Theory and Techniques for Automatic Generation of Vulnerability-Based Signatures

David Brumley, James Newsome, and Dawn Song
Carnegie Mellon University
Pittsburgh, PA, USA
{dbrumley, jnewsome, dawnsong}@cmu.edu

Hao Wang and Somesh Jha
University of Wisconsin-Madison
Madison, WI, USA
{hbwang, jha}@cs.wisc.edu

February 2006
CMU-CS-06-108
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work supported by the National Science Foundation under Grant No. 0448452, the Department of Energy (DOE) Los Alamos National Laboratory under Grant No. W-7405-ENG-36, and the Navy/ONR under Grant No. 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 National Science Foundation, the Department of Energy, or the Navy/ONR.

Keywords: computer security, vulnerability, signature generation, software security, polymorphic worm, metamorphic worm

Abstract

In this paper we explore the problem of creating *vulnerability signatures*. A vulnerability signature matches all exploits of a given vulnerability, including polymorphic and metamorphic variants. Our work departs from previous approaches by focusing on the semantics of the program and vulnerability exercised by a sample exploit instead of the semantics or syntax of the exploit itself. We show the semantics of a vulnerability define a language which contains all and only those inputs that exploit the vulnerability. A vulnerability signature is a representation (e.g., a regular expression) of the vulnerability language. Unlike exploit-based signatures whose error rate can only be empirically measured for known test cases, the quality of a vulnerability signature can be formally quantified for all possible inputs.

We provide a formal definition of a vulnerability signature and investigate the computational complexity of creating and matching vulnerability signatures. We also systematically explore the design space of vulnerability signatures. We identify three central issues in vulnerability-signature creation: how a vulnerability signature *represents* the set of inputs that may exercise a vulnerability, the vulnerability *coverage* (i.e., number of vulnerable program paths) that is subject to our analysis during signature creation, and how a vulnerability signature is *created* for a given representation and coverage.

We propose new data-flow analysis and a novel adoption of existing techniques, such as constraint solving, for automatically generating vulnerability signatures. We have built a prototype system to test our techniques. Our experiments show that we can, using a single exploit, automatically generate a vulnerability signature which is of much higher quality than previous exploit-based signatures. In addition, our techniques have several other security applications, and thus may be of independent interest.

1 Introduction

A *vulnerability* is a type of bug that can be used by an attacker to alter the intended operation of the software in a malicious way. An *exploit* is an actual input that triggers a software vulnerability, typically with malicious intent and devastating consequences. One of the most popular and effective exploit defense mechanisms is signature-based input filtering (also called content-based filtering) [1, 2]. Thus, any improvements in signature generation will likely have widespread impact.

We need *automatic* signature generation techniques because manual signature generation is slow and error prone. Fast generation is 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 [1, 3]. Automatic 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 very difficult for humans at even the source code level [4], let alone COTS software at the assembly level.

Challenges for automatically creating signatures. 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 [5–7]. 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 [8, 9], e.g., an exploit could use different assembly instructions that have the same effect. Our approach does not need to distinguish between polymorphism and metamorphism: both are referred to as polymorphism throughout this paper. Many morphing tools are publicly available to automatically generate polymorphic exploit variants [5, 10, 11]. *Thus, 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 [12]).

Limitations of previous approaches. The importance of the signature generation problem has recently prompted researchers to investigate automatic signature generation techniques. Previous approaches fall into at least one of the following categories: (a) require manual steps, (b) employ heuristics which may fail in many settings, (c) techniques rely on specific properties of an exploit (e.g., return addresses) and are thus not *vulnerability* signatures, (d) are limited by the underlying signature representation they can generate, or (e) only work for specific vulnerabilities in specific circumstances. Previous approaches also do not provide metrics for measuring the intrinsic accuracy of a signature. Instead, previous work measured accuracy based upon empirical trials, e.g., applying a regular expression signature to a known data stream. Empirical trials are insufficient because they do not measure signature matching accuracy against unknown exploits for the same vulnerability.

For example, consider pattern-extraction based methods which syntactically identify bit patterns that appear in attack samples but not in innocuous samples [7, 13–15]. These techniques are either incapable of handling polymorphic worms [13–15], or vulnerable in an adversarial environment in which an attacker can inject false or superfluous tokens, such as an over-learning or "red-herring" attack [7, 16]. Another approach is based on application and exploit semantic information [17–19]. However, these techniques are heuristics-based and rely on specific properties of the exploits such as the value used to overwrite the return address to be invariant. It has been shown previously these heuristics may not work in many real-world vulnerabilities [7, 20].

Finally, a perfect signature — a signature with zero false positives and negatives — must be at least the same language class as the vulnerability language. For instance, if the vulnerability language requires counting to distinguish exploits from benign inputs, a regular expression signature cannot be perfect since regular expressions cannot count [21]. Previous work has not systematically explored the design space of signature creation, instead focusing on a single design point such as creating regular expressions for control-hijacking attacks. Regular-expressions can only recognize simple syntactic properties, thus may not be precise for many vulnerabilities.

Our approach, roadmap, and the central issues. Our approach departs from previous work by analyzing the vulnerability uncovered by a new exploit attack instead of analyzing the exploit. At a high level, our main contribution is a new class of signature, which we call a *vulnerability signature*, that is not specific to details such as whether an exploit successfully hijacks control of the program, but instead whether executing an input will (potentially) result in an unsafe execution state. (Note that control hijacks are only one type of unsafe execution, as are denial of service, subverting data integrity, violating type safety, etc. The set of inputs (exploits) for any single attack type are therefore a subset of all inputs that result in an unsafe execution.)

In this paper we present a formal approach for reasoning about and creating vulnerability signatures. Intuitively, a vulnerability signature matches a *set of inputs* (strings) which satisfies a *vulnerability condition* with respect to a program. A vulnerability condition is a specification of a particular type of program bug, e.g., memory writes should be within the allocated buffer space. We then systematically explore the design space of vulnerability signatures, and identify two important dimensions: which language class the signature is *represented*, in which there is an expressiveness trade-off between matching accuracy and efficiency, and how much of the vulnerability is *covered* by the signature, in which there is a trade-off between the amount of analysis performed and the signature false negative rate. We then develop new techniques for creating vulnerability signatures for different representations. We focus on three representations which highlight the inherent accuracy, efficiency, and creation time trade-offs in the design space: *Turing machine* signatures, *symbolic constraint* signatures, and *regular expression* signatures.

Contributions. This paper presents a systematic approach using a formal model and methods to create vulnerability signatures using static program analysis. We require only a single sample exploit which is used to initially identify the vulnerability. Our automatic signature generation approach is applicable to all vulnerabilities in which the vulnerability condition can be formally specified. Our approach uncovers a rich new domain for representing signatures and new techniques for creating them. In particular:

- We provide a formal definition for vulnerability signatures. Our approach leads to a new perspective where a vulnerability signature can be represented by different language classes with different expressive powers.
- We explore the design space of vulnerability signature and show that there is an inherent trade-off between signature matching and accuracy for different representations. In particular, a perfect signature can be created (Turing machine signatures in Section 2.3), but matching may take an unbounded amount of time. On the other hand, we can create signatures that allow fast matching but are less accurate (regular-expression signatures in Section 2.3).
- We introduce the notion of vulnerability signature coverage. As we will see, one challenge is that a vulnerability may be reachable by an infinite number of paths in the program (in the presence of looping). We show how to iteratively consider each path separately so that signature generation can scale.
- Our methods allow us to identify where a created vulnerability signature approximates a perfect vulnerability signature. Specifically, in our setting, one can identify and control when and how imprecision is introduced. This property makes it easy to quantify the quality of the generated vulnerability signature.
- We develop new static analysis techniques (such as the regular expression data-flow framework in Section 4.4.2), and make novel adoptions of existing techniques such as program chopping and constraint satisfaction to our problem domain.
- We provide a prototype implementation of our techniques and automatically create signatures for several real-world vulnerabilities. Our prototype addresses automatic signatures creation in one of the hardest scenarios: only the program binary is used. We do not require source code or type information, and therefore our prototype is applicable to COTS software.
- Our results show that our techniques automatically generate signatures that are of a much higher quality than previous techniques.

2 Vulnerability Signature

In this section we first give a formal definition of a vulnerability and a vulnerability signature. Intuitively, a vulnerability signature is a representation for the set of inputs which satisfy a specified vulnerability condition. We then explore two dimensions of the design space for vulnerability signatures: signature representation and coverage. Roughly speaking, design points in the signature representation dimension trade-off accuracy and efficiency. Design points in the vulnerability signature creation dimension trade-off creation time for signature coverage, i.e., how many program paths are analyzed.

Motivating Scenario. We motivate our work and approach to vulnerability signatures in the following setting: a new exploit is just released for an unknown vulnerability. A site has detected the exploit through some means such as dynamic taint analysis or stack protection, and wishes to create a signature that recognizes any further exploits. The site can furnish our analysis with the tuple $\{\mathcal{P}, T, x, c\}$ where $\mathcal{P}$ is the program, x is the exploit string, c is a vulnerability condition, and T is the execution trace of $\mathcal{P}$ on x. Since our experiments are at the assembly level, we

```
1  char *get_url(char inp[10]) {
2    char *url = malloc(4);
3    int c = 0;
4    if (inp[c] != 'g' && inp[c] != 'G')
5      return NULL;
6    inp[c] = 'G';
7    c++;
8    while (inp[c] == ' ')
9       c++;
10   while (inp[c] != ' ') {
11     *url = inp[c]; c++; url++;
12   }
13   printf("\%s", url);
14   return url;
15 }
```

Figure 1: Our running example, which returns the URL of a request of the form `[g|G] <url>`, else NULL.

assume $\mathcal{P}$ is a binary program and T is an instruction trace, though our techniques also work at the source-code level. Our goal is to create a vulnerability signature which will *match* future malicious inputs x' without running $\mathcal{P}$.

