Theory and Techniques for Automated Generation of Vulnerability-Based Signatures

David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha

Abstract—In this paper, we explore the problem of creating *vulnerability signatures*. A vulnerability signature is based on a program vulnerability and is not specific to any particular exploit. The advantage of vulnerability signatures is that their quality can be guaranteed. In particular, we create vulnerability signatures from the vulnerable program itself, such that they are guaranteed to have zero false positives by construction. We show how to automate signature creation for vulnerabilities that can be detected by a runtime monitor. There is no one right signature representation for a vulnerability. We introduce a formalism and way of thinking about vulnerability signature generation that is analysis centric instead of representation specific. In particular, a signature can be represented in many ways, from using regular expression to using a full Turing-complete language. Previous systems have mostly focused on a particular point in the design space. We show how to approximate the language of a vulnerability in many different language classes, each of which has unique properties and benefits, by performing analysis on the program binary and vulnerability. Our approach also considers multiple-path vulnerabilities. A multiple-path vulnerability is a vulnerability that can be exploited through several different code paths. For example, a Web server may have a vulnerability in a URL handling routine that is called for many different types of requests. We demonstrate techniques that can create signatures that cover multiple paths an exploit may take. We have had to develop new algorithms to cope with the problem where enumerating vulnerable paths leads to an exponential explosion. We develop a new approach that captures the logical semantics of multiple vulnerable program paths in $O(n^2)$ space (where n is the size of the program) instead of exponential. We provide a formal definition of a vulnerability signature and investigate the computational complexity of creating and matching vulnerability signatures. We systematically explore the design space of vulnerability signatures. We also provide specific techniques for creating vulnerability signatures in a variety of language classes. In order to demonstrate our techniques, we have built a prototype system. Our experiments show that we can, using a single exploit, automatically generate a vulnerability signature as a regular expression, as a small program, or as a system of constraints. We demonstrate techniques for creating signatures of vulnerabilities that can be exploited via multiple program paths. Our results indicate that our approach is a viable option for signature generation, especially when guarantees are desired.

Index Terms—Vulnerability, vulnerability signature, intrusion detection, intrusion prevention, polymorphic worm, Turing machine signature, regular expression signature, symbolic signature.

1 Introduction

A *vulnerability* is a type of bug that can be used by an attacker to alter the intended operation of a software program in a malicious way. An *exploit* is an actual input that triggers a vulnerability, typically with malicious intent and devastating consequences. One of the most popular and effective exploit defense mechanisms is signature-based input filtering. These defense systems are in constant need of new signatures as new vulnerabilities are discovered.

This paper focuses on automated techniques for generating sound signatures. We need *automated* signature generation techniques because manual signature generation is slow and error prone. Automated techniques are important 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 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 [1], let alone COTS software at the assembly level.

The task of automatically constructing signatures is complicated by the fact that there are usually several different *polymorphic* exploit variants that can trigger a software vulnerability [2], [3], [4]. For example, a buffer-overrun vulnerability in a network service may be triggered by many different protocol messages. Another example, sometimes referred to as metamorphism, is that exploit variants may differ syntactically but be semantically equivalent [5], [6], e.g., an exploit could use different assembly instructions that have the same effect.[1] Furthermore, it is fairly straightforward for attackers to produce polymorphic exploit variants using publicly available morphing tools such as Metasploit [7] and CLET [2]. *Thus,*

- *D. Brumley and J. Newsome are with Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15217. E-mail: {dbrumley, jnewsome}@cmu.edu.*
- *D. Song is with the University of California, Berkeley, Soda Hall 675, UC Berkeley, CA 94720-1776. E-mail: dawnsong@cs.berkeley.edu.*
- *H. Wang and S. Jha are with the Computer Science Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706. E-mail: {hbwang, jha}@cs.wisc.edu.*

Manuscript received 8 Sept. 2006; revised 21 May 2007; accepted 13 Aug. 2008; published online 2 Sept. 2008.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0128-0906.
Digital Object Identifier no. 10.1109/TDSC.2008.55.

1. Our approach does not need to distinguish between polymorphism and metamorphism: both are referred to as polymorphism throughout this paper.

to be effective, the signature should be constructed based on the property of the vulnerability, instead of an exploit (this observation has been made by others as well [8], [9]).

Since in many cases, signatures are not only automatically generated but also automatically deployed, we would like generated signatures to have *guarantees*. In particular, we would like the signature to have a guaranteed accuracy. In this paper, we focus on signatures with a zero false-positive rate so that if they are automatically deployed, they will not mistakenly block legitimate traffic. We can make this guarantee because the signature is generated by extracting out the conditions necessary to exploit a vulnerability from the vulnerable program itself. As long as the analysis is sound, the resulting signature will not have false positives. For example, in Section 4.2, we introduce a program with an out-of-bound write vulnerability (Fig. 3). Our techniques extract out the necessary conditions to exploit this vulnerability and turn that information into a signature (shown and discussed in Section 4.4).

We also would like signatures with guaranteed evaluation efficiency in terms of memory and processing time. Efficiency is important because signature-based defense systems may have constrained resources such as memory or processing power. Since no two sites have the same requirements, we would like signature generation algorithms that can offer a wide spectrum of different guarantees, potentially trading off accuracy for efficiency or vice versa.

Our approach, road map, and the central issues. In this paper, we present a formal approach for reasoning about and creating *vulnerability signatures*. The distinguishing characteristic of a vulnerability signature is that it is based upon the semantics of the vulnerability and not on particular characteristics of an exploit. As a result, vulnerability signatures are less susceptible to common evasion methods such as polymorphism. In addition, because vulnerability signature generation is not based on attacker-supplied data, generated signatures are immune to recent attacks where an attacker can mislead the signature generation mechanism itself [10], [11], [12], [13].

At a high level, the contributions of this paper include 1) a formal basis for creating vulnerability-based signatures for a wide variety of vulnerabilities, 2) methods for creating a vulnerability signature in a variety of language classes, each of which has its own guarantees, and 3) an architecture for automated vulnerability signature generation given only a description of the vulnerability and the vulnerable program binary.

First, we establish a formal basis for automated signature generation for a large class of vulnerabilities, namely, those that can be protected against by an execution monitor (EM). Previous works have created signatures for specific types of vulnerabilities, including those that can be detected with dynamic taint analysis [9], [14] and memory protection schemes [9], [15]. We provide an algorithm and approach for automatically generating signatures for any vulnerabilities that can be detected by an EM, thus providing a generalized foundation (Section 3). The formal foundation is not just of theoretical interest. For example, the foundation suggests metrics for comparing the accuracy of signatures in a manner that is not specific to the generation mechanism, as well as techniques for signature generation itself.

Second, we provide algorithms for creating signatures in various language classes. At a high level, a signature is a Boolean function that returns for any input either SAFE or EXPLOIT. We can write a signature in any language class, from Turing-complete languages to regular languages. We explore the trade-offs for representing signatures in each class. Since each class has unique advantages and guarantees, there is no one single best signature class for all scenarios. Therefore, it is important to support signature generation in many different language classes.

We develop a new approach where we can approximate a signature written in a more powerful language with a less expressive language in order to get better guarantees on matching efficiency. For instance, the Vigilante end-to-end system generates from an execution trace essentially straight-line programs as signatures [9], which correspond to our symbolic constraint signatures. We show how to create signatures in other signature representations, how to approximate signatures in one language class with regular expression signatures, and how to handle multiple execution path vulnerabilities. As shown in Section 4.5, one of the central problems is how to concisely represent a Turing machine (TM) signature in a lower class language. Previous techniques such as forward symbolic execution would often lead to an exponential blowup when lowering from a Turing-complete signature to a lower class [16]. We develop new methods (and proved them correct) that reduce the size of symbolic signatures from exponential to at most $O(n^2)$ [17] (where n is the size of the program) and, in practice, nearly linear.

Thus, our techniques address two important design axes in the automatic vulnerability signature generation design space: guaranteed soundness and improved efficiency for representing multiple-path vulnerabilities. In particular, we show how to generate guaranteed sound signatures in a signature class other than a Turing-complete language, e.g., sound regular expression signatures. Sound regular expression signatures are of interest since most commercial signature-based defenses support regular expression signatures.

Finally, we have implemented our approach. We can generate guaranteed sound signatures in a variety of language classes given only the vulnerability specification and the vulnerable program binary (executable). Our focus on the binary level has many advantages: 1) most users only have access to the binary, 2) there may not be time to involve an application vendor when generating the signature, such as in the case of zero-day exploits, and 3) by working at the binary level, the generated signature is precisely faithful to the vulnerability. Working at the binary level is challenging since fundamentally, there are only basic integer types, one globally addressed memory region, goto's instead of structural control flow, and no local variables. Our implementation has addressed these challenges, and we show that we can make signatures for many different vulnerabilities. We are the first to automatically generate guaranteed-sound signatures that cover multiple program paths for exploiting a single vulnerability. Although some engineering challenges remain, our evaluation shows that our techniques are an important step toward automatic generation of accurate and efficient signatures.

2 Related Work

Signatures. We generate vulnerability signatures via binary-program analysis. As shown in Section 3, our approach can generate a signature for any vulnerability whose exploits can be recognized by an EM.

Shield proposes a framework for manually creating vulnerability signatures by modeling network protocols [8]. Modeling the network protocol can be advantageous since exploits are often understood in terms of particular message fields, e.g., an overly long URL. However, Shield signatures 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 signature generation. Overall, the advantage of binary analysis is that the generated signature is guaranteed to be faithful to the vulnerability as it appears in the program.

Vigilante [9] is an end-to-end system that generates signatures (called filters in Vigilante) that are similar to our single-path symbolic constraint signatures from an execution trace. 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 that can be exploited via multiple paths. In addition, we show how to take an execution trace and produce a sound regular expression (and other signature types). Note that Vigilante does not produce regular expression signatures. Since regular expressions are typically better suited for network-based defense, our techniques are likely of interest to Vigilante and similar end-to-end systems.

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 [18]. However, they do not actually generate signatures. They conclude that "token-based byte string signatures 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 that this observation is apt; syntactic details of input strings may be insufficiently general for some vulnerabilities. This observation motivates the need for generating signatures in more expressive language classes (than regular languages), such as our symbolic constraint or TM signatures.

Several researchers have proposed generating signatures via pattern extraction [4], [19], [20], [21], [22]. These approaches have the advantage of requiring relatively inexpensive analysis to generate. However, the down side of pattern-extraction-based signature generation is that the vulnerability itself is never analyzed; only exploits supplied by the attacker are analyzed. As a result, many of these techniques are either incapable of handling polymorphic worms [19], [20], [21], [22] or vulnerable in an adversarial environment [10], [11], [12], [13]. Our techniques, on the other hand, can provide strong guarantees since the signature is based on the vulnerability itself.

Symbolic execution and test case generation. One technique we use is to generate a logical formula representing a particular vulnerable execution path. A related problem is bug finding, where program paths are explored to see if they are vulnerable, such as with DART/CUTE [23], [24] and EXE [25]. Our problem differs in that we *know* where the vulnerability is in the program, as opposed to trying to discover a new bug. As a result, we have adopted more efficient techniques for generating formulas that work backward from the vulnerability point. In particular, we have proven that the size of our formulas over an entire (acyclic) program are at most $O(n^2)$ in the size of the program [17], while forward symbolic execution is exponential in the number of program branches. Another difference is that we use the decision procedure to enumerate satisfying answers to the generated formulas, while bug finding generally only cares if a single answer exists. We currently use STP [26] since it is designed to model bit-level details, which are commonly important in binary analysis; however, our methods are not specific to the decision procedure. In particular, using other decision procedures that return all satisfying answers would enhance the efficiency of our approach. However, we are not aware of any production-quality decision procedures that meet these criteria.

