Analysis and Defense of Vulnerabilities in Binary Code

David Brumley

CMU-CS-08-159

September 29, 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Dawn Song, Chair (U.C. Berkeley and Carnegie Mellon University)
Randal E. Bryant (Carnegie Mellon University)
Peter Lee (Carnegie Mellon University)
Monica Lam (Stanford University)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

The research is supported in part by CyLab at Carnegie Mellon University under grant DAAD19-02-1-0389 from the U.S. Army Research Office, the U.S. Army Research Office under the Cyber-TA Research grant No. W911NF-06-1-0316, the National Science Foundation under Grants No. 0311808, 0433540, 0448452, and 0627511, and by the Graduate Student Fellowship from Symantec, Inc.

The views and conclusions contained here are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARO, CMU, Symantec, the U.S. Government, or the National Science Foundation.

Keywords: Vine, binary analysis, patch-based exploit generation, vulnerability filter generation

Abstract

In this thesis, we develop techniques for vulnerability analysis and defense that only require access to vulnerable programs in binary form. Our approach does not use or require source code. We focus on a binary-centric approach since everyone typically has access to the binary code for the programs they run. Thus, our approach is applicable to a wider audience than previous approaches that require or utilize source code. In addition, the binary itself is often the most faithful encoding of security-relevant details since it is what is actually executed on hardware.

In order to demonstrate the benefits of binary-centric vulnerability analysis and defense, we first develop binary analysis techniques. We have implemented our techniques as part of a binary analysis architecture called *Vine*. We then demonstrate the utility of our approach, and Vine, in two typical applications of vulnerability analysis and defense.

First, we develop binary analysis techniques for reverse engineering a patched vulnerability. More specifically, our techniques enable an attacker to reverse engineer exploits from software patches that fix program bugs and vulnerabilities. We call this *automatic patch-based exploit generation*. We demonstrate automatic patch-based exploit generation on real vulnerabilities using Vine. In our experiments, it only takes a few minutes to generate an exploit from the patched program. We argue one consequence of our results is that current delayed patch distribution architectures (e.g., Windows Automatic Update) may hurt security.

Second, we propose methods and techniques for generating input filters based upon vulnerability analysis. An *input filter* is a recognizer for inputs that exploit a vulnerability. We develop the first automatic techniques for generating input filters with accuracy guarantees even when there may be restrictions on the input filtering language. We demonstrate our techniques by automatically generating input filters from vulnerable binary programs.

Acknowledgments

I thank my wife, family, friends, advisor, coauthors, and colleagues. Each of you have contributed immensely to making this thesis possible. Whether it be a nudge at the right time, or long-term support, each contribution was important.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In this thesis, we develop techniques for the analysis and defense of vulnerabilities when the program is only available in binary form. Vulnerability analysis and defense is not a solved problem. Vulnerable programs are currently one of the leading causes of computer security incidents. Attackers routinely exploit vulnerabilities to compromise computers, carry out financial fraud, menace users, and threaten critical infrastructure.

We develop techniques for a class of vulnerabilities we call *input validation vulnerabilities*. An input validation vulnerability occurs when a program does not sufficiently check user inputs. Figure 1.1 depicts this intuition where the input domain for the program includes inputs that are unsafe (we provide a formal definition in Chapter 4). Specific inputs that trigger a vulnerability, i.e., inputs not in the safe subset, are called *exploits*.

Many typical software vulnerabilities and bugs fall within the scope of input validation vulnerabilities. Input validation vulnerabilities include most vulnerabilities that are commonly exploited by attackers, such as buffer overflows, format string vulnerabilities, integer overflows, and others. Note that input validation vulnerabilities are not limited to

Figure 1.1: An input validation vulnerability occurs when the domain of inputs for a program (white) is a super-set of inputs that are safe (black).

unsafe programming language features. Input validation vulnerabilities can arise in any language when a program mistakenly accepts or processes unintended inputs. For example, even a formally verified program may contain an input validation vulnerability if the initial specification was incorrect.

We focus on vulnerability analysis and defense techniques that only require access to the vulnerable program binary (i.e., executable). Our focus on binary code is motivated by several observations. First, everyone typically has access to the programs they use and run in binary form. The ubiquity of binary code means binary analysis techniques can potentially impact a large number of users.

Second, vulnerability analysis that only requires access to the program binary allows users to respond to modern threats. Current threats such as the Slammer worm have compromised almost all vulnerable hosts within minutes [71]. Future threats may reduce the time to seconds [98]. When a new threat arises, users must be able to rapidly prepare a defense with the information on hand. Unfortunately, since most users do not have easy access to source code, techniques that rely on source code are likely to be too slow. Since binary code is readily available, vulnerability analysis and defensive techniques are likely to be more effective against modern, rapidly developing threats.

Third, binary code analysis allows us to argue about the code that actually runs. Programs in higher-level programming languages are abstractions of what actually executes. Typical abstractions are intended to remove or hide the low-level details such as how variables are allocated, what happens on arithmetic overflow, etc. In security, these same low-level details often matter a great deal. For example, a common difference between a serious vulnerability which allows an attacker to hijack a program and a less serious one that only crashes the program is how variables are allocated on the stack. Binary analysis captures low-level details, thus allowing us to make stronger guarantees about the code that will actually execute, not just the code that was compiled.

1.2 The Running Example

Throughout this thesis, we use the running example shown in Figure 1.2a to help explain concepts and ideas. The example is motivated by a real-life vulnerability in Internet Explorer (IE) 6 from Microsoft [92]. There are three things to keep in mind in this example. First, all integers are assumed to be unsigned 32-bit numbers. Second, all arithmetic is assumed to be performed modulo 2^{32}. Third, even though the running example is shown in something that looks like source code, our techniques analyze binaries. We use a higher-level syntax merely for expository purposes.

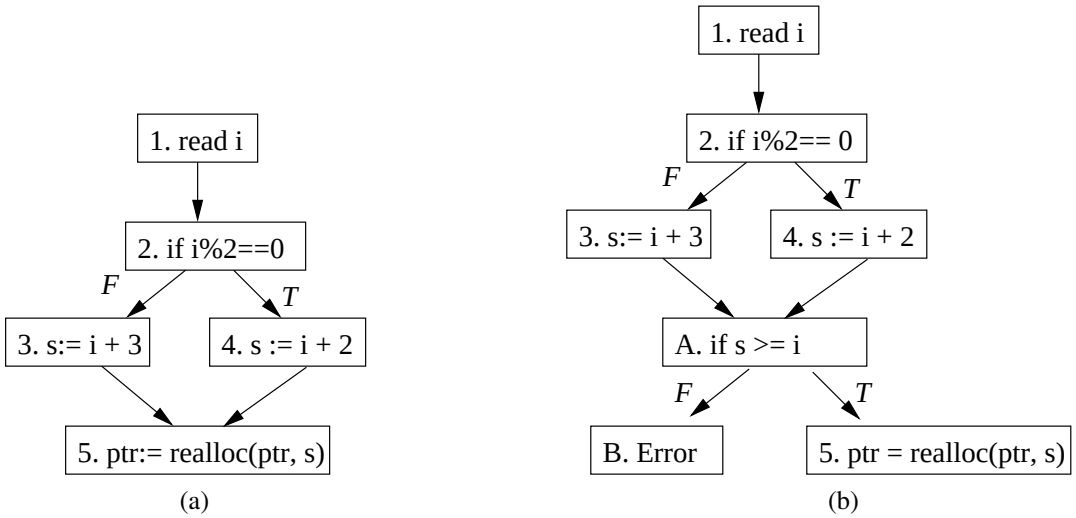


Figure 1.2: Our running example of a vulnerable program (left), and a corrected version (right).

The program first reads in user input `i` on line 1. The input domain to our program is 32-bit integers, therefore $2^{32} - 2$ is a valid input. Next, the input is checked to see if `i` is even on line 2, and if so, line 4 is executed, else line 3. $2^{32} - 2$ is even, so we execute line 4. On line 4, we add 2 to the input, and assign the sum to a local variable `s`. This is where it is important to remember all arithmetic is performed modulo 2^{32}. When `i` is $2^{32} - 2$, then $2^{32} - 2 + 2 \bmod 2^{32} = 0$, so $s = 0$ after line 4. If `i` happened to be odd, e.g., $2^{32} - 1$, something similar could happen along the false branch on line 3.

On line 5, we call `realloc`, which is a C manual memory management routine. `realloc` changes the size of the allocation pointed to by `ptr` to be `s` bytes long. When `i` is $2^{32} - 2$, then the resulting pointer will point to *zero* bytes of allocated memory. Any subsequent dereference of the pointer will in the best case cause the program to crash, or in the worst case, allow an attacker to hijack control of the program. In the real vulnerability in IE 6 [92], an attacker can both crash and hijack control of the program via the same vulnerability.

The programmer intended the sum `s` to be at least as large as the input `i`, written mathematically as $s \geq i$. The program is vulnerable to an integer overflow vulnerability when $\neg(s \geq i)$ at line 5. In this example, any input where $\neg(s \geq i)$ at line 5 is considered an exploit.

Figure 1.2b shows a patched version of the running example which fixes the vulner-

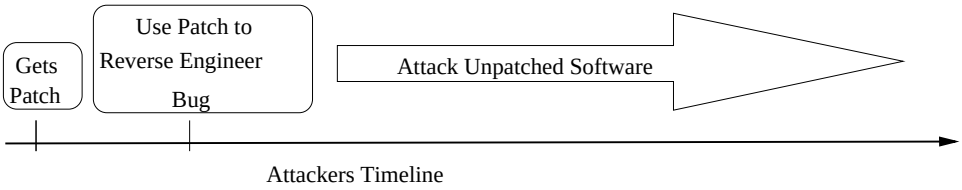


Figure 1.3: The delayed patch attack.

ability. Two new statements are added at lines A and B. Line A introduces a check for overflow. If overflow does not occur, then `realloc` is called. If overflow does occur, the `error` statement on line b is executed. In the real-life vulnerability, the `error` statement corresponds to a new code path introduced in the patched version of IE 6. Thus, the patched version rejects all inputs such that $\neg(s \geq i)$, i.e., when $i \geq 2^{32} - 3$.

1.3 Main Results

In this thesis, our main contributions center around developing practical techniques for vulnerability analysis and defense in binary programs. We demonstrate our techniques on two representative problems: automatic patch-based exploit generation, and automatic input filter generation.

1.3.1 Automatic Patch-Based Exploit Generation (APEG)

Problem Overview. At first glance, releasing a patch that addresses a vulnerability can only benefit security. After all, the patch means there is one less vulnerability overall, thus there must be an increase in security. We show this intuition is naïve. Typical patches reveal some information about the vulnerability and therefore having early access to a patch may confer significant security advantage to an attacker.

We introduce a new type of attack, called the *delayed patch attack*. In the delayed patch attack, an attacker uses a patch P' and an original vulnerable program P to reverse engineer the vulnerability fixed in P. Figure 1.3 depicts the delayed patch attack. The delayed patch attack is important when not everyone will receive a patch at exactly the same time. Current patch distribution practices stagger patch distribution, usually over hours, days, or longer. For example, Gkantsidis *et al*. show that for Windows Update it takes about 24 hours for 80% of the unique observed IPs to notice when a new patch is

available for download [48].

We show that *automatic patch-based exploit generation* (APEG), a type of delayed patch attack, is practical. In the automatic patch-based exploit generation problem, we are given two versions of the same program P and P' where P' fixes an unknown vulnerability in P. The goal is to generate an exploit for P for the vulnerability fixed in P'. For example, if P is the program shown in Figure 1.2a, and P' is Figure 1.2b, APEG uses P' and P to generate an exploit for P.

It has not previously been shown that automatic patch-based exploit generation is possible. Our ability to automatically generate patch-based exploits has important implications for the security landscape. For example, successful APEG means that those who have a patch should be considered armed with an exploit. Current security and patch distribution policies typically assume releasing a patch only helps security.

Approach and Technique Intuition. Our approach for automatic patch-based exploit generation is based on the observation that input-validation bugs are usually fixed by adding the missing sanitizing checks. Intuitively, an input which fails the added check in the patched program is likely an exploit for the unpatched version, since the missing check likely causes the vulnerability.

At a high level, our approach generates exploits by a) differencing the original and patched version to find new checks, b) identifying potentially exploitable code paths to a new check, then c) generating inputs which execute those paths and fail a new check. In addition, we propose 3 techniques for picking possible exploitable code paths for further analysis, each of which is useful in different scenarios.

Results. We have developed and implemented techniques which demonstrate that automatic patch-based exploit generation is practical. Specifically, we automatically generate exploits for 5 vulnerabilities using patches from Microsoft. Each patch addresses a serious security vulnerability. In several cases, the vulnerabilities are widely exploited, indicating the extent at which automatic patch-based exploit generation may affect the security landscape.

Our results also show that each of the 3 approaches have strengths for different vulnerabilities. In each case we are able to generate an exploit, usually within a few minutes. The fastest end-to-end time we were able to generate a verifiable exploit was 5.68 seconds (Section 5.3.2). The slowest end-to-end time when APEG is successful was 186.74 seconds (Section 5.3.3). Our experiments also show that further reducing exploit generation time is possible.

In our evaluation, we automatically generate exploits for several vulnerabilities which previously have no publicly known exploits. In all cases, we can generate exploits which are different from those publicly known. Further, we show we can automatically generate polymorphic variants. Thus, the exploits we generate are not only original, but are likely difficult to detect by existing signature defense systems. We conclude that our evaluation shows automatic patch-based exploit generation is a serious security problem.

APEG Contributions. We show that APEG is practical and that the delayed patch attack is therefore a realistic threat. Our results have several immediate security ramifications. First, a patch distribution scheme that staggers roll out of a patch over a large time window is insecure since the first user to receive a patch may be able to create an exploit before most users have received the patch. Recall that it takes about 24 hours for 80% of the unique IP's using Windows Automatic Update to notice a new patch is available [48]. Since we can potentially generate exploits in minutes or less, this implies that we could potentially compromise most vulnerable hosts before they can even download the patch, let alone apply it. Many large vendors employ similarly staggered patch roll-outs. Therefore, such patch distribution schemes should be considered insecure and redesigned to address the threat of delayed patch attacks.

Second, automatic patch-based exploit generation provides a way for users to discover which vulnerabilities are addressed by a patch. A common vendor practice is to be vague about which vulnerabilities a patch addresses (or not disclose them at all). A possible reason for this is to limit the information given to potential attackers. For example, the Microsoft MS07-030 security bulletin states that the patch addresses "two privately reported vulnerabilities in addition to other security issues identified during the course of the investigation." Such vague descriptions may not give users enough information to decide whether to apply a patch or not. However, an automatically generated exploit based upon the patch can be used by users to help identify what vulnerabilities are addressed, and how easy it is to generate an exploit for the vulnerability, and therefore how important it may be to apply the patch.

Finally, the techniques themselves are useful for other applications. Automatic patch-based exploit generation is related to the more general problem of goal-based automatic test case generation because both want to generate an input which reaches a particular line of code. Our techniques explore the practicality of proposed static and dynamic approach to binary code. We show that neither static nor dynamic alone is sufficient, and a combined approach is often necessary in practice. Our techniques also suggest that generating test cases that exercise the different between two program versions (as opposed to generating test cases for a single version) is a useful direction for software testing.

1.3.2 Automatic Filter Generation

Problem Overview. Input filtering is currently one of the most popular and effective defense systems for software vulnerabilities. Input filters are recognizers for inputs which will exploit a vulnerability. Input filtering defenses check each input against a list of filters, dropping any packets that match. Input filtering allows even vulnerable systems to operate even under attack.

In the *filter generation problem*, we are given a vulnerable program P and a description of the vulnerability. The goal is to generate a filter that recognizes all inputs x such that $P(x)$ would exploit the vulnerability. In our running example, we are given the program in Figure 1.2a, and a description of the vulnerability that says an exploit is an input such that $\neg(s > i)$ on line 5. A recognizer would match $i \geq 2^{32} - 3$.

Input filtering defense systems are in constant need of new filters as new vulnerabilities are discovered. Therefore, a natural question is whether we can automatically generate input filters. We need *automated* filter generation techniques because manual filter generation is slow and error-prone. Automated techniques are necessary because previously unknown ("zero-day") or unpatched vulnerabilities can be exploited orders of magnitude faster than a human can respond, such as during a worm outbreak. Automated techniques also have the potential to be more accurate than manual efforts because vulnerabilities tend to be complex and require intricate knowledge of details such as realizable program paths and corner conditions. Understanding the complexities of a vulnerability has consistently proven to be very difficult for humans at even the source code level, let alone COTS software at the assembly level.

We also need to know what types of vulnerabilities are amenable to automated filter generation. The gamut of possible vulnerabilities include safety properties, liveness properties, side channels, and others. We must know what types of vulnerabilities are amenable in order to know when a proposed filter-based defense is applicable.

There are usually several different exploit variants that trigger the same vulnerability, which complicates automated filter generation further. For example, a buffer-overrun vulnerability in a network service may be triggered by many different protocol messages. There are several existing tools that given an exploit can generate exploit variants (e.g., Metasploit [3] and CLET [36]). Further, our work on automatic patch-based exploit generation shows that automatic polymorphic exploit generation, where each variant is substantially different, is possible. *Thus, to be effective, the filter should be constructed based on the property of the vulnerability, instead of an exploit* (this observation has been made by others as well [30, 103]).

We would like generated filters to have *guarantees*. In particular, filters should have accuracy guarantees. The whole point of a filter-based defense is to filter out exploits: if a filter cannot make any guarantees that it will filter out exploits, then the system is of dubious benefit. Even when a filter does not filter out all and only exploits, i.e., is not perfect, we still want guarantees for what will be filtered.

We need accuracy guarantees to hold within the constraints of the filter-based defense. In particular, a filter may be written in a variety of language classes, from regular expressions to complete programming languages. A defense system will choose a particular language class in order to meet performance requirements, e.g., network-based defenses typically use regular expressions because regular expressions are fast. As we will see, there is usually a trade-off between the power afforded to the filtering language and the accuracy of the filter itself. Therefore, we would also like filter generation algorithms which can generate accurate (though potentially not perfect) filters with respect to limitations of the filtering language itself.

Previous approaches for automatic filter generation have not offered sufficient guarantees. Most previous work on filter generation is based on machine learning of syntactic properties of exploits [57,59,63,81,96]. These systems typically lack accuracy guarantees. Worse, several of these systems have been shown to be vulnerable to an attacker where an attacker can cause the generated filter to have errors [63,82,86]. In 2008, Venkataraman *et al.* provided a generalized result that shows any machine learning approach to automatic filter generation could be fooled by attackers [102].

In summary, the requirements for filter generation are:

- Automated techniques for filter generation.
- The set of requirements for deciding when a vulnerability is amenable to the proposed techniques.
- Accuracy guarantees for the generated filter within the constraints of the filtering language itself.

Previous work has not addressed all of these requirements.

Approach and Technique Intuition. We formalize filter generation for input validation vulnerabilities. As we will see, our definition of an input validation vulnerability means our techniques filter generation will succeed if there is an execution monitor for detecting exploits.

Our techniques are motivated by the idea that the vulnerable program itself encodes all the details necessary to determine whether the program would be exploited. A filter is a

summary of the conditions under which it would be exploited. More concretely, in order for an input to exploit an input validation vulnerability, it must satisfy a set of control and data dependencies defined largely by the code itself. We take a vulnerability-centric approach to filter generation where we employ program analysis of the control and data dependencies when generating a filter.

Filter-based defenses may require filters to be restricted to a particular language class, e.g., regular expressions. Particular restrictions are often chosen in order to meet specific performance requirements. For example, regular expressions are popular among networked filter-based defenses because they can be checked at network line speeds. In contrast, host-based defenses often allow filters to be expressed in Turing-complete languages.

We develop vulnerability analysis techniques for filter generation for several different language classes. As we will see, the filter language inherently offers a trade-off between accuracy and efficiency.

First, we propose techniques for automatically generating error-free filters. An error-free filter recognizes exactly the set of unsafe inputs. One result of our formalism is an error-free filter for an arbitrary input validation vulnerability requires an unconstrained filtering language. We call such filters *Type-T* filters, since they require a Turing-complete filtering language. Executing a Type-T filter on an input in the domain of the vulnerable program will return either safe, indicating the input is safe, else EXPLOIT, indicating the input would exploit the vulnerable program. We then propose techniques for automatically generating filters with other restrictions. We propose techniques for generating filters restricted to Boolean formulas, called *Type-B filters*. We also demonstrate techniques for generating regular expressions filters (Type-R filters) that still retain some accuracy guarantees.

Type-T, Type-B, and Type-R filters are all of interest because they offer performance/accuracy guarantee trade-offs that are important in real-world settings. Our approach shows that only Type-T filters can have perfect accuracy for arbitrary input validation vulnerabilities. However, Type-T filters offer few other guarantees. Type-B and Type-R offer performance guarantees, but may not have perfect accuracy. We develop techniques that generate filters where we guarantee one-sided errors (previous techniques usually had both false positives and false negatives). In particular, we focus on techniques for sound filter generation, where if the filter says an input is an exploit, the input is guaranteed to be an exploit. Our techniques can easily be adapted for generating complete filters which filter out all exploits, but may make mistakes about safe inputs.

Results. We present a new approach and framework for filter generation. Our approach roots filter generation, both in theory and in practice, on program analysis. Our approach provides guarantees on both accuracy and performance for a wide-variety of filter types.

At a high level, we: (1) develop a formal basis for the filter generation problem, (2) provide methods for automatically generating filters in a variety of language classes, each of which has its own guarantees, and (3) develop an architecture for automated vulnerability filter generation given only a description of the vulnerability and the vulnerable program binary.

In our evaluation, we show that we can create filters that have guaranteed perfect accuracy for a number of real vulnerabilities. We also show that we can create filters that are guaranteed to have zero false positives, i.e., are sound, when performance instead of accuracy is the primary goal.

Contributions. At a high level, our approach provides the first formal model for automatic vulnerability filter generation. Previous approaches did not codify the class of vulnerabilities for which their techniques are applicable. Further, our approach provides a metric for the security gained or lost from restricting the language class. The metric we propose is in terms of the number of potential exploitable code paths a filter covers. Previous approaches did not provide techniques for different restrictions on language classes or formalize the trade-offs when restrictions are introduced. For example, previous work that generated regular expressions [57, 59, 63, 81, 96] never stated how accurate the filter was compared to a perfect filter. Our approach provides one logical way for making such comparisons.

Our approach shows program analysis techniques can be applied to filter generation. Previous work in automatic filter generation did not make this connection. As a result, our approach enables the security community to take advantage of research and advancements in these fields, and potentially vice versa. For example, we develop new techniques for computing the weakest precondition for computing symbolic constraint filters for binary programs [24]. This advancement benefits other security applications, and is of interest to formal methods.

Last, in practice most current filters are hand-crafted by well-trained experts. In our partnership with industry affiliates, we have shown that our filters are of much higher quality that even those hand-crafted filters, while being quicker to generate. As a result, industry is now improving filters found in commercial software based upon our results.

In all our experiments, we verify our approaches work on real vulnerabilities in production binary code. Thus, our results demonstrate that users can generate filters themselves

without help from other parties such as vendors or network operators.

1.3.3 Vine: Our Binary Analysis Infrastructure

Problem Overview. Binary program analysis is attractive because binary code is everywhere, and the binary code is faithful to the low-level details of what will actually execute on hardware. However, there are two primary challenges to performing accurate and faithful analysis on modern binary code: the engineering challenges associated with analyzing binary code, and the new code analysis problems arising because binary code lacks many abstractions found in higher-level languages.

The first major challenge is that binary code is complex. Program analysis needs to model the complexity accurately in order for the analysis itself to be accurate. However, accurately modeling the sheer number and complexity of instructions in modern architectures poses a significant challenge. Popular modern architectures typically have hundreds of different instructions, with new ones added at each processor revision. Furthermore, each instruction can have complex semantics, such as single instruction loops, instructions which behave differently based upon the operand values, and implicit side effects such as setting processor flags. The IA-32 manuals describing the semantics of x86 [53] weigh in at over 11 pounds!

More concretely, consider the problem of determining the control flow in the following x86 assembly program:

```
// instruction dst, src
add a, b    // a = a+b
shl a, x    // a << x
jz target     // jump if zero to address target
```

The first instruction, `add a,b`, computes `a := a+b`. The second instruction, `shl a,x`, computes `a := a << x`. The last instruction, `jz a`, jumps to address `a` if the processor zero flag is set.

One problem is both the `add` and `shl` instruction have implicit side effects. Both instructions may calculate up to *six* other bits of information that are stored as processor status flags. In particular, they may calculate whether the result is zero, the parity of the result, whether there was carry, whether than was an auxiliary carry, whether the result is signed, and whether overflow occurred are all also calculated and stored as status flags.

Conditional control flow, such as the `jz` instruction, is determined by the implicitly calculated processor flags. For example, both `add` and `shl` may set the zero conditional

control flow used by the subsequent `jz`. A seemingly straight-forward is to determine when the jump on line 3 is taken. However, answering this question turns out not to be straight-forward. The `shl` instruction behaves *differently* depending upon the operands: it only updates the zero flag if `x` is not zero. Thus, sometimes `add` may determine subsequent control flow, and sometimes `shl` will.

The second major challenge is binary code is different than source code. Thus, we must develop and only use program analyses that are suitable for the unique characteristics of binary code. In particular, binary code lacks abstractions that are often fundamental to source code and source code analysis, such as:

- **Functions.** The function abstraction does not exist at the binary level. Instead, control flow in a binary program is performed by jumps. For example, the x86 instruction `call x` is just syntactic sugar (i.e., shorthand) for storing the number in the instruction pointer register `eip` at the address named by the register `esp`, decrementing `esp` by the architecture word size, then loading the `eip` with number `x`. Indeed, it is perfectly valid in assembly, and sometimes happens in practice, that one may call into the middle of a "function", or have a single "function" separated into non-contiguous pieces. The lack of a function abstraction poses significant scalability challenges to binary analysis.
- **Memory vs. Buffers.** Binary code does not have buffers, it has *memory*. While the OS may determine a particular memory page is not valid, memory does not have the traditional semantics of a user-specified type and size. One implication of the difference between buffers and memory is that in binary code there is no such thing as a buffer overflow. While we may say a particular store violates a higher-level semantics given by the source code, such facts are inferences with respect to the higher-level semantics, not part of the binary code itself. The lack of buffers means we have to conservatively reason about the entire memory space (unless proven otherwise) at each operation.
- **No Types.** New types cannot be created or used since there is no such thing as a type constructor in binary code. The only types available are those provided by the hardware: registers and memory. Even register types are not necessarily a good choice, since it is common to store values from one register type (e.g., 32-bit register) and read them as another (e.g., 8-bit register). The lack of types means we cannot use type-based analysis, which is often instrumental in scaling traditional analyses.

Vine: Our Binary Analysis Infrastructure We have designed and developed *Vine,* a language and infrastructure which facilitates analysis of x86 system binaries (executables). The central design principle for Vine is to provide a faithful and extensible infrastructure

for performing security-relevant program analysis on binaries.

Vine differs from disassemblers, decompilers and other binary analysis platforms by treating assembly as a first-class language. While higher-level languages have types, functions, pointers, loops, and local variables, Vine is geared towards assembly analysis where there fundamentally are few types, no functions, one globally addressed memory region, gotos and stack frames instead of local variables. By treating assembly as a first class language, Vine is faithful to the semantics of the binary code.

Contributions. Our binary-centric approach means that our methods will be applicable to a wide audience. In practice, this means users of the code do not need to cooperate with external parties (such as the software vendor) to achieve a security goal, which is a significant security benefit. The ability to analyze low-level details in practice means that we can make fine-grained judgements about program behavior, e.g., whether a vulnerability poses a denial-of-service risk vs. a vulnerability that can allow an attacker to hijack control of the program. The lack of abstractions also means that a binary-centric approach provides a "worse case" scenario for testing software analysis techniques.

The common Vine infrastructure allows us to realize these benefits for both APEG and filter generation. In addition, the Vine infrastructure fuels 11 different security research projects. For example, Vine is used to analyze malicious software, find differences in how two programs implement the same network protocol specification, and to reverse engineer network protocols.

1.4 Thesis Outline

This thesis is organized into several parts. In the first part, we cover the Vine infrastructure, as well as the algorithms we will use in the remainder of this thesis. In Chapter 2, we introduce Vine, our architecture for conducting binary analysis. We introduce Vine first because it introduces our syntax for expressing binary programs in later chapters. In Chapter 3, we introduce our algorithms and definitions used throughout this thesis. Note we will use algorithms and syntax developed in this part of the thesis in subsequent parts often without further explanation.

In the second part, we detail our techniques for input validation vulnerability analysis and defense. We first define input validation vulnerabilities in Chapter 4. In Chapter 5, we describe our techniques for automatic patch-based exploit generation. In Chapter 6, we show how to automatically generate input filters.

In the final part, we provide an overall discussion and our concluding remarks.

Part I

Binary Program Analysis

Chapter 2

The Vine Binary Analysis Platform

2.1 Overview

Our binary-centric approach to software security requires the ability to perform program analysis on binary code. A program analysis (whether it be static or dynamic) is an algorithm for determining the effects of a set of statements in the programming language under consideration. In particular, a binary-centric approach requires that we 1) be able to analyze each instruction in a manner faithful to its semantics, and 2) provide a way of encoding an algorithm over those instructions.

One approach is to disassemble binary code into a sequence of assembly instructions, then perform program analysis directly over the assembly instructions. This type of assembly-specific approach is naïve because each analysis would have to individually understand the semantics of the assembly, which is difficult.

For example, consider the basic program analysis problem of determining when the conditional jump is taken on line 3 of Figure 2.1. In this example, all operands are 32-bit registers, and all arithmetic is therefore performed mod 2^{32}. Instruction 1 computes the sum `eax := eax+ebx` (mod 2^{32}). Instruction 2 computes `eax = eax` $\ll$ `edx`. Both instructions may set the overflow status register `OF`. The `add` instruction will set the `OF` flag if `eax + ebx` $\geq 2^{32}$. The `shl` instruction will set the `OF` flag if `edx` is 1 *and* the top two bits of `eax` are not equal on line 2. The instruction on line 3 tests to see if `OF` is set, and if so, jumps to `target`, else executes the next sequential instruction.

In order to determine when the jump on line 3 is taken, an analysis must reconstruct all side-effects of `add` and `shl`. The proper control flow diagram is shown in Figure 2.1. As demonstrated by this three line program, even though a program may look simple,

```
// instr dst,src
1. add eax, ebx
2. shl eax, edx
3. jo target
4. ....
...
target: ...
```

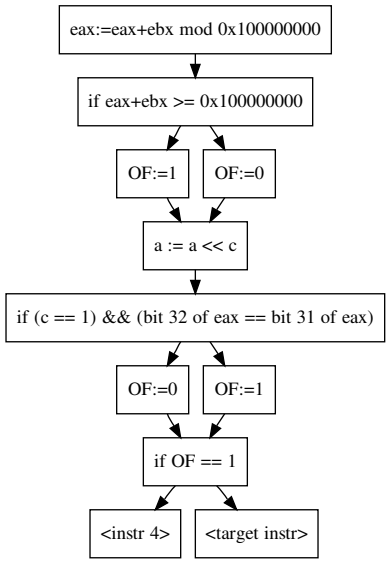


Figure 2.1: A three instruction x86 program, and its corresponding control flow graph. Note the logic for deciding when the jump on overflow instruction (statement 3) is taken, when is often omitted by other binary analysis platforms.

determining the effects may be complicated. An assembly-specific approach would require each analysis to reason about the complex semantics.