In addition, we want to create signatures quickly since in many scenarios signatures must be deployed almost immediately after detection to be of any value. Therefore, we take an iterative approach that generates successively better signatures. Each successive signature will match more exploit variants *without requiring further exploit samples*.

Running example. Throughout this paper, we use the running example given in Figure 1. Our example is in a C-like language for clarity; our implementation operates on program binaries. The example returns the URL when the request begins with the 'G' or 'g' keyword, else NULL is returned. In our example, we will assume $x =$ `g /AAAA`. The corresponding trace is $T = \{1, 2, 3, 4, 6, 7, 8, 9, 8, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11\}$ where each number is the corresponding line number in Figure 1. The vulnerability condition is a heap overflow, which the input x satisfies (i.e., the program is exploited) on the 5^{th} iteration of line 11 since the URL is 5 characters long while only 4 characters were allocated.

2.1 Background Definitions

A program $\mathcal{P}$ consists of a finite sequence of instructions $< i_1, ..., i_k >$. Instructions are referred to by their number, so in the program $\mathcal{P} =$ `<inc x, dec y>`, we can refer unambiguously to `dec y` as instruction 2. A variable in a program is any location read or written. In a high level language, variables are declared and are usually given nice names such as "x" and "sum". At the binary level, any register, stack location, or heap location that may be read or written is a variable. Each variable is assigned a unique name (derived from a possibly arbitrary scheme), e.g., register `%ebp` could be referred to as variable v_{ebp}, stack location $0x80$ is variable v_{s0x80}, heap location $0xc$ is variable v_{h0xc}, etc. For clarity, we denote the variable name for as in, i.e., the input variables names $in_0, in_1, ..., in_k$ denote the first k input locations.

A program state, denoted $\mathcal{M}$, is an assignment of values to program variables (uninitialized variables are given a special distinguished value $\perp$). We denote the initial state of a program variables before execution by $\mathcal{M}^0$, and in general the state before executing instruction i as $\mathcal{M}^i$. We denote updating variable v_i in $\mathcal{M}$ with value a as $\mathcal{M}[v_i/a]$.

The execution trace of a program $\mathcal{P}$ on input $x = x_0x_1x_2...x_k$ is denoted denoted $T_{\mathcal{P}(x)}$. Formally, the execution trace consists of a sequence of program states $< \mathcal{M}^0[in_0/x_0, in_1/x_1, ..., in_k/x_k], \mathcal{M}^1, \mathcal{M}^2, ... >$. In practice, the execution trace includes only the input and the instruction addresses executed, from which we can infer the necessary program state.

For now, we assume all programs are deterministic. This assumption is met by many programs of interest. We revisit this assumption in Section 6.

2.2 Vulnerability, Vulnerability Condition, Vulnerability Language, and Vulnerability Signature Definitions

Intuitively, a vulnerability in a program is a particular instruction which, when executed with malicious or malformed operands, will result in an unsafe execution state. The instruction number, called the *vulnerability point*, helps to distinguish among potentially many vulnerabilities within a program. Typical unsafe program states include an overwritten return address, dead-lock, type safety violations, etc. We refer to a program in an unsafe execution state as in the EXPLOIT state.

A vulnerability point, while necessary, is not sufficient for uniquely specifying a single vulnerability. Consider a function like `strcpy` which copies a source buffer to a destination buffer. `strcpy` is only unsafe when the source buffer is smaller than the destination buffer. Thus, we may need to provide an additional *vulnerability condition*, along with the vulnerability point, to uniquely identify a vulnerability. The vulnerability condition for a `strcpy` vulnerability would specify that the destination buffer should be at least as large as the source buffer.

The vulnerability condition c, for program $\mathcal{P}$ in a state specified by $\mathcal{M}_{\mathcal{P}}$, returns whether executing instruction i would result in an EXPLOIT state. In our setting, $\mathcal{M}_{\mathcal{P}}$ will be given by the trace. We model the vulnerability condition c as the function:

$$c : \mathcal{P} \times \mathcal{M}_{\mathcal{P}} \times i \rightarrow \{\text{EXPLOIT}, \text{BENIGN}\}$$

We allow c to have its own local static memory to allow for vulnerabilities that may depend on additional information not present in the current program state. [1] For example, c's local memory may specify which memory locations are "tainted" by user input. As another example, consider a simple vulnerability condition c_{stack} for stack buffer overflow vulnerabilities. Then c_{stack}, in order to detect a vulnerability, must know which memory cells contain stack addresses. Such cells must be specified in c's memory, either by providing it externally, or in our case, deriving that information automatically from the trace.

We assume we are given the vulnerability condition. We assume this because vulnerability conditions have already been extensively studied by others, and we do not duplicate their work. For example, the operational semantics for what constitutes a vulnerability are often specified as part of an exploit detector, static bug-finding tools, program correctness and verification tools, and for programming languages. The semantics consist of an unambiguous set of rules that describe possible state transitions that may lead to an unsafe execution state. In our scenario, c is just an implementation of those semantics. Thus, our contribution is not how to specify the vulnerability condition itself, but instead how such semantics can be adopted to the problem of creating a vulnerability signature.

When not given the vulnerability point ℓ explicitly, we automatically derive ℓ by sequentially processing the trace and recording as the vulnerability point the first time c returns EXPLOIT. For example, an exploit detector such as Stackguard [22] will only tell us that a stack return address was overwritten, not which instruction overwrote the stack address. To process the trace, we use x to initialize the initial memory state $\mathcal{M}^0_{\mathcal{P}}$. We then feed c the instructions in the execution trace to find the exact vulnerability point:

$$\begin{aligned} c(\mathcal{P}, \mathcal{M}^0_{\mathcal{P}}, i_0) &\rightarrow \text{BENIGN} \\ c(\mathcal{P}, \mathcal{M}^1_{\mathcal{P}}, i_1) &\rightarrow \text{BENIGN} \\ &\dots \\ c(\mathcal{P}, \mathcal{M}^{\ell-1}_{\mathcal{P}}, i_\ell) &\rightarrow \text{EXPLOIT} \end{aligned}$$

A *vulnerability* $\mathcal{V}$ is 3-tuple $(\mathcal{P}, \ell, c)$, where $\mathcal{P}$ is a program, ℓ is the vulnerability point, and c is the vulnerability condition. We say an input x *exploits* a vulnerability $\mathcal{V}$ if c returns EXPLOIT after executing instruction ℓ in $\mathcal{P}$, written $\mathcal{P}_\ell(x) \models \mathcal{V}$. The *language* of a vulnerability $L_{\mathcal{P},\ell,c}$ consists of the set of all exploits:

$$L_{\mathcal{P},\ell,c} \doteq \{x \in \Sigma^* \mid \mathcal{P}_\ell(x) \models \mathcal{V}\}$$

i.e., all inputs that cause the vulnerability condition to return EXPLOIT when executing line ℓ.

[1] The local memory could also be passed as an argument to c, which c could then modify and return to the caller.

```
1  char *url = malloc(4);
2  int c = 0;
3  if (inp[c] != 'g' && inp[c] != 'G')
4    return BENIGN;
5  c++;
6  while (inp[c] == ' ') c++;
7  while (inp[c] != ' ') {
8     if (c >= 4) return EXPLOIT;
9     *url = inp[c]; c++; url++;
10 }
11 return BENIGN;
```

Figure 2: The TM signature for our running example.

A *vulnerability signature* is a matching function MATCH which for an input x returns either EXPLOIT or BENIGN without running $\mathcal{P}$. A *perfect* vulnerability signature satisfies the following property:

$$\text{MATCH}(x) = \begin{cases} \text{EXPLOIT when } x \in L_{\mathcal{P},\ell,c} \\ \text{BENIGN when } x \notin L_{\mathcal{P},\ell,c} \end{cases}$$

Soundness and completeness for signatures. We define completeness for a vulnerability signature MATCH to be $\forall x : x \in L_{\mathcal{P},\ell,c} \Rightarrow \text{MATCH}(x) = \text{EXPLOIT}$, i.e., MATCH accepts everything $L_{\mathcal{P},\ell,c}$ does. Incomplete solutions will have false negatives. We define soundness as $\forall x : x \notin L_{\mathcal{P},\ell,c} \Rightarrow \text{MATCH}(x) = \text{BENIGN}$, i.e., MATCH does not accept anything extra not in $L_{\mathcal{P},\ell,c}$. Unsound solutions will have false positives. A consequence of Rice's theorem [21] is that no signature representation other than a Turing machine can be both sound and complete, and therefore for other representations we must pick one or the other. In our setting, we focus on soundness, i.e., we tolerate false negatives but not false positives. In Section 6 we show how to reformulate our algorithm to generate complete but unsound signatures.