Other areas. Although the focus of our work is to protect vulnerable programs by automatically generating signatures, our approach adopts techniques from nonsecurity-related research areas such as model checking [27], delta debugging [28], [29], [30], compilers [31], bug finding, decision procedures and theorem provers, and program verification. We show in this paper how these techniques can be extended or enhanced to analyze binary programs and produce vulnerability signatures that are capable of detecting polymorphic exploit variants. In particular, we use the work of Leino in efficiently computing the weakest preconditions as a basis for our approach [32]. Barnett and Leino also provides an alternate approach for calculating the weakest precondition of unstructured programs by adding new statements to the guarded command language (GCL) (e.g., `goto`) [33].

3 Vulnerability Signatures

A vulnerability signature recognizes inputs that exploit a vulnerability. At a high level, the distinguishing trademark of a vulnerability signature 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 signature is indifferent to the specifics of the attacks.

In this section, we provide formal definitions for the class of vulnerabilities for which we create signatures, the representation classes we consider for signatures, and how the signature representation classes compare.

3.1 Background and Definitions

Informally, a vulnerability is a "bad thing" that can happen in a program. A vulnerability signature should match program inputs that cause the bad thing to happen. In this section, we formalize the notion of "bad things," vulnerabilities, and what types of vulnerabilities are targeted by our signature generation techniques.

We adopt the notation from the work of Schneider [34]. 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 paper), 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. Let $\sigma[..k]\tau$ represent a finite execution $\sigma[..k]$ followed by τ. Let $P(i) : \sigma$ denote that the execution of P on input i results in execution σ.

The set of "bad things" that we should disallow is specified in a security policy, which is a Boolean predicate on the space of program executions, represented as $\mathcal{Q}$. 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 a security policy if $\forall \sigma \in \Sigma_P : \mathcal{Q}(\sigma) = true$.

In this paper, we specifically consider the class of policies that can be enforced using an EM [34]. EM-enforceable policies are security policies that can be enforced considering only a single execution of a program. Typical examples of EM policies include firewalls, access control policies, and control flow and memory integrity. In particular, EM-enforcement mechanisms enforce policies that are safety properties. A security policy $\mathcal{Q}$ is a safety property if

$$\mathcal{Q}(\sigma) = false \Rightarrow (\exists k : (\forall \tau \in \psi : \mathcal{Q}(\sigma[..k]\tau) = false)). \quad (1)$$

This definition implies that $\mathcal{Q}$ is a safety property if when it is false, one can prove it false in finite time (i.e., $\mathcal{Q}$ is characterized by a set of finite execution prefixes $\sigma[..k]$) [34], [35]. In particular, a violation of a security policy that is a safety property cannot be "undone" at some point in the future. This makes sense for EM enforcement since at any step in the execution, an EM should be able to decide if the current state violates the policy without considering future executions (or other executions).

3.2 Vulnerability Signatures for EM-Enforceable Policies

In this paper, we focus on techniques for generating vulnerability signatures for vulnerabilities for which exploits can be recognized by an EM. For simplicity, we use the term *vulnerability* to denote this class of vulnerabilities.[2] For example, specific previous instances of signatures generated from EMs include dynamic taint analysis [9], [14] and memory protection schemes [9], [15]; our approach can be viewed as a general framework for these schemes.

We focus on generating signatures for a single vulnerability. For example, $\mathcal{Q}$ may specify control flow integrity within the program [36], i.e., control flow during execution acts as intended by the programmer. However, a program with multiple buffer overflows will violate this property at multiple places. We will generate a signature for each buffer overflow independently.

In order to precisely specify a single vulnerability, we define two auxiliary predicates, the *vulnerability condition* c and the *vulnerability point* v_p, which together specify a single vulnerability as $\langle v_p, c\rangle$. The vulnerability condition is satisfied when the security policy is violated:

$$P(i) : \sigma \vdash c(\sigma) = true \Leftrightarrow \mathcal{Q}(\sigma) = false. \quad (2)$$

The vulnerability point is the point where an execution goes wrong. Equation (1) implies that for all executions such that $Q(\sigma) = false$, there is a k such that $Q(\sigma[..k]) = false$. We defined v_p as the program instruction corresponding to the first such k. The pair $\langle v_p, c\rangle$ thus specify all inputs that result in violating the safety policy at a particular point in the program.

The *language of a vulnerability* $\mathcal{V}$ is characterized by the set of all input strings x in the domain of the program that satisfy the vulnerability condition at the vulnerability point:

$$\mathcal{V}_{\langle v_p,c\rangle} \doteq \{\forall x \in dom(P) | P(x) : \sigma \vdash c(\sigma[..v_p]) = true\}. \quad (3)$$

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

Given a specification of a single vulnerability $\langle v_p, c\rangle$, our goal is to automatically generate a signature $\mathcal{S}$ that recognizes exploits, i.e., members of the language $\mathcal{V}$. We model a signature as a Boolean function over the input domain of P:

$$\forall x \in dom(P) : \mathcal{S}(x) \rightarrow \{\text{SAFE}, \text{EXPLOIT}\}.$$

A *sound* signature returns EXPLOIT only for inputs that would really exploit the program. Unsound signatures have false positives by mistaking legitimate inputs for exploits. A sound signature is defined as

$$\mathcal{S}(x) = \text{EXPLOIT} \Rightarrow P(x) : \sigma \vdash c(\sigma[..v_p]) = true. \quad (4)$$

If a signature will always return EXPLOIT for a real exploit, then the signature is *complete*:

$$\forall x \in \text{dom}(P) : P(x) : \sigma \vdash c(\sigma[..v_p]) = true \Rightarrow \mathcal{S}(x) = \text{EXPLOIT}. \quad (5)$$

An incomplete signature has false negatives because it may miss some exploits.

We say a vulnerability signature perfectly recognizes the language of the vulnerability when

$$\mathcal{S}(x) = \text{EXPLOIT} \Leftrightarrow x \in \mathcal{V}.$$

Note that soundness and completeness are guarantees: when we say a signature is sound, then we are guaranteeing no false positives with respect to the vulnerability specification $\langle v_p, c\rangle$, and similarly for complete signatures.

Throughout this paper, we focus on generating sound but possibly incomplete signatures. We take this approach since, as we will see, we cannot always have a sound and complete signature. Sound but potentially incomplete signatures are well suited for low-risk environments where it is better to miss exploits than deny a legitimate user access. Complete but unsound signatures are well suited for high-assurance environments where it is better to block an input that cannot be explicitly verified as safe. In Section 7, we describe how a simple transformation allows us to generate complete but potentially unsound signatures.

Formalism discussion. The above formalism is useful for mathematically describing vulnerability signatures for a

2. There may exist other classes of security policies and vulnerabilities for which signatures can be created. However, we do not address other classes in this work.

TABLE 1
Summary of Approximate Bounds for the Three Vulnerability Signature Representations We Consider for a Program of Length N and Signature Size S

Representation	Generation	Signature Size	Operations		
			Matching	Minimization	Equivalence
Turing machine Sig.	poly(N)	poly(N)	Undecidable	Undecidable	Undecidable
Symbolic Constraint Sig.	poly(N)	poly(N)	PSPACE-complete	exp(S)	exp(S)
Regular Expression Sig.	poly(N) - exp(N)	exp(N)	O(N)	O($S \log S$)	O($S \log S$)

Poly(X) denotes a function polynomial in X, and exp(X) denotes a function exponential in X.

large class of vulnerabilities without regard to a specific vulnerability, representation, or signature generation algorithm. Mathematical precision is useful in a number of ways.

First, it exactly describes what we do and do not target by our vulnerability signature algorithms. In particular, at any point, we can say that a program is exploited or not exploited. For example, in a multistage attack, multiple messages are required to exploit the server. We only return EXPLOIT when the server is exploited; we do not look into the future to determine if possible future messages might exploit the server.

Second, the formalism is useful as an abstraction for comparing the accuracy of a generated signature. For example, we propose in Section 3.4 that we can measure the completeness of a signature with respect to a perfect signature for the vulnerability language.

Third, by basing signature generation on EM-enforceable policies, we arrive at a natural way to perform signature composition (also called signature merging). Suppose we have two EM-enforceable policies $\mathcal{Q}_1$ and $\mathcal{Q}_2$. EM policies can be composed, and a single EM policy for both policies is given by their conjunction $\mathcal{Q}_1 \wedge \mathcal{Q}_2$. The corresponding vulnerability signature for a violation of either EM policy is then given by $\mathcal{S}_1 \vee \mathcal{S}_2$, since a violation of either policy should be detected. Note that non-EM security policies may not compose, so signatures for vulnerabilities of those policies may also not compose. In practice, this means that we are justified in composing signatures by taking their disjunction, e.g., for regular expressions, vulnerability signature composition is carried out by the "|" operator.

3.3 Signature Representation Classes: Properties and Trade-Offs

A vulnerability signature that perfectly recognizes all exploits can be constructed by constructing an EM for the security policy. After all, an EM for a safety property already recognizes all safety violations, so all that needs to be changed is for the EM to return EXPLOIT instead of terminating the program. Equation (1) implies that evaluating such a signature on exploits will return EXPLOIT. However, a signature constructed in this way will not return SAFE unless the original program terminates, since there are no guarantees about executions that do not violate the safety property.

In a perfect world, a signature would be sound and complete while using almost no resources. Unfortunately, a perfect signature—a signature that is both sound and complete—must be at least of the same language class as the vulnerability language. In most cases, this means a perfect signature must be written in a Turing-complete language. However, signatures in Turing-complete languages offer few performance guarantees. In practice, we often want signatures that offer minimum efficiency guarantees. In this section, we examine three signature classes, TM signatures, *symbolic constraint* signatures, and *regular expression* signatures, and describe the trade-offs between accuracy and efficiency among the classes.

Signature operations. We consider the efficiency of the operations shown in Table 1. The operations include generation time, signature size, matching time, minimization time, and equivalence time. The generation time and the signature size are specific to our algorithm, while all other measurements are general bounds. Note that signature minimization takes a signature $\mathcal{S}$ and computes the smallest signature $\mathcal{S}_{min}$ in the same class language as $\mathcal{S}$. A minimized signature takes the least amount of space and is generally more efficient to match. Signature equivalence is to determine whether two signatures $\mathcal{S}_1$ and $\mathcal{S}_2$ match the same language. Signature equivalence is useful in many scenarios, e.g., an administrator receives two signatures from different parties and wants to know if they are for the same vulnerability.

3.3.1 Turing Machine (TM) Signatures

A TM signature is a signature written in a Turing-complete language. At a high level, any EM-enforceable policy can be turned into an initial TM signature by inlining the EM vulnerability condition at the vulnerability point. An initial signature can then be refined to eliminate unnecessary computation (see Section 4.4). Unfortunately, TM signatures inherit the performance guarantees of Turing-complete languages, namely, almost none. In particular, determining whether a TM signature will return an answer is undecidable, along with minimization and equivalence.