Worse, many architectures allow instruction prefixes, which can further complicate the semantics. For example, in x86 a `rep` prefix turns an instruction essentially into a single instruction loop, e.g., `rep` i s,d will repeatedly execute operation i on arguments s and d. The exact semantics (per Intel [53]) of `rep` are shown in Figure 2.2. An assembly-specific approach would have to consider this complicated logic for any instruction that may carry to `rep` prefix. If analyses are written directly on assembly, each analysis would duplicate this logic.

If there were only a few instructions, perhaps the complexity would not be too onerous. Modern architectures, however, typically have hundreds of instructions. x86, for example, has well over 300 instructions (recall the Intel x86 manuals [53] weigh over 11 lbs).

Overall, an assembly specific approach is unattractive because writing analyses over modern complex instruction, such as the logic found in Figures 2.2 and 2.1, tends to be tedious and error-prone. For example, the current version of DynInst [85] (version 5.2) and Phoenix [67] (April 2008 SDK Build), two popular architectures for binary analysis, will not create a correct control flow graph for the three line program. Verifying a program

```
IF AddressSize = 16
THEN
   Use CX for CountReg;
ELSE IF AddressSize = 64 and REX.W used
   THEN Use RCX for CountReg; FI;
ELSE
   Use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
DO
    Service pending interrupts (if any);
    Execute associated string instruction (x);
    CountReg ← (CountReg  1);
    IF CountReg = 0
    THEN exit WHILE loop; FI;
    IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
       or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
    THEN exit WHILE loop; FI;
OD;
```

Figure 2.2: The semantics of the `rep` instruction according to Intel [53]. Modern architectures often have hundreds of instructions that have complex semantics like `rep`.

analysis is correct over a large and complicated instruction set seems even more difficult.

2.1.1 Desired Properties

We would like a platform for analyzing binary code that 1) supports writing analyses in a concise and straight-forward fashion, 2) provides abstractions for common semantics across all assemblies, and 3) is architecture independent when possible.

We want an architecture that supports writing analyses in a concise, straight-forward fashion because that makes it easier to write analyses that are correct. We do not want an analysis to have to tangle with complicated instruction semantics: that should be the job of the architecture.

We would also like to provide abstractions that are common to typical program analyses. There are many recurring abstractions. One is to be able to iterate over each instruction type. Another is to build a control flow graph. Yet another is finding data dependencies. We would like to build a single platform so that a new program analysis can easily reuse common abstractions.

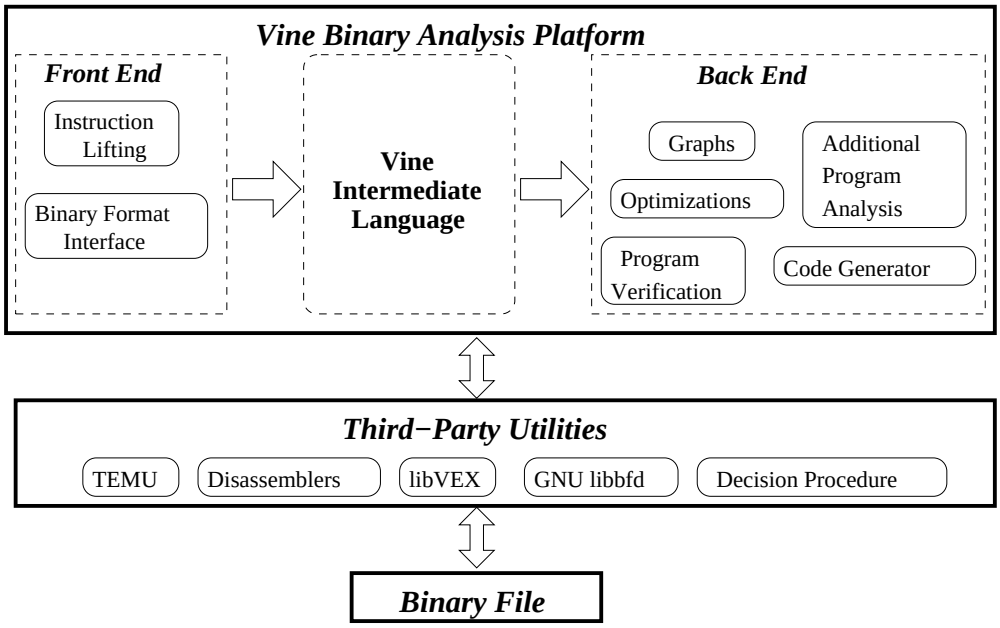


Figure 2.3: TheVine binary analysis architecture and components. Vine is divided into a front-end, which is responsible for lifting instructions to the Vine IL, and a platform-independent back-end for analyses.

Finally, we would like to be architecture independent. Architecture independence would allow us to easily re-target the entire platform to a new architecture without changing the analyses. The more architectures we can handle, the more widely our techniques will be applicable.

2.1.2 Vine Overview

The Vine platform is designed to facilitate faithful security-relevant binary program analysis by 1) reducing complex instruction sets to a single, small, and formally specified intermediate language that supports writing concise, easy-to-understand analyses 2) providing a set of core program analyses abstractions, and 3) being architecture independent when possible in order to support easy re-targeting.

Figure 2.3 shows a high-level picture of Vine. The Vine platform is divided into a platform-specific front-end and a platform-independent back-end. At the core of Vine is a platform-independent intermediate language (IL) for assembly. Assembly instructions in the underlying architecture are lifted up to the Vine IL via the Vine front-end. All analyses

are performed on the platform-independent IL in the back-end.

We lift to the IL by using open-source utilities to parse the binary format and produce assembly. The assembly is then lifted up to the Vine IL in a syntax-directed manner. The Vine front-end currently supports lifting x86 [53] and ARMv4 [12] to the IL.

The Vine back-end supports a variety of core program analyses and utilities. The back-end has utilities for creating a variety of different graphs, such as control flow and program dependence graphs. The back-end also provides an optimization framework. The optimization framework is usually used to simplify a specific set of instructions. We also provide program verification capabilities such as symbolic execution, calculating the weakest precondition, and interfacing with decision procedures. Vine can also write out lifted Vine instructions as valid C code via the code generator back-end.

For example, one may want to analyze an execution trace. The execution trace can be lifted to Vine, then optimized to remove instructions from the trace that are not relevant to the analysis. The analysis could then verify properties of the trace by using program verification routines. Finally, the trace itself could be written out as a valid C program, compiled down to assembly, and then run as a straight-line program.

2.1.3 Chapter Outline

In the remainder of this chapter, we provide a description of the Vine IL in 2.2. We use notation from the IL throughout the rest of this thesis. The remainder of the chapter describes the implementation and formal semantics of Vine. These sections describe how Vine addresses the engineering challenges of faithful binary analysis. However, this material is not essential to understanding the remainder of this thesis.

2.2 The Vine Intermediate Language

At the core of Vine is the Vine intermediate language (IL), shown in Table 2.1. The IL is the target language during lifting, as well as for writing program analyses. In Chapter 2.2.4 we give basic type-checking rules which disallow certain syntactically valid but nonsensical expressions, and in Chapter 2.2.5 we provide the operational semantics. In the remainder of this section we give an informal description and motivation for constructs in the IL.

The base types in the Vine IL are 1, 8, 16, 32, and 64-bit registers (i.e., n-bit vectors), and memories. A memory type is qualified by its endianness, which can be either `little`

program	::=	*decl* instr**
instr	::=	*var* = *exp* | `jmp` *exp* | `cjmp` *exp*,*exp*,*exp* | `halt` *exp* | `assert` *exp* | `label` *integer* | `special` id_s
exp	::=	`load`(*exp*, *exp*, τ_{reg}) | `store`(*exp*, *exp*, *exp*,τ_{reg}) | *exp* $\Diamond_b$ *exp* | $\Diamond_u$ *exp* | *const* | *var* | `let` *var* = *exp* `in` *exp* | `cast`(*cast_kind*,τ_{reg},*exp*)
cast_kind	::=	`unsigned` | `signed` | `high` | `low`
decl	::=	`var` *var*
var	::=	(string, id_v, τ)
$\Diamond_b$	::=	$+, -, *, /, /_s, \text{mod}, \text{mod}_s, \ll, \gg, \gg_a, \&, \mid, \oplus, ==, \neq, <, \leq, <_s, \leq_s$
$\Diamond_u$	::=	$-$ (unary minus), ! (bit-wise not)
value	::=	*const* | $\{ n_{a1} \rightarrow n_{v1}, n_{a2} \rightarrow n_{v2}, \ldots \}$: τ_{mem} | $\perp$
const	::=	$n : \tau_{\text{reg}}$
τ	::=	τ_{reg} | τ_{mem} | Bot | Unit
τ_{reg}	::=	`reg1_t` | `reg8_t` | `reg16_t` | `reg32_t` | `reg64_t`
τ_{mem}	::=	`mem_t` ($\tau_{\texttt{endian}}$, $\tau_{\texttt{reg}}$)
τ_{endian}	::=	`little` | `big` | `norm`

Table 2.1: The Vine Intermediate Language. (Since '|' is an operator, for clarity we use commas to separate all operator elements.)

(e.g., for little-endian architectures like x86), `big` (e.g., for big-endian architectures such as PowerPC), architectures, and `norm` for normalized memory. We describe memory normalization below. A memory type is also qualified by the index type, which must be a register type. For example `mem_t(little, reg32_t)` denotes a memory type which is little endian and is addressed by 32-bit numbers.

There are three types of values in Vine. First, Vine has numbers n of type τ_{reg}. Second, Vine has memory values $\{n_{a1} \rightarrow n_{v1}, n_{a2} \rightarrow n_{v2}, ...\}$, where n_{ai} denotes a number used as an address, and n_{vi} denotes the value stored at the address. Finally, Vine has a nonsense value $\perp$. $\perp$ values are not exposed to the user and cannot be constructed in the presentation language. $\perp$ is used internally to indicate a failed execution.

Expressions in Vine are free of side-effects. The Vine IL has binary operations $\Diamond_b$ (note "&" and "|" are bit-wise), unary operations $\Diamond_u$, constants, `let` bindings, and casting. Casting is used when indexing registers under different addressing modes. For example,

the lower 8 bits of `eax` in x86 are known as `al`. When lifting x86 instructions, we use casting to project out the lower-bits of the corresponding `eax` register variable to an `al` register variable when `al` is accessed.

In Vine, both `load` and `store` operations are pure. While `load` is normally pure, the semantics of `store` are often not. Each `store` expression must specify what memory to load or store from. The resulting memory is returned as a value. For example, a Vine store operation is written `mem1 = store(mem0, a, y)`, where `mem1` is the same as `mem0` except address `a` has value `y`. The advantage of pure memory operations in Vine notation is it makes it possible to syntactically distinguish what memory is modified or read. One place we take advantage of this is in SSA (see 2.4) where both scalars and memory have a unique single static assignment location.

A program in Vine is a sequence of variable declarations, followed by a sequence of instructions. There are 7 different kinds of instructions. The language has assignments, jumps, conditional jumps, and labels. The target of all jumps and conditional jumps must be a valid label in our operational semantics, else the program terminates in the error state ($\perp$). Note that a jump to an undefined location (e.g., a location that was not disassembled such as to dynamically generated code) results in the Vine program halting with $\perp$ (see 2.2.5). A program can halt normally at any time by issuing the `halt` statement. We also provide `assert`, which acts similar to a C assert: the asserted expression must be true, else the machine halts with $\perp$.

A `special` in Vine corresponds to a call to an externally defined procedure or function. `special` statements typically arise from system calls. The `id` of a `special` indexes the kind of special, e.g., what system call.

The semantics of `special` are up to the analysis; its operational semantics are not defined (Chapter 2.2.5). We include `special` as an instruction type to explicitly distinguish calls that alter the soundness of an analysis. A typical approach to dealing with `special` is to replace `special` with an analysis-specific summary function written in the Vine IL that is appropriate for the analysis.

2.2.1 Normalized Memory

The endianness of a machine is usually specified by the byte-ordering of the hardware. A little endian architecture puts the low-order byte first, and a big-endian architecture puts the high-order byte first. (We discuss bi-endian memory operation support in 2.6.)

We must take endianness into account when analyzing memory accesses. Consider the assembly in Figure 2.4a. The `mov` operation on line 2 writes 4 bytes to memory in little

```
 // x86 instr dst,src
1. mov [eax], 0xaabbccdd
2. mov ebx, eax
3. add ebx, 0x3
4. mov eax, 0x1122
5. mov [ebx], ax
6. sub ebx, 1
7. mov ax, [ebx]
```

(a)

address	memory
eax	0xdd
eax+1	0xcc
eax+2	0xbb
eax+3	0xaa
eax+4	

(b)

address	memory
eax	0xdd
eax+1	0xcc
eax+2	0xbb
eax+3	0x22
eax+4	0x11

(c)

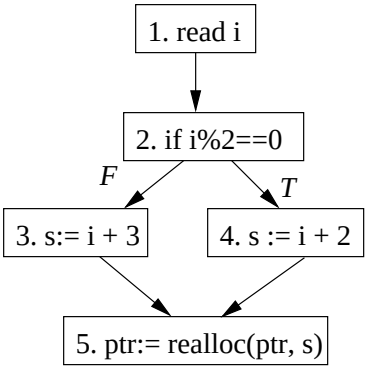
Figure 2.4: (a) shows an example of little-endian stores as found in x86 that partially overlap. (b) shows memory after executing line 1, and (c) shows memory after executing line 5. Line 7 will load the value 0x22bb.

```
1. mem4 = let mem1 = store(mem0,eax, 0xdd, reg8_t) in
          let mem2 = store(mem1, eax+1, 0xcc, reg8_t) in
          let mem3 = store(mem2, eax+2, 0xbb, reg8_t) in
              store(mem3, eax+3, 0xcc, reg8_t);
...
5. mem6 = let mem5 = store(mem4, ebx, 0x22, reg8_t) in
            store(mem5, ebx+1, 0x22, reg8_t)
...
7. value = let b1 = load(mem6, ebx, reg8_t) in
           let b2 = load(mem6, ebx+1, reg8_t) in
           let b1' = cast(unsigned, b1, reg16_t) in
           let b2' = cast(unsigned, b2, reg16_t) in
              (b2' ≪ 8) | b1';
```

Figure 2.5: Vine normalized version of the store and load from Figure 2.4a.

endian order (since x86 is little endian). After executing line 2, the address given by `eax` contains byte `0xdd`, `eax+1` contains byte `0xcc`, and so on, as shown in Figure 2.4b. Lines 2 and 3 set `ebx = eax+2`. Line 4 and 5 write the 16-bit value `0x1122` to `ebx`. An analysis of these few lines of code needs to consider that the write on line 4 overwrites the last byte written on line 1, as shown in Figure 2.4c. Considering such cases requires additional logic in each analysis. For example, the value loaded on line 7 will contain one byte from each of the two stores.

We say a memory is *normalized* for a b-byte addressable memory if all loads and stores are exactly b-bytes and b-byte aligned. In x86, memory is byte addressable, so a normalized memory for x86 has all loads and stores at the byte level. The normalized form for the write on Line 1 of Figure 2.4a in Vine is shown in Figure 2.5. Note that the




```
// address instr dst, src
0x8048419 push    ebp  // setup stack
0x804841a mov     ebp,esp
0x804841c sub     esp, 0x18
0x804841f lea     eax, [ebp-0x8]
0x8048422 mov     [esp], eax
0x8048425 call    0x80483f4 // call read
0x804842a mov     eax, [ebp-0x8]
0x804842d and     eax, 0x1  // if i%2==0
0x8048430 test    eax,eax
0x8048432 jne     0x804843f
0x8048434 mov     eax,[ebp-0x8]
0x8048437 add     eax, 0x2  // s := i+2
0x804843a mov     [ebp-0x4], eax
0x804843d jmp     0x8048448
0x804843f mov     eax,[ebp-0x8]
0x8048442 add     eax,0x3   // s := i+3
0x8048445 mov     [ebp-0x4], eax
0x8048448 mov     eax, [ebp-0x4]
0x804844b mov     [esp+0x4], eax
0x804844f mov     eax, [ebp+0x8]
0x8048452 mov     [esp],eax
0x8048455 call    0x8048338 // call realloc
0x804845a mov     [ebp+0x8],eax
0x804845d leave
0x804845e ret
```

Figure 2.6: Our running example from 1.2 on the left, and the corresponding x86 assembly in Intel syntax on the right.

subsequent load on line 7 is with respect to the current memory `mem6`.

Normalized memory makes writing program analyses involving memory easier. Analysis is easier because normalized memory syntactically exposes memory updates that are otherwise implicitly defined by the endianness. As a result, analyses do not have to reason explicitly about overlapping memory, byte order, etc. The Vine back-end provides utilities for normalizing all memory operations.

2.2.2 Vine and the Running Example

Figure 2.6 shows the assembly for the running example from Chapter 1 (which is reproduced as part of the figure). The assembly (given in Intel syntax) shown is for the running example compiled as a single function which is passed in `ptr`, with `read` and `realloc` as external calls. Each assembly line contains the instruction address followed by the instruction with operands.

The first 4 instructions in Figure 2.6 implement the function prologue, which sets up

```
label 0x8048419; //push   ebp
  ESP = ESP - 4;
  mem = store(mem, ESP, EBP, reg32_t);
label 0x804841a; //mov    ebp,esp
  EBP = ESP;
label 0x804841c; //sub esp, 0x18
  ESP = ESP-24;
  /* 'sub' eflags omitted */
label 0x804841f; //lea    eax, [ebp-0x8]
  EAX = (EBP+0xFFFFFFF8);
label 0x8048422; //mov    [esp], eax
  mem = store(mem, ESP, EAX, reg32_t);
label 0x8048425; //call   0x80483f4
  ESP = ESP-4;
  mem = store(mem, ESP, 0x804842A, reg32_t);
  jmp(0x80483f4);
label 0x804842a; //mov    eax, [ebp-0x8]
  EAX = load(mem, EBP+0xFFFFFFF8,reg32_t);
label 0x804842d; //and    eax, 0x1
  EAX = EAX & 1;
  /* 'and' eflags omitted */
label 0x8048430; //test   eax,eax
  temp = EAX & EAX;
  ZF:reg1_t = temp == 0;
  /* SF and PF code omitted */
label 0x8048432; //jne    0x804843f
  cjmp(ZF,0x8048434,0x804843F);
label 0x8048434; //mov    eax,[ebp-0x8]
  EAX = load(mem, EBP+0xFFFFFFF8, reg32_t);
label 0x8048437; //add    eax, 0x2
  EAX = EAX + 2;
  /* 'add' eflags omitted */
label 0x804843a; //mov    [ebp-0x4], eax
  mem = store(mem, EBP+0xFFFFFFFC, EAX, reg32_t);
label 0x804843d; //jmp    0x8048448
  jmp(0x8048448);
label 0x804843f; //mov    eax,[ebp-0x8]
  EAX = load(mem, EBP+0xFFFFFFF8, reg32_t);
label 0x8048442; //add    eax,0x3
  EAX = EAX+3;
  /* 'add' eflags omitted */
label 0x8048445; //mov    [ebp-0x4], eax
  mem = store(mem, EBP+0xFFFFFFFC, EAX, reg32_t);
label 0x8048448; //mov    eax, [ebp-0x4]
  EAX = load(mem, EBP+0xFFFFFFFC, reg32_t);
label 0x804844b; //mov    [esp+0x4], eax
  mem = store(mem, ESP+4, EAX, reg32_t);
label 0x804844f; //mov    eax, [ebp+0x8]
  EAX = load(mem, EBP+8, reg32_t);
label 0x8048452; //mov    [esp],eax
  mem = store(mem, ESP, EAX, reg32_t);
label 0x8048455; //call   0x8048338
  ESP = ESP-4;
  mem = store(mem, ESP, 0x804845A, reg32_t);
  jmp(0x8048338);
label 0x804845a; //mov    [ebp+0x8],eax
  mem = store(mem, ESP+8, EAX, reg32_t);
label 0x804845d; //leave
  ESP = EBP+4;
  EBP = load(mem, EBP, reg32_t);
label 0x804845e; //ret
  target = load(mem, ESP, reg32_t);
  ESP = ESP+4;
  jmp(target);
```

Figure 2.7: The assembly from Figure 2.6 in the Vine IL.

the stack frame. The variable `i` is assigned by the compiler to register `eax`. Instruction 0x804822-0x804825 push the argument `eax` onto the stack, and then call the function corresponding to `read` at address 0x80483f4. Instruction 0x804842d-0x8048430 correspond to the test `if i %2 == 0`. The compiler implemented the reduction modulo 2 as bitwise "and". Instructions 0x8048437-0x804843d implement the true branch where `s := i+2`. The compiler assigned the variable `s` the register `eax` in assembly, which is put on the local stack frame slot `ebp-0x4`. Instructions 0x8048442-0x8048445 implement the false branch where `s := i+3`, again storing the result in `ebp-0x4`. Line 0x804844b-0x48455 correspond to the call to `realloc`, followed by the function epilogue.

Figure 2.7 shows the Vine IL for the assembly in Figure 2.6. For simplicity, we note where `eflags` code is calculated, but do not include the logic in the Figure. Each assembly instruction is lifted in a syntax-directed manner.

Derived Forms

`cjmp(`e, e_t, e_f`)`	$\doteq$	`jmp(let` x `= cast(signed,` e`) in` $(x\&e_t)\|((!x)\&e_f)$`)`

`assert(`e`)`	$\doteq$	`cjmp(`e`, <next instr>, <assert fail>)`
		`...`
		`label <assert fail>: halt` $\perp$

$e_1(:\tau)\ /_s\ e_2(:\tau)$	$\doteq$	$-((-e_1)/e_2)$ when msb($e_1 : \tau$) = 1 and msb($e_2 : \tau$) = 0. $-(e_1/(-e_2))$ when msb($e_1 : \tau$) = 0 and msb($e_2 : \tau$) = 1. $(-e_1)/(-e_2)$ when msb($e_1 : \tau$) = msb($e_2 : \tau$) = 1. e_1/e_2 when msb($e_1 : \tau$) = msb($e_2 : \tau$) = 0
$e_1(:\tau)\ \mathrm{mod}_s\ e_2(:\tau)$	$\doteq$	$e_1 - (e_2 * (e_1/_s e_2)$

Auxiliary Helpers

bits(`reg`i`_t`)	$\doteq$	i
msb($e : \tau$)	$\doteq$	$e \gg (\text{bits}(\tau) - 1)$

Table 2.2: Derived forms in Vine. $e : (\tau)$ denotes that expression e has type τ.

The IL reduces complex x86 instructions to a simplified language. For example, the x86 `push ebp` instruction at 0x8048419 in the IL is de-sugared as decrementing the register `esp` by one word, then storing `ebp` at the resulting address in memory. Another example is in assembly, there is no syntactic relationship between the `test` instruction at 0x8048430 and the conditional jump `jne` at 0x804843f. The IL, however, explicitly shows `test` sets the `ZF` flag, which the raised conditional jump checks.

2.2.3 Derived Forms

There are several derived forms in Vine, as shown in Table 2.2. A derived form is an instruction in our IL that can be rewritten in terms of more fundamental instructions. We only show the derived forms that are non-obvious but useful (e.g., we do not derive all operators in terms of NAND and NOR gates).

For example, in the IL a `cjmp(`e`,` e_t`,` e_f`)` can be considered a derived form for a `jmp` where e_t is chosen when e is true, and e_f is chosen when e is false. This idea is expressed in Vine as jumping to the address returned from `let` x `= cast(signed,` e`) in` $(x\&e_t)|(!x\&e_f)$. Similarly, an `assert` e statement in our language is a derived form that either jumps to the next instruction when e is true, else jumps to a canonical

failed execution point that terminates the program.

We detail the derived forms in Table 2.2.3 as a matter of simplifying the formal semantics of the IL. We could remove all derived forms from the IL itself, resulting in fewer syntactic terms. However, we leave the derived forms in the IL because they tend to simplify logic for commonly occurring syntactic structures. For example, `cjmp` captures the recurring idea of a bi-directional jump conditioned on a predicate. While we could write an analysis with `cjmps` removed, writing such an analysis over the program would be somewhat unnatural. In many cases, the analysis itself may end up keeping track of which jumps are conditional and which are not.

2.2.4 Vine Typing

The Vine IL syntax described in Table 2.1 is designed to be simple. A literal interpretation, however, allows several nonsensical operations such as left-shifting two memory variables. We type-check Vine programs using the rules shown in Table 2.3 to weed out obviously nonsensical operations. We emphasize that type-checking to show program safety is outside the scope of Vine.

The type-checking rules are straight-forward. Vine requires that binary and unary operations be on scalars of the same register type (except shifts where the shift amount need only be an integer). We type-check `hi` and `low` casts such that the resulting type is a smaller register type (in terms of the sub-typing relationship REG-SUBTYPE), as expected. `unsigned` and `signed` casts must be to a wider register type. One issue with unnormalized memory is that it may have multi-byte accesses, e.g., a store on x86 could be of 32, 16, or 8 bits. Normalized memory, on the other hand, only allows stores and loads of bytes. We define an auxiliary predicate "mem_compat" to ensure that normalized memory is accessed appropriately. Also note that as a convenience we assume 64-bit addressing in the rules, but allow smaller addressable memories via the `reg-subtype` sub-typing rule.

A program type-checks if all instructions type-check under the declared variable types. We require all variables be declared because explicitly typed languages are easier to check.

2.2.5 Operational Semantics

The operational semantics for the Vine IL are shown in Table 2.4. The abstract machine configuration is given by the tuple (Π, Δ, p, i) where Π is the list of instructions, Δ is the variable context, p is the instruction pointer, and i is the current instruction. We write $\Delta' =$

Context

$\Gamma \quad var \to \tau$ Typing context of $(x : \tau)$ pairs where x is of type τ.

Program and Instructions

$$\frac{\forall i \in i^* | (x_i : \tau_i)^* \vdash i : ()}{\cdot \vdash (x_i : \tau_i)^* i^* : ()}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash x : \tau}{\Gamma \vdash x := e : ()} \text{ ASSIGN} \qquad \frac{\Gamma \vdash e : \texttt{reg64\_t}}{\Gamma \vdash \texttt{jmp(}e\texttt{)} : ()} \text{ JMP} \qquad \frac{\Gamma \vdash e : \tau \quad \tau \in \tau_{\text{reg}}}{\Gamma \vdash \texttt{halt(}e\texttt{)} : ()} \text{ HALT}$$

$$\frac{\Gamma \vdash e_1 : \texttt{reg1\_t} \quad \Gamma \vdash e_2 : \texttt{reg64\_t} \quad \Gamma \vdash e_3 : \texttt{reg64\_t}}{\Gamma \vdash \texttt{cjmp(}e_1\texttt{, } e_2\texttt{, } e_3\texttt{)} : ()} \text{ CJMP}$$

$$\frac{\Gamma \vdash e : \texttt{reg1\_t}}{\Gamma \vdash \texttt{assert(}e\texttt{)} : ()} \text{ ASSERT} \qquad \frac{}{\Gamma \vdash \texttt{label } n : ()} \text{ LABEL} \qquad \frac{}{\Gamma \vdash \texttt{special } \mathsf{id}_s : ()} \text{ SPECIAL}$$

Auxiliary Definitions

$$\frac{}{\texttt{reg1\_t <: reg8\_t <: reg16\_t <: reg32\_t <: reg64\_t}} \text{ REG-SUBTYPE}$$

$$\frac{\tau_1 = \texttt{big} \mid \texttt{little} \quad \tau_2 \in \tau_{\text{reg}}}{\text{mem\_compat}(\tau_1, \tau_2)} \qquad \frac{}{\text{mem\_compat}(\texttt{norm}, \texttt{reg8\_t})}$$

$$\frac{k = \texttt{unsigned} \mid \texttt{signed} \quad \tau_1 <: \tau_2}{\text{cast\_compat(k, } \tau_1, \tau_2)} \qquad \frac{k = \texttt{hi} \mid \texttt{low} \quad \tau_2 <: \tau_1}{\text{cast\_compat(k, } \tau_1, \tau_2)}$$

Expressions

$$\frac{\Gamma \vdash e_1 : \texttt{mem\_t(}\tau_1\texttt{, } \tau_2\texttt{)} \quad \text{mem\_compat}(\tau_1, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{load(}e_1\texttt{, } e_2\texttt{, } \tau\texttt{)} : \tau} \text{ LOAD} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ VAR}$$

$$\frac{\Gamma \vdash e_1 : \texttt{mem\_t(}\tau_1\texttt{, } \tau_2\texttt{)} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_4 \quad \text{mem\_compat}(\tau_1, \tau_4)}{\Gamma \vdash \texttt{store(}e_1\texttt{, } e_2\texttt{,}e_3\texttt{)} : \texttt{mem\_t(}\tau_1\texttt{, } \tau_2\texttt{)}} \text{ STORE}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \tau_{\text{reg}} \quad \Diamond_b \notin \ll, \gg, \gg_a}{\Gamma \vdash e_1 \Diamond_b e_2 : \tau} \text{ BINOP}_1 \qquad \frac{\Gamma \vdash e : \tau \quad \tau \in \tau_{\text{reg}}}{\Gamma \vdash \Diamond_u e : \tau} \text{ UNOP}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \tau, \tau_2 \in \tau_{\text{reg}} \quad \Diamond_b \in \ll, \gg, \gg_a}{\Gamma \vdash e_1 \Diamond_b e_2 : \tau} \text{ BINOP}_2$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau, \tau_1 \in \tau_{\text{reg}} \quad \text{cast\_compat}(\textit{cast\_kind}, \tau_1, \tau)}{\Gamma \vdash \texttt{cast(}\textit{cast\_kind}\texttt{, } \tau\texttt{, } e\texttt{)} : \tau} \text{ CAST} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, (x : \tau_1) \vdash e_2 : \tau}{\Gamma \vdash \texttt{let } x \texttt{ := } e_1 \texttt{ in } e_2 : \tau} \text{ LET}$$

Table 2.3: Type-checking rules for Vine.

Contexts

Instruction	Π	$n \mapsto$ *instr*	Maps an instruction address to an instruction.
Variable	Δ	$id \mapsto$ *var*	Maps a variable ID to the variable instance.

Notation

$\Delta \vdash e \Downarrow v$	Expression e evaluates to value v given variable context Δ as given by the expression evaluation rules.
$(\Pi, \Delta, p, i) \rightsquigarrow (\Pi, \Delta', p', i')$	An execution step. Π is the instruction context, p and p' are the pre and post step program counters, i and i' are the pre and post step instructions, and Δ and Δ' are the pre and post step variable contexts.