As we show in Section 2.3, the signature for $L_{\mathcal{P},\ell,c}$ can be represented in many different ways ranging from Turing machines which are precise, i.e., accept exactly $L_{\mathcal{P},\ell,c}$, to regular expressions which may not be precise, i.e., have an error rate.

Application to the Motivating Scenario. In our motivating scenario, the site has detected an exploit and can provide us with an instruction trace $T =< i_1, i_2, i_3, ..., i_k >$ and the initial input x. The vulnerability condition for each kind of vulnerability (e.g., stack overwrite, misuse of tainted data, etc.) is specified ahead of time (the next section addresses specifying the vulnerability condition). In our motivating scenario, we also assume we are given the vulnerability point ℓ and an appropriate vulnerability condition c for the exploit detected.

In our running example, the vulnerability condition should return EXPLOIT if there is an out-of-bounds pointer dereference on line 11. One implementation of such a vulnerability condition is to shadow each reach pointer creation, manipulation, and dereference with a shadow safe pointer (similar as done with CCured [23]). A typical safe pointer is a structure `SafePtr(base, offset, size)` which contains the original base address of the pointer, the current offset, and the total size.

At this point, we have a specification for the vulnerability: $(\mathcal{P}, \ell, c)$. Note we also have a Turing machine which recognizes exactly the vulnerability language: the vulnerable program itself using c at each stage of the execution. The remaining task is to create an appropriate vulnerability signature from the vulnerability condition and the vulnerable program.

2.3 Signature Representation Classes

The vulnerable program along with the vulnerability condition recognize exactly the language of the vulnerability. Therefore, a perfect signature can be created by inlining the vulnerability condition into the original program. Even

```
(inp[0] = 'g' ∨ inp[0] = 'G') ∧
[(inp[1:5] != ' ') ∨
(inp[1] =' '∧ inp[2:6] !=' ') ∨
(inp[1:2] =' '∧ inp[3:7] !=' ') ∨
(inp[1:3] =' '∧ inp[4:8] !=' ') ∨
(inp[1:4] =' '∧ inp[5:9] !=' ') ]
```

Figure 3: The symbolic constraint signature for our running example.

though such a signature is perfect, there are no guarantees on how long it will take to match a given input. In practice, though, matching speed and other factors may be important as well.

A perfect signature — a signature that is both sound and complete — must be at least the same language class as the vulnerability language. Our approach allows us to trade-offs between accuracy and other factors such as efficiency by representing the language of a vulnerability $L_{\mathcal{P},\ell,c}$ with a less expressive language. A representation of the vulnerability language in a lower class language is an estimation and cannot be exact, thus may be sound, complete, or neither. We pick three concrete language classes which highlight the fundamental trade-offs between accuracy and matching efficiency: *Turing machine* signatures, *symbolic constraint* signatures, and *regular expression* signatures. We define each vulnerability signature representation, provide algorithmic complexity bounds for various signature operations, and provide algorithms for constructing each signature type.

Turing machine signatures. A Turing machine (TM) signature is a program which recognizes the vulnerability language. A Turing machine signature S can be created by including those instructions which lead to the vulnerability point with the vulnerability condition algorithm inlined. Paths that do not lead to the vulnerability point will return BENIGN, while paths that lead to the vulnerability point and satisfy the vulnerability condition return EXPLOIT. [2] TM signatures can be precise, e.g., a trivial TM signature with no error rate is emulating the full program. A TM signature for our running example is given in Figure 2:

Symbolic constraint signatures. A symbolic constraint signature is a set of Boolean formulas which approximate a Turing machine signature. The approximation is loop free, but may have universal and existential quantifiers. Unlike Turing machine signatures, matching (evaluating) a symbolic constraint signature on an input x will always terminate. Symbolic constraint signatures only approximate constructs such as loops and memory updates statically. As a result, symbolic constraint signatures may not be as precise as a Turing machine signature.

Let `x:y` represent an inclusive range, e.g., `inp[1:5]` means input bytes 1 through 5, inclusive. Then the symbolic constraint signature (after considerable simplification for readability) for our running example is given in Figure 3.

This signature states that the ten-byte input matches the signature if the first input byte is 'G' or 'g', followed by anywhere from 0 to 4 space characters, followed by at least 5 non-space characters. At least 5 non-space characters are needed to overflow the 4-byte allocated `url` buffer. Note that this signature is created by unrolling the loops on lines 8-9 and 10-12 of the TM signature. Although in our example we can statically infer how many times to unroll the loop, in general such inferences are not possible and an upper bound to unroll loops must be provided (this is the same approach taken by bounded model checkers [24]).

Regular expression signatures. Regular expressions are the least powerful signature representation of the three, and may have a considerable error rate in some circumstances. [3] For example, a well-known limitation is regular expressions cannot count [21], and therefore cannot succinctly express conditions such as checking 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. Our algorithm for computing a regular expression signature creates sound signatures, e.g., zero false positives.

The regular expression signature we would produce for our running example (using the data-flow techniques described in Section 4.4) is `[g|G][ ]*[^␣]{5,}`, which matches any input that begins with 'g' or 'G', followed by

[2] A path in a program is a path in the program's control flow graph.

[3] We use a GNU-style syntax for regular expressions. See Appendix .1.

Representation	Creation	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($S \log S$)

Table 1: Summary of approximate bounds for the three vulnerability signature representations we consider for a program of length N and signature size S. poly(X) denotes a function polynomial in X, and exp(X) denotes a function exponential in X.

zero or more spaces, followed by at least 5 or more (represented as $\{$`5,`$\}$) non-space characters .

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 [25]. One of the main contributions from our construction is any language class may be used to represent a signature. The user is free to pick the appropriate representation for their situation.

2.4 Signature Operations and Efficiency

Many factors may factor into how a signature representation is chosen. We provide bounds for 5 important properties of each signature type: creation time (for our algorithm), resulting signature size, matching time, minimization time, and equivalence time. Table 1 summarizes the results. Signature minimization takes a signature S and computes the smallest signature S_{min} in the same class language as S which recognizes the language L_S. A minimized signature takes the least amount of space, and is generally more efficient to match. Signature equivalence is determining whether two signatures S_1 and 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. The remaining properties are straight-forward.

Signature *merging* is an important operation. In our model, merging signatures S_1 and S_2 is equivalent to performing a single analysis of $L_{S_1 S_2} = L_{\mathcal{P},\ell_1,c_2} \cup L_{\mathcal{P},\ell_2,c_2}$, that is, the union of the languages for both vulnerabilities. The union operation for TM signatures is done by creating a new condition $c_{12} = c_1 \vee c_2$, and considering $L_{\mathcal{P},\ell_1 \vee \ell_2,c_{12}}$. The union operation for symbolic constraints is the disjunction of the individual constraints, i.e., either constraint system could be satisfied. The union operation for regular expression is the "or" (|) operator.

TM signature bounds. First, we prove that matching time for Turing machine signatures is undecidable. A Turing machine signature is a program which accepts the language of the vulnerability $L_{\mathcal{P},\ell,c}$. The Turing machine signature matching (TSM) problem is:

Given a program $\mathcal{P}$, a condition c, and an input x, determine whether $x \in L_{\mathcal{P},\ell,c}$.

TSM is trivially undecidable. Let c be the condition expressing that the program halts. In this case, checking that an input $x \in L_{\mathcal{P},\ell,c}$ (which is equivalent to checking that $\mathcal{P}$ halts for some input x) is undecidable.

TSM minimization and equivalence both require deciding non-trivial properties of Turing machines, thus undecidable by Rice's Theorem [21].

Symbolic constraint signature bounds. Symbolic constraint signatures can be viewed as a Turing machine signature on a finite domain. [4] 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 $\mathcal{B}$.

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

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

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

[4]Note that if we disallow quantifiers, then a symbolic constraint signature may be more efficiently evaluated.

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

Let $\mathcal{P}$ be a non-recursive Boolean program. Since all the variables can take values from a finite domain, without loss of generality we can assume that $\mathcal{P}$ 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 $\mathcal{P}$ appears only *once* on the left-hand side of an assignment statement (this can be done by converting $\mathcal{P}$ to SSA-form [26]). Since $\mathcal{P}$ is non-recursive we can inline all the function calls, and hence assume that $\mathcal{P}$ does not have function calls.

The *model checking problem (MCP)* is defined as follows:

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

It is well known that MCP is PSPACE-complete [24, 27]. If there exists a state reachable from I in M that satisfies, 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 $M_{\mathcal{P}}$ corresponding to the Boolean program $\mathcal{P}$, where the state variables of $M_{\mathcal{P}}$ are the variables of $\mathcal{P}$, the transition relation of $M_{\mathcal{P}}$ corresponds to the transformers of the statements in $\mathcal{P}$. An initial state I of the model $M_{\mathcal{P}}$ corresponds to an assignment of the input variables. Let f be a Boolean formula on the state variables of $M_{\mathcal{P}}$. Let x be an input to the program $\mathcal{P}$, and let I_x be the initial state $M_{\mathcal{P}}$ 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 a Boolean formula f_c over the variables of $\mathcal{P}$. It is easy to see that $M, I_x \models f_c$ iff $x \in Sat(\mathcal{P}, c)$. Hence, we have a PSPACE algorithm for checking if an input x is in the set $Sat(\mathcal{P}, c)$. 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 $\mathcal{P}_M$ whose variables correspond to the state variables of M and statements correspond to the transitions of M. Moreover, $M, I \models f$ iff $x_I \in (Sat(\mathcal{P}, c_f)$, 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. $\Box$.