The lack of guarantees does not mean that TM signatures are unimportant. Since a perfect signature must be in the same language class as a vulnerability and most vulnerabilities are likely Turing complete, TM signatures serve as an important landmark. For example, while ultimately, a sound and complete TM signature may not be practical, they serve as a good starting point in our approach. By starting with a sound and complete TM signature, we only need to be careful to maintain the properties we want, such as soundness, at each step. An approach that does not start with a sound signature has a more significant burden to make similar guarantees.

3.3.2 Symbolic Constraint Signatures

A symbolic constraint signature is a Boolean formula, which in our approach approximates a TM signature. The approximation is loop free but may have universal and existential quantifiers. Unlike TM signatures, matching (evaluating) a symbolic constraint signature on an input x will always terminate. Symbolic constraint signatures only approximate constructs such as loops. As a result, symbolic constraint signatures cannot be both sound and complete.

Symbolic constraint signatures can be viewed as a TM signature on a finite domain.[3] The signature S is a Boolean program, i.e., all the variables in the program can only take values from a finite domain. The Boolean program matching (BPM) problem is

> Given a Boolean formula $\mathcal{B}$, an assignment of values (string) x, determine whether x satisfies $\mathcal{B}$.

We prove that the BPM problem is PSPACE-complete in the Appendix.

3.3.3 Regular Expression Signatures

Regular expressions are the least powerful signature representation of the three and may have a considerable error rate in some circumstances. For example, a well-known limitation is that regular expressions cannot count [37] and therefore cannot succinctly express conditions such as checking that a message has a proper checksum or even simple inequalities such as $x[i] < x[j]$. However, regular expression signatures are widely used in practice because matching a regular expression is efficient.

Regular expression signatures are well understood and are the primary mechanism many production NIDSs use to detect exploits. A regular expression signature can be matched in $O(n)$ time (where n is the size of the input) when represented as a deterministic finite automaton (DFA), then signatures can be matched in $O(|n|)$ time (where n is the size of the input). A signature can be minimized in time $O(S \log S)$, where S is the size of the DFA [38]. Equivalence of two signatures $\mathcal{S}_1$ and $\mathcal{S}_2$ is done by first computing the minimum DFA of each signature and then checking if the states and transitions are the same.

3.3.4 Other Signature Types

Signatures can be represented in other language classes. For example, the call/return semantics of procedures can be represented accurately as a context-free language [39]. Finding efficient methods for creating signatures in other language classes such as a context-free language is an interesting open problem.

3.4 Monomorphic Execution Path (MEP) and Polymorphic Execution Path (PEP) Signature Coverage

We introduce the notion of vulnerability signature coverage in which we create a vulnerability signature with respect to only a subset of vulnerable programs. The ability to consider a subset of paths to a vulnerability (as opposed to all program paths an exploit may follow) is important, since creating a signature for all program paths that lead to the vulnerability may be too expensive. For example, in order to scale it may be necessary to take an iterative approach of generating an initial signature which considers only a few paths, and incrementally updating it to include more paths via signature composition.

First, consider a single path in the program an input may take that satisfies the vulnerability condition, which we call *Monomorphic Execution Path* (MEP) coverage. Our initial MEP is usually the path taken by the sample exploit. Within an MEP, for each conditional branch encountered, one target is an instruction leading toward the vulnerability point, while the other target is a state SAFE. An MEP is therefore a straight-line program. At the vulnerability point, the vulnerability condition is evaluated, which returns either SAFE or EXPLOIT. The vulnerability signature consists of all inputs that reach the EXPLOIT state. 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 reach the vulnerability point, and the vulnerability condition evaluates to EXPLOIT (others have noted this as well [18]). For example, exploits usually have a payload that executes arbitrary attacker code. A straight-line program will return EXPLOIT for exploits with different payloads because the execution of different variants only differs *after* the vulnerability condition has been satisfied.

A *Polymorphic Execution Path* (PEP) coverage includes many different paths (i.e., many MEPs) to the vulnerability point. A *complete* PEP coverage includes all paths to the vulnerability point. Therefore, a complete PEP coverage signature accepts all inputs $\in \mathcal{V}$, i.e., the signature is complete.

In our setting, we think about program paths that can reach the vulnerability in terms of the program's control flow graph (CFG) $G = (V, E)$. The CFG of a program consists of a node $v \in V$ for each statement and an edge (v_i, v_j) if there is control flow from statement i to j. Complete PEP coverage could include the entire program. However, many paths in the program may not be relevant to a vulnerability.

The instructions relevant to the vulnerability constitute a subgraph of the CFG. Intuitively, any instruction that could be executed from where the input is read in to the vulnerability is included as a relevant instruction. In program analysis, the subgraph is called the "chop" of the program with respect to a source node (where the input is read in) and a sink node (the vulnerability point) [40], [41]. One metric for judging the false-negative ratio of a signature is to determine how many different vulnerable code paths—i.e., code paths from the chop—are handled by the signature. Therefore, we can use the chop as a metric for measuring the completeness of a vulnerability signature.

We describe our algorithm for computing the chop in Section 4.4.2. Note that a particular chopping algorithm may be imprecise, e.g., due to limitations of pointer analysis, path sensitivity, etc. However, an imprecise chop means that strictly more statements are included than necessary to evaluate whether a particular input is SAFE or EXPLOIT; thus, chop precision is an efficiency issue and not a correctness issue. As a result, minimizing the chop using program analysis and optimization techniques is an important area of research. Our experiments indicate that even using our rough chopping scheme, we can create signatures that evaluate (i.e., match) in microseconds.

3. Note that restricting the logic will change the theorem, e.g., if quantifiers are disallowed then signature evaluation may be more efficient.

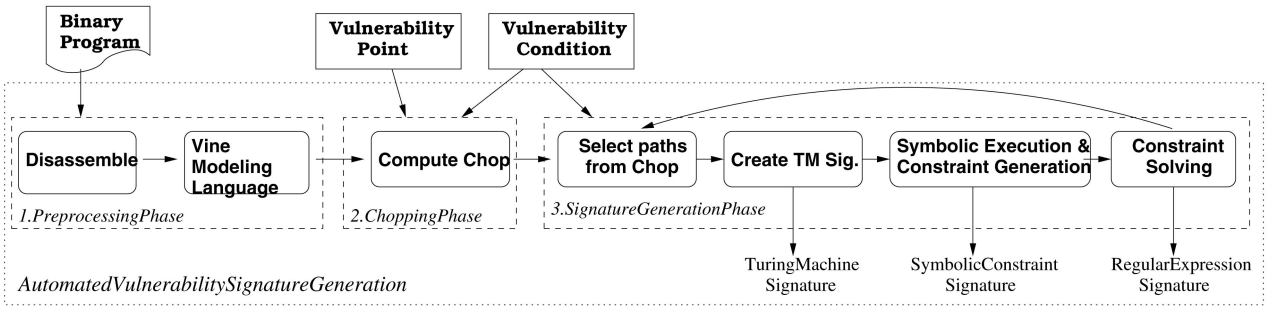

Fig. 1. A high-level view of the steps to compute a vulnerability signature.

4 Approach and Techniques for Automated Vulnerability Signature Generation

In this section, we describe our approach and techniques for generating vulnerability signatures. Our techniques depart from the traditional approaches of analyzing the exploit or its behavior. Instead, we take a *program-centric* approach for generating signatures where we synthesize the vulnerability signature from the vulnerable program. The central benefits of our approach are the following: 1) signature accuracy can be guaranteed, and 2) signatures can be created in a language class appropriate for the particular user requirements.

We are given a vulnerable binary program P and a vulnerability specification $\langle v_p, c\rangle$. In addition, our techniques also require a single sample exploit x. We preprocess the program and translate it into an easier-to-analyze modeling language. The steps to compute the vulnerability signature are the following (Fig. 1):

- Translate the $\times$86 binary program to our modeling language.
- Synthesize a TM signature from the program by inlining the vulnerability condition at the vulnerability point and computing the *chop* of the program model. Stop if this is the final representation.
- Synthesize a symbolic constraint signature from the TM signature. Stop if this is the final representation.
- Synthesize a regular expression signature from the symbolic constraint signature. Stop if this is the final representation.

In this section, we describe each step in detail.

4.1 Input to Signature Generation

We focus on generating signatures for known vulnerabilities. The input to our signature generation algorithm is the vulnerable program P, the vulnerability specification $\langle v_p, c\rangle$, and a sample exploit x. The sample exploit requirement addresses the practical problem of calculating signatures for programs that have multiple types of inputs. For example, a typical program may read in a configuration file, set global variables, and only then read in the user input. We use the trace to determine which inputs can come from the attack and which inputs are from nonattack system calls such as reading in configuration files. Let v_i be the point where the input is first read in in the sample trace. We compute our signatures with respect to the program state at v_i. If v_i is not the initial start of the vulnerable program, then there may be a state at v_i, e.g., the contents of memory, that is relevant in determining whether an input is an exploit or not. For example, a signature for a vulnerability in a Web server should be with respect to the state the Web server is in when the input would be processed.

Note that we do not care how the vulnerability point, condition, and sample exploit are determined: this is addressed by work in exploit detection. For example, in the Vigilante end-to-end system, new exploits are discovered by EMs that participate in the Vigilante system. The vulnerability condition c would be the condition imposed by the Vigilante EM. Another option is to use honeypots to collect exploits. Given the exploit x, the v_p can be ascertained by first generating an execution trace of $P(x) : \sigma$ and then applying c on each step in the trace σ [42]. From (1), we know that there will be a single instruction v_p that results in the program being exploited.

4.2 Vine: Our Modeling Language

We have developed the language shown in Fig. 2, called Vine, in order to model assembly and facilitate analysis of assembly code. We translate $\times$86 instructions into this language. The translation does not depend on the vulnerability and thus can be performed as a preprocessing step ahead of time. The formal semantics of this language are available in [43]; here, we give an informal presentation.

Instructions	i	::=	$*(r_1) := r_2 \mid r_1 := *(r_2) \mid r := v \mid r := r_1 \diamond_b v \mid r := \diamond_u v \mid$ `label` l_i
			$\mid$ nop $\mid$halt $\mid$ fail $\mid$ `jmp` $\ell \mid$ `ijmp` $r \mid$ `if` r `jmp` ℓ_1 `else jmp` ℓ_2
Operations	$\diamond_b$	::=	$+, -, *, /, \ll, \gg, \&, \mid, \oplus, =, <>, <, \leq$ (Binary operations)
	$\diamond_u$	::=	$\neg, !$ (unary operations)
Operands	v	::=	n (an integer literal) $\mid$ r (a register/variable) $\mid$ ℓ (a label)

Fig. 2. Our modeling language, called Vine. In our implementation, we translate $\times$86 assembly instructions into this language.

```
1  orig_pc = pc;
2  *sp = pc; // assembly ''call vulnerable'' inst
3  jmp vulnerable;
4  ...
5  label vulnerable: // start of vulnerable function
6    dst = sp - 4;  // set up stack space for dst
7    if *input <> 'g' jmp ret_label else jmp copy_loop;
8  label copy_loop: // copy non-null elements of src to dst
9    *dst = *input;
10   dst = dst + 1;
11   input = input + 1;
12   if *input <> 0 jmp copy_loop else jmp ret_label;
13 label ret_label:
14   saved_pc = *sp;  // assembly ''ret'' inst
15   ijmp saved_pc;
```

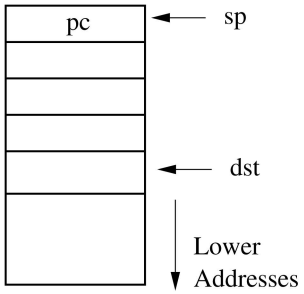

Fig. 3. Our running example. On the left, a vulnerable program that will overtime the saved return address if `*input` is greater than 4 bytes is shown. On the right, the memory configuration after executing line 2 is shown.