Instructions

$$\frac{\Delta \vdash e \Downarrow v \quad \Delta' = \Delta[x \leftarrow v] \quad \Pi[p+1] = i}{\Pi, \Delta, p, \texttt{x = } e \rightsquigarrow \Pi, \Delta', p+1, i} \text{ ASSIGN}$$

$$\frac{\Delta \vdash e \Downarrow v \quad \Pi[v] = i}{\Pi, \Delta, p, \texttt{jmp(}e\texttt{)} \rightsquigarrow \Pi, \Delta, v, i} \text{ JMP}$$

$$\frac{\Pi[p+1] = i}{\Pi, \Delta, p, \texttt{label } n \rightsquigarrow \Pi, \Delta, p+1, i} \text{ LABEL}$$

$$\frac{\Delta \vdash e \Downarrow v}{\Pi, \Delta, p, \texttt{halt } e \rightsquigarrow \text{terminate with } v} \text{ HALT}$$

No rule for `special`

$$\frac{p \notin \Pi}{\Pi, \Delta, p, i \rightsquigarrow \text{terminate with } \bot} \text{ NO-INST}$$

Expressions

$$\frac{\Delta \vdash e_1 \Downarrow v_1 \quad \Delta \vdash e_2 \Downarrow v_2 \quad v = v_1 \Diamond_b v_2}{\Delta \vdash e_1 \Diamond_b e_2 \Downarrow v} \text{ BINOP} \qquad \frac{\Delta \vdash e_1 \Downarrow v_1 \quad v = \Diamond_u v_1}{\Delta \vdash e_1 \Downarrow v} \text{ UNOP}$$

$$\frac{}{\Delta \vdash v \Downarrow v} \text{ VALUE} \qquad \frac{\Delta[x] = v}{\Delta \vdash x \Downarrow v} \text{ VAR} \qquad \frac{\Delta \vdash e_1 \Downarrow v_1 \quad \Delta' = \Delta[x \leftarrow v_1] \quad \Delta' \vdash e_2 \Downarrow v}{\Delta \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow v} \text{ LET}$$

$$\frac{\Delta \vdash e_1 \Downarrow v_1 \quad (v_1 : \tau_{\text{mem}}) \quad \Delta \vdash e_2 \Downarrow v_2 \quad v_1[v_2] = v}{\Delta \vdash \texttt{load(}e_1\texttt{, } e_2\texttt{, } \tau\texttt{)} \Downarrow v} \text{ LOAD}$$

$$\frac{\Delta \vdash e_1 \Downarrow v_1 \quad (v_1 : \tau_{\text{mem}}) \quad \Delta \vdash e_2 \Downarrow v_2 \quad \Delta \vdash e_3 \Downarrow v_3 \quad v = v_1[v_2 \leftarrow v_3]}{\Delta \vdash \texttt{store(}e_1\texttt{, } e_2\texttt{, } e_3\texttt{)} \Downarrow v} \text{ STORE}$$

Table 2.4: Operational Semantics for Vine. $e : (\tau)$ denotes that expression e has type τ.

$\Delta[x \leftarrow v]$ to indicate that Δ' is the same as Δ except that variable x is updated with value v. For simplicity, we use Δ both as a scalar and a memory context. When ambiguous, such as in the STORE rule, we indicate the type of the variable in Δ in parentheses. We write $\Pi[p]$ to indicate the instruction given by address p.

The operational semantics can be read as follows. Each step of the execution is associated with a machine configuration $M = (\Pi, \Delta, p, i)$. A transition is given by $M \rightsquigarrow M'$ where the current configuration M matches the left side of $\rightsquigarrow$ in the conclusion (below the horizontal bar), resulting in a state M' to the right. The transformation from M to M' is given by the rule premise (above the horizontal bar).

ASSIGN and LABEL are sequential instructions that carry out the respective operation, then look up and transition to the next sequential instruction $p + 1$. The semantics of LABEL is a no-op: we use labels for jump targets. ASSIGN updates the variable context Δ resulting in a new context Δ'. As mentioned, there is no rule for `special`; any program with a `special` remaining may get stuck.

Control flow is handled by `jmp` (since `cjmp` is a derived form per Chapter 2.2.3). A `jmp` instruction evaluates the jump target e to a value v, then looks up the instruction associated with v. The NO-INST rule terminates the program in the error state when v is not associated with an instruction. For example, consider the case when a program reads in user input at location v, then issues a jump to v. The user input will be decoded as instructions at run-time. However, since the instruction comes from user input, we cannot include it in the analysis. In Vine, we indicate such possibilities by terminating in error.

2.3 The Vine Front-End

The Vine front-end is responsible for lifting up binary code to the Vine IL. In addition, the front-end interfaces with libraries such as the GNU Binary File Descriptor (libbfd) library for parsing the low-level details of binary files.

Lifting up binary code to the IL consists of three steps:

- **Step 1.** First, the binary file is disassembled. Vine currently interfaces with three disassemblers: IDA Pro [35], a commercial disassembler, a research disassembler from Kruegel *et al.* [60] that can disassemble x86 obfuscated code, and a home-brewed linear-sweep disassembler built on top of GNU libopcodes. Interfacing with other disassemblers is straight-forward.
- **Step 2.** The disassembly is passed to VEX, a third-party library which turns assembly instructions into the VEX intermediate language. The VEX IL is part of the

Valgrind dynamic instrumentation tool [78]. The VEX IL is also similar to a RISC-based language. As a result, the lifted IL has only a few instruction types, similar to Vine. However, the VEX IL is insufficient for performing program analysis because it lacks information about the implicit side effects of instructions, e.g., what `eflags` are set by x86 instructions. This step is mainly performed in order to simplify the development of Vine: we let the existing tool take care of reducing assembly instructions to a basic IL, then step 3 exposes all implicit side-effects so that the analysis is faithful.

- **Step 3.** We lift the VEX IL to Vine. The resulting Vine IL is intended to be faithful to the semantics of the disassembled assembly instructions.

Lifted assembly instructions have all of the side-effects explicitly exposed as Vine instructions. As a result, a single typical assembly instruction will be lifted as a sequence of Vine instructions. For example, the `add eax,0x2` x86 instruction in Figure 2.7 is lifted as the following instructions:

```
tmp1 = EAX;
EAX = EAX + 2;
//eflags calculation
CF:reg1_t = (EAX<tmp1);
tmp2 = cast(low, EAX, reg8_t);
PF = (!cast(low,
              ((((tmp2>>7)^(tmp2>>6))^((tmp2>>5)^(tmp2>>4)))^
              (((tmp2>>3)^(tmp2>>2))^((tmp2>>1)^tmp2)))), reg1_t);
AF = (1==(16&(EAX^(tmp1^2))));
ZF = (EAX==0);
SF = (1==(1&(EAX>>31)));
OF = (1==(1&(((tmp1^(2^0xFFFFFFFF))&(tmp1^EAX))>>31)));
```

The lifted Vine instructions explicitly detail all the side-effects of the `add` instruction, including all 6 `eflags` that are updated by the operation. As another example, an instruction with the `rep` prefix (whose semantics are in Figure 2.2) is lifted as a sequence of instructions that form a loop.

In addition to binary files, Vine can also lift an instruction trace to the IL. Conditional branches in a trace are lifted as `assert` statements so that the executed branch is followed. This is done to prevent branching outside the trace to an unknown instruction. The TEMU [5] dynamic analysis tool currently supports generating traces in the Vine trace format.

2.4 The Vine Back-End

New program analyses are written with respect to the Vine IL. Vine provides a library of common analyses and utilities, which can be used as building blocks for more advanced analyses. In this section we describe the utilities and analyses provided in the Vine back-end.

Evaluator. Vine has an evaluator which implements the operational semantics in 2.2.5. The evaluator allows us to execute programs without recompiling the IL back down to assembly. For example, we can test a raised Vine IL for an instruction trace produced by an input by evaluating the IL on that input and verifying we end in the same state.

Graphs. Vine provides routines for building and manipulating control flow graphs (CFG), including a pretty-printer for the graphviz DOT graph language [2]. Vine also provides utilities for building data, control, and program dependence graphs [73].

One issue when constructing graphs of an assembly program is determining the successors of jumps to computed values, called *indirect* jumps. Resolving indirect jumps usually involves a program analysis that require a CFG, e.g., VSA [13]. Thus, there is a potential circular dependency. Note that an indirect jump may potentially go anywhere, including the heap or code that has not been previously disassembled.

Our solution is to designate a special node as a successor of unresolved indirect jump targets in the CFG. We provide this so an analysis that depends on a correct CFG can recognize that we do not know the subsequent state. For example, a data-flow analysis could widen all facts to the lattice bottom. Most normal analyses will first run an indirect jump resolution analysis in order to build a more precise CFG that resolves indirect jumps to a list of possible jump targets. Vine provides one such analysis, called Value Set Analysis [13].

Single Static Assignment. Vine supports conversion to and from single static assignment (SSA) form [73]. SSA form makes writing an analysis easier because every variable is defined statically only once. Note we convert both memory and scalars to SSA form. We convert memories so that an analysis can syntactically distinguish between memories before and after a write operation instead of requiring the analysis itself to maintain similar bookkeeping. For example, in the memory normalization example in Figure 2.2.1, an analysis can syntactically distinguish between the memory state before the write on line 1, the write on line 5, and the read on line 7.

Chopping. Given a source and sink node, a program chop [54] is a graph showing the statements that cause definitions of the source to affect uses of the sink. For example, chopping can be used to restrict subsequent analyses to only a portion of code relevant to a given source and sink instead of the whole program. We describe our implementation of chopping in more detail in 3.4.

Data-flow and Optimizations. Vine provides a generic data-flow engine that works on user-defined lattices. Vine also implements several data-flow analyses. Vine currently implements Simpson's global value numbering [95], constant propagation and folding [73], dead-code elimination [73], live-variable analysis [73], integer range analysis, and VSA [13].

We have implemented value set analysis [13]. Value set analysis is a data-flow analysis that over-approximates the values for each variable at each program point. Value-set analysis can be used to help resolve indirect jumps. It can also be used as an alias analysis. Two memory accesses are potentially aliased if the intersection of their value sets is non-empty.

Optimizations are useful for simplifying or speeding up subsequent analyses. For example, we have found that the time for the decision procedure STP to return a satisfying answer for a query can be cut in half by first using program optimization to simplify the query [21].

C Code Generator. Vine can generate valid C code from the IL. For example, one could use Vine as a rudimentary decompiler by first raising assembly to Vine, then writing it out as valid C. The ability to export to C also provides a way to compile Vine programs: the IL is written as C, then compiled with a C compiler.

The C code generator implements memories in the IL as arrays. A `store` operation is a store on the array, and a `load` is a load from the array. Thus, C-generated code simulates real memory. For example, consider a program that is vulnerable to a buffer overflow attack. It is raised to Vine, then written as C and recompiled. An out-of-bound write on the original program will be simulated in the corresponding C array, but will not lead to a real buffer overflow.

Program Verification. Vine supports program verification in two ways. First, Vine can convert the IL into Dijkstra's Guarded Command Language (GCL), and calculate the weakest precondition with respect to GCL programs [37]. The weakest precondition for

a program with respect to a predicate q is the most general condition such that any input satisfying the condition is guaranteed to terminate (normally) in a state satisfying q. Currently we only support acyclic programs, i.e., we do not support GCL `while`.

Vine also interfaces with decision procedures. Vine can write out expressions (e.g., weakest preconditions) in CVC Lite syntax [1], which is supported by several decision procedures. In addition, Vine interfaces directly with the STP [46] decision procedure through calls from Vine to the STP library.

2.5 Implementation of Vine

The Vine infrastructure is implemented in C++ and OCaml. The front-end lifting is implemented primarily in C++, and consists of about 17,200 lines of code. The back-end is implemented in OCaml, and consists of about 40,000 lines of code. We interface the C++ front-end with the OCaml back-end using OCaml via IDL generated stubs.

The front-end interfaces with Valgrind's VEX [78] to help lift instructions (see Chapter 2.3), GNU BFD for parsing executable objects, and GNU libopcodes for pretty-printing the disassembly. Each disassembled instruction is lifted to the IL, then passed to the back-end.

The implemented Vine IL has several constructors in addition to the instructions in Figure 2.1:

- The Vine IL has a constructor for comments. We use the comment constructor to pretty-print each disassembled instruction before the IL, as well as a place-holder for user-defined comments.
- The Vine IL supports variable scoping via blocks. Vine provides routines to descope Vine programs via α-varying as needed.
- The Vine IL has constructs for qualifying statements and types with user-defined attributes. This is added to help facilitate certain kinds of analysis such as taint-based analysis.

The Vine IL implementation supports `load` and `store` via a single constructor: `mem`. If a `mem` appears on the left-hand side of an assignment, it is a `store`. If it appears on the right, it is a `load`. Thus, the semantics are identical to Figure 2.1. The initial choice to use `mem` instead of `load` and `store` was made because it reduced 2 syntactic elements to a single syntactic element. However, over time, our experience has been that distinguishing loads from stores by the syntactic name is useful for programmers when thinking about

analysis. We have implemented `store` and `load` construct in SSA form, and will replace `mem` with `store` and `load` throughout the IL in the next revision of Vine.

The Vine IL also has an expression constructor called `unknown`. The intention was that if we encountered an instruction we could not properly lift, but we knew which registers the operation used, we could lift it up as an `unknown`. In practice, `unknown` is almost never used, and could be removed from the IL.

2.6 Discussion

2.6.1 Why Design a New Infrastructure?

At a high level, we designed Vine as a new platform because existing platforms are a) defined for higher-level languages and thus not appropriate for binary code analysis, b) designed for orthogonal goals such as binary instrumentation or decompilation, and/or c) unavailable to use for research purposes. As a result, other tools we tried (e.g., DynInst [85] version 5.2, Phoenix [67] April 2008 SDK, IDA Pro [35] version 5, and others) were inadequate, e.g., could not create a correct control flow graph for the 3 line assembly shown in Figure 2.1. Another reason we created Vine is so that we could be sure of the semantics of analysis. Existing platforms tend not to have publicly available formally defined semantics, thus we would not know exactly what we are using.

Formal semantics are important for several practical reasons. We found that defining the semantics of Vine was helpful in catching design bugs, unsupported assumptions, and other errors that could affect many different kinds of analysis. In addition, the semantics are helpful for communicating how Vine works with other researchers. Further, without a specified semantics, it is difficult to show an analysis is correct. For example, in [22] we show our proposed assembly-level alias analysis [22] is correct with respect to the operational semantics of Vine.

2.6.2 Limitations of Vine

Vine is designed to enable program analysis of binary program states. Therefore, analyses that depends upon more than the operational semantics of the instruction fall outside the scope of Vine. For example, creating an analysis of the timing behavior of binary programs falls outside the current scope of Vine.

Some architectures, such as later versions of ARM, are bi-endian where the endian-

ness is specified by a register status flag and can be changed mid-program. Vine currently supports big and little endian, but does not support bi-endian architectures. Adding support is straight-forward by changing `store` and `load` instructions to have an additional field indicating the endianness (which can then be removed by normalizing memory appropriately). If we annotate individual loads and stores, we would need to remove the endianness as part of the memory type declaration in order to prevent the semantics allowing type-incompatible loads and stores on memory.

Vine does not strongly associated an assembly instruction to the corresponding sequence of lifted IL instructions. Currently Vine prefixes a sequence of IL instructions with a comment containing the pretty-printed assembly. However, an analysis itself does not know which IL instructions arose from which assembly. Although users of Vine often seem to want to tie IL instructions to assembly instructions, it is unclear how to accomplish this in flexible way that is also non-intrusive to analyses themselves.

One principle in Vine is to note explicitly in the IL any unhandled instructions during lifting. For each architecture supported, there may be many instructions which Vine cannot lift. Such instructions are noted in the IL as either `special` or `unknown`. For example, Vine does not currently support floating point numbers. The central reason lies in properly defining the semantics of floating point with respect to rounding. This is a problem we plan on tackling in future versions of Vine.

2.6.3 Is Lifting Correct?

Assembly instructions are lifted to Vine in a syntax directed manner. One may view that the Vine IL is a model of the underlying assembly. There is a chance, however, that lifting could produce incorrect IL. Although it is impossible to say that all assembly instructions are correctly lifted, the advantage of Vine's design is that only the lifting process needs to understand the semantics of the original assembly.

We have developed several measures to make sure our IL is correct. First, we have a series of unit tests for IL translations. A unit test executes both the real instruction and the lifted IL in a know pre-state and verifies both terminate in the same post-state. Second, we have performed larger scale tests that verify the executions consisting of millions of instructions produce the same output value on the corresponding IL.

2.6.4 Size of Lifting IL for a Program

A single assembly instruction will typically be translated into several Vine instructions. Thus, the resulting Vine program will have more statements than the corresponding assembly. For example and roughly speaking, x86 assembly instructions are raised to be about 7 Vine instructions: 1 Vine instruction for the direct effect, and 6 for updating processor-specific status flags.

In our experience, the constant-size factor in code size is worth the benefits of simplifying the semantics of assembly. Assembly instructions are designed to be efficient for a computer to execute, not for a human to understand. The IL, on the other hand, is designed to be easy for a human to understand. We have found even experienced assembly-level programmers will comment that they have a hard time keeping track of control and data dependencies since there are few syntactic cues in assembly to help. Vine, on the other hand, obviates all data and control dependencies within the code.

2.7 Related Work

Other Binary Analysis Platforms. Vine is designed to 1) have a formal, well-defined IL, 2) explicitly expose the semantics of complex assembly in terms of the simpler Vine IL, and 3) be easy to re-target. There are several other binary analysis platforms which may have some similarity to Vine, do not fulfill all requirements.

Phoenix is a program analysis environment developed by Microsoft as part of their next generation compiler [67]. One of the Phoenix tools allows Microsoft compiled code to be raised up to a register transfer language (RTL). A RTL is a low-level IR that resembles an architecture-neutral assembly.

Phoenix differs from Vine in several ways. First, Phoenix can only lift code produced by a Microsoft compiler. Second, Phoenix requires debugging information, thus is not a true binary-only analysis platform. Third, Phoenix lifts assembly to a low-level IR that does not expose the semantics of complicated instructions, e.g., register status flags, as part of the IR [68]. Fourth, the semantics of the lifted IR, as well as the lifting semantics and goals, are not well specified [68], and thus not suitable for our research purposes.

The CodeSurfer/x86 platform [14] is a proprietary platform for analyzing x86 programs. At the core of CodeSurfer/x86 is a value-set analysis (VSA) [13]. We have also implemented VSA in Vine. CodeSurfer/x86 was not made available for comparison, thus we do not know whether it supports a well-defined IL, is re-targetable, or what other com-

parable aspects it may share with Vine.

Decompilation. Decompilation is the process of inverting the compilation of a program back to the original source language. Generally, the goal of decompilation is to recover a valid program in the original source language that is semantically equivalent (though need not be exactly the same) as the original source program.

Program analysis is often an important component in the decompilation process. Cifuentes has proposed using data and control flow analyses as part of decompilation [28]. Van Emmerik has shown that analysis on SSA flow graphs is helpful in a variety of tasks such as reconstructing types and resolving indirect jumps [41]. Vine also has SSA; thus it should be straight-forward to implement Van Emmerik's algorithms. Mycroft has proposed type inference for recovering C types during decompilation [74]. Adding type inference is a possible future direction for Vine.

Binary Instrumentation. Binary instrumentation is a technique to insert extra code into a binary that monitors the instrumented program's behavior (e.g., [6, 61, 65, 75, 78, 85, 97]). Instrumentation is performed by inserting jumps in the original binary code to the instrumentation code. The instrumentation code then jumps back to the original code after executing. The instrumentation code must make sure that the execution state is the same before and after the jump in order for the instrumentation to be transparent.

Although many binary instrumentation tools provide a limited amount of program analysis, the end-goal is instrumentation, not facilitating analyses. For example, Pin [65] calculates register liveness information. However, general static analysis is outside the scope of such tools. For instance, instrumentation tools generally do not expose the semantics of all instructions such as register status flag updates.

Other Program Analysis Platforms. Vine shares many of the same goals, such as modularity and ease of writing correct analyses, as other program analysis platforms. For example, CIL [77] and SUIF [4] are both excellent platforms for analyzing C code. However, using higher-level program analysis platforms is inappropriate for binary code because binary code is fundamentally different. While higher-level languages have types, functions, pointers, loops, and local variables, assembly has no types, no functions, one globally addressed memory region, and goto's. The Vine IL and surrounding platform is designed specifically to meet the challenges of faithfully analyzing assembly.

2.8 Conclusion

The Vine binary analysis platform was designed to 1) support writing analyses in a concise and straight-forward fashion, 2) provide abstractions for common assembly-level semantics, and 3) be architecture independent when possible. Vine supports writing analyses concisely by lifting assembly up to a simplified intermediate language. The intermediate language is formally specified, allowing us to know the exact semantics of each lifted instruction. The Vine platform provides utilities for data and control dependency analysis, flow graphs, and other routines that are commonly found in many different kinds of program analyses.

The Vine platform is currently used by 11 different security research projects. In the remainder of this thesis, we will discuss several such projects, and show in detail how Vine helps address two vulnerability analysis problems: the automatic patch-based exploit generation problem and the automatic filter generation problem.

Chapter 3

Algorithms

In this chapter, we first present our adaption of the weakest precondition algorithm proposed by Flanagan and Saxe [44] to binary code. Our adaption avoids the non-traditional semantics proposed by [16], and includes a new proof of correctness that slightly generalizes [44, 62]. We then present our adaption of Jackson *et al.*'s chopping algorithm [54] to our setting. Our adaption addresses the problem that chops produced by [54] were inappropriate for subsequent program analysis.

3.1 Weakest Precondition

3.1.1 Introduction

A common problem in formal verification is to automatically derive the conditions under which a program assertion may hold. For example, we may want to know under what conditions a pointer in a program may be NULL, under what conditions an arithmetic

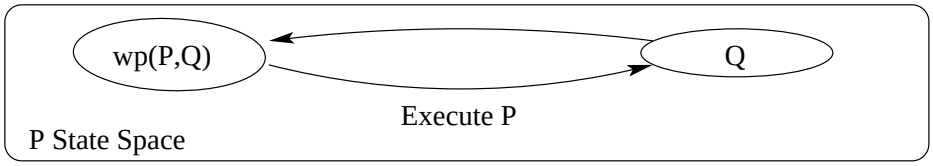
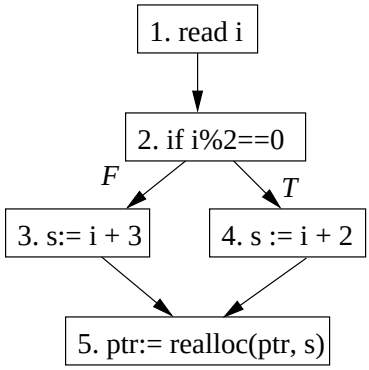


Figure 3.1: The intuition for the weakest precondition $wp(P, Q)$ is that $wp(P, Q)$ describes the program pre-states that, upon executing program P, will guarantee termination in a state satisfying Q.



(a)

```
// Line 1 stores i at address EBP+0xFFFFFFF8
 EAX = load(mem, EBP+0xFFFFFFF8,reg32_t);
 EAX = EAX & 1; // if i % 2 == 0
 temp = EAX & EAX;
 ZF:reg1_t = temp == 0;
 cjmp(ZF,0x8048434,0x804843F);
label 0x8048434;
 EAX = load(mem, EBP+0xFFFFFFF8, reg32_t);
 EAX = EAX + 2;  // s = i + 2
 mem = store(mem, EBP+0xFFFFFFFC, EAX, reg32_t);
 jmp(0x8048448);
label 0x804843f;
 EAX = load(mem, EBP+0xFFFFFFF8, reg32_t);
 EAX = EAX+3;   // s = i+3
 mem = store(mem, EBP+0xFFFFFFFC, EAX, reg32_t);
label 0x8048448; // puts final value for i in EAX
 EAX = load(mem, EBP+0xFFFFFFFC, reg32_t);
```

(b)

Figure 3.2: Our running example, along with the corresponding Vine IL for lines 2-4.

operation may overflow, etc. The weakest precondition is a formal verification technique for answering such questions. Figure 3.1 graphically depicts this intuition. The assertions we may ask of the program are called post-conditions, denoted by Q. Informally, a post-condition is a description of the program state of interest at a particular line of code. The weakest precondition of program P with respect to a post-condition Q, denoted $wp(P, Q)$, is (informally) a description of the program's initial state that lead to an execution satisfying the post-condition Q.

In our running example (reproduced in Figure 3.2), we can use the weakest precondition to derive the conditions in which overflow occurs. The post-condition Q would be $s \leq i$ on line 5 since anytime overflow occurs the sum will be less than the initial input. The weakest precondition $wp(P, s < i)$ is a Boolean formula that is satisfied by all input i such that overflow occurs.

More formally, the state space of a program is the set of all possible assignments of values to variables declared by the program (i.e., the Cartesian product of the domain of all variables). Note that scalars and memory cells are considered variables. For example, the state space for the Vine IL program fragment in Figure 3.2 is $\mathrm{dom}(\texttt{EAX}) \times \mathrm{dom}(\texttt{ZF}) \times \mathrm{dom}(\texttt{mem}) \times \mathrm{dom}(\texttt{EBP}) \times \mathrm{dom}(\texttt{temp})$ assuming that these are the only variables declared. A program can be viewed as an encoding of state transitions. A condition on the state space is a first-order logic Boolean predicate over the state space of the program. Unless explicitly stated otherwise, we assume Boolean formulas are quantifier-free. A post-condition Q is a predicate on the final termination state space.

The *weakest precondition* for a program P and post-condition Q is the weakest possible initial state (i.e., the most general) for the program that will result in termination satisfying Q. More formally, the weakest precondition $wp(P, Q)$ is a Boolean formula over the state space of P which is satisfied by all states (i.e., assignments of values to program variables) such that executing P results in normal termination in a state satisfying Q. [1]

The weakest precondition is calculated from the program itself. The calculation works backward from the program final state. At a high level, we are given the condition Q_j at step j, and we calculate the conditions Q_{j-1} that must be met at step $j-1$ for Q_j to be true at step j.

Intuition using the Running Example. Recall that in the running example (Figure 1.2a), an exploit satisfies the predicate $s < i$ when executing line 5. Let line 5 be the final state (using our chopping algorithm from 3.4). The weakest precondition $wp(P, s < i)$ is a predicate satisfied by inputs which cause overflow. Looking backwards, we see that line 5 can be reached by executing either line 3 or 4. We (for now) calculate the weakest precondition separately for each previous step (i.e., for each path). Consider the former case: a control flow stepping from line 4 to 5. On line 4, `s` is assigned `i+2`. Our calculation substitutes all occurrences of s at step i with its definition `i+2`. Recall that all arithmetic is performed on 32-bit integers, therefore all arithmetic is performed modulo 2^{32}. The condition that must be met at line 4 for $s < i$ to be true at line 5 is therefore $i+2 \mod 2^{32} < 3$. Similarly, for line 3 we get $i+3 \mod 2^{32} < 3$ since on line 3 `s` is defined as `i+3`. At this point, we have calculated the weakest precondition for both lines 3 and 4. At line 2, we have the confluence of the two paths. The weakest precondition for line 2 says the condition calculated at line 3 is true when `i % 2 <> 0`, and true for the condition calculated at line 4 when `i %2 == 0`. More formally, the final weakest precondition is:

$$(i\%2 == 0 \Rightarrow i + 2 \mod 2^{32} < i) \wedge (i\%2 <> 0 \Rightarrow i + 3 \mod 2^{32} < i)$$

Any value of `i` satisfying the above formula will (by the weakest precondition calculation) result in overflow on line 5. In particular, since all integers are assumed 32-bits, the values $i = \{2^{32} - 1, 2^{32} - 2, 2^{32} - 3\}$ satisfy the formula. One can verify in this simple example that this is the complete set of solutions to the calculated predicate, and also the complete set of exploits.

In the corresponding Vine IL, the post-condition would be similar. We would calculate the weakest precondition for `EAX` on line 17 to be less than the initial input location,

[1] A condition c_1 is weaker than c_2 when $c_1 \Rightarrow c_2$.

s ::= lval := e | **assert** e | **assume** e | $s; s$ | $s \square s$ | **skip**

Table 3.1: The guarded command language (GCL) fragment we use.

`mem[EBP + 0xFFFFFFF8]`. The weakest precondition is calculated in a similar manner over the Vine IL program.

Overview. The weakest precondition semantics were first proposed by Dijkstra [37], and have since had a long and successful history in program verification. However, there are two challenges to adapting the weakest precondition to our setting for binary code. First, the traditional approach (due to Dijkstra [37]) is performed on structured Guarded Command Language (GCL). A program is *structured* if the program control flow only contains sequences, selection such as `if-then-else` constructs, and repetition such as `while` statements. Programs that use `goto` or `jump` are not structured. The Vine IL, like assembly, is unstructured. Second, the traditional algorithm (3.2) may result in a predicate that is exponential in the size of the program. Such large predicates are unwieldy to reason about, and in some cases, too large for typical solvers to even syntactically process.

We adapt the algorithm from [44] to calculate the weakest precondition, which will produce a final precondition that is at most $O(n^2)$. Our adaption generalizes the weakest precondition to unstructured code, while avoiding the non-traditional semantics presented in [17]. We then generalize the correctness proof proposed in [62].

In the remainder of this chapter, we first describe the traditional weakest precondition semantics and algorithm. We then describe how we solve the challenges of adapting the weakest precondition to binary code.

3.2 The Traditional Weakest Precondition Semantics

3.2.1 The Guarded Command Language

The weakest precondition is calculated over the guarded command language (GCL), shown in Table 3.1 (we show only the fragment relevant to our work). Statements s in the language include assignments of expressions to L-values (e.g., registers and memory cells), "**assert** e" statements, which checks that expression e is true and fails if it is false, "**assume** e" statements, which adds an assumption that e is true, sequences of statements, and the choice statement "$s_1 \square s_2$" which executes either s_1 or s_2. Note GCL expressions are

$$\frac{}{wp(x := e, Q) : Q[e/x]}\ \text{WP-ASSIGN} \qquad \frac{}{wp(\textbf{assume}\ e, Q) : e \Rightarrow Q}\ \text{WP-ASSUME}$$

$$\frac{}{wp(\textbf{assert}\ e, Q) : e \wedge Q}\ \text{WP-ASSERT} \qquad \frac{wp(s_2, Q) : Q_1 \quad wp(s_1, Q_1) : Q_2}{wp(s_1; s_2, Q) : Q_2}\ \text{WP-SEQ}$$

$$\frac{wp(s_1, Q) : Q_1 \quad wp(s_2, Q) : Q_2}{wp(s_1 \square s_2, Q) : Q_1 \wedge Q_2}\ \text{WP-CHOICE}$$

Table 3.2: The traditional predicate transformers for calculating the weakest pre-condition.

side-effect free.

A program written in GCL may either terminate normally, or it may "go wrong". A program written in GCL terminates normally when none of the assertions fail. A program "goes wrong" if an assertion does fail.

Translating structured programs into the GCL is straight-forward. For example, the if-then-else statement in the pseudo-code in Figure 3.2 is translated in the GCL as:

pseudo-code	GCL
if i%2==0 then s=i+2 else s=i+3;	(**assume** i%2 == 0; s := i+2) □ (**assume** i%2 ≠ 0;s := i+3)

Why not while? In addition to the statement in Table 3.1, the traditional GCL defines a repetition statement **while** e **do** s **od**. We do not include this fragment in our approach since we are concerned with fully automatic methods for calculating the weakest precondition. The weakest precondition calculation for **while** requires that we derive a loop invariant, which cannot always be found through completely automatic means.

In order to allow for completely automatic calculation of the weakest precondition, we follow the same approach as in [44, 62] and assume all loops are unrolled a fixed number of times. We mention in subsequent chapters how unrolling loops may affect specific results, as well as how the fixed number may be chosen, in specific usage scenarios in the following chapters.

3.2.2 The Traditional WP Algorithm

The weakest precondition $wp(P, Q)$ is calculated from a program in GCL. The calculation is performed by defining a predicate transformer for each type of statement in the language.

The predicate transformers transform the post-condition Q into the weakest precondition by considering each statement in the program.

Table 3.2 shows the predicate transformers for the GCL (Table 3.1). The rules are written as an inductive calculation. The conclusion (below the bar) is written as $s : Q$, where weakest precondition statements matching s will produce weakest precondition Q. The condition Q is calculated using the rule premise (above the bar).