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

Proof: Let $\mathcal{P}$ be a Boolean program. The main complication is that in general $\mathcal{P}$ can have recursive calls. In this case we can construct a pushdown system PS corresponding to $\mathcal{P}$, such that $PS, I_x \models f_c$ iff $x \in Sat(\mathcal{P}, c)$ (I_x and f_c have exactly the same meaning as in the previous theorem. Fortunately, the model checking problem for pushdown systems can be performed in PSPACE [28]. This proves that TSM for general Boolean programs is in PSPACE. Since TSM for non-recursive Boolean programs is PSPACE-hard, TSM for general programs is also PSPACE-hard. Therefore, TSM is PSPACE-complete. $\Box$.

Regular Expression Bounds. Regular expression signatures are well understood, and are the primary mechanism many production NIDS systems use to detect exploits. If we assume the regular expression is represented via a deterministic finite automata (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 [29]. Equivalence of two signatures S_1 and 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 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 program paths an exploit may follow. The ability to consider 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. In order to be scalable, our signature creation techniques take an iterative approach where we successively improve signatures by first considering a small coverage, and then incrementally increasing our coverage to include more program paths to the vulnerability.

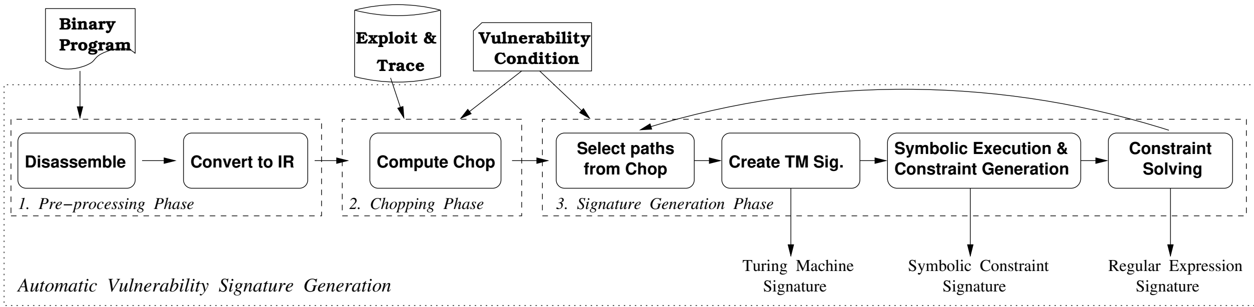

Figure 4: A high-level view of the steps to compute a vulnerability signature.

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 path is usually the path taken by the sample exploit. An MEP covers only those program instructions executed by an exploit on a single path to the vulnerability point, excluding statements with no effect on the computation, e.g., line 6 in the sample exploit is semantically a no-op with respect to the vulnerability since the value written is never read. Within an MEP, for each conditional branch encountered, one target is an instruction leading towards the vulnerability point, while the other target is a state BENIGN. An MEP is therefore a straight-line program. At the vulnerability point the vulnerability condition is evaluated, which returns either BENIGN 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 exists many other inputs $x' \neq x$ that both reach the vulnerability point and the vulnerability condition evaluates to EXPLOIT. For example, exploits usually have a payload which executes arbitrary attacker code. A straight line program will return EXPLOIT for exploits with different payloads because the execution of different variants only differ *after* the vulnerability condition has been satisfied.

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 $\in L_{\mathcal{P},c}$, i.e., the signature is complete. More formally, complete coverage is obtained by generating a signature for a *chop* [30, 31] of the program, which includes all instructions that may be executed between a `read` statement, where an exploit may be read in, and the vulnerability point. A chop has two distinguished nodes: v_{init} and v_{final}. The node v_{init} corresponds to the input `read` statement (if multiple input `read` statements exist, then v_{init} is an abstract node that is connected to each `read` statement in the control flow graph). v_{final} corresponds to the inlined vulnerability condition branch returning EXPLOIT. We outline our algorithm for computing the chop in Section 4.2.

In our signature-creation algorithm, we initially begin with the MEP path consisting of those instructions executing in the exploit trace T. We then compute a program chop of the vulnerability, where v_{init} is the initial read of the sample exploit, and v_{final} is the vulnerability point. The chop contains all possible execution paths from where an exploit was read (in the trace) to the vulnerability point. We then initially create a signature S for the MEP path given by the execution trace, and then iteratively improve S by considering other paths.

For our running example, the MEP coverage consists of the instructions executed in the trace. The complete PEP coverage consists of lines 1-12, excluding line 6.

4 Automatic Vulnerability Signature Creation

Figure 4 depicts the overall architecture of our vulnerability-signature generation system. Our algorithm for computing a vulnerability signature for a program $\mathcal{P}$, a vulnerability condition c, a sample exploit x, and the corresponding instruction trace T is shown below. In this section we detail how we perform each of the three steps in the algorithm.

Algorithm 1: Vulnerability-signature generation algorithm.

Input : $\mathcal{P}$: binary program; T: instruction trace from running $\mathcal{P}$ on input x; c: a vulnerability condition for $\mathcal{P}$

Output : S: vulnerability signatures for $\mathcal{P}$ for the vulnerability condition c

Step 1. *Pre-process the program before any exploit is received:*
Disassembling the program $\mathcal{P}$ (Section 4.1).
Converting the assembly into an intermediate representation (IR) (Section 4.1).

Step 2. *Compute a chop with respect to the trace T. The chop includes all paths to the vulnerability point including that taken by the sample exploit (Section 4.2).*

Step 3. *Compute the signature:*
Compute the Turing machine signature (Section 4.3.1). Stop if this is the final representation.
Compute the symbolic constraint signature from the TM signature (Section 4.3.2). Stop if this is the final representation.
Compute the regular expression signature from the symbolic constraint signature (Section 4.4).

4.1 Disassembling the Binary Program and Converting to the IR

We first disassemble the binary and identify function boundaries. We do not require the symbol table as functions can be identified via their prologue and epilogue. Next, we convert the disassembled instructions into an intermediate representation (IR). The IR disambiguates instructions by making implicit hardware side-effects explicit. Although this step is seemingly straight-forward, it is actually fairly involved. The main complication we address is modern architectures such as x86 implicitly set and test hardware registers, which can affect program execution, i.e., these tests and sets do not appear explicitly in the assembly. For example, the overflow flag may be automatically set when executing arithmetic operation, then later tested by a conditional jump. Another complication is the same register may be indexed in different modes, e.g., `al` is the lower 8 bits of the `eax` register, so any instruction affecting `al` must simultaneously affect `eax` in the IR.

More concretely, the x86 instruction set contains over 60 instructions that perform via hardware test or set operations on the `EFLAGS` register. Extra IR statements must be added to almost all operations to reflect the updates done in hardware. Worse, which statements to add is specific to the particular mode of the operands. The x86 architecture has 8-bit mode, 16-bit mode, etc., which is set depending upon the format of the instruction operands. For example, `add %ax, %bx` is an addition in 16-bit mode since the registers specified are 16-bits long. Overflow, the carry flag, and other implicit hardware-assisted effects must then be set with respect to 16-bits. A very similar instruction `add %eax, %ebx` is 32-bit mode, and implicit hardware effects must be done with respect to 32-bits.

We perform the remaining steps — program chopping and vulnerability signature creation — on the IR statements.

4.2 Computing the Chop on the IR

We first compute the chop [30, 31] of the vulnerability with respect to the exploit x and trace T (as discussed in Section 3). Note that we developed an imprecise chopping algorithm which works without pointer analysis. Pointer analysis such as value-set analysis could be used to improve precision [32]. The result of the chop is a smaller program $\mathcal{P}'$ in which every path begins at the `read` statement in the trace and ends at the vulnerability point. We can then select in the signature generation step any set of paths in $\mathcal{P}'$ and compute a signature.

We perform a chop on the program's callgraph. The chop contains all functions that may be executed between reading in the exploit and the vulnerability point. The chop is performed by essentially doing a reachability analysis so that any function in a call sequence that may reach the vulnerability point is included.

A callgraph is a directed graph where each function is a vertex, and edges represent the caller-callee relationship of functions. We perform the following algorithm on the callgraph to create the chop given start IR statement v_{init}, which is the `read` statement for the exploit in the trace, and the vulnerability point v_{final} in the trace T. Let F_{init} and F_{final} be the functions enclosing the v_{init} and v_{final} nodes respectively. Note that there is at least one path from F_{init} to F_{final}: the one that appears in the trace. We then add an extra edge from F_{final} to F_{init}, resulting in a loop in the callgraph. We then calculate the strongly connected component (SCC) containing F_{init} and F_{final}. This SCC is the chop, since it contains all reachable functions from F_{init} to F_{final}.

One problem we must deal with at the binary level is the widespread use of indirect jumps, e.g., `jmp %*eax` [5]. Note that some indirect jumps correspond to source code constructs such as function pointers, while some are compiler-generated as optimizations. The central issue is a target of an indirect jump could potentially be any other instruction [6]. As a result, any control flow graph (including dependency graphs) would have an edge from each indirect jump to all other instructions.