Our language has assignments ($r := v$), binary and unary operations ($r := r_1 \diamond_b v$ and $r := \diamond_u v$, where $\diamond_b$ and $\diamond_u$ are binary and unary operators), loading a value from the memory into a register ($r_1 = *(r_2)$), storing a value ($*r_1 := r_2$), direct jumps (jmp ℓ) to a known target label (label ℓ), indirect jumps to a computed value stored in a register (ijmp r), and conditional jumps (if r then jmp ℓ_1 else jmp ℓ_2).

Our implementation translates $\times$86 into this language in a syntax-directed fashion (Section 5). Like assembly, our language is not structural and does not have procedures. For example, a `call` instruction in assembly is modeled as a two-part operation that stores the current instruction address in memory and then `jmp` to the callee's address. A `ret` retrieves a 32-bit address index by the variable `sp` and performs an `ijmp`. Recovering higher level semantics such as procedure boundaries is not necessary for our techniques, though it can increase efficiency.

One important point to keep in mind is that the translated program is not the assembly itself; it is a semantically equivalent model for all well-defined executions. When assembly is not safe, e.g., out-of-bound writes, the operational semantics of our modeling language are safe. In other words, just like in assembly, you can write unsafe programs in our modeling language and analyze that program. However, when you go to run a program written in the modeling language, safety is dynamically checked to ensure that no unsafe operations really happen.

While the full operational semantics of the modeling language are outside the scope of this paper (they are available as part of other work [44]), an example is helpful to clarify the distinction. For example, an *assembly* program may have an out-of-bound read or write to memory. The corresponding program in the *modeling language*, however, simulates the effects of the reads and writes safely by raising a runtime error for any out-of-bound access (as opposed to assembly, which would just make the access with no error).

Example. Fig. 3 shows a typical buffer overflow in Vine. Fig. 3a shows the program text, while Fig. 3b shows the memory configuration after executing line 6. This example is a simplified version of a classical buffer overflow. In the example, we assume that the variable `sp` corresponds to the stack pointer, `pc` corresponds to the instruction pointer, and `input` contains the address of an input source buffer. We assume that `input` is the only input into this program.

On line 1, we record the value of the current program counter. On lines 2-3, an assembly `call` statement is issued, which in our language consists of two statements: one that saves the current instruction pointer and one that transfers control to the called function "vulnerable." The vulnerable function allocates 4 bytes of space for buffer `dst` on line 5 by decrementing the stack pointer. The function copies bytes from the `input` buffer to the `dst` buffer on lines 8-12 until NULL (0) byte in `src` is reached. To make things interesting, this copy only happens if the first byte of `input` begins with "g." Finally, on lines 14-15, we show the function epilogue, where we read and jump to the saved return address on the stack. This program has a vulnerability if there is not a NULL within the first 4 bytes.

In our running example, the safety property states that the saved return address from executing the `call` statement should be equal to the address on the stack at the time of `ret`. In our setting, the vulnerability condition is then $c = saved\_pc <> orig\_pc$, and the vulnerability point is line 14. This is a typical vulnerability condition. Note that an alternate vulnerability condition would be that writes to `dst` should stay within the allocated stack frame, e.g., $dst < sp$.

Our goal is to create a vulnerability signature for this vulnerability. A perfect signature would match all inputs greater than 4 bytes long that begin with "g." A sample exploit for our vulnerable program is "gaaaa."

4.3 Generate an Instruction Trace

We first execute $P(x)$ to generate an instruction trace σ. The instruction trace contains the address of each instruction

executed, along with the value of all instruction operands. Instruction traces can be efficiently generated for most modern architectures including ×86 via hardware [45], [46] or software [47], [48], [49]. Although the number of instructions executed may be large, the corresponding trace can be efficiently represented [50], [51].

Given sample input "gaaaa," the instruction trace for Fig. 3 would include instructions $\sigma = 1, 2, 3, 5, 6, 7, 8, \{9, 10, 11, 12\}^5, 13, 14$.

4.4 Generating a TM Signature

4.4.1 Generate an MEP TM Signature

We create the MEP TM signature with respect to the path followed in the instruction trace. Therefore, the initial signature will match the sample exploit and certain exploit variants such as changing the exploit payload.

We create the TM signature by translating each instruction executed into a straight-line program. We then insert two special return labels for EXPLOIT and SAFE. We then replace all conditional and indirect jumps in the program with checks to make sure that the execution path is followed. In our running example, each time line 12 is executed except the last, we would replace the jump to `ret_label` with a jump to `safe`, where `safe` is a label corresponding to the SAFE state. We then inline the vulnerability condition at the vulnerability point. If c returns true, the signature jumps to EXPLOIT; else, it issues a jump to SAFE. In our example, this results in the following code being inserted after line 14:

```
if saved_pc <> orig_pc jmp exploit else jmp safe
label exploit: return exploit
label safe: return safe
```

The result is a TM signature that recognizes exploits for the same path the sample exploit followed. Again, note that although there may be an out-of-bound write in the original program, this does not correspond to an out-of-bound write in the model, as the modeling language is safe (while assembly is not). In our implementation, we compile down the resulting signature to object code and then evaluate it on inputs by running the program.

4.4.2 Generate a PEP TM Signature

We compute a TM signature for multiple paths by first inlining the vulnerability condition at the vulnerability point. However, instead of considering the single execution path in the trace, we compute all program paths from the exploit input to the vulnerability point. We first prepare a CFG of the program where each vertex is a statement and each edge represents a possible flow of control between two statements. Note that in the CFG, call sites are linked up with the called functions (this type of CFG is sometimes called a supergraph [52]).

We then "chop" the CFG so that only paths starting at where exploits are read in to the vulnerability condition are considered. Let v_i be where an input is read in, e.g., as determined by the sample exploit. We compute the chop from v_i to the vulnerability point v_p. The chop includes at least all instructions that are relevant to the vulnerability, i.e., could be executed starting at v_i and producing a value that is used to reach v_p and satisfy c.

A precise chop is performed by only considering how input values propagate during execution. Noninput variables are given the same value as in the trace. However, safely determining which variables are affected by input requires data dependency and alias analysis. Calculating precise data dependency and alias analysis is very challenging at the assembly level. We sidestep this issue by developing a safe approximation of chopping appropriate for our problem domain, which only considers control flow. The result of our chop may include more instructions than necessary due to the lack of dependency analysis (thus, less efficient to evaluate) and may be incomplete due to the indirect control flow we could not resolve.

The chop is constructed on the full CFG. We are given two distinguished nodes: v_i, where input is read, and v_p, the vulnerability point. There must be at least one path in the CFG from v_i to v_p, namely, the one taken in the exploit trace. We add an edge (v_p, v_i) to the CFG. This creates a cycle in the CFG. We then calculate the strongly connected component (SCC) containing v_p and v_i. The SCC is our chop, since it contains all reachable statements (based on control flow) from v_i to v_p.

Note that our chopping algorithm is conservative: by considering data dependencies, we could remove edges and statements but never add statements. Thus, although adding data dependency analysis would reduce the size of the TM signature, our method still produces a sound signature. For indirect jumps, if we cannot resolve the target accurately, we add a special return status (UNKNOWN in our implementation) that indicates signature failure. Signature failure does not mean that an input is SAFE or EXPLOIT; it means that the signature generation algorithm could not analyze the control path.

4.5 Generate a Symbolic Constraint Signature

We compute the symbolic constraint signature for multiple paths from the TM signature. At a high level, we generate constraints for input variables such that inputs satisfying the constraints would exploit the vulnerability. The symbolic constraint signature is an approximation of the TM signature because we may have to statically estimate the effects of loops. The symbolic constraint system is a logical formula that represents the effects of executing the program for any input. The formula is satisfied (i.e., true) for inputs that are exploits. The steps for computing the symbolic constraint signature are the following:

1. Transform the input TM signature into an acyclic program P'.
2. Translate the acyclic TM program P' into the GCL P'_g. This step is needed so that we can take advantage of optimizations described below.
3. Replace conditional jumps in the GCL program that go outside the chop.
4. Calculate the *weakest precondition* $\mathcal{S} = wp(P'_g, c)$ for exploiting the acyclic GCL program. $\mathcal{S}$ is a symbolic constraint signature formula.

4.5.1 Create an Acyclic Program

The first step is to transform the TM signature into a finite program by creating an upper bound on the number of

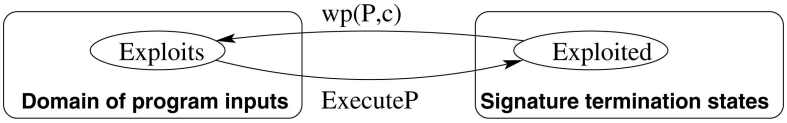

Fig. 4. For a program P and a vulnerability c, the weakest precondition $wp(P, c)$ describes the inputs which, upon execution, result in an exploited state.

times any single loop executes. This step is needed in order to guarantee signature termination. We locate loops via standard analysis, which outputs for each loop the loop conditional and the loop back-edge [31]. For each loop, we create a fresh loop counter variable and replace the back-edge with a check to make sure that the loop counter does not exceed an upper bound. At this point, we have a program P' that is guaranteed to terminate. We can output P' as the signature if we do not wish to create a regular expression signature.

4.5.2 Translate into the GCL

At this point, we could iterate over each execution path in the modified TM signature and perform forward symbolic execution to create the symbolic constraint signature [16], [25]. However, creating a compact representation for multiple paths an exploit may take is key for creating a succinct vulnerability signature. For instance, in a loop-free program with b branches, there are $O(2^b)$ program paths. Forward symbolic execution calculates a separate formula for each path, which results in an exponential-size formula. For example, if there are two conditional jumps, then there are four program paths, and forward symbolic execution would generate $f = f_1 \wedge f_2 \wedge f_3 \wedge f_4$, where f_i represents the formula for path i.

In order to efficiently generate a symbolic signature, we calculate the *weakest precondition* for the TM signature to return EXPLOIT. At a high level, the weakest precondition for a program P and a postcondition Q is a recursive calculation for generating a formula that is satisfied for all inputs that, when executed, cause the program to terminate in a state satisfying the specified postcondition Q. For example, if the program is $P : y = x + 1$ and $Q : 2 < y < 5$, then $wp(P, Q)$ is $1 < x < 4$. In our setting, the postcondition is the vulnerability condition, and the weakest precondition $wp(P, c)$ is a predicate on inputs that satisfy the vulnerability condition. The generated formula will be true for inputs that exploit the given vulnerability. The net result of this process is depicted in Fig. 4.

The advantage of using the weakest precondition is that we can take advantage of a logical transformation that guarantees that the generated formula is at most $O(n^2)$, where n is the number of statements in the program. Thus, we reduce the size of the generated signature from exponential using forward symbolic execution to at most quadratic and, in practice, more near linear. We have a formal proof of this in our companion paper [17]. Intuitively, these bounds are possible because the acyclic TM signature itself is almost a logical formula that recognizes exploits. It is only "almost" a formula because variables and memory cells may be assigned more than once, and functions cannot have such updates. The weakest precondition is slightly larger than the program because we must translate such updates into logical updates where variables are equated and not assigned. The formula is much smaller than the one created with symbolic execution because the calculation summarizes paths instead of enumerating all possible paths.