Example. The derivation of the the weakest precondition using the rules from Table 3.2 for our running example is:

$$\frac{\dfrac{(Q_1 \doteq \text{(i\%2) == 0} \Rightarrow \text{(i+2 } \bmod 2^{32}) \leq \text{i)}}{wp(\textbf{assume } \text{i\%2==0,(i+2 } \bmod 2^{32}) \leq \text{i}) : Q_1}}{wp(\textbf{assume } \text{i\%2 == 0; s := i+2, s} \leq \text{i}) : Q_1} \quad \frac{\dfrac{(Q_2 \doteq \text{(i\%2)} \neq 0 \Rightarrow \text{(i+3 } \bmod 2^{32}) \leq \text{i)}}{wp(\textbf{assume } \text{i\%2}\neq 0, \text{ (i+3 } \bmod 2^{32}) \leq \text{i}) : Q_2}}{wp(\textbf{assume } \text{i\%2} \neq 0; \text{ s := i+3, s} \leq \text{i}) : Q_2}$$

$$wp((\textbf{assume } \text{i\%2 == 0; s := i+2}) \square (\textbf{assume } \text{i\%2} \neq 0\text{;s := i+3}), \text{s} \leq \text{i}) : Q_1 \wedge Q_2$$

The final calculated condition $Q_1 \wedge Q_2$ is:

$$\text{((i\%2) == 0} \Rightarrow \text{(i+2 } \%2^{32}) \leq \text{i)} \wedge \text{((i\%2)} \neq 0 \Rightarrow \text{(i+3 } \%2^{32}) \leq \text{i)}$$

which after logical and arithmetic simplification is $i + 3 \bmod 2^{32} \leq i$. As desired, the set inputs that cause overflow — $\{2^{32} - 3, 2^{32} - 2, and 2^{31} - 1\}$, are the only values of i that satisfy the weakest precondition.

3.3 Efficient Weakest Preconditions

3.3.1 Converting Vine to the GCL

The first thing we must do in order to calculate the weakest precondition is write Vine programs in the GCL. Since Vine expressions are side-effect free, all Vine expressions are also GCL expressions. Vine assignment instructions become GCL assignments. Vine **assert** instructions becomes GCL **assert** statements. However, there is no straight-forward conversion for jumps, since the GCL is a structured programming language. Previously, this led to new predicate transformers in order to adapt the weakest precondition to unstructured code [16].

Our approach is to convert the unstructured control flow in Vine into the GCL. As a result, we do not need to create new predicate transformers. Our algorithm for converting to the GCL takes as input a Vine IL program CFG G and an integer k. Recall from 3.2.1 that our weakest precondition semantics assume a loop-free program (similar to [16, 44, 62]). We first unroll all loops in the CFG k times, producing an acyclic CFG G'.

The main idea behind our approach is that an acyclic CFG G' can be rewritten as a structured program. The CFG of an acyclic program has only two structures: sequences and (2-way) branches. We observe that all nodes have either 0, 1, or 2 successors because all control flow is either a jump to a single successor, a conditional jump to two possible successors, or a terminal node. Thus, a node with two successors corresponds to a GCL choice ($\Box$), a node with a single successor is part of a sequence, and a node with zero successors is the CFG canonical exit node and corresponds to a final **skip** instruction in the GCL program. For example, a structural analysis of the Vine IL in Figure 3.2 would recover the if-the-else structure shown in the CFG in Figure 3.2, and create the GCL program:

(**assume** (i%2) ==0; s := i+2) $\Box$ (**assume** $\neg$ (i%2) == 0; s := i+3;); ptr := realloc(ptr,s);

Algorithm 1 Our algorithm for creating a GCL program from an acyclic CFG.

1: **for all** v in a topological order rooted in the CFG exit of G^T **do**
2: **if** |succ(v)| = 0 **then**
3: g := to_gcl(v)
4: **else if** |succ(v)| = 1 **then**
5: g := M(succ(v)); to_gcl(v) // A sequence
6: **else**
7: v_1, v_2 := succ(v) // v is a conditional jump with targets v_1 and v_2
8: (g_{suffix}, g_1, g_2) := split_suffix(M(v_1), M(v_2))
9: e = branch_predicate(v, v_1) // e is the conditional jump guard
 // Result is choice, with statements in confluence of branch as common suffix
10: s := (**assume** e; g_1) $\Box$ (**assume** $\neg e$; g_2); g_{suffix}
11: **end if**
12: M[v] := g
13: **end for**

Our algorithm for converting a CFG to a GCL program is shown as Algorithm 1. The algorithm considers three cases for a vertex v: a) v has zero successors, corresponding to the canonical exit node, b) v has one successor v', thus corresponds to a sequence of instructions, and c) v has two successors, corresponding to a conditional jump.

The algorithm uses a few helper functions:

- succ(v): the successor list of v in the CFG.
- to_gcl(v): Translates a sequential statement in Vine at v to the corresponding sequential GCL statement g.
- M: A map from a vertex v to the GCL statement for all executions starting at v.
- split_suffix(g_1, g_2): returns the tuple $(s_{\text{suffix}}, g_1 - g_{\text{suffix}}, g_2 - g_{\text{suffix}})$, where g_{suffix} is the common GCL statement suffix for g_1 and g_2, and $g_i - g_{\text{suffix}}$ is statement i with the suffix g_{suffix} removed. For example:

```
let g1= assert i%2 ≠ 0;s:=i+3; ptr:=realloc(s)
let g2= assert i%2 == 0; s:=i+2; ptr:=realloc(s)
(ptr:=realloc(s),(assert i%2 ≠ 0;s:=i+3),(assert i%2 == 0; s:=i+2))
        = common_suffix(g1, g2)
```

- condition_of(v, v_1), which returns the branch predicate for the edge (v, v_1). In our semantics, $\neg e$ is then the guard for (v, v_2).

Example. Consider running Algorithm 1 on Figure 3.2 where the topological sort order is {5,4,3,2}. Line 12 will always map the current vertex to the statement g. Our run will set g as follows:

1. $v = 5$. g is set to to_gcl(ptr:=realloc(s))= ptr:=realloc(s) by line 3.
2. $v = 4$. Line 5 executes, where succ(4) = 5, and M[5] = ptr:=realloc(s) and to_gcl returns s:=i+2. Therefore, g is the GCL sequence s:=i+2; ptr:=realloc(s).
3. $v = 3$. Line 5 executes, where again we look up M[5], to_gcl returns s:=i+3, and g is set to the GCL sequence s:=i+3;ptr=realloc(s);.
4. $v = 2$. In this case, line 2 executes where $v_1 = 3$ and $v_2 = 4$. We call split_suffix on the values of g from when $v = 3$ and $v = 4$. g_{suffix} is the common suffix ptr:=realloc(s), g_1 is s:=i+3, and g_2 is s:=i+2. The branch predicate returned on line 9 is i%2==0. The complete GCL statement is:

```
(assume (i%2) ==0; s := i+2) □
(assume ¬ (i%2) == 0; s := i+3;); ptr := realloc(ptr,s);
```

Analysis. Let $G = (V, E)$. Computing the transpose of a graph can be done in time $O(|E|)$. The running time of the algorithm includes a topological sort, which can be done

in $O(|V| + |E|)$. Because each node is visited at most once, the running time is linear in the size of the graph.

For correctness, the important case to consider is for a node with two successors. Suppose v has successors v_1 and v_2, and a subsequent join point v_j. v_j must exist because we have a canonical exit, which itself is not a branch. Note that because vertices are visited in topological order of the graph transpose, all successors of a node will be visited before the node itself. Therefore, we know v_j was visited before v_1 or v_2, and v_1 and v_2 will have as a common suffix the GCL program from v_j onward. Line 8 splits the GCL for v_1 and v_2 at the suffix point, and then line 10 builds an appropriate GCL statement for the branch-join behavior.

3.3.2 Efficiently Calculating the Weakest Precondition

Calculating the weakest precondition using the traditional algorithm from Table 3.2 may result in a formula exponential in the size of the program. In this section we explain the causes and solutions for exponential blowup, as well as our correctness proof. The final algorithm will calculate the weakest precondition for GCL in size $O(n^2)$ where n is the number of statements in the original GCL program.

3.3.2.1 Causes of Exponential Blowup

The predicate transformer WP-ASSIGN may cause exponential blowup in the weakest precondition. To see why, consider the calculation:

wp(b:= a+a; c:= b+b;d:= c+c,d<5)

The WP-ASSIGN rule performs substitution for each sequence. Initially, d is replaced with $c + c$ per WP-ASSIGN. Each occurrence of c is replaced with $b + b$, and then each occurrence of b is replaced with $a + a$. The final weakest precondition predicate, $a + a + a + a + a + a < 5$, contains 8 a's.

The second source of exponential formula growth is with the WP-CHOICE rule. As can be seen from the rule, the post-condition Q is duplicated by the post-condition on each branch. Duplicating the post-condition potentially doubles the size of the weakest precondition at each branch point.

3.3.2.2 Efficiently Calculating the Weakest Precondition

We can prevent exponential blowup due to WP-ASSIGN by replacing each assignment `x = e` with an equality statement **assume** `x = e`. The rule for **assume** does not cause exponential blowup because substitution is not performed. However, in order for this technique to work, we must address the issue that the semantics of equality are different than assignment. Equality is a mathematical assertion, while the semantics of assignment are an update. In particular, assignment allows for multiple updates to the same variable, while equality binds a name to an expression. If every variable is assigned only once, then assignments are the same as equality.

We can replace assignments with **assume** if we first ensure each variable is only assigned once. Flanagan and Saxe first proposed this solution, and called it *passification* [44]. Passification transforms the program into a semantically equivalent version in which all program variables are assigned only once. This condition is easily met by (acyclic) programs by converting the program into SSA form, then pushing ϕ assignments up into the respective branches. [2] Our running example is already passified since for any execution s is only assigned once.

A passified program is at most $O(n^2)$ where n is the number of statements in the original program. In practice, the resulting passified program will be linear in size since SSA is almost always linear in practice [44]. As we will see, the size of the passified program is the size of the weakest precondition for (acyclic) programs.

In order to prevent blowup due to WP-CHOICE, [44, 62] proposed to make use of the *weakest liberal precondition* (wlp). The weakest liberal precondition is the weakest condition which guarantees that the post-condition is met if the program terminates, i.e., the same as the weakest precondition, except the program may not terminate. The inference rules for the weakest liberal precondition are the same as for the weakest precondition except for:

$$\frac{}{wlp(\textbf{assert } E, Q) : E \Rightarrow Q} \text{ WLP-ASSERT}$$

The relationship between the weakest precondition and weakest liberal precondition is:

$$wp(P, Q) \Leftrightarrow wp(P, true) \wedge wlp(P, Q) \tag{3.1}$$

Equation 3.1 (due to Dijkstra [37]) can be read as is that the weakest precondition for a program to terminate in a state satisfying Q ($wp(P, Q)$) is the same as for *if* the program terminates it satisfies Q (the $wlp(P, Q)$ term) *and* it terminates (the $wp(P, true)$ term).

[2] [44] actually propose to use dynamic single assignment. For our purposes, SSA is sufficient.

$$\frac{wp_t(P) : Q_1 \quad wlp_f(P) : Q_2}{wp(P, Q) : Q_1 \wedge (Q_2 \vee Q)}\ \text{WP}$$

$$\frac{wp_t(s_2) : Q_1 \quad wp_t(s_1) : Q_2 \quad wlp_f(s_1) : Q_3}{wp_t(s_1; s_2) : Q_1 \wedge (Q_2 \vee Q_3)}\ \text{WP}_\text{T}\text{-SEQ}$$

$$\frac{wlp_f(s_1) : Q_1 \quad wlp_f(s_2) : Q_2}{wlp_f(s_1; s_2) : Q_1 \vee Q_2}\ \text{WLP}_\text{F}\text{-SEQ}$$

$$\frac{wp_t(s_1) : Q_1 \quad wp_t(s_2) : Q_2}{wp_t(s_1 \square s_2) : Q_1 \wedge Q_2}\ \text{WP}_\text{T}\text{-CHOICE}$$

$$\frac{wlp_f(s_1) : Q_1 \quad wlp_f(s_2) : Q_2}{wlp_f(s_1 \square s_2) : Q_1 \wedge Q_2}\ \text{WLP}_\text{F}\text{-CHOICE}$$

$$\frac{}{wp_t(\text{assert } e) : e}\ \text{WP}_\text{T}\text{-ASSERT} \qquad \frac{}{wlp_f(\text{assert } e) : \neg e}\ \text{WLP}_\text{T}\text{-ASSERT}$$

$$\frac{}{wp_t(\text{assume } e) : \text{true}}\ \text{WP}_\text{T}\text{-ASSUME} \qquad \frac{}{wlp_f(\text{assume } e) : \neg e}\ \text{WP}_\text{T}\text{-ASSUME}$$

Table 3.3: The Final Rules for Efficient Weakest Preconditions.

An essential insight is that passified programs do not change state, i.e., assignment-free programs are pure expressions. Therefore, if a passified program starts in a state satisfying Q and nothing goes wrong, it will end in a state satisfying Q. This is expressed in the following identity which is true for all assignment-free programs:

$$wlp(P, Q) \Leftrightarrow wlp(P, false) \vee Q \tag{3.2}$$

The reason this identity is important is that the post-condition Q does not appear in the weakest precondition calculation, thus is not duplicated along branches in WP-CHOICE.

Putting together equation 3.1 and equation 3.2, we get:

$$wp(P, Q) \Leftrightarrow wp(P, \text{true}) \wedge (wlp(P, false) \vee Q) \tag{3.3}$$

The Final Algorithm. Table 3.3 shows the final rules for efficient weakest precondition calculation given equations 3.2 and 3.3. One thing we must be careful of is not to calculate the weakest liberal precondition multiple times, e.g., once as part of WP and once as part of WP_T-SEQ. The final algorithm for program P is:

1. Passify P by converting to SSA and back, pushing ϕ statements up along the respective branches. Let P' be the passified version of P.
2. Compute $\texttt{let}\ \ Q_i = wlp_f(P'_i)$ for each statement sequence in P'.
3. Compute $Q_{P'} = wp_t(P')$, using Q_i from above for each wlp_f sub-computation.
4. The final weakest precondition is $Q_{P'} \wedge (Q_0 \vee Q)$ where Q_0 is $wlp_f(P')$ from above.

The final weakest precondition is at most $O(n^2)$ in the size of the original program, where n is the number of statements.

3.3.2.3 Proof of Correctness

Proof of Equation 3.2. The argument above holds for assignment-free, acyclic programs if we can prove Equation 3.2 holds (since Equation 3.1 follows from the definition of wp and wlp [37]). Further, we claim the total weakest precondition predicate be at most $O(n^2)$ where n is the number of statements in the original, un-passified, loop-free program.

We prove the following generalization of Equation 3.2 of any two predicate Q_a and Q_b:

Lemma 1. For all assignment-free acyclic programs P, $\forall Q_a, Q_b | wlp(P, Q_a) \vee Q_b : Q \Leftrightarrow wlp(P, Q_b) \vee Q_a : Q$.

Equation 3.2 generalizes Lemma 1 (thus generalizing [62]) with $Q = Q_a$ and $Q_b = false$, i.e., $wlp(P, Q) \equiv wlp(P, Q) \vee false \Leftrightarrow wlp(P, false) \vee Q$.

Proof. We provide the proof for the forward direction. The backward direction is similar. Our proof is by induction on the derivation of $\mathcal{D} = wlp(P, Q_a) \vee Q_b : Q$. One slight problem is our derivation rules for wlp do not provide for logical connectives, e.g., the $\vee Q_b$. We show the augmented rules here.

Case: $\mathcal{D} = \dfrac{}{wlp(\textbf{assume}\ \text{E}, Q_a) \vee Q_b : (E \Rightarrow Q_a) \vee Q_b}$
where $P = \textbf{assume}\ \text{E}$ and $Q = (E \Rightarrow Q_a) \vee Q_b$. We show that $wlp(\textbf{assume}\ \text{E}, Q_b) \vee Q_a : Q$.

$(E \Rightarrow Q_a) \vee Q_b$	given.
$wlp(\textbf{assume}\ \text{E}, Q_b) \vee Q_a : (E \Rightarrow Q_b) \vee Q_a$	by rule.
$(E \Rightarrow Q_a) \vee Q_b \Leftrightarrow (E \Rightarrow Q_b) \vee Q_a$	by truth table.

Case: $\mathcal{D} = \dfrac{}{wlp(\textbf{assert}\ \mathrm{E}, Q_a) \vee Q_b : (E \Rightarrow Q_a) \vee Q_b}$.

Symmetric to above.

Case: $\mathcal{D} = \dfrac{wlp(s_1, Q_a) \vee Q_b : Q_1 \quad wlp(s_2, Q_a) \vee Q_b : Q_2}{wlp(s_1 \square s_2, Q_a) \vee Q_b : Q_1 \wedge Q_2}$

where $P = s_1 \square s_2$ and $Q = Q_1 \wedge Q_2$. We show $wlp(s_1 \square s_2, Q_b) \vee Q_a : Q_1 \wedge Q_2$.

$wlp(s_1, Q_b) \vee Q_a : Q_1$	by inductive hypothesis (IH).
$wlp(s_2, Q_b) \vee Q_a : Q_2$	by IH.
$wlp(s_1 \square s_2, Q_b) \vee Q_a : Q_1 \wedge Q_2$	by rule.

Case: $\mathcal{D} = \dfrac{wlp(s_1, Q_a) \vee Q_b : Q_1 \quad wlp(s_2, Q_1) : Q}{wlp(s_1; s_2, Q_a) \vee Q_b : Q}$

where $P = s_1; s_2$. We show $wlp(s_1; s_2, Q_a) \vee Q_b : Q$

$wlp(s_2, Q_a) \vee Q_b : Q_1$	by IH.
$wlp(s_1, Q_1) : Q$	given.
$wlp(s_1; s_2, Q_a) \vee Q_b : Q$	by rule.

□

This proof shows that our desired property naturally follows directly from the algorithm itself, only appealing to logical equivalence once for **assume** and once for **assert**.

$\mathbf{O(n^2)}$ Size Proof. We need to first establish two lemmas about the size of wlp_f and wp_t computations. Let $|P|$ denote the size of P in the number of statements, and $|Q|$ be the number of non-constants (since constants can be immediately removed) in Q.

Lemma 2. $|wlp_f(P)| \leq |P|$

Proof. By structural induction on the derivation $\mathcal{D}$ of wlp_f.

Case: $\mathcal{D} = \dfrac{wlp_f(s_1) : Q_1 \quad wlp_f(s_2) : Q_2}{wlp_f(s_1 \square s_2) : Q_1 \wedge Q_2}$

$\lvert Q_1 \rvert \leq \lvert s_1 \rvert$	by IH.
$\lvert Q_2 \rvert \leq \lvert s_2 \rvert$	by IH.
$\lvert Q_1 \wedge Q_2 \rvert \leq \lvert s_1 \rvert + \lvert s_2 \rvert$	by above.

Case: $\mathcal{D} = \dfrac{wlp_f(s_1) : Q_1 \quad wlp_f(s_2) : Q_2}{wlp_f(s_1; s_2) : Q_1 \vee Q_2}$

Symmetric to above.

Case: $\mathcal{D} = \overline{wlp_f(\text{assert } e) : \neg e}$ Immediate from rule.

Case: $\mathcal{D} = \overline{wlp_f(\text{assume } e) : \neg e}$ Immediate from rule.

□

Lemma 3. Given a variable $Q_i = wlp_f(P_i)$ for each statement of the s sub-sequence in P, $|wp_t(P)| \leq |P|$

Proof. The proof is symmetric to Lemma 2 where we apply the IH on each sub-computation. The only change is we must make use of Q_i when called to compute wlp_f:

Case: $\mathcal{D} = \dfrac{wp_t(s_2) : Q_1 \quad wp_t(s_1) : Q_2 \quad wlp_f(s_1) : Q_3}{wp_t(s_1; s_2) : Q_1 \wedge (Q_2 \vee Q_3)}$

$\|Q_1\| \leq \|s_1\|$	by inductive hypothesis (IH).
$\|Q_2\| \leq \|s_2\|$	by IH.
$\|Q_3\| = 0$	by using the appropriate let-bound variable.
$\|Q_1 \wedge (Q_2 \vee Q_3)\| \leq \|s_1\| + \|s_2\|$	by above.

□

Given Lemma 3 and 2, we can prove the desired result:

Theorem 1. The size of the weakest precondition is $O(n^2) + |Q|$ for post-condition Q and program P with n statements using the above algorithm.

Proof. In step 1, we passify the program P. Passifying a program results in a new GCL program P' that is $O(n^2)$ the size of P.

In step 2, we calculate $Q_i = wlp_f(P'_i)$ for all sub-statements of the passified P'. The total size is the sum of all individual Q_i. The length of each sub-statement is monotonically decreasing. Lemma 2 tells us each Q_i is therefore also monotonically decreasing. Since there are n sub-statements of length $0...n$, we have $(\Sigma_i |Q_i|) \leq 2|P'| - 2$.

In step 3, we calculate $Q_{P'} = wp_t(P')$. Lemma 3 tells us $|Q_{P'}| \leq |P'|$ because we can use the Q_i from step 2 to satisfy the premise of the lemma.

Therefore, $|Q_{P'} \wedge (Q_0 \vee Q)| \leq 2|P'| - 2 + |P'| + |Q|$, which is $O(n^2) + |Q|$. □

3.4 Strongly Connected Component (SCC)-Based Chopping

3.4.1 Overview

We are often only interested in a subset of possible execution paths. For efficiency, we would like to restrict program analyses to only those paths of interest. For example, when analyzing an input-validation vulnerability, we may only care about execution paths from where user input is read in to a vulnerable line of code. An analysis based solely upon only those paths may run significantly faster than a comparable whole-program analysis.

We present *SCC-based chopping*, which will create a sub-program given a source and a sink program location. The resulting sub-program is called the chop, and contains all execution paths starting at source that may influence the value computed at sink. Subsequent analysis performed on just the chop will be correct with respect to the overall program. Since the chop is potentially smaller than the whole program, the subsequent analysis will be potentially faster.

SCC-based chopping is different than previous work in chopping [54, 88] in that the resulting chop is appropriate for subsequent analysis such as computing the weakest precondition. Previously proposed chopping techniques (e.g., [54, 88]) were designed primarily for understanding and debugging program behavior. As a result, previous techniques did not produce a chop that was suitable for subsequent program analyses such as weakest precondition and other techniques.

3.4.2 Background

A *program slice* [105] for a given sink statement s_k is obtained by deleting all statements that are irrelevant to the computation at s_k. A statement s_i is irrelevant to s_k when s_k is not data or control dependent on s_i. A *data dependence* exists between two nodes u and v in the control flow graph if u computes a value that is referenced by v. For example, in Figure 3.3, statement 3 and statement 5 are both data dependent upon statement 1 since they use a value computed on 1. Statement 1 has no data dependencies.

A node y is *control dependent* on a node x if x branches to u and v, and from u there is a path to the canonical CFG exit that avoids y, and from v every path to exit executes y. Figure 3.4 shows this relationship. Standard compiler textbooks (e.g., [10, 11]) provide efficient algorithms for calculating both control and data dependencies.

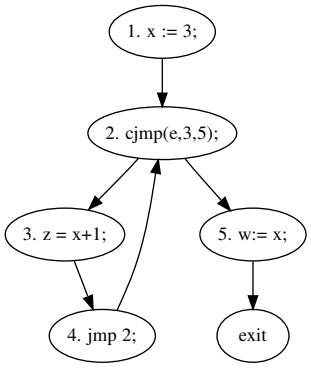


Figure 3.3: An example control flow graph where the weakest precondition would be incorrect based upon a chop from [54] with source as 1 and sink as 5.

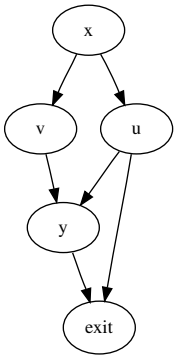


Figure 3.4: y is control dependent upon x.

Chopping was first proposed by Jackson and Rollins [54] as a generalization of program slicing. A program chop is defined with respect to a given source s_{source} and sink s_{sink} statement. The chop is the intersection of the forward slice of all statements that are dependent upon s_{source} and backward slice of statements that are needed to compute s_{sink}.

Slicing and chopping can be performed on the program dependence graph (PDG) [54, 88]. A PDG consists of all labeled control and data dependencies. At a high level, chopping and slicing algorithms walk the PDG, marking each node visited. Any node not visited is removed from the resulting chop [54, 88] or slice [105].

Chopping based upon the PDG is inappropriate for our uses because it may remove control flow that is relevant for additional program analysis, such as program verification. Consider the example CFG shown in Figure 3.3, and suppose we want to compute the chop with respect to line 1 and 5. Using existing chopping algorithms, the chop of Figure 3.3 will result in:

```
1. x :=3;
5. w := x;
```

The chop will not contain the loop with instructions 2-4 since there is no control or data-dependency with these statements. However, subsequent program analyses may need to know about the removed instructions. For example, the weakest precondition will depend upon whether the removed loop terminates, thus calculating the weakest precondition on the above code may be incorrect.

In general, chopping using previous algorithms may remove loop-based control flow that is important to subsequent analyses. We next propose a variation of chopping, which we call SCC-based chopping, that addresses this issue.

3.4.3 Our Chopping Algorithm: SCC-Based Chopping

Our approach to chopping produces a sub-graph that contains all possible control-flow paths between a given source and sink, not just control dependencies. Our chopping algorithm given program P, source s_{source}, and sink s_{sink} consists of the following steps:

1. Build a control flow graph G for P. We describe the case where each CFG node corresponds to one statement. It is straight-forward to extend the subsequent steps to basic blocks.
2. Create a temporary edge $(s_{\text{sink}}, s_{\text{source}})$ if one does not already exist. The back-edge ensures that if s_{sink} is reachable from s_{source}, then there is also a cycle.

3. Compute the strongly connected component (SCC) containing s_{sink} and s_{source}. If there is no such SCC, then there is no path between the two nodes (due to step 2), and the chop is empty.
4. Let G_c be the chop graph. G_c is built by iterating over each vertex v in G. Each vertex v that is in the SCC is in G_c.
   (a) If $(v_1, v_2) \in G$, and v_1 and v_2 are in G_c, then add edge (v_1, v_2) to G_c.
   (b) If $(v_1, v_2) \in G$, and $v_1 \in G_c$ but $v_2 \notin G_c$, then we must fix up the statement at v_1 so it doesn't jump to v_2. Note v_1 must be a conditional branch: if v_1 was sequential, then v_2 is always in the same SCC as v_1. We fix up the statement at v_1 by asserting that the branch predicate (v_1, v_2) is false.
5. Perform dead-code elimination on G_c. We perform dead-code elimination that is initialized by marking the value computed at s_{sink} as live-out. We also mark values used by `assert` as live.

By basing our chopping on the SCC, we know that after step 4 all statements reachable from s_{source} to s_{sink} are in G_c. Thus, our SCC-based chop includes all statements that may execute between s_{source} to s_{sink}, not just those with control dependencies.

The final step, dead-code elimination, is the same as walking the data-dependencies in the PDG for G_c. As normal, we always mark variables used in `assert` live because the semantics of `assert` for program verification are to establish an invariant on the state space. All other dead-code can be removed since the values computed are never used, thus irrelevant.

3.4.4 Discussion

In Vine, one explicit goal is to provide general program analysis capabilities. Thus, we developed SCC-based chopping since it is more general in our context. However, [54] may produce a smaller chop for acyclic graphs. We have also implemented the intra-procedural variant of [54] in Vine. This version is only called when the input CFG to chop is acyclic.

Chops and slices are often criticized for being large or imprecise. Traditionally, large or imprecise chops are bad because the chop is used to explain code. In our case, we are using chopping to make subsequent analyses more efficient, not as an explanation. Thus, even large chops are not an issue as the chop can only improve subsequent efficiency, and never hurt it.

Part II

Vulnerability Analysis and Defense

Chapter 4

Vulnerability Analysis and Defense

4.1 Input Validation Vulnerabilities

In this thesis we focus on *input validation vulnerabilities*. Intuitively, an input validation vulnerability is characterized by the ability to decide at each step of the execution whether or not the program is in a safe state. We call such vulnerabilities *input validation vulnerabilities* because the programmer should have included checks that prevent transitions to an unsafe state. Figure 4.1 depicts this relationship where the input domain is larger than the set of inputs that are safe.

The class of input validation vulnerabilities includes typical well-known vulnerability classes such as buffer overflows, format string vulnerabilities, and integer overflow vulnerabilities. More generally, input validation vulnerabilities include any vulnerability whose exploits can be detected via a run-time check. For example, if exploits can be detectable by popular mechanisms such as taint-based analysis, firewalls, VM-based logging, non-executable bits on memory pages, etc., then the underlying vulnerability is an input validation vulnerability. Vulnerabilities whose exploits cannot be detected via a run-time monitor fall outside the scope of input validation vulnerabilities. For example,

Figure 4.1: An input validation vulnerability occurs when the domain of inputs for a program (white) is a super-set of safe inputs (black).

timing vulnerabilities against cryptographic algorithms are usually characterized by how long it takes to execute steps, not whether or not the program at each execution step is in a safe state.

Note programs written in "safe" languages may also contain input validation vulnerabilities. While it is true that safe languages and practices can help prevent vulnerabilities, it is possible to get input validation wrong even in a safe language .

In the remainder of this chapter we define input validation vulnerabilities more formally. We do so in order to be unambiguous. Although vulnerability research has a long history, there is little agreement or standardization of the definition of a vulnerability, exploit, etc. Our formalization below attempts to address this issue within the scope of our work.

4.2 Background: Safety Policies Enforceable by an Execution Monitor

We define input validation vulnerabilities partially in terms of an execution monitor (EM) enforceable safety property. We adapt notation from Schneider [91] to define safety policies which are enforceable by an execution monitor. Let ψ denote the universe of all finite and infinite execution sequences. How execution sequences are represented is unimportant for formalization purposes: common representations include program states (which we use in this thesis), system steps, and atomic action sequences. Let σ and τ represent a single finite or infinite execution, and let $\sigma[..k]$ represent a finite execution involving the first k steps, and $\sigma[k]$ represent the k'th step. Let $\sigma[..k]\tau$ represent a finite execution $\sigma[..k]$ followed by τ. Let $P(i) : \sigma$ denote the execution of P on input i results in execution σ.

An EM-enforceable policy defines what constitutes a bad execution (i.e., an exploit) by a Boolean predicate χ on the space of program executions. A target program P defines $\Sigma_P \subseteq \psi$ corresponding to possible infinite and finite executions of P. A program is safe with respect to the policy if $\forall \sigma \in \Sigma_P : \chi(\sigma) = \text{true}$.

EM-enforceable policies are policies which can be enforced considering only a single execution of a program. More formally:

$$\chi(\sigma) = \mathit{false} \Rightarrow (\exists k : (\forall \tau \in \psi : \chi(\sigma[..k]\tau) = \mathit{false})) \tag{4.1}$$

This definition means that at each step of execution of a program, we can determine whether an input violates the policy by only considering the execution thus far. This

notion matches with our previous informal idea that at each step of the execution we can determine whether the program is in a safe state or not.

4.3 Input Validation Vulnerability Definitions and Formalism

Definition 4.3.1. An *input validation vulnerability* is given by the tuple (Φ, v_p) where Φ is a safety policy enforceable by an execution monitor, and v_p is a uniquely labeled program point. We call Φ the *vulnerability condition*, and v_p the *vulnerability point*.

