Moscow? (Security survey of key internet hosts & various semi-relevant reflections), December 1996. Published online at `http://www.fish.com/survey/`.

[58] R. Feiertag, K. Levitt, and L. Robinson. Proving multi-level security of system design. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 57–65. IEEE, 1977.

[59] G. Fink. Discovering security and safety flaws using property-based testing, 1995. Ph.D. thesis.

[60] G. Fink and M. Bishop. Property based testing: A new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4), July 1997.

[61] G. Fink and K. Levitt. Property based testing of privileged programs. In *10th Annual Computer Security App. Conf.*, pages 154–163. IEEE Comp. Soc. Press, December 1994.

[62] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997. Proceedings of Infinity'97.

[63] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for Unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996.

[64] S. Garfinkel and G. Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, 1996. Second edition.

[65] Serban Gavrila. Penetration analysis tools: Final technical report, August 1999.

[66] A.K. Ghosh and T. O'Connor. Analyzing programs for vulnerability to buffer overrun attacks. In *Proceedings of the National Information Systems Security Conference*, October 1998.

[67] A.K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proc. IEEE Symp. on Security and Privacy*, pages 104–114, May 1998.

[68] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Sixth USENIX Security Symposium Proceedings*, pages 1–12, San Jose, CA, July 1996.

[69] S.L. Graham, M.A. Harrison, and W.L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

[70] Biswaroop Guha and Biswanath Mukherjee. Network security via reverse engineering of TCP code: Vulnerability analysis and proposed solutions. In *Proceedings of the IEEE Infocom '96 Conference*, 1996.

[71] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, Mar.–Dec. 1993.

[72] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *SAS'94: Static Analysis Symp.*, pages 223–237. Springer-Verlag, 1994.

[73] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997. From *CAV'93: Computer Aided Verification.*

[74] M. Handjieva. Abstract interpretation of constraint logic programs using convex polyhedra. Technical Report LIX/RR/96/06, LIX, Ecole Polytechnique, May 1996.

[75] M. Handjieva. STAN: A static analyzer for CLP(R) based on abstract interpretation. In *SAS'96: Static Analysis Symp.*, 1996.

[76] W.H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. Software Engineering*, SE-3(3):243–250, May 1977.

[77] N. Heintze. Set based analysis and arithmetic. Technical Report CS-93-221, Carnegie Mellon Univ., 1993.

[78] G. Helmer. Incomplete list of Unix vulnerabilities. `http://www.cs.iastate.edu/~ghelmer/unixsecurity/unix_vuln.html`.

[79] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, pages 113–159, 1992.

[80] D.S. Hochbaum and J.S. Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM J. Computing*, 23(6):1179–1192, December 1994.

[81] D. Holland.
`http://www.hcs.harvard.edu/~dholland/computers/netkit.html`.

[82] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[83] E. Hyvönen. Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artificial Intelligence*, pages 71–112, 1992.

[84] R. Bisbey II and D. Hollingsworth. Protection analysis project final report. Technical Report ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute, May 1978.

[85] `http://www.infilsec.com/vulnerabilities/`.

[86] Computer Security Institute. Issues and trends: 2000 CSI/FBI computer crime and security survey, March 2000.

[87] Daniel Jackson. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[88] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. Technical Report 1210, IRISA, October 1998.

[89] S.C. Johnson. Lint, a C program checker. Technical report, Bell Laboratories, 1978. Computer Science Technical Report 65.

[90] R. Jones and P. Kelly. Bounds checking for C.
`http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html`.

[91] O. Kirch. The poisoned NUL byte, October 1998. Post to the `bugtraq` mailing list.

[92] klog. The frame pointer overwrite. *Phrack Magazine*, 9(55), September 1999.

[93] Calvin Ko. *Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach.* PhD thesis, U.C. Davis, September 1996.

[94] Calvin Ko. Logic induction of valid behavior specifications for intrusion detection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 142–153, Oakland, CA, May 2000. IEEE.

[95] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the Tenth Computer Security Applications Conference*, pages 134–144, Orlando, FL, December 1994. IEEE Computer Society Press.

[96] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, CA, May 1997. IEEE.

[97] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[98] C. Lengauer. Loop parallelization in the polytope model. In *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[99] O. Lhomme. Consistency techniques for numeric CSPs. *IJCN'93: 13th Intl. Joint Conf. on Artificial Intelligence*, 1, 1993.

[100] Raymond Lo, Karl Levitt, and Ronald Olsson. MCF: A malicious code filter. *Computers & Security*, 14(6):541–566, 1995.

[101] G. Lueker, N. Megiddo, and V. Ramachandran. Linear programming with two variables per inequality in poly-log time. *SIAM J. Computing*, 19(6):1000–1010, December 1990.

[102] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. *SIGPLAN Notices*, 17(6):114–119, 1982. From *1982 Symp. on Compiler Construction*.