The weakest precondition is calculated over the GCL. Statements in the GCL are given by the following grammar:

$$\mathrm{s} ::= \mathrm{lval} := e \mid \mathbf{assert}\ e \mid \mathbf{assume}\ e \mid s; s \mid s \,\square\, s.$$

Although this language looks simple, it is powerful enough to reason about general-purpose programming languages [53], [54]. Statements s in the language are assignments of expressions to l-values (e.g., registers and memory cells), "**assert** e," which checks that expression e is true and fails if it is false, "**assume** e," 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. A program written in GCL terminates normally iff none of the assertions fail.

Because GCL is structural, it is straightforward to translate a structural language into GCL. For example, the program `if e then A else B` is translated into the GCL as "$(\mathbf{assume}\ e; A) \square (\mathbf{assume}\ \neg\, e; B)$." However, the binary programs we analyze are not structural because of jumps. Therefore, previous work on translating a program to GCL does not apply in our work with binary programs.

In order to work on binaries, we have developed a new approach for translating an unstructured binary program into the GCL. Our algorithm is a type of structural analysis on the CFG of the unstructured binary where we create an appropriate GCL based upon the structure of the CFG. The details of the algorithm are unimportant here; a full description for the interested reader is available in [17]. The output of this step is the (acyclic) GCL program P'_g.

4.5.3 Replace Conditionals

The weakest precondition calculation will calculate a formula that is true for all inputs that reach the vulnerability point and satisfy the vulnerability condition. We alter P'_g so that any input that will not exploit the program is a "failed" execution by replacing all conditional jumps outside the chop to be **assert** statements. This step ensures that the weakest precondition is true only when an input satisfies the vulnerability condition at the vulnerability point. A SAFE input will either fail an assert or not satisfy the vulnerability condition.

4.5.4 Calculate the Weakest Precondition

We then calculate the weakest precondition on the acyclic TM program in a syntax-directed manner. The rules for calculating the weakest precondition are shown in Fig. 5, where Q is any Boolean predicate. These rules can be read as an algorithm where the ":" separates inputs from

$$\frac{}{wp(x := e, Q) : Q[e/x]}\ \text{WP-ASSIGN} \qquad \frac{}{wlp(x := e, Q) : Q[e/x]}\ \text{WLP-ASSIGN}$$

$$\frac{}{wp(\textbf{assume}\ E, Q) : E \Rightarrow Q}\ \text{WP-ASSUME} \qquad \frac{}{wlp(\textbf{assume}\ E, Q) : E \Rightarrow Q}\ \text{WLP-ASSUME}$$

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

$$\frac{wp(s_2, Q) : Q_1 \quad wp(s_1, Q_1) : Q_2}{wp(s_1; s_2, Q) : Q_2}\ \text{WP-SEQ} \qquad \frac{wlp(s_2, Q) : Q_1 \quad wlp(s_1, Q_1) : Q_2}{wlp(s_1; s_2, Q) : Q_2}\ \text{WLP-SEQ}$$

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

Fig. 5. Algorithm for calculating the weakest precondition (wp) and weakest liberal precondition (wlp). The weakest liberal precondition rules are essentially the same as the weakest precondition except for **assert**. The weakest liberal precondition rules are used as a subcalculation when computing the weakest precondition.

outputs. For example, we calculate the weakest precondition $wp(\textbf{assert}\ e \square s, Q)$ as follows:

$$\frac{\dfrac{}{wp(\textbf{assume}\ e \square s, Q) : e \Rightarrow Q} \quad \dfrac{\cdots}{wp(s, Q) : Q_1}}{wp(\textbf{assume}\ e \square s, Q) : e \Rightarrow Q \wedge Q_1}.$$

In order to get the quadratic bound, when calculating the weakest precondition, we avoid introducing redundancy into the formula. For example, in symbolic execution, when a path branches, two separate formulas are created: one for the true branch f_1 and one for the false branch f_2. However, both formulas share the same prefix; thus, when calculating whether either can be satisfied as $f_1 \vee f_2$, the common prefix is considered twice.

If we calculated the weakest precondition naively using just the WP rules in Fig. 5, we would introduce redundancy in two ways: 1) the postcondition is duplicated as part of WP-CHOICE, which corresponds to duplicating the postcondition for independent branches, and 2) WP-ASSIGN can cause an exponential blowup during variable substitution. To see why assignment is a problem, consider a calculation of the following form:

$$wp(x_1 := x_0 * x_0; x_2 := x_1 * x_1; x_3 := x_2 * x_2;, x_3 < 5).$$

The weakest precondition using WP-ASSIGN would eventually substitute all variables for x_0, resulting in the exponential formula $x_0 * x_0 * x_0 * x_0 * x_0 * x_0 * x_0 * x_0 < 5$. We can eliminate duplication due to WP-CHOICE by ensuring that all variables and memory locations are only assigned to once and then replacing the assignment with **assume**s. We use a form of single static assignment (SSA) to make sure that all variables are only assigned once [31]. Note that memory references are also translated into SSA form, e.g., a write creates a new memory variable, and reads are performed from the current memory incarnation. Translating into SSA form results in a GCL program at most $O(n^2)$ larger than the original non-SSA program.

We remove the redundancy due to the postcondition by calculating

$$\mathcal{S} \doteq wp\left(P'_g, c\right) \equiv wp\left(P'_g, true\right) \wedge \left(wlp\left(P'_g, false\right) \vee c\right) \quad (6)$$

instead of $wp(P, c)$ directly. Note that this equation is only valid for assignment-free programs, i.e., have undergone the above transformation. The final redundancy is removed from WP-CHOICE since the postcondition is always a constant and thus is not duplicated along each branch. The resulting signature is at most quadratic in size due to the conversion to SSA. [17]

We output $\mathcal{S}$ as our signature, which is a Boolean formula that is true for all inputs that would exploit the vulnerability in the acyclic program.

4.6 Generate a Regular Expression Signature

We generate a regular expression signature from the symbolic constraint signature by using a decision procedure to enumerate inputs that satisfy the constraint system. A decision procedure, when given a formula, will return whether the formula is satisfiable or not. If satisfiable, the decision procedure will return an assignment of values to variables that satisfy the formula. By construction, the symbolic constraint signature is a Boolean formula that is only satisfied by inputs that would exploit the program. In other words, a satisfying answer *is* an input that exploits the program.

More specifically, suppose in the symbolic constraint signature, $\mathcal{S}$ accepts an n byte input $x = x_1, x_2, x_3, \ldots, x_n$. Querying the decision procedure on $\mathcal{S}$ will return a satisfying answer $v = v_1, v_2, v_3, \ldots, v_n$, where v denotes a particular constant value. By construction, v is an exploit. We then iteratively query if $\mathcal{S} \wedge \neg v$ is satisfiable, i.e., if there is a satisfying answer that is not one we already know about. When satisfiable, we get a new satisfying answer v', and the signature is $v \vee v'$ or, in regular expression notation, $v|v'$.

The above approach will return a new v on each iteration until we have exhausted the inputs that the symbolic constraint signature recognizes. This procedure is guaranteed to terminate since a symbolic constraint signature is over a finite domain. However, exhaustively querying may take a long time. One thing we have found in practice is that often, particular bytes for an exploit are unconstrained, e.g., in many buffer overflows, the only constraint on values for input bytes is that they are not NULL. We have found that querying the decision procedure specifically to find such input bytes is an important optimization.

Divide-and-conquer. The number of variables to consider within a single path may be very large, e.g., millions of variables at the assembly level. The decision procedure must simultaneously reason about all variables, including any possible alias relationships. Reducing the complexity of the formula by providing alias information can significantly improve performance. For example, in an HTTP request, the method (e.g., GET, HEAD, etc.) is often processed independent of the URL, e.g., in our evaluation for Atphttpd, an HTTP request must begin with either the keyword "head" or "get," but it does not matter which, i.e.

$$(\text{method} = \text{“GET”} \vee \text{method} = \text{“HEAD”}) \wedge (..\text{other conditions}..).$$

If we can prove that values for the variable "method" never affect "url," then the decision procedure can solve each conjunct separately. However, if "method" and "url" could be aliased, then the decision procedure must consider both conjuncts simultaneously.

In our evaluation, we show that partitioning formulas into independent subclauses can significantly reduce the time to produce regular expressions. In our implementation and all experiments but one, we let the decision procedure reason about aliasing. However, we do perform one experiment where we manually specify that such conjuncts can be independently considered in order to demonstrate the potential speedup.

5 Implementation

We have implemented a prototype system, called Vine, to evaluate our techniques for automatically generating signatures. Our implementation works with both Linux (ELF) and Windows (PE) binaries. Our implementation does not require debugging information or a symbol table. Vine is divided into four components: a trace collector, a lifting component, an analysis component, and a decision procedure interface component.

We have implemented a trace collector on top of QEMU [55], a whole system emulator. We have modified QEMU to track how specified external inputs such as keyboard and network inputs flow into the operating system. Instructions whose operands are derived from external inputs are assigned a special "taint" flag, which indicates to our remaining system that those are potential exploit inputs. The output log contains for each instruction executed its address and the operand values for each instruction.

The lifting component reads a native binary, parses the binary format, disassembles code segments to assembly, and lifts the assembly to an intermediate representation (IR) based on the language shown in Fig. 2. We interface with two disassemblers: IDA-Pro, a commercial disassembler, and the disassembler described in [56].

The advantage of the Vine language is that it makes the $\times$86 instruction set easier to analyze. For example, $\times$86 has single instruction loops (e.g., the `rep` prefix) and implicit side effects (many instructions implicitly update or test the `eflags` register); the same register may be addressable in multiple ways (e.g., the `al` register addresses the lower 8 bits of `eax`) or may have different behaviors for different operands (e.g., shifting by 0 does not set `eflags`, but other values will). All these behaviors are translated into a more manageable set of statements. Our implementation handles almost all $\times$86 instructions with the exception of floating-point operations. The lifting component includes about 16,500 lines of C/C++ code.

The analysis component is responsible for subsequent analysis discussed in Section 4, as well as type-checking, control flow and call graph construction, data-flow analysis, miscellaneous optimizations, and a compiler for our language which generates C code. The C code can then be compiled with any C compiler. The analysis component is written in OCaml and is about 28,000 lines of code. The final component interfaces our OCaml routines with a decision procedure. We currently interface with STP [25], [26], [57] and CVCL [58], both of which are satisfiability-oriented decision procedures.

Indirect jumps. One problem we must deal with at the binary level is the widespread use of indirect jumps, e.g., `jmp` $\%\star$ `eax`.[4] Indirect jumps pose a potential problem when computing the chop in that they could potentially go anywhere. If we assume that they could go anywhere, then the chop would include the entire program.

The most common type of indirect jump in IA-32 programs is the `ret` instruction, which uses a return address previously stored on the stack, usually by the `call` instruction that called the function containing the `ret`. In our implementation, we assume that the program intends `ret` to return to the callee when computing the chop. When we compile our code to native $\times$86, we generate code that checks this assumption.

There may be indirect jumps that do not correspond to `ret` instructions, e.g., function pointers in the source language may be implemented as indirect jumps in assembly. Since we do not know which function may be called, we rewrite these jumps as a runtime error during signature evaluation. This is a conservative approach in order to keep the signature sound. We leave as future work implementing an analysis to resolve nonreturn indirect jumps at compile time instead of during signature evaluation.