In definition 4.3.1, Φ specifies what it means for a program execution to be safe. However, Φ alone is insufficient for specifying a single vulnerability, since there may be many places in the program where Φ may be violated. For example, Φ may disallow buffer overflows, but there may be many buffer overflows in a single program. In order to isolate a single vulnerability, we also include the specific program location. In our binary-centric setting, v_p is an instruction address, and Φ is a security policy that can be checked at every step of the execution.

Note that our formulation allows for vulnerability composition since EM policies are composable, while non-EM policies may not be [91]. Given two policies χ_1 and χ_2, the composition of the policy is given by the conjunction $\chi_{1\wedge 2} \doteq \chi_1 \wedge \chi_2$. A violation of the composition is defined as:

$$\chi_{1\wedge 2}(\sigma) = \text{false} \doteq \chi_1(\sigma) = \text{false} \vee \chi_2(\sigma) = \text{false} \tag{4.2}$$

Definition 4.3.2. An *exploit* for an input validation vulnerability (Φ, v_p) is an input x such that: $P(x) : \sigma \wedge \Phi(\sigma[v_p]) = \text{false}$. The execution path σ is an *exploitable execution path*.

Note that there may be many different exploits for a single vulnerability. For example, it is not uncommon for there to be exploits that crash the program, steal passwords, and hijack control for a single vulnerability.

We also want to describe the set of all exploits, i.e., the set of unsafe inputs. We call this the *vulnerability language* $\mathcal{V}_{(\Phi,v_p)}$.

Definition 4.3.3. The *vulnerability language* $\mathcal{V}_{(\Phi,v_p)}$ for vulnerability (Φ, v_p) is the set of inputs in the domain of the program which satisfy the vulnerability condition at the vulnerability point:

$$\mathcal{V}_{(\Phi,v_p)} \doteq \{\forall x \in dom(P) | P(x) : \sigma \vdash \Phi(\sigma[..v_p]) = \text{false}\} \tag{4.3}$$

Note that the vulnerability is defined such that Φ is false at exactly instruction v_p. Inputs that trigger other vulnerabilities before v_p will not necessarily be within $\mathcal{V}$.

In Figure 4.1, the language of the vulnerability is the set difference between the input domain of the program and the set of safe inputs. The vulnerability language for our running example (Figure 1.2a), where valid inputs are between 0 and $2^{32} - 1$, is $i = \{2^{32} - 3, 2^{32} - 2, 2^{32} - 1\}$.

Chapter 5

Automatic Patch-Based Exploit Generation

5.1 Introduction

In this chapter, we introduce a new type of attack we call the *delayed patch attack*. In the delayed patch attack, an attacker uses a patch P' and an original vulnerable program P to reverse engineer the vulnerability fixed in P. Figure 5.1 depicts the delayed patch attack. Software vendors currently do not protect against delayed patch attacks.

There are several questions we may ask about a delayed patch attack. First, what information about a potentially unknown vulnerability is revealed by a patch? Second, how quickly can that information be derived from the original and patched program? Third, what advantage does that information yields to attackers? Previous work has not addressed these questions (e.g., fuzz testing and others discussed in Section 5.6).

We show that *automatic patch-based exploit generation* (APEG), a type of delayed

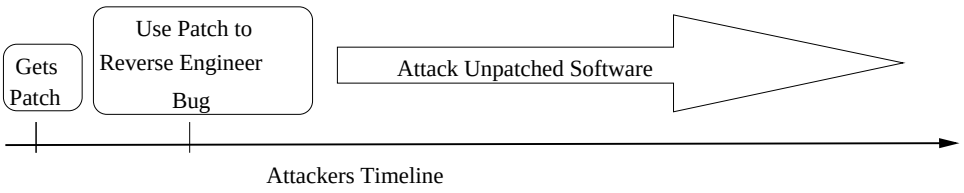


Figure 5.1: The delayed patch attack.

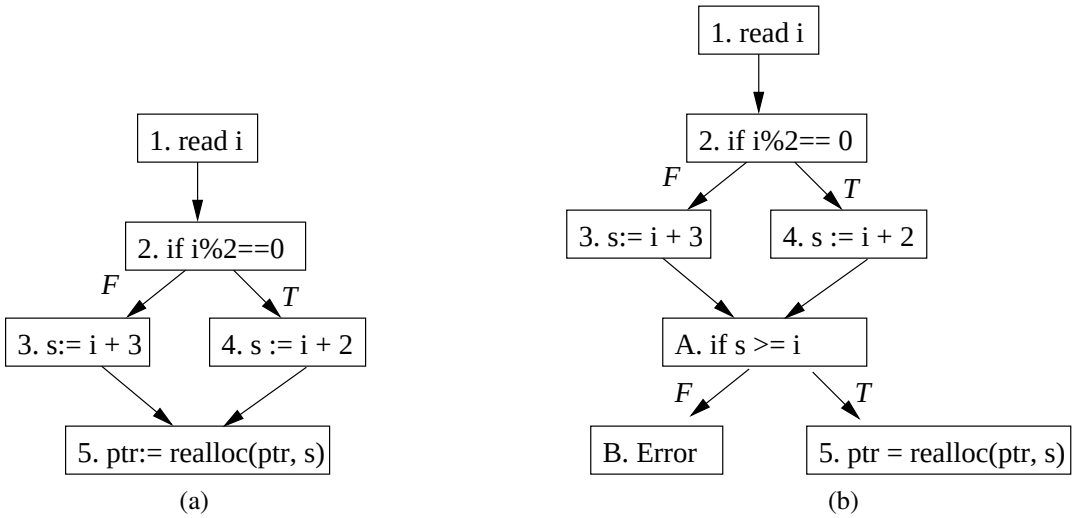


Figure 5.2: Our running example of a vulnerable program (left), and the patched version (right).

patch attack, is practical. The intuition behind our approach is that in order to address an input validation bug, a patch P' will add the missing validation checks from P. Thus, inputs that fail these new checks in P' are exploits for P. For example, consider our integer overflow running example, reproduced in Figure 5.2. Recall that any value of $i \geq 2^{32} - 3$ is an exploit because it will trigger integer arithmetic overflow on line 2 or 4. The patched program P' adds a check for overflow on line (A). Any input which is an exploit for P will fail the inserted check in P'. In Figure 5.2, the goal would be to first discover the check added on line (A) of P', then generate a value for `i` such that P'(`i`) fails the check and leads to the `ERROR` state.

Once we have identified the new check, our goal is the same as in goal-based automatic test-case generation: generate an input that executes the specified line of code (in our case, fails the check). We adopt approaches from automatic test-case generation: a static approach where we calculate the weakest precondition to execute a line of code (similar to [44]), and a dynamic approach (similar to [26, 49]). While each approach will work in some cases, neither approach works in all cases. We propose a new combined static and dynamic approach, and show it can generate exploits when a purely static or purely dynamic approach will not.

We show that automatic patch-based exploit generation is possible as demonstrated by our experiments using five Windows programs that have recently been patched. Our

techniques for APEG may not always work for any number of reasons. However, a fundamental tenet of security is to estimate the capabilities of attackers in a conservative manner. Under this assumption, APEG should be considered practical, and those who have received a patch should be considered armed with an exploit.

We successfully demonstrate APEG, and show in our experiments that 1) patches revealed enough information about a the fixed vulnerability to generate an exploit, 2) exploits were automatically generated in minutes and potentially less, and thus 3) those who first receive the patch have a significant security advantage. To put this in perspective, consider that modern threats, such as the Slammer worm, have empirically demonstrated that once an exploit is available, most vulnerable hosts can be compromised in minutes [72]. However, modern patch distribution takes hours or longer [48]. Therefore, our results imply that those who first receive a patch could potentially compromise most vulnerable hosts, e.g., by using the exploit as part of a worm.

Successful APEG motivates the need for defenses against delayed patch attacks. In particular, our work indicates that current patch distribution schemes that stagger patch roll-out over large time periods require rethinking. We describe several directions for improving current patch distribution practices.

5.2 Automatic Patch-Based Exploit Generation

In this section, we describe our approach and steps for automatic patch-based exploit generation.

5.2.1 Overview of Our Techniques

In APEG, we are given the vulnerable program P and a patch P' that fixes the vulnerability. We are also given a list of possible EM-enforceable safety policies $\Phi*$, P, and P', and the goal is to generate an input for which P violates a policy $\Phi \in \Phi*$ at instruction k, while P' does not. Note that we are given the list of possible safety policies, but not told which one is violated, where k is in P'.

In our setting, P and P' can be either an executable program or library. Addressing APEG for libraries is important since a) library vulnerabilities may often be exploited by multiple programs which use the library, and b) on many OS's, security updates are often related to libraries. For example, we conducted a survey of patches released from Microsoft in 2006 and found 84% of the security-related updates were changes in libraries.

If P is a library, then the generated exploit x is a valid set of arguments to an exported (e.g., callable) function in the library, while if P is a program, x is an input to the program.

An input x that fails the added validation check is called a *candidate exploit* for P. We call x a candidate exploit because a new check may not correspond to a real vulnerability. We verify a candidate exploit by checking that $P(x)$ violates ones of the listed safety policies $\Phi*$, i.e., observing the execution of $P(x)$ within an execution monitor. Our approach therefore attempts to generate inputs which would fail the new (input validation) checks added to P'.

Following this intuition, our approach for APEG is:

1. Identify the new check k in P'. A check will be a conditional jump. For now, assume there is only a single added conditional jump; we discuss the more general case in Section 5.5. Without loss of generality, assume that when a new check k evaluates to false, P' takes a new code path not present in P.
2. Construct a candidate exploit x which will fail the new check k. We take a program verification approach where we:
   (a) Calculate a predicate satisfied by inputs that execute potential vulnerable program paths to k (e.g., using the weakest precondition).
   (b) Each satisfying solution to the predicate is a candidate exploit x.
3. Verify the candidate exploit by evaluating $\forall \Phi \in \Phi* : P(x) : \sigma[k] \vdash \Phi(\sigma[k])$. If $\Phi(\sigma[k]) = \text{false}$, then x is verified as an exploit for P.

We describe these steps in detail in the remainder of this chapter.

5.2.2 Differencing Two Binaries Using an Off-The-Shelf Tool

The first step of our patch-based exploit generation is to difference P and P' to find new validation checks that are added in P'. Several tools exist for differencing binaries which are reasonably accurate and can be used to determine what new checks exist [38,40,43,90]. We use eEye's Binary Diffing Suite (EBDS) [40] in our implementation since it is freely available, and filter the reported differences to report only new conditional checks.

Note that previous work only finds syntactic differences between two binaries (e.g., similar to GNU diff but with binary code). Previous work did not address the more difficult and problem of automatically generating candidate exploits.

Our approach does not assume that the differencer only outputs semantic differences. For example, if P has the check $i > 10$, and P' has the check $i - 1 > 9$, the differencer

may report the latter is a new check. As a result, the list of new checks based on the syntactic analysis is a super-set of new checks added to P'. Our approach will (correctly) fail to produce an exploit for semantically equivalent differences. Semantically equivalent differences, such as the above, are weeded out by the verification step.

The differencer also indicates whether the true or false branch of a new check corresponds to a new path. In our exposition, we assume a new path corresponds to failing the check. For example, in Figure 5.2 EBDS would report the false branch of the new check on line 5 introduces a new path, and we infer that $\neg(s \geq i)$ is the check that should fail.

The remaining steps are performed on each identified new check. Of course our approach benefits from better differencing tools which output fewer and more semantically meaningful checks, as fewer iterations are needed. In our evaluation, we measure the number of new checks reported by the tool, but assume the attacker can process each new check in parallel. This is realistic since attackers often have many (perhaps hundreds or thousands of) compromised hosts they can use for checking each reported difference.

If there is a need to prioritize which new checks are tried first for APEG, we have found that one effective scheme for prioritizing is to try new checks that appear in procedures that have changed very little. The eEye tool already provides a metric for how much a procedure has changed between P and P'.

5.2.3 Generating Solvable Predicates

The next step is to generate a predicate that is satisfied by inputs that execute and fail the new check. However, the number of truly exploitable paths is often only a fraction of all paths to the new check. We then want to find solutions to the predicate (i.e., an assignment of values to predicate variables for which the predicate is true). However, there are two competing factors. If we conservatively build a predicate over all potential paths to a new exploit, the predicate may be large and therefore not be solvable. If we optimistically build a predicate for a single potential path, the predicate may not capture a truly exploitable path and thus have no solution. At a high level, therefore, there is a trade-off between the number of paths covered by a predicate, and how likely the predicate is to have a solution.

We consider three approaches to selecting paths to include in the predicate: 1) a dynamic approach which considers only a single path at a time (e.g., similar to DART [49] and EXE [26]), 2) a static approach which builds a predicate based on paths in a control flow graph (e.g., similar to Flanagan and Saxe [44]). Neither approach works in all cases. We then develop 3) a combined dynamic and static approach to address the limitations of a purely static or purely dynamic approach.

5.2.3.1 Generating Predicates with Dynamic Analysis

Let x be a known input. The dynamic approach generates a predicate representing the constraints for any input that executes the same path as $P(x)$. The dynamic analysis consists of three parts. First, we execute $P'(x)$ and record each instruction executed up to the identified new check. Second, the trace is then lifted to the Vine IL. Third, we compute the predicate by computing the weakest precondition on the IL to fail the new check.

This approach assumes we have an input that executes the new check, e.g., the new check appears along a commonly executed path. Such normal inputs can be found by examining logs of normal inputs, fuzzing, or other techniques. Of course, a normal input will likely satisfy the new check; otherwise, it is already a candidate exploit.

For example, suppose we run Figure 5.2b on the input 4. Our instruction trace would contain {`1,2,4,A,5`}. After lifting the trace to the Vine IL, and computing the weakest precondition to fail the check at `A`, we would get the predicate:

$$i\%2 == 0 \wedge s = i + 2\%2^{32} \wedge \neg(s \geq i) \tag{5.1}$$

(Note i is a variable.)

To be efficient, we only record instructions (including all of their explicit and implicit operands) dependent upon inputs (via taint-based dynamic analysis [32, 64, 84, 99]) since we only tackle vulnerabilities which can be exploited via user input. The size of the weakest precondition will be linear in the size of the trace. A solution is returned by the solver only if the path chosen is exploitable. If no solution is returned, then the chosen dynamic path is not exploitable, and we pick a new path.

The dynamic approach produces predicates that are typically the smallest of the three approaches. Since small predicates are generally the easiest to solve, the dynamic approach is usually the fastest for producing candidate exploits. The ASPNet_Filter vulnerability in our evaluation (Section 5.3) is an example demonstrating real-world utility of the dynamic approach. In ASPNet_Filter, the vulnerability is in a web-server and the new check is added along a common code path which is executed by most URI requests. In this case, it was relatively easy to obtain one benign input since common HTTP requests executed the new check.

The dynamic approach does not produce a candidate exploit if the code path is not exploitable. This happens when the sample input execution path contains the new check, but no input that takes the same execution path will fail the check. For example, the dynamic approach failed for the PNG vulnerability in our experiment because the predicate had no solution. At a high level, the path predicate was of the form $x \wedge \neg x$ where condition x appeared along the path, and $\neg x$ was the condition for failing the new check.

5.2.3.2 Generating Predicates with Static Analysis

The static approach builds a predicate over a control flow graph of P'. The resulting predicate, therefore, is satisfied if any path in the CFG is exploitable. The DSA_SetItem vulnerability in our evaluation is an example where a purely static approach works. The advantage of the static approach is it does not need a sample input. However, it may produce a predicate that is large, thus difficult to solve. The DSA_SetItem vulnerability in our evaluation is an example where a purely static approach works.

In the static approach, we first raise P' to the Vine IL. Since we are only concerned with paths that execute the new check, we remove Vine statements in the Vine IL for other paths. We achieve this by computing the chop (Chapter 3.4) for the CFG up to failing the new check. We then compute the weakest precondition (Chapter 3.1) over the chop.

In order to compute the weakest precondition, we either need to know loop invariants or unroll loops a fixed number of times. We take the latter approach since it can be completely automated. Note a dynamic approach (e.g., Section 5.2.3.1) also only considers loops a fixed number of times.

The predicate generated will be over all program paths in the chopped CFG. The predicate generated using the static approach for Figure 5.2b is:

$$((i\%2) == 0 \Rightarrow (i + 2\%2^{32}) \leq i) \wedge ((i\%2) \neq 0 \Rightarrow (i + 3\%2^{32}) \leq i) \qquad (5.2)$$

The static approach fails when the generated predicate cannot be solved. For example, in our experiments the solver we use ran out of memory before generating a candidate exploit for the IGMP vulnerability.

Program Optimizations Make Predicates Easier to Solve. In our experiments, the time to find a solution to the predicate is a significant fraction of the total exploit generation time. Therefore, techniques that reduce the time to generate a solution are likely to have an impact on the total exploit generation time.

We have found that common program optimizations can reduce the time for the solver to find an answer. We have not previously found this mentioned in related literature. Our experience has shown three common reasons predicate may take longer to solve: 1) symbolic memory writes where memory stores and loads are given by variables instead of constants, 2) algebraic simplifications that can be performed, and 3) common sub-expressions that are recomputed. We use three Vine optimizations to simplify the Vine program before generating the predicate: we perform memory value-set analysis [13], perform as much algebraic simplification as possible, and remove redundant sub-expressions using global

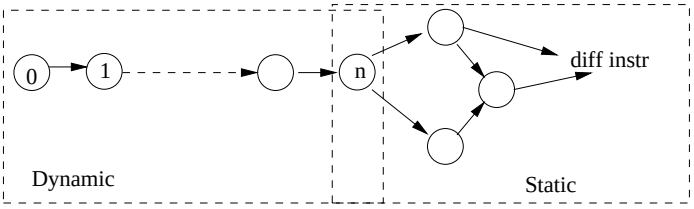


Figure 5.3: A graphical depiction of path selection for predicate generating using a combination of dynamic and static information.

value numbering [95]. In our evaluation, we show these optimizations can double the speed at which predicates are solved. (Note that these optimizations can also be applied in the dynamic case.)

5.2.3.3 A Combined Dynamic and Static Approach to Predicate Generation

The generated predicate must cover all instructions for an exploitable path in order for the solver to generate a candidate exploit. The dynamic approach considers only a single program path to the new check, generates small predicates, and requires we know an input that executes the new check. The static approach generates a predicate that covers more paths, but the predicate may be larger. At a high level, the difference between the two is that the dynamic approach uses a trace to select a path, while the static approach uses the CFG to select the paths for predicate generation.

We propose a *combined approach* where we combine the dynamic and static approach to select paths for predicate generation. Although both the static and dynamic approach alone have been used previously to generate predicates, we are unaware of other work that propose and demonstrate the feasibility of a combined approach in practice.

Figure 5.3 shows the mixed approach graphically. The combined approach offers a balance between the efficiency offered by the dynamic approach and the path coverage offered by the static approach.

Suppose we have a trace containing executed instructions $0..k$. Let instruction $0 \leq n \leq k$ be a dynamic execution, and let there be a path from n to the new check, as shown in Figure 2. We call n the *mix point*, since it is where we will mix the path selection from the dynamic and static approaches. We combine dynamic and static path selection by first truncating the execution trace at instruction n to create the "stick" end. We create the lolly end by chopping off the program using the successor of n as the chop start and the new check as the chop sink. The two pieces are put together by adding the edge from n from

the dynamically selected path to its successor in the CFG. A predicate over the resulting paths is satisfied by exploits which first execute the straight-line program path up to n, and then any subsequent path from n to the new check.

The intuition why this works is that if we lifted the entire chop from instruction 0 to the new check, then the particular path taken by dynamic analysis is a path in the chop. Therefore, the path up to some step n in the dynamic trace to the chop is also a path. In the worst case, all paths from n to the new check are infeasible, i.e., there is no input that takes the path $0..n$ and then the successor $n + 1$ to the new check.

For example, in our evaluation of the IGMP vulnerability, we combine an execution path that cannot be turned into an exploit with a chop of the procedure that contains the new check to create the predicate. The predicate generated by the combined approach is solvable, while the predicate generated by a dynamic approach alone has no solution, and one generated by a static approach alone is not solvable.

Automatic Combined Execution. An automatic combined approach requires automatically deciding the mix point. In Figure 5.3, the question is which point should we choose as n. Of course one pre-requisite is we should choose an n such that there is a path in the CFG from n to the new check. However, there still may be many such instructions in the trace.

We propose an iterative approach, which first pick the mix point closest (in terms of CFG) distance to the new check. If the generated predicate has no solution, then the path is not exploitable. The iterative approach then picks a mix point with the next closest distance, and repeat until a solution (i.e., candidate exploit) is found.

In our experiments, we found a good heuristic that is quicker than the iterative approach is to choose the mix point at the beginning of a procedure. In our experiments, Microsoft provides information about procedure boundaries to all developers. Procedures are intended to perform a specific task independent of the remaining code. Therefore, by mixing at procedure points, the combined approach includes overall tasks, instead of specific code paths. One implementation advantage of choosing procedure boundaries is that it is relatively straight-forward to implement automatic mixing; we only need to set up a jump to the static approach of the procedure at the desired mix point in the trace. When procedure information is not available, then other heuristics may be of benefit. In the worst case, one can still use the basic backtracking method for finding a mix point.

5.2.4 Generating Candidate Exploits from the Predicate

Answers that satisfy the predicate are, by construction, candidate exploits. In our approach, we use a solver to find satisfying answers. In our implementation, we interface with the decision procedure STP [46]. Other solvers could be used, though in our binary-centric setting we require a solver that supports bit-level operations, such as $\oplus$ and logical shifts, that are commonly found in binary code.

If the solver returns that a satisfying solution does not exist, then the respective path is not exploitable. Another possibility is that solving the predicate requires too many resources, e.g., takes too much time or requires too much memory. In this case, we set fixed resource limits. If the resource limits are exceeded, we move on and build a predicate for a different set of paths. For example, the mix point can be changed so that fewer paths are included. In 5.3.4 we evaluate how the changing the mix point effects how long it takes the solver to generate a candidate exploit.

Generating Exploit Variants. Our approach allows us to enumerate candidate (polymorphic) exploit variants of the paths covered by a predicate Q. Suppose x satisfies Q. Let $Q'(X) = Q(X) \wedge (X <> x)$. Q' is satisfied by all inputs except x that fail the check and execute a path in Q. Therefore a satisfying answer x' such that $Q'(x') = true$ is a candidate exploit variant. The above process can be repeated as desired.

Note that the generated exploit will execute some path covered by the predicate. If the predicate covers over more than one path, then the exploits generated may execute different code paths. A straight-forward modification to the predicate generation procedure can be made to assert whether particular code paths are, or are not, followed.

5.2.5 Verifying a Candidate Exploit

We verify the candidate exploit x by checking if a safety policy Φ from the list of policies $\Phi*$ is violated when executing $P(x)$. In our implementation, we use an off-the-shelf detector which employs dynamic taint analysis as a black box for Φ for memory safety vulnerabilities. Using other types of exploit detectors is also possible. The candidate exploit is verified when the detector indicates the program is exploited. If the verifier never indicates exploit, then we can iterate the procedure over different code paths (and also over different checks as discussed in 5.5).

5.3 Evaluation

In this section, we evaluate our approach on five different vulnerable Microsoft programs which have patches available. Our experiments highlight that each approach for predicate generation — dynamic, combined, and static — is valuable in different settings. We show that we can generate exploits when no public exploit is available (to the best of our knowledge) for the ASPNet_Filter, IGMP, and PNG vulnerabilities. We also show that we can generate polymorphic exploit variants.

We focus on reporting our results on generating exploits for the new check which is exploitable, as discussed in Section 5.2.2. We also report the order in which the exploitable check would be found using the least-changed heuristic from Section 5.2.2.

5.3.1 Vulnerability and Exploit Description

DSA_SetItem Integer Overflow Vulnerability. The DSA_SetItem routine in comctl32.dll performs memory management similar to `realloc` [92]. At a high level, the vulnerable function takes in a pointer p, a size for each object s, and a total number of objects n, and then calls `realloc(p,` $s * n$`)`. An overflow can occur in the multiplication $s * n$, resulting in a smaller-than-expected returned pointer size. Subsequent use of the pointer at best causes the application to crash, and at worst, can be exploited to hijack control of the application. `DSA_SetItem` can be called both directly and indirectly by a malicious webpage via the `setSlice` JScript method. In practice, this vulnerability is widely exploited on the web either by overtly malicious sites, or legitimate but hacked web sites [76].

The patched version adds logic to protect against integer overflow. In particular, it adds a check that error-handling code is called if the product is greater than 2^{31} (i.e., is positive).

EBDS took 371.9 seconds to perform the differencing. 21 functions were found changed, and 5 new functions were added. Given the least-changed heuristic, the exploitable check would be the third check tried.

Exploit Generated: The exploits we generated caused a denial of service attack, e.g., Internet Explorer crashed. Any ϕ that can detect pointer misuse is suitable: we used TEMU [8]. We could also specify specific memory locations to overwrite. Determining the specific address for a successful control hijack requires predicting the processes memory layout, which changes each time the process is invoked. Attackers currently do this by essentially repeatedly launching an attack until the memory layout matches what the exploit expects. We similarly repeatedly launched the attack until we achieved a successful control hijack.

ASPNet_Filter Information Disclosure Vulnerability (MS06-033; Bugtraq ID#18920; CVE-2006-1300). The ASPNet_Filter dynamically linked library (DLL) is responsible for filtering ASP requests for the Microsoft .NET IIS Server, and is vulnerable to an information disclosure attack. The module filters sensitive folder names from a URI request during processing so that information contained in these folders is not disclosed upon response. These folders are automatically built using ASP.NET's default template. For example, `App_Data`, `App_Code`, and `Bin` are used to store data files, dynamically compiled code, and compiled assemblies, respectively. An exploit for this vulnerability would allow the attacker to view files under these folders. This is a serious vulnerability because scripts in these directories often contain sensitive information, such as passwords, database schemas, etc. To the best of our knowledge, there are no public exploits for this vulnerability.

The unpatched version performs proper filtering for URI requests that use forward slashes ('/'), but not backslashes ('\'). The patched version fixes this vulnerability by checking for '\' and flipping them to '/'.

EBDS took 16.6 seconds to perform the differencing. One new function was added, along with 4 changes to existing procedures to call the new function. The exploitable check using the least-changed heuristic would be the first one tried.

Exploit Generated: The exploit we generated was able to read files in the protected directories. Currently we have not implemented a ϕ that detects such attacks, so we verified the generated candidate exploit manually.

IGMP Denial of Service Vulnerability (MS06-007; Bugtraq ID#16645; CVE-2006-0021). The IGMP (Internet Group Management Protocol) protocol is used for managing the membership of multi-cast groups. An exploit for this vulnerability is an IGMP query packet with invalid IP options. The invalid options can cause the IGMP processing logic to enter an infinite loop. Since IGMP is a system-level network service, an exploit will freeze the entire vulnerable system. The patch adds checks in the IGMP processing routine for invalid IP options. To the best of our knowledge, there is no public exploit for this vulnerability. [1]

EBDS took 157.08 seconds to difference the patched and unpatched code in `tcpip.sys`. Only one function was changed. Using the least-changed heuristic, the exploitable check would be first.

The exploit we generated successfully caused the denial-of-service. Currently we have

[1] An EBDS [40] tutorial discusses this vulnerability. However, they do not create an exploit.

not implemented a ϕ that detects deadlock due to an infinite loop, thus we verified our candidate exploit manually.

GDI Integer Overflow Vulnerability (MS07-046; Bugtraq ID#25302; CVE-2007-3034). The Windows Graphic Device Interface (GDI) is the core engine for displaying graphics on screen. The GDI routine responsible for showing meta-file graphics is vulnerable to an integer overflow. The integer overflow can subsequently lead to a heap overflow, which at best causes a system crash, and at worst, can result in a successful control hijack.

The patch addresses the integer overflow by adding 5 additional checks when loading a meta-file. The unpatched version is exploitable when any one of the 5 checks fails.

EBDS took 109 seconds to difference the patch and unpatched version. EBDS identified the 5 additional checks. Since an exploit can fail any of the 5 checks, an exploitable check would be tried immediately using the least changed heuristic.

Exploit Generated: The exploit we initially generated caused a denial-of-service. This vulnerability is similar to DSA_SetItem; we can specify what to overwrite in the heap structure, but the location of the heap structure depends upon the process layout. Thus, a successful control hijack required repeatedly launching the attack. Any ϕ that detects pointer misuse is appropriate; we used TEMU [8].

PNG Buffer Overflow Vulnerability (MS05-025; Bugtraq ID#13941; CAN-2005-1211). PNG (Portable Network Graphics) is a file format for images utilized by many programs such as Internet Explorer and Microsoft Office programs. Each PNG image contains a series of records which specify different properties of the image, e.g., whether the image is indexed-color or gray-scale, the alpha channel, etc. In the indexed-color mode, the record format specifies an additional alpha channel byte value for each indexed color. A heap-based buffer overflow occurs in early Microsoft implementations when the number of alpha channel bytes exceeds the number of pre-specified colors.

The patched version adds additional checks to validate PNG record fields. To the best of our knowledge, there are no public exploits for this vulnerability.

The total time to difference the two versions was 27.05 seconds. Changes were only reported in the vulnerable procedure, with the exploitable check being the first using the least changed heuristic.

Exploit Generated: The exploit we generated initially caused the program to crash, similar to GDI and DSA_SetItem. Again, we use TEMU [8] to confirm candidate exploits, but any ϕ that detects pointer misuse is also possible. This attack is on the heap, and also

	DSA_SetItem	ASPNet_Filter	GDI
Trace	4.99	4.50	9.92
Predicate	0.52	0.14	0.41
Solver	0.17	6.93	0.01
Total	5.68	11.57	10.34

Table 5.1: Time to generate an exploit using the dynamic approach. All times are in seconds.

required us to repeatedly launch the attack to achieve successful control hijack.

5.3.2 Patch-Based Exploit Generation using Dynamic Analysis

We successfully generated exploits for the DSA_SetItem, ASPNet_Filter, and GDI vulnerabilities using dynamic analysis. For DSA_SetItem, we recorded the execution trace of IE 6 loading a valid web-page that calls the setSlice ActiveX control method, which in turn calls DSA_SetItem. For ASPNet_Filter, we recorded IIS processing an HTTP request from a log file. For GDI, we created an image within a PowerPoint presentation, then saved the image in the Windows meta-file format. We recorded the execution of a small GDI application loading the saved file. All execution traces were recorded using TEMU [8].

Table 5.1 shows an overview of our results. All times in the table are in seconds. The "Trace" row shows the amount of time it took to generate a trace using TEMU. The "Predicate" row shows the amount of time to lift the trace to our modeling language and produce the predicate. The "Solver" row indicates how long it took the solver to solve the predicate.

The total time to generate an exploit after diffing is under 12 seconds in all experiments. If we include diffing time, then the total exploit generation time for DSA_SetItem is 377.58 seconds, ASPNet_Filter is 28.17 seconds, and GDI is 119.34 seconds.