In order to deal with the widespread indirect jumps in binaries without pointer analysis, when creating the callgraph we make the target of each indirect jump a special node IJMP. Our algorithm for computing the chop then will essentially ignore indirect jumps until a chop is computed. After computing the chop, we constrain each indirect jump so that the target is within the chop. One limitation of this approach is that technically we could be incorrectly excluding a function that only appears as the target of an indirect jump. The indirect jump problem may or may not disappear once we have implemented function pointer analysis. It remains unclear (and a point of future work) how precisely such analysis will be able to pin down the targets of indirect jumps.

4.3 Computing the Signature

We compute the signature with respect to the chop. We compute a PEP signature by iteratively considering single MEP paths (except in our data-flow analysis optimization). Our iterative method works because we can pick any path or set of paths within the chop, perform our analysis, and output the corresponding vulnerability signature. The complete PEP coverage signature (Section 3) is then the analysis of all paths in the chop. We begin by describing how we compute the complete PEP TM signature, which in turn becomes input to symbolic constraint and regular expression signature generation.

4.3.1 Turing Machine Signature Generation

MEP Turing machine signature generation. Our initial MEP Turing machine signature is created 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 initial MEP TM signature by reading in the instruction trace and including the corresponding IR statements.

Sequential instructions in the trace correspond to sequential statements in the MEP Turing machine signature. Conditional branch statements have exactly two targets in the IR. Any branch that does not lead towards the vulnerability point returns BENIGN. EXPLOIT is only returned if the vulnerability point is reached and the vulnerability condition is satisfied.

We encode the vulnerability condition as a function, and then inline calls to the vulnerability condition into the program. Theoretically, difficult vulnerabilities may require inserting the vulnerability condition at each instruction. (Note that we are not running the program with the vulnerability condition inlined, we are inlining it for generating a signature.) In practice, we may not need to inline the entire vulnerability condition. Also, when the vulnerability is local to a specific region of the program, we need only inline the condition for that region. In general, algorithms for making these optimizations can be specified in conjunction with the vulnerability condition. For example, in our running example we know from static analysis that the size of the `url` buffer is 4, therefore any write to a memory location greater than 4 is unsafe. We need only simply insert a check at each memory write to `url` to see if the index is greater than 4. Figure 5 shows the MEP vulnerability signature we would return for our running example with the vulnerability check inlined.

PEP Turing machine signature generation. A PEP Turing machine signature is created similar to an MEP Turing machine signature. The PEP signature first computes the chop, and then computes which jump targets cannot lead to the vulnerability point via standard graph reachability analysis. Paths that terminate or cannot lead to the vulnerability point return BENIGN. We also insert a call to the vulnerability condition function at the vulnerability point, which

[5] A direct jump is of the form `jmp` c where c is a constant

[6] An indirect jump target could potentially be any location in the executable image, even to the middle of a previously disassembled instruction! In our problem setting, we assume the binaries were not maliciously compiled or obfuscated.

```
char *url = malloc(4);
int c = 0;
if (inp[c] != 'g' && inp[c] != 'G')
  return BENIGN;
c++;
if (inp[c] != ' ') return BENIGN;
c++;
if (inp[c] == ' ') return BENIGN;
*url = inp[c]; c++; url++;
if (inp[c] == ' ') return BENIGN;
*url = inp[c]; c++; url++;
if (inp[c] == ' ') return BENIGN;
*url = inp[c]; c++; url++;
if (inp[c] == ' ') return BENIGN;
return EXPLOIT;
```

Figure 5: The MEP TM signature for our running example.

returns either BENIGN or EXPLOIT. Figure 2 shows the complete PEP vulnerability signature with the vulnerability condition inlined.

4.3.2 Symbolic Constraint Signature Generation

A symbolic constraint signature is a set of constraints an exploit of the vulnerability must satisfy. We use the TM signature as the input to symbolic constraint signature generation, and at a high level generate constraints that represent meeting the correct conditionals in the TM to reach the vulnerability point and satisfy the vulnerability condition. The symbolic constraint signature is an approximation of the TM signature because we may have to statically estimate the effects of loops and memory updates as constraints on the input. The symbolic constraint system is built up by symbolically evaluating the TM signature program on symbolic inputs instead of actual inputs (values).

More formally, we build up the constraints based upon symbolically executing paths in the TM. Each function in the TM signature is represented by a control flow graph (CFG), which is a direct graph $(V, E, v_{entry}, v_{exit})$ where each IR instruction is a node in V, each transfer of control between instructions is an edge in E, and v_{entry}, v_{exit} are distinguished entry and exit nodes. Conditionals in the control flow graph become constraints to take the appropriate branch to reach the vulnerability point and satisfy the vulnerability condition.

Single Static Assignment (SSA) form. We must convert the IR into a single static assignment (SSA) [26] form prior to symbolic constraints (this step can be performed during the pre-processing phase). Normally, memory locations and registers are destructively updated many times in the lifespan of a program, e.g., $x = x + 1$ destructively updates the x on the right-hand side (RHS) when assigning to x on the left-hand side (LHS). However, symbolic execution requires each variable be treated as a single logical entity that is assigned to only once. SSA form is a semantically equivalent form of the program which satisfies this criteria. The SSA form of sequential statements is just a unique renaming of each LHS. For example, $x = x + 1$ becomes $x_2 = x_1 + 1$. For control statements, SSA introduces a special assignment called ϕ-functions which merges several possible definitions of a variable into one. For example, the if-then-else statement:

$$\textbf{if } x < 2 \textbf{ then } z = 10; \textbf{ else } z = 20;$$

becomes

$$\textbf{if } x_0 < 2 \textbf{ then } z_1 = 10; \textbf{ else } z_2 = 20; \; z_3 = \phi(z_1, z_2);$$

where z_3 is assigned z_1 on the true branch and z_2 on the false branch.

MEP symbolic execution. We perform MEP symbolic execution by evaluating the MEP TM signature. Recall that the MEP TM signature is a straight-line program. Then there is a single path $\pi = v_{entry}, v_1, ..., v_{exit}$ that goes through the vulnerability point and the vulnerability condition. All other paths will end up returning BENIGN and need not be considered. The result of symbolic execution on π is a set of constraints on input variables that when met results in an execution from v_{entry}, through the vulnerability point and the inlined vulnerability condition to v_{exit}.

We begin by creating symbolic input variables $i_0, ..., i_n$ where n is the length of the symbolic input to consider, e.g, n is initially the length of the sample exploit x. Each statement is then executed on these inputs, resulting in a symbolic formula at each step. There are three fundamental operation types to evaluate symbolically: *memory updates*, *arithmetic operations*, and *branch predicate evaluation*. Symbolic execution of arithmetic operations is simply a substitution procedure. For example, $x = a + i_0; y = x * z$ becomes $y = (a + i_0) * z$.

A memory store operation is an assignment of a value to a symbolic memory location (stack and heap assignments are handled in a uniform fashion). We adopt a model similar to UCLID [33] for handling memory updates. The initial state of M is given by m_0. Reads and writes are modeled as λ expressions, where a write to memory location A with value D yields a new M':

$$M' = \lambda addr.\text{ITE}(\text{addr} = A, D, M[\text{addr}])$$

The result of a write is an if-then-else (ITE) λ-expression. A subsequent read behaves as follows: the address to read is applied as the argument to the write λ-expression. If the supplied address matches A, then D is returned, else we recurse to the next memory address given by $M[\text{addr}]$.

Consider the path π to the EXPLOIT state. Without loss of generality, we assume each branch predicate (such as `je` (jump if equal) or `jz` (jump if zero)) $v_i \in \pi$ evaluates to true in order to create the desired total path π. A branch predicate forms an arithmetic constraint (with some expressions perhaps involving memory reads and writes) relating the symbolic execution to some constant, e.g., `jz y` where $y = (a+i_0)*z$ as before and `jz` is "jump if zero" results in the constraint $(a + i_0) * z = 0$. Constraints evaluate to a constant because machine instructions only allow comparison of an expression to a constant. The total symbolic formula is then just the conjunction of each branch predicate.

The constraint system consisting of the conditions on each branch predicate in π is returned as the desired signature. Optionally, constraint systems can be simplified, which consists of deducing how multiple constraints can be collapsed into a single constraint.

PEP symbolic execution. PEP symbolic execution is similar to the MEP case, except we must deal with loops. Loops are handled by computing fixed points. However, in data-flow analysis a widening operator is used to guarantee that the iteration to compute the fixed-point terminates [34, 35]. Currently, we use the following algorithm to handle loops:

- First, we identify induction variables [26, Chapter 14] in each loop. For example, the induction variable for the first `while` loop in Figure 1 is `c`. We also compute the bounds on the induction variable, e.g., the bound on the induction variable `c` is `c` ≥ 1.
- Assume that an induction variable is used to index the input array in the condition used in the `while` loop. The condition used in the `while` loop along with the bounds on the induction variable gives us the desired result. For the first while loop in Figure 1 the condition that is generated is:

$$(\texttt{inp}[c] =' \ ') \text{ where } (c >= 1)$$

4.4 Regular Expression Signature Generation

4.4.1 Computing MEP Regular Expression Signatures

One method for generating a regular expression is to solve the constraint system S to a set $x : x \in S$ and or (|) together all members, e.g., if $S = \{00, 01, 21\}$, then the regular expression is 00|01|21 [7]. This method is explored heavily in test-case generation literature [36–40]. We adopt this approach to our problem setting.

[7] A reader may notice this expression is precise, and wonder when the solution will not be precise. The answer is as precise as the symbolic representation, e.g., if the symbolic representation only unrolls a loop once, then the regular expression signature will not reflect inputs that may cause the loop to be executed more than one time.