6 Evaluation

In this section, we evaluate our techniques. First, we evaluate creating signatures in different language classes for a single execution path (MEP). We then evaluate our approach for creating signatures in different language classes for many execution paths (PEP).

6.1 Experiment Setup

Our evaluation is performed under Linux on a machine with an Intel 2.4-GHz Core Duo processor and 3 Gbytes of addressable memory. The vulnerability condition we use is that in our signature, the simulated return address should not be overwritten (similar to the condition in Section 4.2). In each case, we first obtain an exploit sample trace to pinpoint where the user input is first read in, as well as the vulnerability point, the instruction that overwrites the return address.

4. A direct jump is of the form `jmp` c, where c is a constant.

TABLE 2
Performance Numbers for MEP Signature Creation and Signature Evaluation Time in Seconds

	Blaster	Slammer	Atphttpd	Samba
Create TM Sig (seconds)	3.064s	15.517s	19.091s	9.970s
TM Sig Size (# stmts)	518801	1574296	2096915	1290216
Optimized TM (# stmts)	117860	281345	488214	148901
Translate TM Sig to C	2.11s	5.26s	9.71s	2.79s
Compile TM C Sig	14.99s	51.88s	120.51s	20.84s
TM Sig Evaluation	0.21s	0.005s	0.008s	0.018s
Create Symbolic Sig (seconds)	0.166s	0.388s	0.161s	0.207s
Create Initial Regex Sig (seconds)	38s	212s	293s	103s

The total number of statements in the trace is the same as the second-row TM signature size.

We evaluate six different programs for creating MEP signatures: Atphttpd, passlogd, Ghttpd, Samba, Windows DCOM RPC Interface, and the Microsoft SQL Server. Atphttpd is a small Web server written in C [59]. We evaluate Atphttpd 0.4b compiled for Linux, which has a standard sprintf-style buffer overflow in the URL processing logic [60]. Passlogd is a syslog message server and has a buffer overflow in the `sl_parse` message parsing routine [61]. Ghttpd is another small Web server. We evaluate version 1.4.3 [62]. Samba is a server that implements the SMB/CIFS file and print sharing protocol. We evaluate Samba 2.2.8 compiled for Linux, which has a buffer overflow in the `call_trans2open` procedure [63]. The Windows DCOM RPC interface in Windows 2000 and Windows XP is vulnerable to a buffer overflow. We refer to this vulnerability as "blaster" since the blaster worm is based on an exploit for this vulnerability [64]. Finally, the Microsoft SQL Server 2000 contains multiple buffer overflows. We target the overflow exploited by the Slammer worm [65].

For each of these vulnerabilities, the vulnerability point is the point where an overflow may take place, e.g., the vulnerable call to `sprintf` in the Atphttpd vulnerability and a particular `mov` instruction in the passlogd vulnerability. The vulnerability condition is that the instruction or function call will result in writing to an area known to be outside the destination buffer, such as a saved stack pointer or return address.

6.2 Monomorphic Execution Path

We first generated MEP TM signatures, symbolic constraint signatures, and regular expression signatures for several of the vulnerabilities. Table 2 summarizes our results.

We begin by converting the instruction trace into the Vine language and creating a single-path TM signature. Considering only the code path taken by a sample exploit has previously been shown to generate effective signatures [9], [42]. The time to generate a TM signature was under 20 seconds in all cases. Profiling reveals that most of this time is spent in converting the trace to straight-line code. This step is currently implemented by executing the Vine translation on our prototype Vine evaluator to determine the exact path followed and is only necessary due to a previous limitation of our infrastructure. It is straightforward to eliminate this step by obtaining the path directly from the given execution trace. With this step removed, generating the TM signature takes no more than 1.6 seconds.

We also give the total number of statements of the TM signature for the single path. Many instructions along the single path are irrelevant to the particular vulnerability. We perform a dead-code elimination step to remove these statements. The number of statements after dead-code elimination [31] is also shown. Dead-code elimination removes any statement that computes a result that is never subsequently referenced. We also report the time to generate C code for a TM signature and compile the resulting C code using gcc 4.1. Finally, we measure the evaluation time of the TM signatures. The evaluation time is averaged over 1,000 runs including both safe and exploit input samples. The numbers indicate that TM signature evaluation is efficient. We then generate a symbolic constraint signature. This step took less than half of a second in all cases.

Surprisingly, our PEP signatures (Section 6.3) produced signatures that were smaller and more efficient to evaluate. The reason for this is an artifact of our implementation: given a trace, we convert the whole trace to a signature (and then perform dead-code elimination), including each instruction each time it is executed. Since an instruction may be executed multiple times, e.g., as part of multiple function call invocations, this is inefficient. However, when creating a full PEP TM signature, we have access to the whole program and convert each instruction only once, resulting in a smaller and more efficient signature. This implementation limitation will likely be addressed in future work.

Regular expression generation. We gave the symbolic constraint signature to the decision procedure STP to generate an example of an input that would satisfy the constraints. Table 2 shows the time to generate the initial regular expression, i.e., the time to check if the symbolic constraint signature formula is satisfiable. The total time to generate an initial regular expression is 293 seconds for Atphttpd, 38 seconds for Blaster, 212 seconds for Slammer, and 103 seconds for Samba.

We investigated the dominating time factors for regular expression signature generation and found that one primary case was reasoning about memory operations. In our current infrastructure, STP must reason about the entire formula, including all potential alias relationships in the generated constraints. One obvious trick is to provide more information to STP so that it can consider clauses in the formula independently (Section 4.6). We manually inspected the formula for Atphttpd and found that we could divide it up into 10 distinct terms, i.e., 10 clauses could be reasoned about independently. Partitioning the formulas into 10 components and solving each independently reduces the total time to 0.1216 second [16]. Thus, this experiment indicates that sound assembly alias analysis and the ability to automatically partition formulas would be extremely valuable in a production system.

Next, we evaluated how well our techniques work at generalizing regular expression signatures. Vigilante has previously created an MEP signature for both Blaster and Slammer [9] but did *not* create regular expressions or symbolic signatures. In that work, Costa et al. manually inspected their single-path signature to summarize the

necessary conditions for an exploit to match their generated TM signature. In order to automatically generate a regular expression that matched their description, we let STP iterate for 3 days to discover the value ranges for each byte. STP answered 2,189 queries for Blaster and 181 for Slammer. The resulting automatically created regular expression signature confirmed the criteria specified by Costa et al. For example, the signature restricted the first byte of Slammer to be 0x4, while allowing most other bytes to be any nonzero value.

Comparison with other signatures. We compare the quality of our generated signature for Atphttpd with signatures created via pattern extraction by Hamsa [22]. Hamsa [22] extracts signatures from a pool of traffic using machine learning techniques to extract common tokens common to only exploits. Although Atphttpd has two vulnerable code paths, one via an HTTP "GET" request and one via an HTTP "HEAD" request, both signatures appear to have been generated using only "GET" requests (neither identified "HEAD" as a token). Hamsa includes, among others, the token "$\mathrm{HTTP/1.1 \setminus r \setminus n}$" in their signature for Atphttpd. Our analysis and formulas indicate that this token does not need to appear in an exploit. In fact, Atphttpd does not even check the HTTP method field where "$\mathrm{HTTP/1.1 \setminus r \setminus n}$" would be checked.

These examples highlights the trade-offs one gets from different approaches. Hamsa and similar signature generation algorithms are fast but do not analyze the vulnerability and thus cannot make any guarantees. Our approach can give guarantees but requires more expensive analysis. In practice, both approaches are valuable. One interesting future line of work is to use our techniques to verify signatures created by less expensive methods.[5]

6.3 Polymorphic Execution Path TM Evaluation

Next, we consider multiple execution paths. In this section, we focus on vulnerable Linux servers. We do this because analyzing all paths in the Windows servers is complicated by the fact that the core functionality is strewn through multiple dynamically loaded libraries.[6] For example, the Slammer vulnerability point is inside a DLL and not in the server executable. Due to the research nature of our current infrastructure, we do not address simultaneous disassembly of multiple DLLs and executables.

We created TM signatures for three Linux programs: Atphttpd, Ghttpd, and Passlogd. Each program can be exploited via multiple program paths. We create these signatures by calculating all paths reachable from the input statement in the sample trace to the vulnerability point. Table 3 shows our results. We first measure the total number of statements for the executable in our modeling language. These numbers show that with our current prototype, the total signature is about 9 percent of the original code size. In practice, the C compiler will further significantly reduce the total size. For example, we implemented the global value numbering and dead-code elimination optimizations for our language [31], which reduced the total number of statements to 1 percent of the code size. Since these optimizations are standard in compilers, we currently let the C compiler perform them instead of duplicating the work in our implementation. These numbers indicate that TM signatures in our experiments could be only 1 percent of the original vulnerable program (i.e., 1 percent of the initial signature where c is inlined at v_p). We again measure how long it takes to generate the TM signature, convert the TM signature to C, and compile the C code. We also measure evaluation time, averaged over 1,000 inputs that are both benign and malicious.

TABLE 3
Performance for Creating TM Multipath Signatures for All Vulnerable Paths

	Atphttpd	Ghttpd	Passlogd
Total program stmts	13846	132323	121520
Create PEP TM Signature (sec)	0.676s	0.549s	0.273s
TM Sig Size (stmts)	9087	12697	2101
Translate TM Sig to C (sec)	0.029s	0.88s	1.585s
Compile TM C Sig (sec)	0.88s	0.61s	0.05s
Evaluate TM Sig (sec)	0.00064s	0.00021s	0.00136s

The time to generate a TM PEP signature is much smaller than that for a single path. The reason for this is that the MEP signature contains a statement for each instruction actually executed. For example, if a loop contains 100 instructions and is executed four times, there are 400 instructions. The PEP signature, on the other hand, replicates the loop semantics and therefore does not have the unnecessary duplication. In addition, the MEP signature contains a statement for instructions executed as part of a library call. For example, Atphttpd calls `strcmp` in several places, and each time, every instruction executed in `strcmp` is included in the signature. The PEP signature, on the other hand, does not consider instructions that are part of called libraries. Instead, the PEP signature uses a stub function to call the actual library itself during signature evaluation. For example, we set up a stub to call `strcmp` in the Atphttpd signature.

6.4 Polymorphic Execution Path Symbolic Constraint Evaluation

In this next experiment, we created a PEP constraint symbolic signature for Atphttpd and Samba. We first consider creating a PEP symbolic signature where libraries are not analyzed. Similar to a TM signature, if a PEP symbolic constraint signature is the final signature representation, the signature can call external functions directly. However, if we wish to create a regular expression signature, we must include the logical semantics of library calls directly in the formula. This is a typical problem in model checking, which is often solved by creating *summary functions* of called procedures.

Table 4 show our results where the symbolic constraint signature considers both when summary functions are both available and when they are not available. These results are

5. Another possible extension would be to use our techniques to generate exploits that do not match known signatures; we leave that extension to the black hat community.

6. Vigilante also points out this issue [9].

TABLE 4
Symbolic Signature Size for Full Path Coverage Using Weakest Preconditions and Forward Symbolic Execution

Program	No Summaries			library summaries		
	# Stmts	WP	Forward Exec.	# Stmts	WP	Forward Exec.
Atphttpd	4.16×10^{11}	4.32×10^{11}	5.75×10^{18}	15834	33517	96658
samba	2.78×10^{7}	3.00×10^{7}	$> 2^{64}$	2.40×10^{7}	2.72×10^{7}	7.72×10^{18}

We consider both with and without summary functions for dynamic libraries.

also reported in [17]. The idea is that if we were to generate regular expressions, we would use these formulas as a basis. We unrolled each loop once; thus, the signature considers all branches. Thus, this measures how big formulas would be if we were to cover all possible branches in the signature. We also measure the size of the formula generated with forward symbolic execution, as is commonly done in bug finding [16], [25], [66], [67], [68].