We were not able to generate exploits using the dynamic approach for the IGMP and PNG vulnerabilities. For IGMP, we recorded the execution of Windows processing the sample IGMP message from [33]. The identified new checks were executed. However, the predicate was not satisfiable by any input that failed the new check. The reason is that the particular execution path taken was already constrained so the added check could never fail (i.e., was redundant along that path). For PNG, we were not able to generate an

	DSA_SetItem		GDI	
	no opt	opt	no opt	opt
To Vine	1.35	1.45	3.61	3.97
Predicate	2.48	0.87	3.45	1.02
Solver	182.91	81.15	19.61	21.42
Total	186.74	83.47	26.67	26.41

Table 5.2: Time to generate exploit using the static approach. All times are in seconds.

exploit for a sample execution trace for the same reason: the path predicate prevented the new check from ever failing. In particular, the execution of PNG involves the calculation of a CRC-32 check-sum. There were no other inputs along the chosen path that satisfied the check-sum while failing the new check.

5.3.3 Patch-Based Exploit Generation using Static Analysis

We were able to generate exploits for the DSA_SetItem and GDI vulnerabilities using a purely static approach. For DSA_SetItem, the static model included setSlice and DSA_SetItem. For GDI, the vulnerable procedure `GetEvent` is reachable by the explored API `CopyMetaFileW`. Thus, our static model consisted of these two functions.

Table 5.2 shows an overview of our results. All times in the table are in seconds. We include in this table the time to generate the Vine IR and chop of all static paths to the new check under the "To Vine" row. For each vulnerability, we also consider two cases: with and without the optimization on the model discussed in Section 5.2.3.2.

Without optimization, we were able to generate exploits for DSA_SetItem in 186.74 seconds. When we enable optimizations, the time to generate the model increases, but the subsequent steps are much faster. In particular, the optimizations for DSA_SetItem reduce the time to generate an exploit from the predicate by about 55%. We believe further optimizations would likely further reduce the solution time. For GDI, the optimizations had less effect, saving .26 seconds overall.

We enumerated 3 different exploits for the DSA_SetItem vulnerability. In particular, we enumerated both the public exploit, and 2 new exploit variants.

One way to compare the advantage of the static approach is to measure the number of paths to the new check included in the predicate. A similar predicate using the dynamic

	DSA_SetItem	IGMP	GDI	PNG
Trace Gen	4.99	10.14	9.92	103.28
To Vine	1.42	2.58	3.36	0.58
Predicate	0.31	12.57	0.27	0.28
Solver	4.79	3.78	0.26	0.14
Total	11.51	29.07	13.57	104.28

Table 5.3: Time to generate an exploit using the combined approach. All times are in seconds.

approach alone would require enumerating each path. There are 6 exploitable paths to the new check for DSA_SetItem in the static model we consider. There are about 1408 total paths in the static model for the GDI vulnerability.

We were not able to generate exploits statically for the PNG, IGMP, and ASPNet_Filter vulnerabilities. In the ASPNet_Filter vulnerability, there are system calls not currently supported by our predicate generator. The standard solution is to generate summaries of the effects [24, 29]. A manual analysis indicates that simply omitting the various calls would likely still result in a predicate that generates exploits. We leave exploring such extensions as future work. We could not generate exploits statically for all paths for the PNG and IGMP vulnerabilities because the solver ran out of memory trying to solve the generated constraints.

5.3.4 Patch-Based Exploit Generation using Combined Analysis

We successfully generated exploits using the combined approach for DSA_SetItem, IGMP, GDI, and PNG. In our experiments, we use the heuristic to mix at procedure boundaries.

Table 5.3 show our results when we mix using the dynamic trace from Section 5.3.2 up to the vulnerable procedure. The static approach generates a predicate for the vulnerable procedure. The two are then spliced together.

The combined approach works for IGMP and PNG, whereas the purely dynamic and purely static approaches do not. In both cases the purely dynamic approach fails because the executed path in the trace is not exploitable. In both cases the static approach also fails because the solver runs out of memory. The combined approach offers a way to build a predicate for a subset of potentially exploitable paths without enumerating them

Dyn:Static	Predicate Size	Solver Time	# Paths
4:1	309250	18.94	496
3:2	310414	22.77	496
2:3	6549513	Out of Mem	10416

Table 5.4: Results for changing the mix point at different points in the call path to the vulnerable procedure. The predicate size is the number of expressions in the predicate. Solver time is in seconds.

individually.

We also measured how mixing reduces the static predicate size for the IGMP vulnerability. The shortest call path to the vulnerable function has length 5: IPRcvPacket → DeliverToUserEx → DeliverToUser → IGMPRcv → IGMPRcvQuery. We consider mixing at IGMPRcvQuery, IGMPRcv, and DeliverToUser, i.e., the predicate consists of all paths through 1, 2, and 3 procedures, and the rest from the dynamic path.

Table 5.4 shows our results. This table shows that using the dynamic predicate for IPRcvPacket → DeliverToUserEx → DeliverToUser and the static for IGMPRcv and IGMPRcvQuery is solvable, while adding all paths for DeliverToUser creates a predicate that is too difficult to solve. It also shows a common behavior when solving predicates in our experience: they are either solvable relatively quickly, e.g., within a few minutes, or they are not solvable within a reasonable amount of time.

5.4 Implications of Automatic Patch-Based Exploit Generation

Our evaluation demonstrates APEG for several vulnerabilities. Since we must conservatively estimate the capabilities of attackers, we conclude APEG should be considered a realistic attack model. The feasibility of automatic exploit generation has important implications on the security landscape.

Therefore, APEG demonstrates that delayed patched attacks should be taken seriously. One of the most immediate implications is to rethink today's patch distribution practices to limit the damage from a delayed patch attack. Current patch distribution techniques use a staggered approach where vulnerable systems will receive a patch at different times. Staggered patch distribution is attractive because it prevents huge traffic spikes when a

new patch is released. Recall that measurements indicate that it takes about 24 hours for Windows Update to see 80% of the unique IPs of hosts checking for a patch [48].These measurements confirm the intuition that not everyone will receive a patch at the same time, with gaps of hours if not longer before even the majority receive the update.

In our results, we are typically able to create exploits from the patch in a matter of minutes, and sometimes seconds. Therefore, APEG could enable those who first received a patch to generate an exploit and compromise a significant fraction of systems *before they even had a chance to download the update.* Note that this is irrespective of whether people actually apply the patch; but whether they even have the opportunity to apply it.

There are many approaches to defend against delayed patch attacks such as APEG. We discuss three directions: 1) make it hard to find new checks (e.g., using obfuscation), 2) make it so everyone can download the update before anyone can apply it (e.g., using encryption), and 3) make it so everyone can download the patch at the same time (e.g., using P2P).

Patch Obfuscation. One approach is to make it difficult to determine what new checks are added to P'. In particular, vendors could obfuscate patches such that the difference between P and P' is very large. This approach would be the easiest to break our particular implementation, since the results of EBDS [40] would contain too many instructions to isolate which checks were added.

The advantage of this approach is obfuscation techniques are widely available. However, there are many challenges to the obfuscation approach. For example, figuring out the level of obfuscation necessary to thwart attackers may be tricky. Simple instruction replacement, e.g., multiplications by 2 with left shifts, may thwart EBDS but not a more sophisticated tool that focused on semantic, not assembly-level syntactic differences [47]. Another problem is the effects of obfuscation should be transparent to legitimate users, e.g., obfuscation that degrades performance is likely unacceptable. Finally, there are theoretically negative results about the possibility of general obfuscation [15]. Thus, obfuscation is not likely to be a complete solution to delayed patch attacks.

Patch Encryption. We could initially encrypt patches so that simply having the patch leaks no information. Then, after a suitable time period, a short decryption key (e.g., 128-bits) is broadcast. This scheme allows all users who have the patch and receive the key to apply it simultaneously. Others have independently arrived at similar ideas [55, 89].

Patch encryption allows vendors to use essentially the same staggered patch distribution architecture while defending against automatic patch-based exploit generation. Si-

multaneously (or near simultaneously) distributing the decryption key is possible since the key is very small, e.g., 64-bits. Therefore, this scheme is potentially fair in the security sense: everyone has the same opportunity to apply the patch before anyone could potentially derive an exploit. One problem is determining when to broadcast the encryption key. Another potential problem is how to handle off-line hosts. A second problem is the actual fixes are delayed from the users perspective, which raises a number of policy issues. There are security-related policy choices, e.g., should patches be encrypted when a zero-day exploit is available to a few attackers, but not all attackers. There are also human-related factors, e.g., people may not like the idea of having a patch that they cannot apply. Further research is needed to answer such questions.

Fast Patch Distribution. It may be possible to change patch distribution so everyone receives the patch at about the same time. For example, Gkantsidis *et al*. propose using a peer-to-peer network for patch distribution in order to reduce the load on patch distribution servers [48]. Such a peer-to-peer system could potentially also distribute patches faster than the centralized model. However, such as scheme would still need to address off-line hosts. It is also unclear whether such a scheme is fast enough to combat APEG.

5.5 Discussion

5.5.1 Generating Specific Exploits

The techniques we describe generate an exploit from the universe of all exploits for a patched vulnerability. At a high level, the solver gets to pick any exploit that satisfies the generated predicate. We can generate particular kinds of attacks in many cases, e.g., generate a control hijack attack.

One problem is we do not initially know what vulnerability is patched, it does not make sense to try to create a specific type of exploit *a priori*. For example, if the unknown vulnerability is an information disclosure vulnerability, it makes no sense to try to create a control hijack exploit, e.g., control hijack cannot be performed in the ASPNet_Filter information disclosure vulnerability.

However, once we know what vulnerability can be exploited, we can extend our approach to generate specific kinds of attacks, as long as we can write the conditions necessary as a constraint over the vulnerable program state space. For example, we can add assertions to the predicate such that any solution must overwrite sensitive data structures.

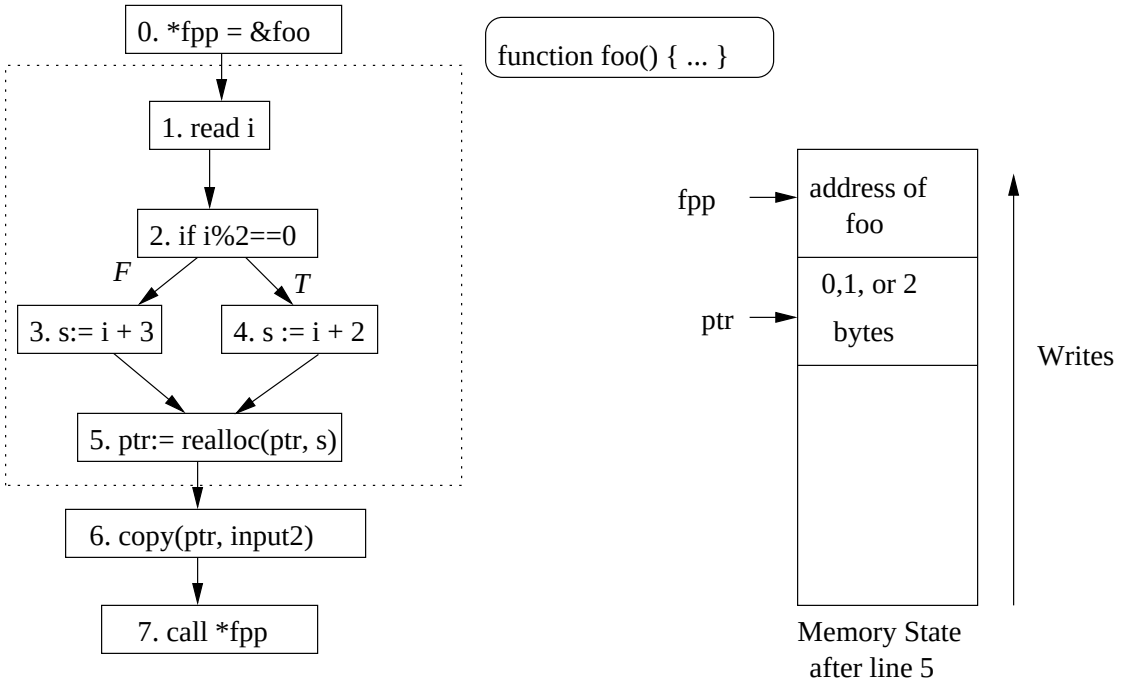


Figure 5.4: Our running example extended to demonstrate creating a specific exploit.

We take a two-step process to creating specific exploits. The first step identifies the kind of vulnerability by generating an exploit. The second step creates a specific exploit for that type of vulnerability.

In the first step, we create an initial exploit x that fails the new checks. Recall we are given a list of vulnerability conditions $\Phi*$. We run the vulnerable program $P(x)$ to determine *which* vulnerability condition Φ is violated. Only then will we know if it makes sense to generate a specific type of exploit, e.g., a control hijack; until we know what the actual vulnerability is by demonstrating initial exploit, we do not know what type of vulnerability is patched. In the second step, we add constraints to the predicate such that a satisfying answer is the type of exploit desired.

For example, consider Figure 5.4, which extends our running example similar to how the real vulnerability works. The code inside the dashed box is the same as the running example; statements 0, 6, and 7 are new. Statement 0 initializes a pointer to a function pointer `fpp` to point to the function `foo`. Statement 6 copies additional user input into the allocated `ptr`. Statement 7 then calls the function pointed to by `fpp`.

Suppose we want to create a control hijack attack of this example. Recall that we will

initially create an exploit that is a value of $i \geq 2^{32} - 3$. This is an exploit because it violates the overflow safety property. The right-hand side of Figure 5.4 show the program memory state after executing statements 0-5 on such inputs. For exploit inputs, `ptr` will point to 0, 1, or 2 bytes of memory. The copy on statement `6` will copy to the memory address given by `ptr`, up to higher addresses. Thus, if more than 2 bytes are read in as `input2`, the function pointed to by `fpp` will be overwritten by user input. The subsequent call on line 7 will then be to a function at a user-defined address.

To automatically create a control-hijack exploit, we perform the following steps:

1. We first create an exploit for the patched bug on line 5. This generates an input x that causes overflow.
2. We next run the program on input x and record the execution trace. The execution trace tells us that the overflow pointer is subsequently used to read in user data.
3. We add constraints to the symbolic predicates as necessary to overwrite the critical structures as desired.
4. We solve the new constraint predicate, which will be a control hijack exploit.

We can perform similar steps to create other specific types of exploits.

5.5.2 Dealing with Multiple Checks

The patch for a single vulnerability may have many new checks in the patched version. In some cases, our techniques will still work as-is, such as in the GDI vulnerability.

In other cases, an attacker may have to consider a specific combination of possible changes. Consider the case where an input has to fail two new checks a and b in sequence to exploit the unpatched version. Using the static approach, we could build a predicate over all potential paths through a and b, e.g., the case where they fail together, but also the case where a succeeds but b fails. However, it could be the case that we cannot generate a solvable predicate that considers all combinations simultaneously. In such situations, we could also potentially explore each combination using a variety of mixed or dynamic methods. This approach would generate smaller predicates, but the predicate would have to have the right combination of failed checks to produce a verifiable exploit. We leave it as future research to determine the best strategies in such situations.

From the security standpoint, however, we must conservatively assume attackers are lucky. The combinatorial problem seems unlikely to pose a significant obstacle in enough cases that we should not consider the possible number of combinations itself a defense against APEG.

5.5.3 Other Applications of Our Techniques

Our techniques have applications in other areas. For example, automatic deviation detection is concerned with the problem of finding any input i for programs P_1 and P_2 such that the behavior of $P_1(i)$ is different than $P_2(i)$. In our scenario, $P_1 = P$ and $P_2 = P'$, and the deviation input $i = e$ such that P_1 is exploited but P_2 is not. Previous work focused on deviation detection from a single dynamic trace [19]; we consider multiple paths.

We expect our techniques, especially combined dynamic and static predicate generation, will be applicable to many similar problems that require modeling multiple program paths. Most previous work that requires generating a predicate to represent a program path only focus on a single path for scalability reasons. Our work shows for the first time that scaling up to multiple paths in binary programs is possible. In particular, applying the combined static and dynamic approach to other settings is an interesting avenue to explore.

Our techniques generate an input that demonstrates the difference between two versions of a program. We expect such techniques may be useful to software developers. For example, a software developer could use our techniques to generate test cases for just the differences between two program versions.

5.6 Related Work

Fuzzing to find inputs which crash programs essentially tries random or guided semi-random inputs on a program [45,50,66,69,70]. Fuzzing tools have recently become popular as a way of finding exploits for programs, e.g., fuzzing found numerous vulnerabilities in the Month of Browser Bugs [7]. Recently, fuzzing techniques have been augmented to produce particular kinds of exploits, e.g., control-hijack exploits for buffer overflow vulnerabilities [66]. Unlike fuzzing, our approach is goal-oriented: we find an input that reach a specific line of code (the new check). Instead of searching for vulnerabilities at random, we use the patch as a guide to generate exploits. Fuzzing and similar techniques also only consider P, thus do not address generating exploits from patches.

We use an off-the-shelf differencer to identify changes. Research in finding semantic differences, such as BinHunt [47], would help winnow down the number of new checks for which we try exploit generation. Others have speculated that automatic patch-based exploit generation may be possible, e.g., [39]. To the best of our knowledge, there is no previous work that demonstrates it is possible.

Our techniques are closely related to automatic test case generation, which has a long

history (e.g., [18, 51, 52, 58]). Our techniques are most closely related to goal-based test generation (e.g., [51]) where inputs are automatically generated that will execute a given goal statement in the program.

Recently, DART [49] and EXE [26] work along these lines by incrementally exploring execution paths and letting the execution itself generate the predicate. However, DART and EXE achieve this by rewriting source code; our techniques only have access to the binary. In addition, DART and EXE consider each path individually, while we can consider multiple paths.

5.7 Conclusion

We have demonstrated that delayed patch attacks should be taken seriously by showing that automatic patch-based exploit generation (APEG) is practical on real-world cases. In our evaluation, we are able, within a few minutes, to automatically generate an exploit given the unpatched and patched program. In order to achieve our results, we developed novel techniques for analyzing potential exploitable paths to a new validation check. Since best security practices dictate that we conservatively estimate the power of an attacker, our results imply that in security-critical scenarios automatic patch-based exploit generation should be considered practical. One immediate consequence we suggest is that the current patch distribution schemes are insecure, and should be redesigned to more fully defend against automatic patch-based exploit generation. We propose several research directions for defending against the delayed patch attack and APEG in particular.

Chapter 6

Automatic Vulnerability Filter Generation

6.1 Introduction

One of the most popular and effective defense mechanisms for protecting vulnerable programs against exploits is input filtering (a.k.a. content-based filtering or signature-based filtering). Without input-based filtering, all inputs are processed directly by the vulnerable application. Input-based filtering introduces a filter check step before a vulnerable application receives an input (Figure 6.1). A filter is a function from the input domain of the vulnerable application to the labels {unsafe, safe }. If a filter labels an input as unsafe, it is dropped ("filtered") from the program input stream.

Input filtering is attractive because it allows vulnerable programs to continue to operate

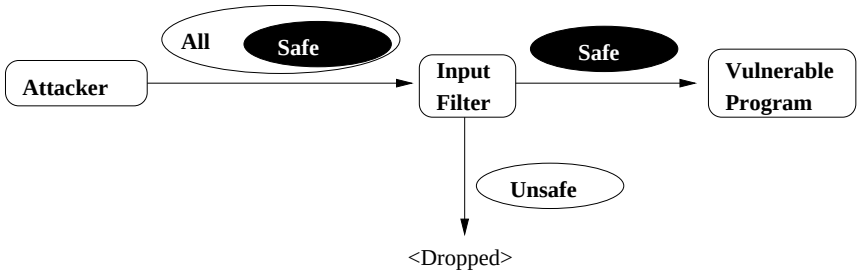


Figure 6.1: Input-based filtering introduces a filter check before an application receives an input. Inputs deemed unsafe are dropped.

even while under attack. Filter-based defenses will protect vulnerable programs without requiring the program itself to be replaced with a patched version. Note that the difference between a patch and a filter is that a patch is created by a human to perform semantically correct actions, while a filter simply drops offending inputs. Although in most situations patching is eventually the best course of action, filters are an important line of defense until patches can be developed, tested, and deployed.

At the core of filter-based defenses are the filters themselves. Since the purpose of a filter-based defense is to make the system safer, we are primarily interested in developing accurate filters. Defenses based upon inaccurate filters are of dubious benefit. Generating accurate filters, however, is challenging because we typically do not have access to all exploits that we wish to filter. If we knew all exploits, our filter could simply be the disjunction of all exploit strings. Thus, we want to generate filters that protect against exploits we have yet to see.

Further, we need accuracy to be guaranteed. Accuracy guarantees are needed to ensure that the cost of the additional filter check on inputs is worthwhile. Filters with no guarantees are not appropriate because they may offer no real benefit, and only serve to introduce delays by the filter check step. There are two important types of guarantees for generated filters. First, a filter is *sound* if all inputs labeled as unsafe are guaranteed to exploit the vulnerability. A soundness guarantee means that a filter will never interfere with legitimate traffic. Second, a *complete* filter is guaranteed to label all exploits as unsafe. A completeness guarantee means that the filter will completely protect the vulnerable application from all exploits. A perfectly accurate filter — a filter with no errors — is both sound and complete.

6.1.1 Limitations of Previous Approaches

At a high level, previous approaches are insufficient because: 1) they offer no filter accuracy guarantees, and/or 2) an attacker can fool the algorithm into generating inaccurate filters, and/or 3) they are inefficient, and/or 4) they have limited applicability.

The predominant previous approach for automatic filter generation has focused on using machine learning techniques to generate a filter [57,59,63,81,96,104]. At a high level, this line of research has focused on extracting patterns from known exploits. The patterns become the filters, and any input that matches the pattern is considered unsafe.

Machine learning-based techniques cannot offer filter accuracy guarantees for the class of input-validation vulnerabilities [102]. Worse, filters generated using machine learning techniques are vulnerable to attacks which fool the generator into generating a filter with

errors [82,86,102]. Researchers have demonstrated that attacks against the filter generator succeed in practice [82, 86]. Further research has shown that attacks against the filter generator are fundamental to all approaches using machine learning [102].

A second popular approach for filter generation is to use heuristics to generate a filter from a sample execution of an exploit [64,84,106]. However, heuristics-based techniques cannot provide any guarantees on filter accuracy. Further, these heuristics may not work in many real-world vulnerabilities [31,81].

Concurrent to our work, Costa *et al.* have proposed techniques from an execution trace that generate filters that do not use heuristics [29,30] as part of an overall end-to-end defense system. This work makes a positive step by providing a soundness guarantee for filters expressed as Boolean formulas. We provide a generalized formulation and formalization, develop techniques that are more efficient in the specific cases they handle, and extend guarantees to other language classes such as Turing-complete and regular expressions.

Although there has been significant research in filter generation, there is currently no standard definition for what it means for a filter to have an accuracy guarantee. For example, [29,30,57,59,63,64,81,84,96,104,106] state filter accuracy (at least in part) by evaluating the filters on known traces. However, guarantees cannot be derived by empirical observations: tomorrows exploits may look nothing like those seen today (as observed by [63,82,86,102]).

6.1.2 Our Approach

Our approach starts by first formalizing input filter generation for input validation vulnerabilities. Formalization is necessary in order to define the desired accuracy guarantees. We then show how this formalization leads to a natural method for generating filters based upon program analysis of the vulnerable application. We call our filters *vulnerability-based* filters. The distinguishing characteristic of a vulnerability filter is that it is based upon the semantics of the vulnerability.

At a high level, we are given a vulnerability specification (Φ, v_p) for a vulnerable program P. By our definition of an input-validation vulnerability (see 4.3.1), any input that exploits P must violate Φ at instruction v_p. We generate filters by extracting from the program itself the conditions for an execution to violate Φ at instruction v_p. The filter is accurate because the analysis is faithful to the semantics of the vulnerability.

6.1.3 Contributions

Our techniques address the four primary limitations of previous approaches. First, accuracy is guaranteed by the filter construction algorithm. Second, the filter generation algorithm cannot be fooled by attackers. Third, our techniques are efficient. Fourth, our techniques are more widely applicable because they provide an explicit mechanism for addressing the fundamental accuracy vs. performance trade-off.

We establish the first formal approach for automatic vulnerability filter generation for input validation vulnerabilities. Previous work has not provided any formal foundation for reasoning about the class of vulnerabilities to which the techniques are applicable. In addition to suggesting directions for filter generation, our foundation also provides metrics for comparing the accuracy of filters in a manner which is not specific to the generation mechanism.

A filter is a recognizer for the set of unsafe inputs. However, which language class we write the filter in provides limits to the fundamental accuracy and efficiency of a filter. We explore the trade-offs for representing filters among the three classes. Previous work has only focused on single language class, usually regular expression filters. We show how to generate filters with a variety of filter language restrictions. In particular, we show how to generate filters with guarantees when the filtering language is unrestricted, restricted to Boolean formulas, and restricted to regular expressions.

Finally, we demonstrate our methods empirically. We show that we can automatically generate accurate and efficient filters for real-life vulnerabilities.

6.2 Vulnerability Filters

We generate vulnerability filters, which are based upon analysis of the vulnerability in the program. Our technique is contrasted with exploit-centric approaches such as [57, 59, 63, 81, 96, 102, 104], which focus on generating a filter from exploit samples. At a high level, the distinguishing advantage of a vulnerability filter is that it does not depend on the behavior of a given exploit. For example, a stack-based buffer-overflow vulnerability may be exploited in several ways, including code injection, return-to-libc attacks, or simply crashing the program. A vulnerability filter is indifferent to the specifics of the attacks.

We focus on techniques for generating vulnerability filters for input validation vulnerabilities (defined in 4). For simplicity, we use the term *vulnerability* to denote this class of vulnerabilities, and *exploit* to denote an input that satisfies definition 4.3.2.

6.2.1 Definitions

We formalize filters in terms of the vulnerability itself. Previous work has measured filters with respect to empirically measured accuracy. Our approach defines what exactly the filter should be filtering in terms of the underlying input validation vulnerability.

Definition 6.2.1. A *vulnerability filter* for program P and vulnerability specification (Φ, v_p) maps inputs in the domain of P to either safe or unsafe:

$$\mathcal{F} : \mathrm{dom}(P) \rightarrow \{\mathsf{safe}, \mathsf{unsafe}\}$$

For example, a vulnerability filter for our running example in Figure 1.2a would be a function from 32-bit integers to {unsafe, safe }.

Let V be the language of the vulnerability (Φ, v_p) we wish to filter. We define errors on filters as:

Definition 6.2.2. A *false positive* occurs when a safe input is marked as unsafe: $\mathcal{F}(x) = \mathsf{unsafe} \wedge x \notin V$. We say a filter is *sound* if it has zero false positives, i.e., $\mathcal{F}(x) = \mathsf{unsafe} \implies x \in V$.

For example, a sound filter for our running example would only return unsafe when $i = \{2^{32} - 3, 2^{32} - 2, 2^{32} - 1\}$. A false positive occurs when the filter returns unsafe for any other input.

Definition 6.2.3. A *false negative* occurs when an unsafe input is marked safe: $\mathcal{F}(x) = \mathsf{safe} \wedge x \in V$. We say a filter is *complete* if it has zero false negatives, i.e., $x \in V \implies \mathcal{F}(x) = \mathsf{unsafe}$.

In our running example, a complete filter must return unsafe for all $i = \{2^{32} - 3, 2^{32} - 2, 2^{32} - 1\}$. A false negative would be returning safe for one of these inputs.

Definition 6.2.4. A *perfect filter* recognizes exactly the language of the vulnerability: $\forall x \in V : \mathcal{F}(x) \rightarrow \mathsf{unsafe} \wedge \forall x \notin V : \mathcal{F}(x) \rightarrow \mathsf{safe}$.

Note: The above formalism relates that filter accuracy is dependent upon the semantics of the vulnerability. This connection, though it may seem straight-forward, has previously not been shown. In particular, previous work focused on only empirically determining filter accuracy. The problem with an empirical approach is it does not account for an attacker who at a later time may generate a new, unknown variant [57, 59, 63, 81, 96, 104]. In other words, an empirical approach does not capture the notion of how accurate a filter will be in the future; it only captures the effectiveness in a particular measurement at a particular time. Using the above, however, we can characterize a filter by its false negative and positive ratio with respect to the vulnerability language.

6.2.2 Filter Language Classes and Trade-offs

The above definitions suggest that a perfect filter must be in at least the same language class as the vulnerability language. That is, if the language of the vulnerability is Turing-complete, then a perfect filter must be written in a Turing-complete language. We explore filter generation where we can vary the restrictions on the filter language class. Previous work in filter generation has focused on generating filters in a single language class, e.g., regular expression filters [57, 59, 63, 81, 96, 104] or a Boolean expression [29, 30].

We focus on three different language classes for writing the filter itself. First, Type-T filters are filters written in Turing-complete programming languages. Second, Type-B filters are filters limited to Boolean formulas. Third, Type-R are filters limited to regular expressions.

The reason we wish to explore many types of filters is because a single filter class may not always be appropriate. For instance, Type-R filters are fast to evaluate. However, there is no perfect Type-R filter for a vulnerability that involves arithmetic because regular expressions cannot count.

6.2.2.1 Type-T Filters

A Type-T filter is a filter that is expressed in a Turing-complete language. Type-T filters can be perfect since they place no restrictions on the filter language. A Type-T filter is a program, and Type-T filter evaluation is performed by running inputs on the program.

Type-T filters can have perfect accuracy. As a consequence of having no language restrictions, however, Type-T filters do not offer performance guarantees. For example, in our construction Type-T filters may not terminate on safe inputs if the original program does not terminate. Type-T filters are a good choice when filter accuracy is the most important priority, or when it can be verified that the Type-T for a vulnerability meets any performance requirements (e.g., through static analysis).

6.2.2.2 Type-B Filters

A Type-B filter is a filter limited to Boolean formulas. Unlike Type-T filters, Type-B filters may not be perfect if the language of the vulnerability is not a Boolean expression. Filter evaluation on an input is performed by assigning the input values to the Boolean formula variables, and checking if the formula is then *true* or *false*. Inputs that evaluate to *true* are considered unsafe and subsequently filtered.

Representation	Generation	Filter Size	Operations		
			Filtering	Minimization	Equivalence
Type-T	poly(N)	poly(N)	Undecidable	Undecidable	Undecidable
Type-B	poly(N)	poly(N)	PSPACE - O(S)	exp(S)	exp(S)
Type-R	exp(N)	exp(N)	O(N)	O($S \log S$)	O($S \log S$)

Table 6.1: Summary of bounds for the three vulnerability filter representations we consider for a program of length N and filter size S. Generation time and filter size are shown for our techniques. poly(X) denotes a function polynomial in X, and exp(X) denotes a function exponential in X.

A Type-B filter is only well-defined to inputs of a predetermined finite length. If a Type-B filter has n input variables, then any input $> n$ is outside the domain of the filter. The filter defense system must then choose whether such inputs are unsafe or safe. In exchange for this limitation, Type-B filters have performance guarantees. For example, they are guaranteed to terminate.

In our approach the Type-B filter is the weakest precondition for the Type-T filter to return unsafe. The weakest precondition only considers loops a fixed number of times. The technique ensures that a Type-B filter is sound since it only returns unsafe for inputs that a perfect Type-T filter would return as unsafe.

6.2.2.3 Type-R Filters

Type-R filters are limited to regular expression. Type-R filters are evaluated by performing regular expression matching: if an input matches the filter, it is considered unsafe and filtered.

A regular expression filter may have a considerable error-rate because the language of most vulnerabilities is likely not regular. However, regular expression filters are widely used in practice because regular expression matching is efficient. Type-R filters are well understood, and are the primary mechanism of many commercial network-based filter defenses.