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. We address this problem by decomposing an MEP single-path solution into smaller sub-paths we can consider independently. Let $\pi = v_{entry}v_1...v_{exit}$ be an MEP path. A sub-path is a sequence of instructions $\pi_i = v_i v_{i+1}...v_\ell \in \pi$ in the CFG. A sub-path π_i can be independently evaluated with respect to another sub-path π_j if no computation in π_i could ever affect a computation in π_j, and vice-versa. Formally, we partition $\pi = \pi_1\pi_2\pi_3...\pi_j$, where each π_i is a partition with no data dependencies with another sub-path π_j. A data dependency exists between π_i and π_j if π_i computes a value that π_j uses.

Since no computation in one sub-path π_i could affect a computation in another sub-path π_j, each sub-path can be independently solved, then the final solution can be combined. The solution to each sub-path π_i is computed as above by solving the corresponding constraint system for the sub-path. The full path π is then the conjunction ($\wedge$) of all sub-paths.

MEP solution. Our approach allows us to divide a single MEP into possibly several smaller sub-problems. Let the MEP path $\pi = \pi_1\pi_2...\pi_n$ correspond to evaluating the symbolic input in order $i_1, ...i_n$. Since sub-paths are independent, we can always reorder the sub-paths so this is the case. Then the signature for an MEP is the concatenation of the solution for each sub-path. If π_i has solution S_i, then the resulting signature is $S = S_1S_2...S_n$.

4.4.2 Computing PEP Regular Expression Signatures

We consider two approaches for computing a PEP solution. The first method considers each MEP path within a PEP independently, and solves the symbolic constraints exactly. The second method is an optimization based on data-flow analysis which can be applied to portions of the PEP control-flow graph when certain conditions stated below are met. The data-flow analysis optimization works on basic blocks instead of paths and does not require access to a constraint solver.

Exact PEP solution. The PEP solution iteratively explores paths, and then solves them as an MEP solution. We note that in practice one would likely create an initial MEP signature for the sample exploit, then process other paths in the background. This approach generates an initial narrow signature quickly, then continues to refine it as we perform more analysis.

PEP data-flow optimization. In many cases we may be able to determine: (a) the data dependencies partition a vulnerability into two or more components (w.r.t. the CFG and data-dependency graph), and (b) some of these components do direct comparisons with input values. For example, many protocols have keywords or have constant values for specific fields which the input is simply compared against.

We use data-flow analysis to efficiently compute the language recognized by such components. Since each component has no data dependencies with other components, the solution to each component can be inlined into the complete PEP or MEP solution. At a high level, data-flow analysis iteratively processes a CFG until a fixed point of data-flow facts is reached. Data-flow analysis is widely used in compilers and is highly efficient.

The high-level idea of our data-flow analysis is to calculate what values input variables may be in order to reach a given program point (a program point is a point just before or after a basic block). For instance, an implementation of HTTP may require the variable `input` to begin with the case-insensitive keyword "GET" before processing a HTTP request URL. The data-flow analysis should produce a mapping right before the program statements that process the URL request such that `input[0]` $\rightarrow \{g, G\}$, `input [1]` $\rightarrow \{e, E\}$, and `input[2]` $\rightarrow \{t, T\}$. Thus, if there is a vulnerability at the beginning of the subsequent URL-parsing code, our data-flow analysis will be used to create a signature such that any exploit must begin with case-insensitive "GET".

Here, we focus on our data-flow analysis: interested readers unfamiliar with data-flow analysis should consult standard text-books such as Aho *et. al.* [41] for more information. Let G' be a sub-graph of G with a designated entry and exit node, and let V be a list of variables in G'. Data-flow analysis produces a map $M : V \rightarrow I$ at each program point where I is the input domain for the program (e.g., I is ASCII characters for text-based protocols). Let `jmp` `(`$v = \alpha$`) then` bb_1 `else` bb_2 represent a conditional jump that jumps to location bb_1 if $v = \alpha$, else it jumps to location bb_2.

A data-flow analysis requires three things:

1. A set of values X of data-flow facts. X is our setting contains an element $\top$, the power-set of 8-bit ASCII characters, and a special value $\perp$ denoting the unknown state. The intuition is that the data-flow maps $v \rightarrow \top$ if

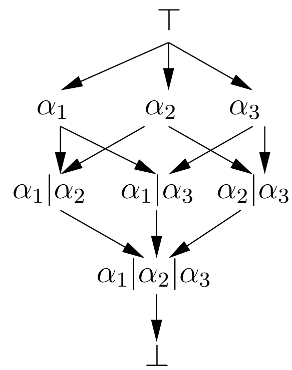

Figure 6: A Hasse diagram of our $\sqcap$ operator defined over sets of 3 values. A downward edge exists if $x \leq y$ (in the partial ordering). For example, if a basic block has incoming edges labeled $i = \alpha_1$ and $i = \alpha_2$ for variable i, then $i = \alpha_1|\alpha_2$ because if i is either value that program point will be reached w.r.t. i.

	$\top$	α	$\beta(\neq \alpha)$	$\perp$
$\top$	$\top$	α	β	$\perp$
α	α	α	$\alpha\|\beta$	$\perp$
$\beta(\neq \alpha)$	β	$\alpha\|\beta$	β	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table 2: The $\sqcap$ operator.

v is not compared to anything, $v \rightarrow \perp$ if we cannot determine what v is compared to, and $v \rightarrow \{\alpha_1, \alpha_2, ..., \alpha_n\}$ if v is compared to α_1 and α_2 and so on.

2. A binary meet operator $\sqcap$ which defines how data-flow facts from multiple predecessors are combined at the beginning of each basic block. Our meet operator is defined in Table 2. Figure 6 shows a Hasse diagram of our $\sqcap$ operator over a restricted domain. (Note that the $\sqcap$ operator is similar to the standard definition for the super-set lattice.)

3. A transfer function f which describes how data-flow facts are calculated and updated within a block. Our transfer function f is:

$$f : \text{out}[\ell] = \text{gen}[\ell] \cup (\text{in}[\ell] - \text{kill}[\ell])$$

Informally, the *in* set for a basic block is the $\sqcap$ of all incoming edge labels. Define $Flow_T(G)$ and $Flow_F(G)$ to be the true and false edges in the CFG, respectively. Then *in* can be more formally specified as:

$$\text{in}[\ell] = \begin{cases} \sqcap!(\text{out}[\ell']) & |(\ell, \ell') \in \text{Flow}_F(G) \\ \sqcap\text{out}[\ell'] & |(\ell, \ell') \in \text{Flow}_T(G) \end{cases}$$

Then we say a block *bb generates* $v \rightarrow \alpha$ on the edge (bb, bb_1) and $v \rightarrow !\alpha$ on the edge(bb, bb_2). We say a block *bb kills* $v \rightarrow \alpha$ if v is compared to some $\beta \neq \alpha$.

Note that $\perp$ acts as a widening type, e.g., $v = \perp$ indicates we do not know what value v must have to reach that program point. If there is a comparison between a constant and an unknown input variable, then *all* variables are updated with $\perp$. If there is a comparison with a known input variable v but an unknown value, we widen $v = \perp$, essentially indicating we don't know what value v could take on.

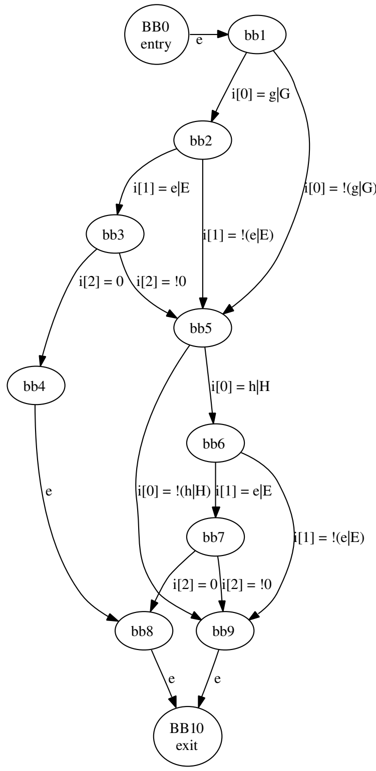

Figure 7: Data-flow facts of the CFG

Consider the following code:

```
if (strncmp(input, "ge", 2) == 0 || strncmp(input, "he", 2) == 0)
    return 0x1;
else
    return 0x0;
```

The CFG of the compiled-down version of this code is given in Figure 7. The results of our data-flow analysis over this program is given in Table 3. We can read this table as such: if bb_8 is considered the vulnerability point, then the input string that reaches this node is given by the regular expression [h|H|g|G][e|E].

5 Implementation and Evaluation

We have implemented a prototype system to evaluate our techniques for automatically generating signatures. In this section we briefly discuss implementation details of our prototype, and then present our evaluation results. Our evaluation results show that even an MEP vulnerability signature is of far higher quality than signatures generated with previous approaches. We focus on creating regular expression signatures since they require generation of the Turing machine and symbolic constraint signature.