Our results show a number of things. First, formulas built using forward symbolic execution are too large to be useful. For Samba, the size of the formula exceeded the 64-bit counter. This demonstrates that forward symbolic execution is likely not a good candidate to use for PEP signature generation. Second, formulas built with WP are only a bit larger than the original number of statements. This measurement concurs with other applications of WP at the source code level [69]. Third, without function summaries, even efficient algorithms are likely to generate formulas too big for a decision procedure to consider. If we assume summary functions, then the formula size is within the realm of current decision procedures.

7 Discussion

In this section, we discuss our approach, future work, and alternate directions.

Using heuristics. Our approach is geared toward guaranteeing soundness for generated signatures. Our experiments indicate that our approach for generating TM signatures and symbolic constraint signatures for a large number of paths is practical. Regular expression signature generation currently seems best suited when only a few paths are considered at a time.

Although heuristics to improve signature quality may be attractive in practice, from a research point of view, it is equally as important to push how far we can get without them. Part of the reason our formalism is useful is because it defines exactly the class of vulnerabilities for which we can generate accurate signatures. However, we can only make such guarantees for fixed-length inputs currently. This seems to be part of the nature of providing guarantees for programs: even established fields such as model checking often only consider a finite number of steps. In practice, however, heuristics may be useful in some scenarios. For example, we have previously proposed heuristics such as automatic widening of input bytes to be anything after a finite number of queries [16]. Vigilante proposes additional heuristics, which work well in their test cases [9].

Vulnerability conditions. In our paper, we inline the vulnerability condition at the vulnerability point. In the worst possible case, the safety policy predicate may need to be evaluated at each execution step. We have found, as others, that such draconian interposition is often not necessary [34]. The vulnerability conditions we have found most useful are often the simplest, such as limiting the range of a particular memory write instruction.

Regular expressions. Our experiments indicate a few things. First, it would be useful for decision procedures to return more than one satisfying answer at a time, i.e., a decision procedure for ALL-SAT instead of simply SAT. Interfacing with an appropriate ALL-SAT solver is left as future work. Second, our experiment on Atphttpd indicates that alias analysis could offer huge performance benefits. For example, regular expression signature generation for Atphttpd without alias analysis took almost 300 seconds, while with alias information, it took less than 1 second [16]. Currently, we construct our queries such that the SAT solver must reason about all potential alias relationships. There is work on alias analysis that is suitable for assembly [44], [70], [71]. We expect adding even simple alias analysis to our infrastructure would increase efficiency by orders of magnitude.

System calls. In our current implementation, we assume that all nonexploit inputs are given exactly the same as the trace for MEP. Note that any system call is treated as an input to the program. For PEP, we may have a system call that is not handled in the execution trace, i.e., an alternate path to the vulnerability may make a system call. In this case, it is unclear what the right solution is. One approach that we have previously taken is to quantify over all possible values of such inputs [17]. This approach is not completely sound, however, since the quantification may be too general. For example, quantification may result in a statement that says a particular file may exist when in fact, on the vulnerable system, it never will. Our current implementation does not automatically quantify anything: it requires the user to specify the effects of the system calls. Such specifications can be done ahead of time or during signature creation. It is not clear whether in practice, there is one single right answer: quantification may be appropriate in some scenarios, and it may be best to allow the user to specify. We leave exploring this issue as future work.

Multistage attacks. Sometimes an attack on a vulnerability may take several steps to accomplish. For example, a buffer overflow may be exploitable only after an otherwise benign input is received. It is unclear in such attacks what inputs are exploits and which are benign, i.e., should we consider the priming message that leaves the server in a

vulnerable state malicious or not? Our approach has the same limitation as EMs: we cannot see into the future. This means that in a multistage, in order for a signature to be complete, it would have to wait until all input messages are received before allowing the signature to make a decision. We leave addressing multistage attacks as future work.

Complete signatures. A complementary approach to sound signature generation is to guarantee complete but perhaps unsound signatures, i.e., *overapproximate* the set of possible exploits. Our approach can easily be adapted to this setting. At a high level, a symbolic signature can be made complete but unsound by calculating the weakest precondition for the negated vulnerability condition, e.g., $wp(P, \neg c)$. The resulting formula will be true for all inputs that do not satisfy c. For example, we could use this approach to find an upper bound on the length of safe input strings. Other techniques are possible, e.g., abstract interpretation techniques such as data-flow analysis have also been proposed [72].

Application to nondeterministic programs. We currently only address programs that execute sequentially in our infrastructure. We consider multithreaded applications in which each request has its own thread sequential since we really only need to analyze a single thread. Many servers fall into this class. At a high level, there is no a priori reason why our techniques will not work with multithreaded applications for EM-enforceable properties. However, we currently have no experimental data on this type of problem; thus, we leave exploring any special issues raised by multithreaded applications as future work.

8 Conclusion

We presented a general framework for generating vulnerability signatures. Given a single sample exploit, we presented techniques for automatically generating a signature that is guaranteed to be sound. Our formulation works for vulnerabilities that can be exploited by multiple paths. In particular, we discuss three distinct types of vulnerability signature representations: TM, symbolic constraints, and regular expressions. We provide theoretical and practical insights into these three signature representations. Our evaluation indicates that our approach is promising.

Appendix

Theorem 1. *The BPM problem is PSPACE-complete.*

We prove this by first establishing that BPM is PSPACE-complete for nonrecursive Boolean signatures and then extend the proof to the recursive case.

Lemma 1. *BPM for a nonrecursive Boolean program is PSPACE-complete.*

Proof. At a high level, signature matching can be reduced to the model checking problem (MCP), which is known to be PSPACE-complete.

Let $\mathcal{B}$ be a nonrecursive Boolean program. Since all the variables can take values from a finite domain, without loss of generality, we can assume that $\mathcal{B}$ only contains Boolean variables (a variable with a finite domain can be encoded using a finite set of Boolean variables). Moreover, we can assume that every variable in $\mathcal{B}$ appears only *once* on the left-hand side of an assignment statement (this can be done by converting $\mathcal{B}$ to SSA form [31]). Since $\mathcal{B}$ is nonrecursive, we can inline all the function calls and hence assume that $\mathcal{B}$ does not have function calls.

The *MCP* is defined as follows:

Given a model M, an initial state I, and a Boolean formula f, determine if there exists a state reachable from I that satisfies f.

It is well known that MCP is PSPACE-complete [27], [73]. If there exists a state reachable from I in M that satisfies f, we will write it as $M,\ I \models f$. With the transformations outlined above, it is easy to see that we can write a finite state model $M_{\mathcal{B}}$ corresponding to the Boolean program $\mathcal{B}$, where the state variables of $M_{\mathcal{B}}$ are the variables of $\mathcal{B}$, and the transition relation of $M_{\mathcal{B}}$ corresponds to the transformers of the statements in $\mathcal{B}$. An initial state I of the model $M_{\mathcal{B}}$ corresponds to an assignment of the input variables. Let f be a Boolean formula on the state variables of $M_{\mathcal{B}}$. Let x be an input to the program $\mathcal{B}$ and let I_x be the initial state $M_{\mathcal{B}}$ that assigns values to the input variables according to x and assigns 0 to all local variables. Assume that the condition c associated with the vulnerability signature can be expressed as a Boolean formula f_c over the variables of $\mathcal{B}$. It is easy to see that $M,\ I_x \models f_c$ iff $x \in Sat(\mathcal{B}, c)$. Hence, we have a PSPACE algorithm for checking if an input x is in the set $Sat(\mathcal{B}, c)$. This proves that TSM is in PSPACE.

Given a model M with a finite set of state variables and a transition relation, it is easy to construct a Boolean program $\mathcal{B}_M$ whose variables correspond to the state variables of M and statements correspond to the transitions of M. Moreover, $M,\ I \models f$ iff $x_I \in Sat(\mathcal{B}, c_f)$, where the input x_I corresponding to the initial state I and the condition c corresponding to the Boolean formula f are constructed as before. Therefore, TSM is PSPACE-hard. Hence, TSM is PSPACE-complete. □

Lemma 2. *BPM for a Boolean program is PSPACE-complete.*

Proof. Let $\mathcal{B}$ be a Boolean program. The main complication is that in general, $\mathcal{B}$ can have recursive calls. In this case, we can construct a pushdown system PS corresponding to $\mathcal{B}$ such that $PS,\ I_x \models f_c$ iff $x \in Sat(\mathcal{B}, c)$, where I_x and f_c have exactly the same meaning as in the previous theorem. Fortunately, the MCP for pushdown systems can be performed in PSPACE [74]. This proves that the TSM for general Boolean programs is in PSPACE. Since the TSM for nonrecursive Boolean programs is PSPACE-hard, the TSM for general programs is also PSPACE-hard. Therefore, TSM is PSPACE-complete. □

Acknowledgments

This material is based upon work supported by the US National Science Foundation (NSF) under Grant 0448452, the US Department of Energy (DoE) Los Alamos National Laboratory under Grant W-7405-ENG-36, and

the Navy/ONR under Grant N00014-01-1-0708. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US National Science Foundation (NSF), the DoE, or the Navy/ONR.