6.2.2.4 Comparison Between Filter Classes

Different defense systems will have different performance versus accuracy requirements. The trade-offs, in turn, affects when a particular filter language class may be appropriate. For example, network-based systems usually favor faster evaluation times, thus typically use Type-R filters. Host-based systems, on the other hand, are typically the last line of defense and thus require high accuracy.

Table 6.1 summarizes the trade-offs between each filter class for various operations. The operations include generation time, filter size, matching time, minimization time, and equivalence time. The generation and filter size are specific to our algorithm, while all other measurements are general bounds.

Filter generation takes in the vulnerability specification (Φ, v_p) and returns a filter. Smaller times are better, since they mean we can deploy a filter more quickly to defend against threats. The generation time for Type-T and Type-B filters is linear in the program size for our methods. We present two algorithms for generating Type-R, one of which will generate filters exponential in the size of the corresponding Type-B, and one which is linear in the size of the corresponding Type-T.

The filter size is the size of the filter with respect to the number of statements N in the original program. Again, smaller is better since filters are generally kept in memory for the filter defense system.

The filtering operation itself determines how quickly a filter will mark an input as either safe or unsafe. For Type-B filters, the filtering time depends upon whether we allow quantifiers. Quantified Boolean formulas are PSPACE-complete (assuming there are free variables in the filter formula), while unquantified Boolean formulas can be evaluated in time linear in the size of the filter.

Filter minimization takes a filter $\mathcal{F}$ and computes the smallest filter $\mathcal{F}_{min}$ in the same language class as $\mathcal{F}$. A minimized filter takes the least amount of space, and is generally more efficient to match. Filter equivalence is used to determine whether two filters $\mathcal{F}_1$ and $\mathcal{F}_2$ match the same language. Filter equivalence is useful in many scenarios, e.g., an administrator receives two filters from different parties and wants to know if they are for the same vulnerability.

6.2.3 Single and Multiple Execution Path Coverage

If the filter language is restricted to a language less powerful than the vulnerability language, then the filter will not have perfect accuracy. Soundness and completeness deter-

mine whether or not the filter makes a mistake. However, we may also want to quantify what exploits are filtered, and which ones may be missed.

We introduce the notion of vulnerability filter coverage in which a filter only matches exploits that execute a subset of vulnerable execution paths. The ability to consider a subset of paths to a vulnerability (as opposed to all program paths an exploit may follow) is important in the case when creating a filter for all program paths that lead to the vulnerability may be too expensive.

First, consider a filter that captures only single vulnerable execution path in the program. The single path may be chosen from a CFG, or the path executed by a sample exploit. The execution path is a straight-line program. A filter corresponding to a single path evaluates the vulnerability condition Φ at the vulnerability point v_p. Note that straight-line programs do not imply that only a single input leads to the vulnerability point: there usually exist many other inputs $x' \neq x$ that both reach the vulnerability point and the vulnerability condition evaluates to EXPLOIT (others have noted this as well [31]). For example, exploits usually have a payload which executes arbitrary attacker code. A straight-line program will return EXPLOIT for exploits with different payloads because the execution of different variants only differ *after* the vulnerability condition has been satisfied.

Multiple execution path coverage includes many different paths. For example, the whole program itself with Φ inlined at v_p is a multiple path coverage filter. However, many paths in the program may not be relevant to a vulnerability. The instructions relevant to the vulnerability constitute a sub-graph of the control flow graph. Intuitively, any instruction which could be executed from where input is read in to the vulnerability is included as a relevant instruction. We use chopping (see 3.4) to create a sub-program which contains only vulnerable paths up from where input is read in (the source node in the chop), to the vulnerability point (the sink node in the chop).

Potentially exploitable paths are a natural metric for determining the false negative ratio of sound filters. For example, the ratio of paths covered vs. not covered in a CFG by a filter gives a method for quantitatively comparing filters. Filters that cover more paths are better, since they will match more exploit variants.

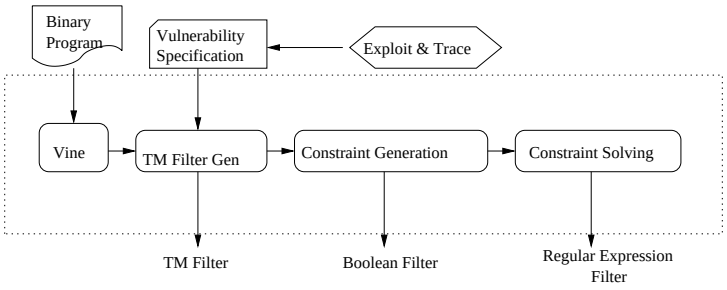


Figure 6.2: A high-level view of the steps to compute a vulnerability filter.

6.3 Approach and Techniques for Automated Vulnerability Filter Generation

6.3.1 Our Approach to Filter Generation

In this section, we describe our approach and techniques for generating vulnerability filters. We are given a vulnerable binary program P and the vulnerability specification (Φ, v_p). Our approach is:

1. Obtain (Φ, v_p) and lift P to the Vine IL.
2. Generate a Type-T filter by inlining the vulnerability condition at the vulnerability point and computing the *chop* of the program model. Stop if this is the final representation.
3. Generate a Type-B filter from the Type-T filter. Stop if this is the final representation.
4. Generate a Type-R filter from from the Type-B filter.

6.3.1.1 Step 1: Lift to Vine

The input to our filter generation algorithm consists of the vulnerable program P and the vulnerability specification (Φ, v_p). Our approach is ambivalent to how the vulnerability specification is determined. We discuss several possibilities for obtaining the specification below.

Previous work has shown how to generate (Φ, v_p) from the execution trace of a single sample exploit. We have shown that the information needed for our vulnerability specification can be derived from a sample exploit [83]. In [83], Φ is a taint-based policy that detects memory overwrite attacks. The specification is obtained from a sample exploit x by

first generating an execution trace $P(x) : \sigma$, then applying Φ on each step in the trace σ to determine v_p [83]. The definition of an EM-enforceable safety policy (Equation 4.1) guarantees that there will be one such instruction, and taint-based analysis is EM-enforceable. Similarly, Costa *et al.* have also proposed the Vigilante end-to-end system [30] which could also be used to obtain the vulnerability specification.

6.3.1.2 Step 2: Type-T Filter Generation

Our approach for generating a perfect Type-T filter is to create the filter by combining Φ with the vulnerable program. The combined program contains an inlined version of Φ when a respective exploit detector execution monitor would evaluate Φ. The combined program will return unsafe for exploits by constructions. In particular, we combine the logic of Φ up to v_p, at which point the combined program will return unsafe if Φ is true. When v_p can no longer be reached via any execution, the combined program returns safe.

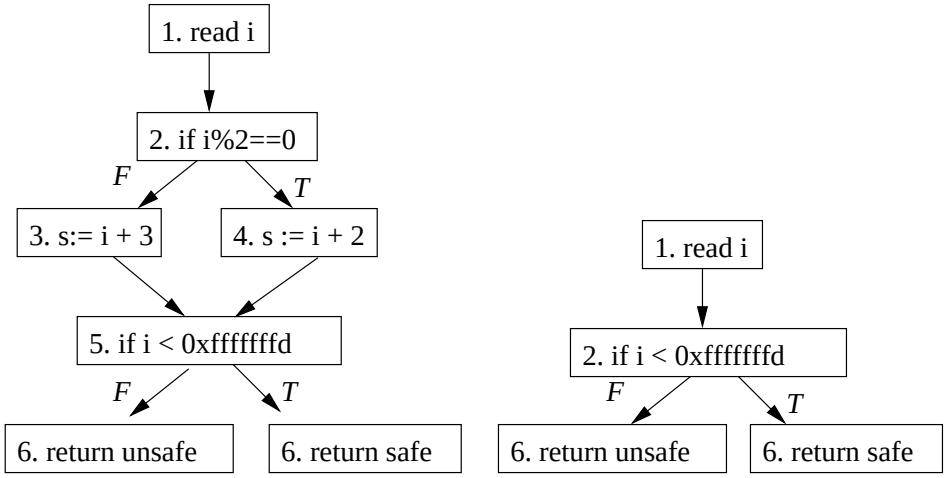
Given v_p, we first calculate the chop from where input is read in (the chop source) to v_p (the chop sink), using SCC-based chopping (see 3.4). The resulting chop will likely be significantly smaller than the initial program.

We next perform program optimizations over the chop. For example, we perform dead code elimination, global value numbering [95], value set analysis [13], constant propagation and folding, and other optimizations on the chop. Optimizations are successful, even on previously optimized code, because in the filter the only relevant operations are those that calculate an intermediate value for deciding whether Φ would be violated at v_p. Many statements in the original program, therefore, are likely to be dead code in the filter.

The final Type-T filter, since it is generated from the vulnerable program itself, will be perfect by construction. In the worst case, the chop will be as large as the whole program, and optimizations will be ineffective at removing any statements. This should not be surprising: the program itself could be the smallest recognizer, e.g., if the vulnerability is halting, a perfect filter may be running the program itself and seeing if it halts. Note also the correspondence between optimizations and making a compact filter: if we had a mechanism for creating a more compact perfect filter, we could turn that mechanism into an optimization routine.

Figure 6.3 shows the perfect Type-T filter we would generate for the running example from Figure 1.2a. Note that optimizations produce a final filter that consists of only a single check.

In our experimental evaluation, we show (the perhaps surprising) result that filters constructed with the above technique on real vulnerabilities are very compact and efficient.



(a) Before Optimizations. (b) After Optimizations.

Figure 6.3: The Type-T we would generate for the running example shown in Figure 1.2a before and after optimizations.

6.3.1.3 Type-B Filter Generation

We generate a Type-B filter from a Type-T filter. Our method for creating a Type-B filter is to write the Type-T filter as a Boolean formula. The filter is an approximation because the Type-B may have to statically approximate the effects of loops. In our approach, we create the Type-B filter by first identifying all loops, and then ensuring each loop is only executed a fixed number of times.

We ensure all loops are only executed k times by a straight-forward rewrite of the Type-T filter. All loops in a flow-graph can be reduced to a while-loop of the form `while(`b`){ instr*; }`. We rewrite while loops as:

```
i := 0;
while(b && i < k)
  instr*;
  i++;
```

where `i` is a fresh variable. Note the above is not needed if we can statically show that a loop will never execute more than a fixed number of times.

We then calculate the weakest precondition for the Type-T filter to return unsafe. The resulting filter is guaranteed to be sound (Definition 6.2.2). The filter is sound because it will, by construction, only return unsafe if the perfect Type-T filter returns unsafe.

The Type-B filter is sound, but may not perfect. The Type-B filter is well-defined only on all inputs that do not execute more than k iterations of any loop. Thus, the filter itself may miss exploits for inputs that require evaluating a loop more than k times. The filter defense decides whether inputs that execute loops more times are unsafe or safe. If zero false positives are the goal, then such inputs should be considered safe. If zero false negatives are the goal, then the inputs should be considered unsafe.

The Type-B filter we would calculate from the Type-T filter in Figure 6.3 is:

$$((i\%2) == 0 \Rightarrow (i + 2\%2^{32}) \leq i) \wedge ((i\%2) \neq 0 \Rightarrow (i + 3\%2^{32}) \leq i) \qquad (6.1)$$

6.3.1.4 Type-R Filter Generation

We create a Type-R filter from the Type-B filter for the vulnerability. Our basic method is an iterative approach where we generate the regular expression by taking the disjunction of generated exploits. An exploit is an input that satisfies Type-B (by the construction of the Type-B). We use a solver to find solutions to the Type-B, and the regular expression being the disjunction of all solutions found so far. For example, the regular expression filter for our running example is the literal string $2^{32} - 3|2^{32} - 2|2^{32} - 1$.

Suppose that the Type-B filter returns unsafe for the n byte input $x = x_1, x_2, x_3, ..., x_n$. Querying the decision procedure to find satisfying answers to the Type-B filter will return a satisfying answer $v = v_1, v_2, v_3, ..., v_n$ where v denotes a particular constant value. By construction, v is an exploit. We then iteratively query if $\mathcal{F} \wedge \neg v$ is satisfiable, i.e., if there exists a satisfying answer that is not one we already know about. When satisfiable, we get a new satisfying answer v', and the filter is $v \vee v'$, or in regular expression notation, $v|v'$.

Our approach to Type-R filter generation will return a new v on each iteration until we have exhausted the inputs that the symbolic constraint filter recognizes. This procedure is guaranteed to terminate since a symbolic constraint filter is over a finite domain.

Enumeration works, but may be inefficient when a variable may take on a range of values. For example, suppose x_1 is only restricted in the Boolean formula to values between 0 and 10. The above procedure would enumerate satisfying answers, meaning that it would take 10 queries to determine that 10 was the value range for x_1. In [22], we propose an analysis to address this issue. The analysis conservatively estimates the set of possible values for a variable at each program point.

The Type-R filter is sound, but may again not be perfect because it may not be complete. Our methods are the first to show how to generate a sound Type-R filter. The Type-R filter is not perfect because it is only well defined on n-length inputs. The filter defense decides whether inputs $> n$ are considered safe or unsafe.

6.3.2 Implementation Details

Many Different Kinds of Inputs. Our approach to filter generation assumes a single instruction that reads in inputs. When desired, we can always define a canonical input, with edges to each input location. In practice, however, a program may have many different kinds of inputs, and many different places where inputs may be read in. Further, we may want to generate a filter with respect to only a subset of input locations. For example, a program may read in a configuration file, set global variables, and only then read in user input. We may want a filter with respect to the point user input is read in, and not for all input variables.

We handle such cases by computing the chop (see 3.4) on the Vine program with respect to a designated input and vulnerability point. In an end-to-end system, e.g., Vigilante [30], the designated input location can be derived from a sample exploit execution. For example, taint-based analysis already assumes a set of known input locations for tainting purposes, and those same locations can be used for filter generation. The filter generator can also use the concrete state with respect to those input locations.

Efficient Signature Evaluation. For efficiency, in our implementation we compile down the Type-T filter in the Vine language to an executable program. This is done in two steps: first we translate the model to C++, then compile the C++ to an x86 binary, which is our input filter.

Simulated Memory. Type-T and Type-B filters may have expressions denoting memory accesses. An accurate filter may explicitly require modeling memory operations. For example, consider a vulnerability which is only exploitable if an alias relationship holds. Statically determining exact alias relationships is undecidable, thus static analysis is insufficient. Similarly, using information from a particular dynamic run is insufficient since aliasing may be different for different input values. Thus, we need to be able to model memory in the input filter itself so that we know the exact relationships for the specific input in question. We model memory by translating any memory operations in the final input filter into a *simulated memory operation*. Our approach maintains the invariant that

each step in input filter evaluation, the simulated memory accurately reflects what the real program's memory state would be at the corresponding step when run on the input.

We translate load and store operations in the model into loads and stores in the simulated memory. In our implementation, the simulated memory is implemented as a hash-map; the reads and writes to simulated memory are implemented as fast hash table look ups and store operations, respectively. Our implementation correctly takes into account endian issues and writes/reads of different sizes by first normalizing memory.

Simulated memory also provides us with input filter safety. It is not unusual for an application to have many vulnerabilities, perhaps some not yet discovered. When we build a model of the vulnerability, it may also include an implicit model of an unknown vulnerability. Simulating memory ensures that a vulnerability in the modeled application does not turn into a vulnerability in the input filter, since the model and how the model is evaluated are completely separate.

Handling Vine Limitations and the NOTSURE Filter State. In our implementation we specifically distinguish implementation limitations by introducing a NOTSURE state to the filter. The filter defense system can equate NOTSURE with either safe or unsafe as appropriate. For example, if we cannot resolve an indirect jump target, we return NOTSURE instead of including the entire program in the filter. Note using NOTSURE allows us to label our potential errors since inputs labeled safe are still guaranteed safe, and inputs labeled unsafe are still guaranteed exploits. In our evaluation, we found NOTSURE states were rare in filters for real-world vulnerabilities.

The Vine `special` Instruction. The Vine language uses the `special` instruction for calls to externally defined functions (see 2.2). Vine itself does not define the semantics of `special`. However, for Type-T and Type-B filters, we have more latitude since filters are evaluated dynamically. We explore four options: 1) call the external code directly, 2) use a summary that replicates the effects of the external code, 3) introduce the new filter state NOTSURE and translate `special` to return NOTSURE, and 4) find the external function and add it to the Vine IL.

Option 1, to call the external code directly, works well when that code does not have external side-effects or introduce auxiliary inputs. This is the case for many calls to utility functions, such as string processing libraries. To call external code, we write stubs that marshal the code's inputs and output between real and simulated memory and registers. These stubs generally need only be written once for common library functions and system calls, and can be reused for many input filters. We implement this approach for many

standard library functions.

Option 2, to write a function summary (as found in [23,24,29]), is preferable to option 1 when the external code has undesired side-effects, such as writing to the file system or network, or introduces auxiliary inputs. For example, if the filter contains a `special` instruction corresponds to writing a file, we prefer to use use a summary function rather than actually creating a file for filter evaluation.

In general, we prefer Option 1 for host-based filters since actually executing most calls makes it possible to check whether an input would exploit a program it its current state. For example, a web-server may only be vulnerable when a requested file is not present. If we use a summary, we have to choose at filter generation time whether to consider the case that the file exists or not. We can execute a call in the filter to check for the filter at evaluation time. This approach would guarantee that the filter is accurate with respect to the exact current state of the vulnerable system.

Option 3, we return NOTSURE to distinguish between cases when we know an input is safe, or when we know it is unsafe, and therefore a limitation of our implementation. We use this as the fall-back in our implementation, such as when no particular choice would ensure soundness.

The final option is to find the externally called function and adding it to the Vine IL for the vulnerability. The call is translated as `special` during lifting because the called code is not available. For instance, many `special` instructions arise from calls to functions in the C standard library. We could include the standard library as part of the Vine IL. The advantage is all further analysis is over all code that may be executed, not just the code in the vulnerable binary. In our evaluation (Section 6.4.3), we show that this option is often not efficient, however, because the size of the additional code is often too large.

Non-determinacy. Our implementation only handles deterministic vulnerabilities. Unfortunately, some real vulnerabilities are non-deterministic, such as MOBB 10 in our evaluation. In MOBB 10, a random number generator determines whether a particular run of the program will be exploitable. In our implementation, we cannot guarantee that a random number picked during input filter evaluation would match the random number picked by the actual program, thus we cannot guarantee soundness. Using NOTSURE is useful for this type of situation.

6.4 Evaluation

In this section we evaluate our techniques on real vulnerabilities. Each measurement is performed on a 2.4 GhZ Pentium D with 2 cores and 2GB of RAM.

6.4.1 Vulnerability Description

APEG Vulnerabilities. We evaluated our filter generation techniques on the vulnerabilities we created exploits for in 5 (see 5.3.1). We use the same vulnerable code paths as from APEG. We use the same code paths so that we can perform measurements on the same vulnerability for each filter type. This measurement, therefore, gives an end-to-end time for generating filters (instead of exploits).

MOBB Vulnerabilities. We next evaluated our techniques for Type-T and Type-B filter generation on a set of six additional web browser vulnerabilities. The client-end web-browser vulnerabilities come from the Month of Browser Bugs (MOBB) website [7]. These bugs are scripting vulnerabilities in Internet Explorer on Windows XP with Service Pack 2 installed, usually due to mishandled memory. A script is a small program written as part of a web page. Modern web-browsers allow for various scripting languages, such as JScript, JavaScript, and VBScript. Scripting languages extend basic HTML with the ability to call native methods on the browser's computer, e.g., ActiveX controls. When the web-browser renders a page containing a script, the script invokes the native method, which is executed with the privileges of the browser. Similar to any type of bug, scripting vulnerabilities are really a type of input-validation problem, where the browser fails to sanity check all inputs supplied by the web page.

We tested our input filters using the Canary web-browser defense system prototype from Symantec. Canary protects against vulnerabilities in Microsoft's Internet Explorer (IE) 6 on Windows XP SP2. Canary intercepts scripting calls to external modules. We insert our input filter at the interception point. The input filter checks whether the input arguments provided by the script are malicious or not. When a script is deemed malicious, our input filter returns unsafe and the script stops executing.

MOBB 10, 13, 15, and 22 each contain denial-of-service vulnerabilities which arise when a script does not properly initialize all data-structures. For example, MOBB 13 is based upon a module which allows PowerPoint-like transitions for images on a web page. This module is frequently used by web-masters to enhance functionality, e.g., to provide a slide show for pictures with transition effects. A script calls the method

Name	Generate (sec)	unsafe Match (μs)	safe Match (μs)
DSA_SetItem	0.53	46	20
GDI	3.64	105	38
PNG	0.66	26	14
IGMP	8.09	65	64
ASPNet_Filter	1.50	3	3
Average	2.88	49	28

Table 6.2: Type-T filters for the five vulnerabilities exploited by automatic patch-based exploit generation (see 5.3.1).

`put_Transition(object, int)` where the second `int` argument specifies the transition to use. The `object` passed in is assumed to have a table of virtual functions, one for each possible transition which can be specified for the int argument. If the object does not have that transition defined, the corresponding virtual function is NULL. However, `put_Transition` does not check for NULL, and a subsequent de-reference crashes the browser.

We use the vulnerability point as specified in the crash message on the MOBB website [7], e.g., for MOBB 13 it is in dxtmsft.dll at `CDXTRevealTrans::put_Transition+0x3a`. The vulnerability condition is `EAX == 0` (since the `EAX` register holds the address at the vulnerability point) for MOBB 10, 13, 22, and 25. The vulnerability condition is `ECX == 0` for MOBB 15.

MOBB 19 arises from a vulnerability in a Microsoft Office ActiveX control, which can be invoked by a script in a browser. The same vulnerability appears in two different versions of MS office. We refer to these vulnerabilities as `owc10` and `owc11` since these are the vulnerable DLL names. The vulnerability point is again taken off the MOBB website. In each case, the vulnerability condition is a range check to see that referenced memory at the vulnerability point should be within bounds of a valid object.

Name	Simple Instrs			Memory Ops		
	Pre	Post	%	Pre	Post	%
DSA_SetItem	534	160	-70%	37	33	-11%
GDI	1376	316	-77%	87	83	-5%
PNG	384	97	-75%	19	18	-5%
IGMP	2620	823	-69%	165	81	-51%
ASPNet_Filter	4849	1122	-77%	381	0	-100%
Average	1953	578	-70%	285	43	-85%

Table 6.3: The size of various size aspects of the Type-T filter before and after optimizations on the chop.

6.4.2 Type-T Filter Evaluation

6.4.2.1 Type-T Filter Evaluation for APEG Vulnerabilities

We first evaluate generating Type-T filters for the vulnerabilities from automatic patch-based exploit generation in 5.3.1. In order to evaluate how quickly we can create a comparable filter, we use the same vulnerable code paths as in Chapter 5.

Table 6.2 show our results. The first column shows the name of the vulnerability. The second column shows the number of seconds to generate a filter. The faster we can generate a filter, the faster a filter-based defense can protect against exploits. The longest it took us to generate a filter was 8.09 seconds. The shortest was .53 seconds.

We also evaluate how long the filters took to match both exploits. The filter that took the longest to match an exploit was the GDI filter, at 105 microseconds. The shortest was the ASPNet_Filter, at 3 microseconds. Finally, we evaluate how long it took the filters to match safe inputs. The safe input numbers give a sense of the best-case overhead for filter checking. The filter that took the least amount of time was the ASPNet_Filter vulnerability, again at 3 microseconds. The longest was the IGMP vulnerability at 64 microseconds.

Filter Accuracy. The filters are guaranteed to filter any exploit that could have been generated with our APEG results. This is guaranteed by our approach to filter generation.

Name	Generate (sec)	unsafe Match (μs)	# NOTSURE
MOBB 10	1.52	13	0
MOBB 13	1.32	15	1
MOBB 15	5.97	11	0
MOBB 19 (owc11)	9.42	6	0
MOBB 19 (owc10)	9.64	6	0
MOBB 22	1.35	10	0
Average	4.87	10.12	-

Table 6.4: Summary of Type-T filter generation results for MOBB Vulnerabilities.

Optimizations. We also measured the effects of program optimization on the final filter size. Table 6.3 shows our results for filter size before and after optimizations. On average, the final filter size is 578 simple Vine IL instructions. A simple instruction is an instruction that has no recursive expressions, e.g., `z= 5 + x` is allowed, but `z = 5 + 10 + 7` is not since the latter involves 2 nested binary addition operations. The average reduction in filter size was 70% due to optimizations. We also specifically measure how many memory operations optimizations remove. Memory operations are expensive in the filter since all operations must be simulated (See 6.3.2). On average, optimizations removed 85% of all instructions. If we leave off the best case, ASPNet_Filter, the average savings is still 30%. Thus, we conclude optimizations significantly improved overall filter size and performance.

6.4.2.2 Type-T FIlter Evaluation for Web-Browser Vulnerabilities

We next generated Type-T filters for the 6 MOBB vulnerabilities. The filter is for all vulnerable program paths. Table 6.4 shows our results. The generation time for filters was again small. The longest a filter took to evaluate was 9.64 seconds for MOBB 19 (owc10). The best case was 1.32 seconds for MOBB 13. We then measure matching times for unsafe. We only show matching time for unsafe because it is an upper bound on filter evaluation time. The longest matching time was 15 microseconds, while the smallest was 6 microseconds. These times are well within the range to be considered realistic for commercial host-based filter defense systems.

Name	Exec. Stmts			Mem Ops		
	Pre	Post	%	Pre	Post	%
MOBB 10	565	93	-84%	20	10	-50%
MOBB 13	1080	99	-91%	120	22	-82%
MOBB 15	402	25	-94%	11	7	-36%
MOBB 19 (owc10)	157	7	-96%	19	2	-89%
MOBB 19 (owc11)	118	7	-94%	11	2	-82%
MOBB 22	574	86	-86%	81	20	-75%
Average	483	53	-89%	44	11	-75%

Table 6.5: The benefit of optimizations to Type-T filters for the MOBB vulnerabilities.

Filter Accuracy. Recall from Section 6.3.1.2 that we introduce the NOTSURE state which roughly corresponds to implementation limitations. We also measured the number of NOTSURE states in our generated filters. Our only NOTSURE state arises from the MOBB 13 filters from a `special` instruction corresponding to a system call.

In MOBB 13, the NOTSURE state is from a `special` instruction corresponding to a Windows API call for obtaining random numbers. Recall MOBB 13 performs a PowerPoint-like transition for web images based upon an integer argument to the vulnerable function. Any integer above 25 indicates a random transition should be taken. The vulnerable function implements this by calling the random number generator to select a transition, then looking in a table based on the random number to locate the appropriate function to implement the transition. The code is only vulnerable when the random number generator picks a transition that is undefined for the passed in object. Thus, any test we insert could potentially lead to unsoundness since the browser and the filter do not have access to the same random bits, thus the filter may not be representative of what the browser may do. In this case we return NOTSURE.

A filter that does not contain NOTSURE is guaranteed to not only be sound, but also *complete*. That is, the filter will return EXPLOIT for *all* input which exploit the vulnerability, while still retaining the zero false-negative guarantee.

We verified that each filter was able to detect the example exploits that were available to us, as well as manually created variations. The only filter that can return NOTSURE is for MOBB 13, which returns NOTSURE only when the calling script specifies to choose a random transition.

Name	Efficient WP		Classic WP	
	Size	Time (s)	Size	Time (s)
DSA_SetItem	11150	0.080	9688	0.085
ASPNet_Filter	7281	0.125	716071	23.234
IGMP	31588	0.348	1535700	15.409
GDI	9480	0.161	53115	0.510
PNG	6248	0.052	805	0.008

Table 6.6: Type-B Size and Time to Generate

Optimizations. Table 6.4.2.2 shows the effect of optimizations on the MOBB filters. On average, optimizations reduced the filter size by an average of 89%. The number of memory operations was reduced by 75% on average. Thus, we conclude optimizations significantly improved overall filter size and performance.

6.4.3 Type-B Filter Evaluation

We evaluated our Type-B filter generation techniques on the five APEG vulnerabilities. Table 6.6 shows our results. The second column in the table measures the size of the generated filter using the efficient weakest precondition algorithm from 3.3. The size is calculated by measuring the number of nodes in the expression tree. For comparison, we also provide the size of the Type-B if we used the classic algorithm from 3.2. The fourth column is how long it took to generate the formula from the Type-T filter using the efficient algorithm. Both experiments were performed on the exact same optimized Type-T filter. [1]

The efficient weakest precondition algorithm created a smaller Type-B filter most of the time, and created it faster all of the time. Note that although in specific cases the efficient weakest precondition algorithm may be larger (e.g., from expansion due to SSA), it is still likely to be a better choice since it guarantees the filter will be at most quadratic, while the classic case may be exponential. In our experiments, the classic algorithm produced a Type-B filter on average 5.94 times as large as when using the efficient weakest precondition algorithm. The time to generate the Type-B filter was on average 28.63 times faster using the efficient algorithm.

[1]Note that the times here are to generate the Type-B filter from the optimized Type-T filter. The formula generation times in APEG 5.3 are longer because they include optimization time.

Function	**# Stmts**	Efficient WP	Classic WP
gethostbyname	7.54×10^{13}	8.17×10^{13}	6.62×10^{18}
perror	5.27×10^{12}	5.71×10^{12}	1.59×10^{18}
realloc	1.69×10^{10}	1.83×10^{10}	4.04×10^{18}
strerror	1.32×10^{12}	1.43×10^{12}	5.37×10^{18}

Table 6.7: Measurements for how much complexity various glibc functions add to the overall signature.

6.4.3.1 Library Calls and Type-B Filters

Recall from 6.3.2 that the Vine IL for a vulnerability may include calls to external libraries. There were four choices for external calls: return NOTSURE, call the external code directly, write a function summary, or locate the external call and add it to the Vine IL. For Type-T filters, all choices are possible. However, since Type-B filters are Boolean formulas, we really only have three options: return NOTSURE, write a function summary, or add the Vine IL for the external call.

One may think that adding the external code is the best answer. Unfortunately, that tends to be impractical in many common cases. Table 6.7 shows how much code we would have to add for several common function calls. We show three different numbers. First, we show the size in terms of number of statements added for the call. Second, we show the size using the efficient weakest precondition from 3.3. Finally, we show the size using the classic weakest precondition algorithm from 3.2 (which is equivalent in size to forward symbolic execution).

By using the efficient weakest precondition we get a huge savings over the classic algorithm. However, the increase in size, is still enormous. The reason is that library calls call are inter-dependent and call one another. For example, `realloc` calls other routines, such as `sprintf`, as part of error-handling. `sprintf` in turn calls more glibc functions, and so on. Thus, calling a single function in glibc tended to result in most of glibc being recursively added.

Thus, for Type-B filters, we expect that most of the time function summaries will be used, or when unavailable, the filter will return NOTSURE [24]. This conclusion is supported by subsequent work by Costa *et al.* [29] where function summaries are used to decrease the size of their Type-B filters.

6.4.4 Type-R Filter Evaluation

Type-R filter generation is performed by giving the Type-B filter to a decision procedure. The solver we use in Vine, STP [46], currently does not support the syntax of efficient weakest preconditions. In particular, it does not support equating memories. Thus, we use the less efficient weakest precondition formulation for generating Type-R filters from Type-B filters.

In 5.3, we show the time to generated satisfying answers to the Boolean formulas, i.e., the Type-B filter, for the 5 vulnerabilities. The time to generate a single satisfying answer ranged from 0.01 seconds (GDI in Table 5.1) to 182.91 seconds (DSA_SetItem see Table 5.2).