Block	in[s]	out[s]
entry	$\{\epsilon, \epsilon, \epsilon\}$	$\{\epsilon, \epsilon, \epsilon\}$
BB1	$\{\epsilon, \epsilon, \epsilon\}$	$\{g\|G, \epsilon, \epsilon\}$
BB2	$\{g\|G, \epsilon, \epsilon\}$	$\{g\|G, e\|E, \epsilon\}$
BB3	$\{g\|G, e\|E, \epsilon\}$	$\{g\|G, e\|E, \emptyset\}$
BB4	$\{g\|G, e\|E, \emptyset\}$	$\{g\|G, e\|E, \emptyset\}$
BB5	$\{!(g\|G), !(e\|E), !(\emptyset)\}$	$\{(h\|H), !(e\|E), !(\emptyset)\}$
BB6	$\{(h\|H), !(e\|E), !(\emptyset)\}$	$\{(h\|H), (e\|E), !(\emptyset)\}$
BB7	$\{(h\|H), (e\|E), !(\emptyset)\}$	$\{(h\|H), (e\|E), (\emptyset)\}$
BB8	$\{(h\|H\|g\|G), (e\|E), (\emptyset)\}$	$\{(h\|H\|g\|G), (e\|E), (\emptyset)\}$
BB9	$\{!(h\|H\|g\|G), !(e\|E), !(\emptyset)\}$	$\{!(h\|H\|g\|G), !(e\|E), !(\emptyset)\}$

Table 3: Result of data-flow analysis

,

5.1 Implementation

Our total prototype for implementing our techniques is about 9000 lines of C++ code. We currently use CBMC [42], a bounded model checker, to help build and solve symbolic constraints to produce regular expression signatures. [8]

Disassembling the program, converting to IR, and obtaining instruction traces. Our binary program disassembler is based upon Kruegel et. al. [43]. We then translate each instruction into the appropriate IR statement via our own translation language.

Instruction traces can be efficiently generated for most modern architectures including x86 via hardware [44, 45] or software [46–48]. An instruction trace contains the instruction address and optionally the value of the operands for each instruction executed. Although the number of instructions executed may be large, the corresponding trace can be efficiently represented [49, 50]. We currently use Pin [47] to create our traces.

Solving the constraint system. We use model checking to solve the system of constraints. We translate the constraints into constraints on C variables and use CBMC [42]). We then assert to the model checker that the vulnerability condition is unsatisfiable. The model checker will either verify the vulnerability condition is unsatisfiable, or solve the constraint system and present a counter-example which, by construction, is a satisfying input. This process can be iterated to exhaustively enumerate all possible satisfying inputs (i.e., exploits). The regular expression signature is the "or" of all satisfying inputs. However, this process may be slow when an input byte may be any of the 2^{256} values. Therefore, we currently apply a widening operator such that any byte that appears to be unconstrained after 3 iterations becomes a wild-card byte. The widening step may introduce false positives, and can be eliminated when desired.

Implementation limitations. Our current implementation is a prototype used for researching automatic signature generation. Although our prototype works in our research setting, there are a number of limitations. As mentioned previously, alias analysis is currently not supported. Specifically, we assume that no two memory locations are aliases. In addition to the possible imprecision this may introduce during symbolic execution, this limitation prevents us from computing a true chop [30, 31]. Our current callgraph-based chopping algorithm is less precise than a true chop, which primarily results in larger MEP and PEP coverages than necessary. Second, we currently create sub-paths based upon control-flow based analysis, which may not accurately identify when two sub-paths are independent (Section 4.4). Finally, our IR transformations do not handle floating point operations, and we currently do not support the entire x86 instruction set (we add operations as needed for our experiments). All these limitations are orthogonal to our problem and can be resolved by implementing known techniques. We currently manually verify none of these problems introduce errors into our results.

[8] We do not use source code despite the fact this is primarily a C model checker.

5.2 MEP Evaluation

5.2.1 ATPhttpd

ATPhttpd is a webserver written in C [51]. ATPhttpd version 0.4b is vulnerable to a common sprintf-style buffer-overflow when an HTTP request is too long. Specifically, an exploit of the ATPhttpd vulnerability must meet the following conditions: (a) the HTTP request method is case-insensitive, and must be either "get" or "head"; (b) the first byte of the requested file name must be '/', and cannot be followed by '/'; (c) the requested filename cannot contain the substring "/../" or end with "/.."; and (d) the requested filename must be over 677 characters long.

We use the exploit sample from [52], which consists of the request `GET /`, followed by the shell code, followed by the HTTP protocol string `HTTP/1.1`. In this experiment, the vulnerability condition given for ATPhttpd is that no pointer should be able to write to a return address.

Signature result and quality. We generated the symbolic constraints, which were partitioned into 10 distinct sub-paths that were analyzed independently. We solved the constraints and create a regular expression in a little over a second, with the average time per partition taking $0.1216s$.

We generated the regular expression signature `[g|G][e|E][t|T][ ]/.{423}//.{3}/.{386}`. This regular expression is almost perfect w.r.t. the necessary conditions to reach the vulnerability as stated previously. In particular, it recognizes that the `get` keyword is case insensitive, and that most bytes can be anything. The bytes that are constraints ("/" and "//" in the signature) are both contained in the exploit and explicitly tested along the MEP vulnerability path that the exploit took. We contrast our signature with previous exploit-specific signature generation approaches [7, 18, 53], which at best only identify small parts of our signature and do not match different exploit variants such as those that crashes the server instead of injecting code. Our signatures will catch many more exploit variants given only a single exploit sample.

5.2.2 BIND

BIND is one of the most popular DNS servers. BIND supports a secret key transaction authentication mechanism where messages are signed with a transaction signature (TSIG) [54]. BIND 8.2.x is susceptible to a stack overflow vulnerability in the TSIG processing code.

The attacker must send a valid DNS transaction signature request in order to exploit this vulnerability [55]. DNS is a binary-based protocol in which all messages are struct-like. DNS (and the exploit) can be TCP or UDP-based, though here we only consider the UDP protocol messages. DNS messages begin with a header, followed by a number of resource records (RR). An exploit of this vulnerability must satisfy the following conditions: (a) the request must be a query, which is represented by byte 2 of the message being 0; (b) there must be questions present, meaning that the field specifying the number of questions (byte offsets 4 and 5) must be greater than zero, and that there must be properly encoded questions starting at offset 12; (c) the field specifying the number of additional resource records (byte offsets 10 and 11) must be greater than zero; (d) The DNS must contain a resource record with the type field set to TSIG, which is 0x00af. Since DNS may have many different resource records in a single request, the specific byte offset for this field is a function of several other fields in the request. We use the TSIG vulnerability exploit from the LION worm [56] as our sample exploit.

Signature result and quality. We generated the symbolic constraints, which again could be partitioned into 10 distinct graphs, which we independently analyzed. The generated regular-expression signature specified that bytes 6-10 must be zero, that bytes 268 and 500, which indicate the end of each query in the exploit, must be 0, that byte 12 must not be 0, which is the first byte of the first query, and finally, that bytes 505 through 507 must be 0x0000fa, which is the 0 byte at the beginning of the additional resource records section, followed by the field type TSIG. We verify that the constructed signature identified all constraints that must be met to exploit the vulnerability. We also verified the false-positive rate of our signature by matching it again against 1,000,000 DNS requests (trace taken from a high-traffic DNS server that serves several top level domains). There were no false positives.

5.3 PEP Evaluation

The chop of ATPHttpd took 30μs and found 88% of all functions were reachable between accepting a connection and the vulnerability point (including all libraries). As mentioned previously, one technique for generating a PEP signature is to consider each MEP path independently. Another technique is to estimate the effects of multiple paths simultaneously. Our current prototype implementation for the latter technique is limited to moderate-sized functions. Unfortunately, the ATPHttpd and BIND vulnerabilities use extremely large library function which consists of several thousand basic blocks. Addressing scalability issues is an important part of our future work. We expect existing state reduction techniques from model checking will help solve this problem.

Here, we evaluate our PEP techniques on synthetic examples. We compile down our running example to a binary, and then calculate the full PEP solution. The regular expression generated is `[g|G][ ]*[^␣]{5,}`. The total time to compute the answer is about 1.5 seconds. Alternatively, our tool can also produce the regular expression for each independent component of the PEP, and then use data-flow facts to produce the final signature. In this setting, our tool runs slightly faster as it does not have to perform symbolic evaluation along all possible paths.

6 Discussion

Other application scenarios. At a high level, our techniques generate an input string that reaches a given instruction in the binary. Several other applications of our techniques that we plan on investigating include:

- Improve existing pattern-extraction signature generation algorithms. The quality of a signature generated by pattern-extraction techniques generally improves as the number of exploit samples increase. Our techniques can be used to iteratively generate a new exploit sample x' that is different than the sample exploit x. In this scenario, we can give x' to the pattern-extractor as a *labeled* exploit, which it then uses to improve an existing signature. Note that in previous scenarios pattern extraction would be limited to only x. In addition, we may be able to label tokens within x which may further help the analysis. Finally, we note that our analysis could also be used to help defend against "red-herring" and "coincidental token" attacks.
- Perform robust vulnerability identification. Often it is not known whether a *known* bug is exploitable. Here, the developer would set v_{init} to the appropriate `read` statement and v_{final} to the line for the bug. Our techniques will generate a sample exploit when possible, confirming whether a bug is exploitable or not.
- Vendor patches often miss all possible paths to a vulnerability. Missing alternate paths is not only a security problem, but can also be an embarrassment to the vendor because even "patched" systems may still be compromised [4]. Our techniques can be adapted to see if a given patch covers all possible ways a vulnerability may be exploited.