REFERENCES

[1] C. Cerrudo, *Story of a Dumb Patch,* http://argeniss.com/research/MSBugPaper.pdf, 2005.
[2] T. Detristan, T. Ulenspiegel, Y. Malcom, and M.V. Underduk, *Polymorphic Shellcode Engine Using Spectrum Analysis,* http://www.phrack.org/show.php?p=61&a=9, 2003.
[3] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymophic Worm Detection Using Structural Information of Executables," *Proc. Int'l Symp. Recent Advances in Intrusion Detection,* 2005.
[4] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically Generating Signatures for Polymorphic Worms," *Proc. IEEE Symp. Security and Privacy,* May 2005.
[5] M. Jordan, "Dealing with Metamorphism," *Virus Bull. Magazine,* 2002.
[6] P. Szor, "Hunting for Metamorphic," *Proc. 11th Ann. Virus Bull. Conf. and Exhibition,* 2001.
[7] "Metasploit," http://metasploit.org, 2008.
[8] H.J. Wang, C. Guo, D. Simon, and A. Zugenmaier, "Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits," *Proc. ACM SIGCOMM '04,* Aug. 2004.
[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worms," *Proc. 20th ACM Symp. Operating System Principles (SOSP),* 2005.
[10] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee, "Polymorphic Blending Attacks," *Proc. 15th Usenix Security Symp.,* 2006.
[11] S. Chung and A. Mok, "Allergy Attack against Automatic Signature Generation," *Proc. Ninth Int'l Symp. Recent Advances in Intrusion Detection (RAID),* 2006.
[12] J. Newsome, B. Karp, and D. Song, "Paragraph: Thwarting Signature Learning by Training Maliciously," *Proc. Ninth Int'l Symp. Recent Advances in Intrusion Detection (RAID '06),* Sept. 2006.
[13] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading Worm Signature Generators Using Deliberate Noise Injection," *Proc. IEEE Symp. Security and Privacy,* May 2006.
[14] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS '05),* Feb. 2005.
[15] Z. Liang and R. Sekar, "Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS),* 2005.
[16] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards Automatic Generation of Vulnerability-Based Signatures," *Proc. IEEE Symp. Security and Privacy,* pp. 2-16, 2006.
[17] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating Vulnerability Signatures Using Weakest Pre-Conditions," *Proc. 20th IEEE Computer Security Foundations Symp. (CSF),* 2007.
[18] J. Crandall, Z. Su, S.F. Wu, and F. Chong, "On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS),* 2005.
[19] H.-A. Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," *Proc. 13th Usenix Security Symp.,* Aug. 2004.
[20] C. Kreibich and J. Crowcroft, "Honeycomb—Creating Intrusion Detection Signatures Using Honeypots," *Proc. Second Workshop Hot Topics in Networks (HotNets '03),* Nov. 2003.
[21] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated Worm Fingerprinting," *Proc. Sixth ACM/Usenix Symp. Operating System Design and Implementation (OSDI '04),* Dec. 2004.
[22] Z. Li, M. Shanghi, B. Chavez, Y. Chen, and M.-Y. Kao, "Hamsa: Fast Signature Generation for Zero-Day Polymorphic Worms with Provable Attack Resilience," *Proc. IEEE Symp. Security and Privacy,* 2006.
[23] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. ACM SIGPLAN Int'l Conf. Programming Language Design and Implementation (PLDI),* 2005.
[24] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *Proc. Fifth Joint Meeting of the European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC/FSE '05),* pp. 263-272, 2005.
[25] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution," *Proc. 13th ACM Conf. Computer and Comm. Security (CCS '06),* Oct. 2006.
[26] V. Ganesh and D. Dill, "A Decision Procedure for Bit-Vectors and Arrays," *Proc. 19th Int'l Conf. Computer Aided Verification Conf. (CAV '07),* Aug. 2007.
[27] E. Clarke, O. Grumberg, and D. Peled, *Model Checking.* MIT Press, 1999.
[28] D.B. Whalley, "Automatic Isolation of Compiler Errors," *ACM Trans. Programming Languages and Systems,* vol. 16, no. 5, pp. 1648-1659, Sept. 1994.
[29] B. Ness and V. Ngo, "Regression Containment through Source Change Isolation," *Proc. 21st Int'l Computer Software and Applications Conf. (COMPSAC '97),* p. 616, 1997.
[30] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" *Proc. Seventh European Software Eng. Conf Held Jointly with the Seventh ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC/FSE '99),* pp. 253-267, Sept. 1999.
[31] S. Muchnick, *Advanced Compiler Design and Implementation.* Academic Press, 1997.
[32] K.R.M. Leino, "Efficient Weakest Preconditions," *Information Processing Letters,* vol. 93, no. 6, pp. 281-288, 2005.
[33] M. Barnett and K.R.M. Leino, "Weakest-Precondition of Unstructured Programs," *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Analysis For Software Tools and Eng. (PASTE),* 2005.
[34] F.B. Schneider, "Enforceable Security Policies," *ACM Trans. Information and System Security,* vol. 3, no. 1, pp. 30-50, Feb. 2000.
[35] L. Lamport and F.B. Schneider, "Formal Foundation for Specification and Verification," *Distributed Systems. Methods and Tools for Specification. An Advanced Course.,* M. Paul and H. Siegert, eds., vol. 190, pp. 203-270, 1985.
[36] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS '05),* pp. 340-353, 2005.
[37] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.
[38] J. Hopcroft, *An $n \log n$ Algorithm for Minimizing the States in a Finite Automaton,* Z. Kohavi, ed., Academic Press, 1971.
[39] T. Reps, "Program Analysis via Graph Reachability," *Information and Software Technology,* vol. 40, nos. 11-12, 1998.
[40] D. Jackson and E. Rollins, "Chopping: A Generalisation of Slicing," *Proc. Second ACM SIGSOFT Symp. Foundations of Software Eng. (FSE),* 1994.
[41] T. Reps and G. Rosay, "Precise Interprocedural Chopping," *Proc. Third ACM SIGSOFT Symp. Foundations of Software Eng. (FSE),* 1995.
[42] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah, "Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software," *Proc. 13th Ann. Network and Distributed System Security Symp. (NDSS),* 2006.
[43] D. Brumley, "Analysis and Defense of Vulnerabilities in Binary Code," PhD dissertation, School of Computer Science, Carnegie Mellon Univ., 2008.
[44] D. Brumley and J. Newsome, "Alias Analysis for Assembly," Technical Report CMU-CS-06-180, School of Computer Science, Carnegie Mellon Univ., 2006.
[45] P. Bosch, A. Carloganu, and D. Etiemble, "Complete $\times 86$ Instruction Trace Generation from Hardware Bus Collect," *Proc. 23rd IEEE EUROMICRO Conf.,* 1997.
[46] P.A. Sandon, Y. Liao, T. Cook, D. Schultz, and P.M. de Nicolas, "NStrace: A Bus-Driven Instruction Trace Tool for PowerPC Microprocessors," *IBM J. Research and Development,* vol. 41, no. 3, 1997.
[47] "Dynamorio," http://www.cag.lcs.mit.edu/dynamorio/, 2008.
[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. ACM SIGPLAN Int'l Conf. Programming Language Design and Implementation (PLDI '05),* June 2005.

[49] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," *Proc. Third Workshop Runtime Verification (RV '03),* July 2003.
[50] A. Milenkovic, M. Milenkovic, and J. Kulick, "N-Tuple Compression: A Novel Method for Compression of Branch Instruction Traces," *Proc. ISCA 16th Int'l Conf. Parallel and Distributed Computing (PDCS),* 2003.
[51] R.A. Uhlig and T. Mudge, "Trace-Driven Memory Simulation: A Survey," *ACM Computing Surveys,* vol. 29, 1997.
[52] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools,* second ed. Addison-Wesley, 2007.
[53] E. Dijkstra, *A Discipline of Programming.* Prentice Hall, 1976.
[54] D. Detlefs, K.R.M. Leino, G. Nelson, and J. Saxe, "Extended Static Checking," Technical Report 159, Compaq Systems Research Center, Dec. 1998.
[55] *QEMU—Open Source Processor Emulator,* http://fabrice.bellard.free.fr/qemu/, 2008.
[56] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," *Proc. 13th Usenix Security Symp.,* 2004.
[57] V. Ganesh and D. Dill, "STP: A Decision Procedure for Bit-Vectors and Arrays," http://theory.stanford.edu/vganesh/stp, 2008.
[58] C. Barrett and S. Berezin, "CVC Lite: A New Implementation of the Cooperating Validity Checker," *Proc. 16th Int'l Conf. Computer Aided Verification Conf. (CAV '04),* R. Alur and D.A. Peled, eds., 2004.
[59] Y. Ramin, "Atphttpd 0.4b," http://jnewsome.net/src/atphttpd.html, 2008.
[60] r code, "Atphttpd Remote Get Request Buffer Overrun Vulnerability,"http://www.securityfocus.com, Bugtraq ID 8709.
[61] dong-h0un U, "Passlog Daemon sl_parse Remote Buffer Overflow Vulnerability," http://www.securityfocus.com, Bugtraq ID 7261, 2008.
[62] pyramid-rp@hushmail.com, "ghttpd log() Function Buffer Overflow Vulnerability," http://www.securityfocus.com, Bugtraq ID 5960, 2008.
[63] D. Defense, "Samba call_trans2open Remote Buffer Overflow Vulnerability," http://www.securityfocus.com/bid/7294/discuss, 2003.
[64] Symantec, *Blaster Worm,* http://www.symantec.com/security_response/writeup.jsp?docid=2003-081113-0229-99, 2003.
[65] Symantec, *W32.sqlexp.worm (Slammer Worm),* http://www.symantec.com/security_response/writeup.jsp?docid=2003-012502-3306-99, 2003.
[66] J. King, "Symbolic Execution and Program Testing," *Comm. ACM,* vol. 19, pp. 386-394, 1976.
[67] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," *Proc. IEEE Security and Privacy Symp.,* 2007.
[68] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating Mimicry Attacks Using Static Binary Analysis," *Proc. 14th Usenix Security Symp.,* 2005.
[69] C. Flanagan and J. Saxe, "Avoiding Exponential Explosion: Generating Compact Verification Conditions," *Proc. 28th ACM Symp. Principles of Programming Languages (POPL),* 2001.
[70] G. Balakrishnan and T. Reps, "Analyzing Memory Accesses in ×86 Executables.," *Proc. 13th Int'l Conf. Compiler Construction (CC '04),* pp. 5-23, 2004.
[71] S.K. Debray, R. Muth, and M. Weippert, "Alias Analysis of Executable Code," *Proc. 15th Ann. Symp. Principles of Programming Languages (POPL '88),* pp. 12-24, 1988.
[72] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, *Theory and Techniques for Automatic Generation of Vulnerability-Based Signatures,* Technical Report CMU-CS-06-108, Computer Science Dept., Carnegie Mellon Univ., Feb. 2006.
[73] A.P. Sistla and E.M. Clarke, "The Complexity of Propositional Linear Temporal Logics," *J. ACM,* vol. 32, no. 3, pp. 733-749, 1985.
[74] A. Bouajjani and O. Maler, "Reachability Analysis of Pushdown Automata," *Proc. Int'l Workshop Verification of Infinite-State Systems (Infinity),* 1996.

David Brumley received the PhD degree from Carnegie Mellon University, Pittsburgh, Pennsylvania, in 2008, and he is an assistant professor. His work focuses on computer security. His interests also include formal methods, compilers, and programming languages.

James Newsome received the bachelor's degree in computer engineering in 2002 from the University of Michigan and the PhD degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, in 2008. He is currently with Carnegie Mellon University. In the course of his thesis work, he has published several papers on automatic detection, analysis, and mitigation of software exploits.

Dawn Song received the PhD degree in computer science from the University of California, Berkeley (UC Berkeley) in 2002. She is an assistant professor at UC Berkeley. Prior to joining UC Berkeley, she was an assistant professor at Carnegie Mellon University from 2002 to 2007. Her research interest lies in security and privacy issues in computer systems and networks. She is the author of more than 70 research papers in software security, networking security, database security, distributed systems security, and applied cryptography. She is the recipient of various awards, including the US National Science Foundation CAREER Award, the IBM Faculty Award, the George Tallman Ladd Research Award, the Sloan Award, the Okawa Foundation Research Grant Award, and Best Paper Awards in top security conferences.

Hao Wang completed the PhD degree from the Computer Sciences Department, University of Wisconsin, Madison, in 2007, where he worked under the guidance of Professor Somesh Jha. He is currently with the Computer Science Department, University of Wisconsin. His primary research interests are static and dynamic techniques for malware analysis, detection, and prevention.

Somesh Jha received the BTech degree in electrical engineering from Indian Institute of Technology, New Delhi, and the PhD degree in computer science from Carnegie Mellon University, Pittsburgh, Pennsylvania, in 1996. Currently, he is an associate professor in the Computer Science Department, University of Wisconsin, Madison, which he joined in 2000. His work focuses on analysis of security protocols, survivability analysis, intrusion detection, formal methods for security, and analyzing malicious code. Recently, he has also worked on privacy-preserving protocols. He has published more than 90 papers in highly refereed conferences and prominent journals. He has won numerous best paper awards. He also received the US National Science Foundation CAREER Award in 2005.