6.5 Discussion

6.5.1 Comparison with Manually Generated Filters

While it is obvious that manual input filter generation is slower than our automated approach, we also found that hand-written input filters are also inaccurate. We obtained the hand-written filters for the MOBB vulnerabilities from a large security company. The authors of the hand-written input filters share our desire for zero false positives. We compared the hand-generated filters to our automatically generated filters. We found several instances where the manually generated input filters had (confirmed) false positives, while our input filter did not. Interestingly, although the goal of the input filter author was zero false positives at the expense of false negatives, the manual input filters tended in reality to be the opposite: have zero false negatives but considerable false positives.

We also experimented with *automatically* finding deviations where our generated input filter was different than the hand-written input filter. In order to do this, we translated the manually-generated input filter for MOBB 10 to the Vine IL, producing a model F_m (manual filter). We then calculated the weakest precondition F_m for the manual filter to return exploit, and similarly calculated F_a for the automatically created filter. We then queried STP on the formula $F_m \neq F_a$. A satisfying answer is an input accepted by F_m but not F_a, or vice versa, i.e., what exploits one filter would recognize but the other does not. Any inputs accepted not accepted by F_m, but accepted by our perfect filter F_a, is a false negative. Similarly, we can find false positives. In this experiment, STP exposed a false positive in the manually generated input filter, but no false negatives (verifying the manually generated input filter had zero false negatives but does have false positives).

Name	Gen Time	Eval Time	False negatives	False Positives
Bouncer	$>$ 24 hrs	1.4×10^{-6}	Yes	No
Our Approach	1.59 sec	250×10^{-6}	No	No

Table 6.8: Comparison between our method for Type-T filter generation and Bouncer [29] for the `ghttpd` vulnerability in Bouncer.

Specifically, the input value 0 for MOBB 10 was a false positive in the manually generated input filter, but not for our input filter. We confirmed this using an unpatched browser.

6.5.2 Iterated Single Path Vulnerability Input Filters

The most closely related work to ours is Bouncer [29], which is the only other system that creates sound input filters based upon analyzing multiple program paths. [2] Bouncer handles memory safety input-validation vulnerabilities.

One significant difference is Bouncer requires *source code* alias information [29, Section 5.2]. However, since we wish to make filter generation possible when source code is not available, their results are not directly applicable to our problem setting. For each execution path, Bouncer generates a filter using information from a single execution to a false-positive free filter [29]. The source-level alias information is needed to remove unnecessary conditions from each path filter. Without the alias information, all conditions executed may be in the path filter. Note that they cannot simply add alias analysis at the binary level without building a static analysis infrastructure such as Vine.

The dynamic approach in general, including the one used by Bouncer [29], has worse bounds on filter size, filter generation time, and filter evaluation time. First, Bouncer enumerates program paths. Let b be the number of branches considered in an acyclic program, e.g., a program executed in a finite number of steps. Then there are $O(2^b)$ branches to explore all possible vulnerable paths in the worst case. The final filter is the disjunction of each path filter. (In Bouncer, the authors do some post-processing on the resulting disjunction, though it is unclear exactly what post-processing is done.) The total filter size and evaluation time is $O(2^b)$.

[2] Note that it is sometimes thought that Bouncer will find new vulnerabilities. This, however, is not correct. Bouncer performs dynamic analysis across multiple paths for the same vulnerability [29, Section 7].

In practice, Bouncer runs into two of these issues: filter generation is slower than our approach, and filters are larger. We generated a filter using our techniques for the `ghttpd` vulnerability discussed in Bouncer. The comparison is shown in Table 6.8.

In Bouncer, filter generation exceeded the 24-hour time limit for `ghttpd`. Overall, in Bouncer two out of four experiments exceeded the 24-hour time limit, and the remaining two took hours [29]. Our static approach considers all paths (since it is based on the CFG), and took a 1.2 seconds. Bouncer measures size by the number of "conditions" in the filter. Bouncer's resulting input filter, after simplification, had over 2000 conditions [29]. In our approach, a condition should correspond to a conditional jump (it is unclear what a condition means in Bouncer). Our filter had only 73 conditions.

Bouncer reports better filter evaluation performance for `ghttpd`. Costa *et al.* report filter evaluation takes between about .8 to 1.5 microseconds [29], depending on the length of an exploit. In our case, filter evaluation takes 250 microseconds. Although we cannot be sure why the filters are faster in Bouncer, there are several mitigating factors that may explain the difference. First, as mentioned, Bouncer uses alias information recovered from the source code level, and alias information could be used to remove memory operations in the filter (note the dynamic approach also need to consider memory operations, since a memory addresses are commonly dependent upon the input, e.g., table look-ups). Second, our filters for `ghttpd` use stubs, which can execute slowly due to the marshaling described in Section 6.3.2. Bouncer uses function summaries. If we switched, we would likely see speedups. Third, even though there are fewer conditions in our filter, the conditions derived by Bouncer may be faster to evaluate. For example, alias information used by Bouncer may remove memory loads, which are relatively expensive. Fourth, further optimizations in Vine could likely reduce filter evaluation time. At a high level, the information recovered with the dynamic approach in Bouncer is just static analysis of straight-line code in our setting, thus we could perform such optimizations. [3]

6.5.3 Implementation Discussion

There are 30 different MOBBs, with about 25 amenable to the defense architecture we used to test MOBB filters. We did not target bugs specific to OS X, Gentoo, or Windows 2000 bugs because the defense system we used to test browser bug filters did not support these platforms. Our analysis infrastructure, however, should port to other OSes with little or no modification since it is geared at x86, not a specific OS.

[3]Note that whatever constants used in a dynamic approach could similarly be used as part of constant propagation and folding in a static approach.

Of the 25 MOBBs possible, we targeted vulnerabilities that did not require significant inter-procedural analysis. This is because of an implementation choice: we currently inline functions so that we do not have to perform inter-procedural analysis. Inter-procedural variants of our analysis are straight-forward to implement. However, it does require a more extensive implementation than our current system. We manually looked through the remaining applicable MOBBs, and did not note any serious problems given an inter-procedural infrastructure.

It would be useful to combine dynamic and static approaches in future work. For example, if we cannot resolve statically an indirect call, we could use dynamic execution to try and figure out at least a subset of possible targets.

As discussed in Section 6.3.2, an important problem is dealing with vulnerabilities that are dependent upon side-effects. For example, consider a vulnerability that requires creating and then reading from a file, and there is no way around this if we want an accurate filter. In our approach, we would return NOTSURE since we cannot model these side-effects. Other defense approaches may be more appropriate. A similar example is MOBB 13, where a vulnerability depends upon the output of a random number generator: the random number returned to the filter may be different than the vulnerable program. In cases such as these, where the vulnerability is non-deterministic, one approach is to let the filter and vulnerable program share the same random bits. We leave investigating such issues to future work.

6.5.4 Type Safety

Our work is orthogonal to type safety; even type-safe programs can have input validation bugs that can benefit from filtering. For example, a type-safe language could mistakenly process inputs that cause it to enter an infinite loop, as in the IGMP vulnerability (as described in 5.3.1).

6.6 Related Work

In security, filters are often called signatures. We avoid this term because it is overloaded, e.g., signatures in cryptography are a very different thing that what we refer to as "filters".

Shield proposes a framework for manually creating vulnerability-based filters by modeling network protocols [103]. Modeling a network protocol can be advantageous since exploits are often understood in terms of particular message fields, e.g., an overly long

URL. However, Shield filters are manually generated. When available, such protocol information could be used in a complementary approach where protocol information is used to help guide static analysis (e.g., provide invariants enforced by the protocol) during filter generation. Overall, the advantage of binary analysis is that the generated filter is guaranteed to be faithful to the vulnerability as it appears in the program.

Vigilante [30] is an end-to-end system which generates filters for a single path. They rewrite the execution trace as a Boolean formula, thus this type of filter is a Type-B filter in our hierarchy. Vigilante shows that this important point in the design space can be effectively used in an end-to-end defense system. Our work generalizes the approach to include vulnerabilities which can be exploited via multiple paths. In addition, we show how to take an execution trace and produce a sound regular expression (and other filter types). Note that Vigilante does not produce Type-R or perfect Type-T filters. Since Type-R filters are typically better suited for network-based defense, our techniques are likely to be of interest to Vigilante and similar end-to-end systems. Similarly, our Type-T filters are of interest to host-based systems. We empirically demonstrate this for Symantec's Canary web-browser defense system with our MOBB filters (See 6.4.2.1).

Crandall *et al.* propose techniques for identifying tokens that must appear literally in an input string to exploit a given vulnerability, and use their techniques to perform an in-depth analysis of several vulnerabilities [31]. However, they do not actually generate filters. They conclude that "token-based byte string filters composed of smaller sub-strings are only semantically rich enough to be effective for content filtering if the vulnerability lies in a part of a protocol that is not commonly used." We believe this observation is apt; syntactic details of input strings may be insufficiently general for some vulnerabilities. This observation motivates the need for generating filters in more expressive language classes (than regular languages), such as Type-T and Type-B filters.

Machine Learning Techniques Several researchers have proposed generating network filters via machine learning techniques [57, 59, 63, 64, 81, 84, 96, 104, 106]. However, the down side of pattern extraction-based filter generation is the vulnerability itself is never analyzed, only exploits supplied by the attacker. As a result, many of these techniques are either incapable of handling exploit variants in practice, e.g., [63, 82, 86]. Venkataraman *et al.* [102] show the limits using any machine learning technique for filter generation in an adversarial environment.

Identifying exploits using semantic information is used by other automatic filter generation approaches [34, 83, 108]. These approaches generally have different goals, and do not provide guarantees on filter accuracy. It may be possible to combine these approaches with our vulnerability-centric static analysis to create a hybrid system.

Binary Analysis We are the first to propose techniques for generating filters based upon static program analysis, and binary analysis in particular. Since dynamic analysis is static analysis of straight-line code, our techniques generalize dynamic approaches [29, 30, 83].

Binary Rewriting and Protection Filters are run before an input is given to an application. Another approach to protecting vulnerable programs is to rewrite the binary code to detect the error and fail at run-time, e.g., [9, 27, 75, 83]. This protects the program by terminating it. Our goal here is to allow the vulnerable program to function during an attack.

The vulnerability condition and vulnerability point may help automatically generate patches at the binary level [93, 94]. Automatic patch generation has a different goal: figure out the right thing to do to prevent an exploit from causing a bad execution, or recover from the execution when it first starts to go wrong. Self-healing techniques ([56] contains an overview of recent advances) are also of interest as a defense, but are complementary to the scope of this work. In many circumstances it will still likely be advantageous to filter exploits from the input stream.

6.7 Conclusion

We presented a general framework for generating filters for input validation vulnerabilities. Given a single sample exploit, we presented techniques for automatically generating a filter with guarantees. Previous work could not make such statements because they lacked the proper framework, definitions, and formalism. We discuss three distinct restrictions we may have on the filter language: Type-T, Type-B, and Type-R filters. We provide theoretical and practical insights into these three filter representations. We then propose filter generation techniques for real-world vulnerabilities for each filter class. Our evaluation indicates that our approach produces better results than manually generated filters as well as filters generated automatically by previous approaches.

Part III

Conclusion

6.8 Perspective on Results

In this section, we discuss our opinions and perspectives on our work.

Vine and the Basic Algorithms. Many of the challenges addressed by Vine are engineering in nature. These engineering challenges are similar to those found when designing any large-scale code analysis framework. We have learned a couple of things. First, the right abstraction within the framework, e.g.,. the IL, often seems obvious and straightforward in retrospect. However, when developing Vine it was often unclear, what abstractions were correct. Several times we had to go back and refine our IL, front-end, and our back-end. We continue to learn from using our implementation and using Vine to address security ideas in real code.

In chapter 3, we present our adaption of algorithms for calculating the weakest precondition and chopping. We adapted algorithms which were known to be the best at least in one sense. Our primary intention was to see how far we could push these algorithms on binary code. Binary code serves as a good test-bed because there are fundamentally few abstractions.

One lesson we take away from our experience is that the efficiency and the speed of each algorithm is dependent upon the overall framework. For example, how quickly and thoroughly we can perform an optimizations depends upon the quality of the chop. The speed at which the weakest precondition can be solved depends heavily upon the optimizations we perform before-hand. Thus, simply looking at the performance of the algorithms by themselves is only a small part of the picture. A much better understanding is gained by seeing how the algorithms interact when solving a larger problem.

Input Validation Vulnerabilities. In chapter 4, we define the notion of an input validation vulnerability. One reason for doing so is to add a level of precision to our approaches. There often seems to be disagreement (even among security researchers) of the definition of a vulnerability, an exploit, and so on. In our experience, it is common for someone to define the notion of a vulnerability or an exploit in a limiting manner. For example, we are often asked about creating a control hijack for an exploit. This sort of question implicitly assumes that only bugs that allow control hijack are vulnerabilities, while not considering information disclosure vulnerabilities for example.

Our definition is instrumental throughout APEG and filter generation. The definition of input validation vulnerabilities helps explain why APEG is successful in practice: the validation checks are missing in P but added in P'. In filter generation, the definition

leads to a method for constructing filters where we essentially inline the EM safety policy. Further, the definition yields important properties of these filters, e.g., that the vulnerability description can be obtained from an exploit sample, that filters are composable, etc.

Automatic Patch-Based Exploit Generation. There are several different interpretations of our results and approach for APEG. From a security standpoint, they show that patches can hurt security, and we thus need to redesign patch distribution architectures. From a program verification standpoint, one message is to focus on the difference between program versions. Program verification often has trouble scaling to entire programs. However, techniques that do not scale for an entire program may still be of value since we only care about a single program point.

Another message from a program analysis standpoint is that even when we restrict our techniques to a single point in the program, state-of-the-art program verification algorithms may still not scale. Our mixed approach for generating exploits is an example where we must mix algorithms in order to get results in real problem instances.

Filter Generation. In Chapter 6, one of the main challenges was to formulate filter generation in as precise and consistent manner as possible. In retrospect, our approach and techniques may seem simple, and perhaps even obvious. On the other hand, years of research had not address issues such as how we can guarantee filter accuracy, can we generate a perfect filter, etc. One possible explanation is that everything flows out naturally because the problem is stated more precisely than previous work. In many cases, even though the technique may seem straight-forward, the result often was not. When we first went about defining Type-T filters, we thought they would only be of theoretic interest as a standard for perfect filters. However, it turned out that in practice perfect filters are often efficient. We initially thought regular expression filters would be of the most important, in part because regular expression filter defenses are extremely popular. However, the vulnerability language for most vulnerabilities we have seen is not regular, and any regular expression is therefore likely to have either a high false positive or false negative rate.

6.9 Common Techniques

At a high level, both APEG and filter generation share many similarities. This correlation should not be surprising: a filter catches exploits, thus a mechanism for generating filter matches is a likely a way to generate exploits.

Automatic patch-based exploit generation uses a patch to locate the vulnerability. Automatic filter generation uses a single sample exploit. Once a vulnerability is located, both then start out by performing a chop. A Type-T filter is built directly on top of the chop. Then, we create Boolean formulas in both approaches where the formula describes the conditions to exploit the vulnerability. Satisfying answers to the formula are exploits. We use a decision procedure to enumerate satisfying answers. The disjunction of exploit strings is a Type-R filter.

At a high level, the core techniques include: a) chopping to isolate paths of interest for security analysis, b) creating a predicate using the weakest precondition, and the c) finding solutions to the formula. In essence, we are formulating security problems as program verification and program analysis problems. Vine provides a way of accomplishing this work-flow on binary code. Other projects enabled by Vine (see 6.11) take advantage not just of the IL or program analysis, but also of the overall work-flow enabled by a binary-centric approach to software security.

We expect filter generation, APEG, and many other similar tasks will benefit from improvements to these techniques. Our work on efficient weakest preconditions (See 3.1) was motivated by this observation. We have also worked on reducing the time to find satisfying answers to the generated formulas. In particular, our work on optimizing the program before generating formulas is geared towards making the formulas themselves easier to solve. Our work in APEG showed that common program optimizations could significantly reduce the time to solve the formula (See 5.3). Presumably the solver could be improvement by implementing such optimizations.

6.10 Static vs. Dynamic Analysis

Our work takes a static approach to security-relevant program analysis where we analyze a binary program. A second popular approach is a dynamic approach. A dynamic approach runs the program on a single input, and then performs analysis on the code that was executed.

Thus, one way of viewing dynamic analysis is static analysis of straight-line code. The code is straight-line since it only includes a specific execution path. The central difference between a static and dynamic approach from this perspective is that in the dynamic case we use an input to select the exact execution path.

We can use information from a dynamic run as part of static analysis. For example, we can unroll loops for the weakest precondition the fixed number of times as executed

in a dynamic run. As another example, we can perform static analysis with respect to the program state at the beginning of a dynamic run.

6.11 Other Work Using Vine

In this section, we briefly discuss several other projects that are using Vine and the techniques developed in this thesis.

Automatic Deviation Detection. Many network protocols and services have several different implementations. Due to coding errors and protocol specification ambiguities, these implementations often contain deviations, i.e., differences in how they check and process some of their inputs. Automatically identifying deviations can enable: 1) the automatic detection of potential implementation errors and 2) the automatic generation of fingerprints that allow to distinguish among implementations of the same network service. The main challenge in this project is to automatically find deviations without requiring access to source code. Note this is similar to APEG, where we use the difference between two programs versions to generate exploits, and to how we found errors in manually generated filters in Section 6.5.1.

However, the binaries themselves encode exactly the protocol implemented. Brumley *et al.* develop techniques for taking two binary implementations of the same protocol and automatically generating inputs which cause deviations [19].

BitScope: Automatically Dissecting Malware The ability to automatically dissect a malicious binary and extract information from it is an important cornerstone for system forensic analysis and system defense. Malicious binaries, also called malware, include denial of service attack tools, spamming systems, worms, and botnets. New malware samples are uncovered daily through widely deployed honeypots/honeyfarms, forensic analysis of compromised systems, and through underground channels. As a result of the break-neck speed of malware development and recovery, automated analysis of malicious programs has become necessary in order to create effective defenses.

Brumley *et al.* have proposed BitScope, an architecture for systematically uncovering potentially hidden functionality of malicious software [20]. BitScope takes as input a malicious binary, and outputs information about execution paths. This information is then be used by supplemental analysis designed to answer specific questions, such as what behavior the malware exhibits, what inputs activate interesting behavior, and dependency

between inputs and outputs.

Replayer: Sound Replay of Application Dialogue The ability to accurately replay application protocol dialogs is useful in many security-oriented applications, such as replaying an exploit for forensic analysis or demonstrating an exploit to a third party. A central challenge in application dialog replay is that the dialog intended for the original host will likely not be accepted by another without modification. For example, the dialog may include or rely on state-specific to the original host such as its host name, a known cookie, etc. In such cases, a straight-forward byte-by-byte replay to a different host with a different state (e.g., different host name) than the original observed dialog participant will likely fail. These state-dependent protocol fields must be updated to reflect the different state of the different host for replay to succeed.

Newsome *et al.* propose the first approach for soundly replaying application dialog where replay succeeds whenever the analysis yields an answer [79].

Sting: An Automatic Defense System against Zero-Day Attacks. Worms such as CodeRed and SQL Slammer exploit software vulnerabilities to self-propagate. They can compromise millions of hosts within hours or even minutes and have caused billions of dollars in estimated damage. How can we design and develop effective defense mechanisms against such fast, large scale worm attacks?

Newsome *et al.* have designed and developed Sting [80, 101], a new end-to-end automatic defense system that aims to be effective against even zero-day exploits and protect vulnerable hosts and networks against fast worm attacks. Sting uses the vulnerability filters proposed as part of this thesis.

Polyglot: Automatic Extraction of Protocol Message Format. Protocol reverse engineering, the process of extracting the application-level protocol used by an implementation, without access to the protocol specification, is important for many network security applications. Currently, protocol reverse engineering is mostly manual. For example, it took the open source Samba project over 10 years of work to reverse engineer SMB, the protocol Microsoft Windows uses for sharing files and printers [100].

Caballero *et al.* have proposed that binary program analysis can be used to aide automatic protocol reverse engineering [25]. The central intuition is that the binary itself encodes the protocol, thus binary analysis should be a significant aide in extracting out the protocol from the binary itself.

6.12 Conclusion

We have shown the security benefits of a principled approach to vulnerability analysis and defense on binary code. In order to demonstrate this approach, we developed an architecture, called Vine, for performing security-relevant binary program analysis. Vine is designed to perform faithful static analysis of binary code. De-compilers, disassemblers, and other binary analysis tools tend not to be faithful to the semantics of binary code. For example, we show a simple, three line program that many other tools will not be able to faithfully analyze.

We have also defined the class of input validation vulnerabilities. We then use Vine to tackle input validation vulnerabilities in binary code.

We show the somewhat unintuitive result that releasing a patch may actually decrease security. In particular, we have shown that a person who receives a patch can reverse engineer the fixed vulnerability and create an exploit. Thus, those who first receive a patch have a significant security advantage. One of the immediate consequences of our work is that current patch distribution schemes should be redesigned.

We also have shown how to automatically defend against input validation vulnerabilities. We have developed methods for automatically generating filters which filter out exploits for input validation vulnerabilities. Input-based filtering is a widely used defense mechanism which introduces a filter check step before a vulnerable application receives an input. Inputs that are matched by the filter are subsequently dropped. Our techniques address four primary limitations of previous approaches. First, filter accuracy is guaranteed by the filter construction algorithm. Second, the filter generation algorithm cannot be fooled by attackers. Third, our techniques are efficient. Fourth, our techniques are more widely applicable because they provide an explicit mechanism for addressing the fundamental accuracy versus performance trade-off.

Overall, the techniques, tools, and new approaches we have developed in this thesis show not only can we better understand vulnerability analysis in binary code, but are also applicable to a wide variety of security problems.

Bibliography

[1] CVC Lite documentation. http://www.cs.nyu.edu/acsys/cvcl/doc/. Page checked 7/26/2008.

[2] The DOT language. http://www.graphviz.org/doc/info/lang.html. Page checked 7/26/2008.

[3] Metasploit. http://metasploit.org.

[4] The SUIF 2 compiler system. http://suif.stanford.edu/suif/suif2/index.html. Page checked 7/27/2008.

[5] TEMU. http://bitblaze.cs.berkeley.edu/temu.html. Page checked 7/27/2008.

[6] On the run - building dynamic modifiers for optimization, detection, and security. Original DynamoRIO announcement via PLDI tutorial, June 2002.

[7] Month of browser bugs website. http://browserfun.blogspot.com, 2006.

[8] The BitBlaze binary analysis project. http://bitblaze.cs.berkeley.edu, 2007.

[9] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 340–353, New York, NY, USA, 2005. ACM Press.

[10] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 2nd edition, 2007.

[11] Andrew Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[12] ARM. *ARM Architecture Reference Manual*, 2005. Doc No. DDI-0100I.

[13] Gogul Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Computer Science Department, University of Wisconsin at Madison, August 2007.

[14] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86 - a platform for analyzing x86 executables. In *Proceedings of the International Conference on Compiler Construction*, April 2005.

[15] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, pages 1–18, 2001.

[16] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[17] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the Conference on Computer Aided Verification*, Lecture Notes in Computer Science. Springer, 2004.

[18] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *ACM Internation Symposium on Software Testing and Analysis*, pages 123–133, July 2002.

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

[20] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Dawn Song. Bitscope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University, March 2007.

[21] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In Wenkee Lee, Cliff Wang, and David Dagon, editors, *Botnet Detection*, volume 36 of *Countering the Largest Security Threat Series: Advances in Information Security*. Springer-Verlag, 2008.

[22] David Brumley and James Newsome. Alias analysis for assembly. Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006.

[23] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[24] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2007.

[25] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, October 2007.

[26] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*, October 2006.

[27] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, pages 147–160, November 2006.

[28] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994.

[29] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of the ACM Symposium on Operating System Principles*, oct 2007.

[30] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating System Principles*, 2005.

[31] Jedidiah Crandall, Zhendong Su, S. Felix Wu, and Frederic Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.

[32] Jedidiah R. Crandall and Fred Chong. Minos: Architectural support for software security through control data integrity. In *Proceedings of the International Symposium on Microarchitecture*, December 2004.

[33] Alan Crosswell. Igmp v3 tcpdump trace. http://www.columbia.edu/~alan/igmp/ex1b/.

[34] Weidong Cui, Marcus Peinado, Helen J. Wang, and Michael Locasto. Shi eldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[35] DataRescue. IDA Pro. http://www.datarescue.com. Page checked 7/31/2008.

[36] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Von Underduk. Polymorphic shellcode engine using spectrum analysis. http://www.phrack.org/show.php?p=61&a=9.

[37] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[38] Thomas Dullein and Rolf Rolles. Graph-based comparison of executable objects. In *Proceedings of the Symposium sur la Securite des Technologies de L'information et des communications*, 2005.

[39] Thomas Dullien. Need new tools. Presentation at the BlackHat Conference, 2006. http://media.blackhat.com/bh-usa-06/video/2006_BlackHat_Vegas-V7-Halvar_Flake-Need_New_Tools.mp4.

[40] eEyE Security. eEye binary diffing suite (EBDS). http://research.eeye.com/html/tools/RT20060801-1.html. Version 1.0.5.

[41] Michael James Van Emmerik. *Single Static Assignment for Decompilation*. PhD thesis, The University of Queensland School of Information Technology and Electrical Engineering, May 2007.

[42] Dawson Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *International Symposium on Software Testing and Analysis*, pages 1–4, 2007.

[43] Halvar Flake. Structural comparison of executable objects. In *Proceedings of the IEEE Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, 2004.

[44] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the Symposium on Principles of Programming Languages*, 2001.

[45] J.E. Forrester and B.P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *4th USENIX Windows Systems Symposium*, 2000.

[46] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Proceedings on the Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 524–536, Berlin, Germany, July 2007. Springer-Verlag.

[47] Debian Gao, Michael Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the International Conference on Information and Communications Security*, pages 238–255, October 2008.

[48] Christos Gkantsidis, Thomas Karagiannis, Pablo Rodriguez, and Milan Vojnovic. Planet scale software updates. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2006.

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

[50] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, February 2008.

[51] Arnaud Gotlieb, Bernard Botella, and Michael Rueher. Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Software Engineering Notes*, 23(2):1998, 1998.

[52] Neelam Gupta, Aditya Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. *ACM SIGSOFT Software Engineering Notes*, 23(6):231–244, November 1998.

[53] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 1-5*, April 2008.

[54] Daniel Jackson and Eugene J. Rollins. Chopping: A generalization of slicing. Technical Report CS-94-169, Carnegie Mellon University School of Computer Science, 1994.

[55] Havard Johansen, Dag Johansen, and Robbert van Renesse. Firepatch: Secure and time-critical dissemination of software patches. In *IFIP International Information Security Conference*, May 2007.

[56] Angelos Keromytis. Characterizing self-healing software systems. In *Proceedings of the International Conference on Mathematical Methods, Models and Architectures for Computer Network Security*, pages 22–33, September 2007.

[57] Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the USENIX Security Symposium*, 2004.

[58] Bogdan Korel. Automated test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990. path-based test set generation.

[59] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the ACM Workshop on Hot Topics in Networks*, 2003.

[60] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the USENIX Security Symposium*, 2004.

[61] James Larus and Eric Schnarr. EEL: machine-independent executable editing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 291–300, 1995.

[62] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[63] Zhichun Li, Manan Shanghi, Brian Chavez, Yan Chen, and Ming-Yang Kao. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resiliance. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[64] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.

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

[66] Jason Medeiros. Automated expoit development: The future of exploitation is here. http://toorcon.org/2007/talks/19/toorcon_whitepaper.pdf, 2007.

[67] Microsoft. Phoenix framework. http://research.microsoft.com/phoenix/. Paged checked 7/31/2008.

[68] Microsoft. Phoenix project architect posting. http://forums.msdn.microsoft.com/en-US/phoenix/thread/90f5212c-f05a-4aea-9a8f-a5840a6d101d, July 2008. Page checked 7/31/2008.

[69] Barton Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[70] B.P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the International Workshop on Random Testing*, 2006.

[71] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. In *Proceedings of the IEEE Symposium on Security and Privacy*, volume 1, 2003.

[72] David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of the IEEE Conference on Computer Communications*, 2003.

[73] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[74] Alan Mycroft. Type-based decompilation. In *European Symposium on Programming*, March 1999.

[75] Susanta Nanda, Wei Li, Lap chung Lam, and Tzi cker Chiueh. BIRD: Binary interpretation using runtime disassembly. In *Proceedings of the IEEE/ACM Conference on Code Generation and Optimization*, March 2006.

[76] Ryan Naraine. Crime rings target ie 'setslice' flaw. http://www.eweek.com/article2/0%2C1759%2C2022805%2C00.asp, 2006.

[77] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the International Conference on Compiler Construction*, 2002.

[78] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation or Building Tools is Easy*. PhD thesis, Trinity College, University of Cambridge, 2004.

[79] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In Rebecca Write, Sabrina De Capitani di Vimercati, and Vitaly Shmatikov, editors, *Proceedings of the ACM Conference on Computer and Communications Security*, pages 311–321, 2006.

[80] James Newsome, David Brumley, and Dawn Song. Sting: An end-to-end self-healing system for defending against zero-day worm attacks. Technical Report CMU-CS-05-191, Carnegie Mellon University School of Computer Science, 2006.

[81] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[82] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, September 2006.

[83] James Newsome, David Brumleyand Dawn Song, Jad Chamcham, and Xeno Kovah. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proc. of the* 13^{th} *Annual Network and Distributed System Security Symposium (NDSS)*, 2006.

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

[85] Paradyn/Dyninst. Dyninst: An application program interface for runtime code generation. http://www.dyninst.org. URL Checked 9/28/2008.

[86] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the IEEE Symposium on Security and Privacy*, may 2006.

[87] Andres Protas and Steve Manzuik. Skeletons in microsoft's closet. BlackHat Europe 2006: http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Manzuik.pdf.

[88] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering*, 1995.

[89] Jim Roskind. Attacks against the netscape browser plus security response philosophy and methods. Private communication and seminar talk.

[90] Sabre Security. Bindiff. http://www.sabre-security.com/products/bindiff.html.

[91] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[92] Secure Science Corporation. Analysis of the WebViewFolderIcon ActiveX integer overflow (setSlice). http://www.mnin.org, 2006.

[93] Stelios Sidiroglou and Angelos D. Keromytis. A network worm vaccine architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Workshop on Enterprise Security*, pages 220–225, June 2003.

[94] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[95] Loren Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University Department of Computer Science, 1996.

[96] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. The EarlyBird system for real-time detection of unknown worms. Technical Report CS2003-0761, University of California, San Diego, August 2003.

[97] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[98] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The top speed of flash worms. In *Proceedings of the ACM Workshop on Rapid Malcode*, October 2004.

[99] G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[100] Andrew Tridgell. How samba was written. http://www.samba.org/ftp/tridge/misc/french_cafe.txt, August 2003. URL Checked on 8/21/2008.

[101] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *Proceedings of the EuroSys Conference*, 2007.

[102] Shobha Venkataraman, Avrim Blum, and Dawn Song. Limits of learning-based signature generation with adversaries. In *Proceedings of the Network and Distributed System Security Symposium*, February 2008.

[103] Helen J Wang, Chuanxiong Guo, Daniel Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM Special Interest Group on Data Communication*, August 2004.

[104] Ke Wang, Janak Parekh, and Salvatore J. Stolfo. Anagram: A content anonaly detector resistant to mimicry attack. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 226–248, September 2006.

[105] Mark Weiser. Program slicing. In *Proceedings of the Conference on Software Engineering*, pages 142–151, 1981.

[106] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.

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

[108] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the USENIX Security Symposium*, 2005.