[103] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Efficient and exact data dependence analysis. In *PLDI'91*, 1991.

[104] John McHugh. Towards the generation of efficient code from verified programs. Technical Report 40, University of Texas, 1984.

[105] B.P. Miller, L. Fredricksen, and B. So. An empirical study of the reliability of Unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[106] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murphy, A. Natarajan, and J. Steidl. Fuzz revisited: a re-examination of the reliability of Unix utilities and services. Technical Report Tech. report CS-TR-95-1268, U. Wisconsin, April 1995.

[107] T.C. Miller and T. de Raadt. strlcpy and strlcat—consistent, safe, string copy and concatenation. In *FREENIX'99*. USENIX Assoc., Berkeley, CA, 1999.

[108] Mudge. Sendmail 8.7.5 vulnerability, September 1996. Post to the `bugtraq` mailing list.

[109] G.C. Necula and P. Lee. The design and implementation of a certifying compiler. In *PLDI'98*, 1998.

[110] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammers. In *Generalized LR Parsing*, pages 61–76. Kluwer Acad. Pub., 1991.

[111] NuMega BoundsChecker.
`http://www.numega.com/products/aed/vc_more.shtml`.

[112] Kurt Olender and Leon Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, January 1992.

[113] J. Patterson. Accurate static branch prediction by value range propagation. In *PLDI'95*, pages 67–78, 1995.

[114] A. Pioli and M. Hind. Combining interprocedural pointer analysis and conditional constant propagation. Technical Report RC 21532, IBM Research, March 1999.

[115] V.R. Pratt. Two easy theories whose combination is hard, September 1977. Unpublished manuscript.

[116] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, January 1998.

[117] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *PLDI'92*, pages 140–151, 1992.

[118] Pure Atria Purify.
`http://www.rational.com/products/purify_unix/index.jtmpl`.

[119] C. Ramakrishnan and R. Sekar. Model-based vulnerability analysis of computer systems. In *2nd Int. Workshop on Verication, Model Checking and Abstract Interpretation*, September 1998.

[120] C. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities, 2000. Unpublished manuscript.

[121] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Proc. 19th IEEE Real-Time Systems Symp.*, pages 200–209, 1998.

[122] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocecural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*. ACM, 1995.

[123] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, Nov.–Dec. 1998.

[124] R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings 2000 IEEE Computer Society Symposium on Security and Privacy*, May 2000.

[125] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI 2000*. ACM Press, 2000.

[126] David A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*. ACM, 1998.

[127] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *11th International Workshop on Distributed Algorithms*, September 1997.

[128] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2(4), March 2000.

[129] A. Schrijver. *Theory of linear and integer programming*. Series in Discrete Mathematics. John Wiley & Sons, 1986.

[130] M. Shapiro and S. Horwitz. The effects of precision of pointer analysis. In *SAS'97: Static Analysis Symp.*, pages 16–34. Springer-Verlag, 1997.

[131] R. Shostak. On the SUP-INF method for proving Presburger formulas. *J. ACM*, 24(4):529–543, October 1977.

[132] R. Shostak. Deciding linear inequalities by computing loop residues. *J. ACM*, 28(4):769–779, October 1981.

[133] Christian Skalka and Scott Smith. Static enforcement of security with types. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 34–45, September 2000.

[134] N. Sosuki and K. Ishihata. Implementation of array bound checker. In *POPL'77*, pages 132–143, 1977.

[135] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 32–41, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

[136] B. Steffen and O. Burkart. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science (TCS)*, 1999. Special Issue for ICALP'97, to appear August 1999.

[137] STeP help: Polyhedral invariant generation. `http://www.cs.yorku.ca/~stateclock/STeP/WWW/polyhedral.html`.

[138] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1–2):31–46, 1987.

[139] C. Verbrugge, P. Co, and L.J. Hendren. Generalized constant propagation: A study in C. In *Compiler Construction, 6th Intl. Conf.*, LNCS 1060, pages 74–90. Springer-Verlag, April 1996.

[140] Werner Vogels, Robbert van Renesse, and Ken Birman. Six misconceptions about reliable distributed computing. In *Proc. 8th Intl. Symp. High Performance Distributed Computing*, pages 213–215. IEEE Comp. Soc., 1999.

[141] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97, Colloqium on Formal Approaches in Software Engineering*, 1997.

[142] D. Volpano and G. Smith. Language issues in mobile program security. In *Mobile Agents and Security*, LNCS 1419, pages 24–43. Springer-Verlag, 1998.

[143] D. Volpano and G. Smith. Secure information flow in a multi-threaded imperative language. In *POPL'98*, pages 355–364. ACM Press, 1998.

[144] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[145] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings 2000 Network and Distributed System Security Symposium.* Internet Society, 2000.

[146] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Computer Aided Verification '98*, pages 88–97. Springer, 1998.

[147] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI'98*, pages 249–257, 1998.

[148] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL'99*, 1999.

[149] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.