Complete but unsound signatures. Every satisfying solution to the generated symbolic equations is an exploit string, thus the signature is sound but not complete. A complete but potentially unsound signature, i.e., no false negatives but false positives, can be created by setting the initial signature to Σ^* and removing any input that leads to BENIGN state.

Identifying sources of signature imprecision. Our construction allows a signature creator to tune accuracy and generation time in several ways. First, the creator has a choice of signature representations. Second, the creator can choose how much information to retain for less expressive representations. For example, when creating a symbolic representation the creator may choose how many times loops are unrolled. Third, the creator can choose how much analysis to perform. For example, when creating a regular expression signature theorem proving can be employed to enumerate every input string that may exploit the program, or faster but less accurate data-flow analysis. We believe these choices allow a creator to gain a fundamental understanding of the overall accuracy of the final generated signature by comparing their generated signature to the perfect TM signature.

Application to non-deterministic programs. We assume that we are writing the vulnerability signature for a deterministic program. In some cases, any non-determinism may not affect the vulnerability condition and is thus irrelevant. For example, analyzing a multi-threaded Internet services will likely still be possible when each request is handled by a single thread. One approach for handling non-determinism is to consider any possible choice that can be made. For example, one approach for analyzing multi-threaded programs is to consider all possible thread inter-leavings. Another approach is to quantify over possibly non-deterministic variables. For example, if a random number generator is consulted, one may be able to quantify the return value as being between zero and the maximum integer size. We leave analyzing non-deterministic programs as future work.

7 Related Work

Signature creation. In Section I we detailed most previous work in this area. Here we mention that Vigilante has independently proposed signatures which are essentially straight-line programs, not regular expressions [57], much like our MEP symbolic constraint signatures. However, Vigilante only creates a signature for the execution path taken by the sample exploit, and does not explore more extensive coverage or other vulnerability signature representations. Newsome *e*t. al. propose vulnerability-specific execution filtering (VSEF) [58]. Although the purpose of a VSEF filter is as a temporary patch, the filter itself can be viewed as a type of Turing machine signature.

Estimating language classes. A significant part of creating a vulnerability signature boils down to conservatively estimating the higher-powered language such as a Turing machine with a lower-power language such as a regular expression. Our techniques provide one way of accomplishing this. For example, Mohri and Nederhof present an algorithm for converting certain context-free languages into regular expressions [59].

Program analysis. We use many static analysis techniques such as symbolic execution [60], abstract interpretation [35], model checking [24], theorem proving [61], data-flow analysis [62], and program slicing [63]. Each of these areas is an active research area in which we can benefit from new or more advanced techniques. It would be impossible to note all related work in static analysis; the reader is referred to [26, 41] for an overview of the subject.

Automatic test case generation research explores the problem of automatically creating an input that reaches a particular point in the program [36–40]. We are interested in a very similar problem where we want to approximate *all* inputs that reach a certain location.

Another closely related area is static analysis of program generated string expressions. This line of work aims at discovering possible strings *generated*, as opposed to accepted by a program. Christensen *et. al.* performed string analysis on Java programs where type information is available [64]. Christodorescu *et. al.* extended Christensen's work to x86 binaries [65]. These techniques are exciting, though more research is needed to apply their techniques to our problem setting. In particular, this approach only handles strings and not other types such as integers.

8 Conclusion

We presented a general framework for obtaining a new type of signature called vulnerability signatures. Given a single sample exploit, we presented techniques for automatically generating a signature of higher quality than previous approaches. In addition, our formulation opens up a wide variety of signature representations. In particular, we discuss three distinct types of vulnerability signature representations: Turing machine, symbolic constraints, and regular expressions. We provide theoretical and practical insights into these three signature representations. We conclude that our approach is promising alternative to exploit-centric techniques.

Acknowledgment

The authors would like to thank Vyas Sekar, Zhenkai Liang, Himanshu Jain, Vinod Ganapathy for their suggestions and comments while preparing this paper. We would also like to thank the program committee and anonymous reviewers of the 2006 IEEE Security and Privacy Symposium for their feedback.

References

[1] D. Brumley, L.-H. Liu, P. Poosank, and D. Song, "Design space and analysis of worm defense systems," in *Proc of the 2006 ACM Symposium on Information, Computer, and Communication Security (ASIACCS)*, 2006.

[2] D. Moore, C. Shannon, G. Voelker, and S. Savage, "Internet quarantine: Requirements for containing self-propagating code," in *2003 IEEE Infocom Conference*, 2003.

[3] S. Staniford, V. Paxson, and N. Weaver, "How to 0wn the Internet in your spare time," in *11th USENIX Security Symposium*, 2002.

[4] C. Cerrudo, "Story of a dumb patch," http://argeniss.com/research/MSBugPaper.pdf, 2005.

[5] 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.

[6] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymophic worm detection using structural information of executables," in *Rapid Advances in Intrusion Detection (RAID)*, 2005.

[7] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Proc. of the IEEE Symposium on Security and Privacy*, May 2005.

[8] M. Jordan, "Dealing with metamorphism," in *Virus Bulletin Magazine*, 2002.

[9] P. Szor, "Hunting for metamorphic," in *Virus Bulletin Conference*, 2001.

[10] "K2, admmutate," http://www.ktwo.ca/c/ADMmutate-0.8.4.tar.gz.

[11] "Metasploit," http://metasploit.org.

[12] H. J. Wang, C. Guo, D. Simon, and A. Zugenmaier, "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," in *Proc. of the 2004 ACM SIGCOMM Conference*, August 2004.

[13] H.-A. Kim and B. Karp, "Autograph: toward automated, distributed worm signature detection," in *Proc. of the 13th USENIX Security Symposium*, August 2004.

[14] C. Kreibich and J. Crowcroft, "Honeycomb - creating intrusion detection signatures using honeypots," in *Proc. of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[15] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proc. of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2004.

[16] J. Newsome, B. Karp, and D. Song, "Paragraph: Thwarting signature learning by training maliciously," 2006.

[17] Z. Liang and R. Sekar, "Fast and automated generation of attack signatures: A basis for building self-protecting servers," in *Proc. of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[18] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

[19] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic diagnosis and response to memory corruption vulnerabilities," in *Proc. of the* 12^{th} *ACM Conference on Computer and Communication Security (CCS)*, 2005.

[20] J. Crandall, Z. Su, S. F. Wu, and F. Chong, "On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits," in *Proc. 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[21] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to automata theory, langauges, and computation*. Addison-Wesley, 2001.

[22] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. of the 7th USENIX Security Symposium*, January 1998.

[23] G. C. Necula, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy code," in *Proc. of the Symposium on Principles of Programming Languages*, 2002.

[24] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. The MIT Press, 1999.

[25] T. Reps, "Program analysis via graph reachability," *Information and Software Technology*, vol. 40, no. 11-12, 1998.

[26] S. Muchnick, *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[27] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," *Journal of the ACM*, vol. 32, no. 3, pp. 733–749, 1985.

[28] A. Bouajjani and O. Maler, "Reachability analysis of pushdown automata," in *Proc. of the International Workshop on Verification of Infinite-State Systems (Infinity'96)*, Pisa (Italy), 1996.

[29] J. Hopcroft, *An $n \log n$ algorithm for minimizing the states in a finite automaton*, Z. Kohavi, Ed. Academic Press, 1971.

[30] D. Jackson and E. Rollins, "Chopping: A generalization of slicing," in *Proc. of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1994.

[31] T. Reps and G. Rosay, "Precise interprocedural chopping," in *Proc. of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.

[32] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proc. of the Intl. Conf. on Compiler Construction*, 2004.

[33] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Proc. Computer-Aided Verification (CAV)*, 2002.

[34] F. Bourdoncle, "Efficient chaotic iteration strategies with widenings," in *Proc. of the International Conference on Formal methods in Programming and their applications (LNCS 735)*, 1993.

[35] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4^{th} ACM Symposium on Principles of Programming Languages (POPL)*, Jan 1977.

[36] C. Cadar and D. Engler, "Execution generated test cases: How to make system code crash itself," Stanford, Tech. Rep. CSTR-2005-04, 2005.

[37] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. of the 2005 International Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[38] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," in *ACM Symposium on Software Testing and Analysis*, 1998.

[39] ——, "A CLP framework for computing structural test data," in *First International Conference on Computational Logic*, 2000.

[40] N. Gupta, A. Mathur, and M. L. Soffa, "Automated test data generation using an iterative relaxation method," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1998.

[41] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[42] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[43] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proc. of the 13th USENIX Security Symposium*, 2004.

[44] P. Bosch, A. Carloganu, and D. Etiemble, “Complete x86 instruction trace generation from hardware bus collect,” in *23rd IEEE EUROMICRO Conference*, 1997.

[45] 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 Journal of Research and Development*, vol. 41, no. 3, 1997.

[46] “DynamoRIO,” http://www.cag.lcs.mit.edu/dynamorio/.

[47] 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,” in *Proc. of 2005 Programming Language Design and Implementation (PLDI) conference*, june 2005.

[48] N. Nethercote and J. Seward, “Valgrind: A program supervision framework,” in *Proc. of the Third Workshop on Runtime Verification (RV’03)*, Boulder, Colorado, USA, July 2003.

[49] A. Milenkovic, M. Milenkovic, and J. Kulick, “N-tuple compression: A novel method for compression of branch instruction traces,” in *Proc. of the* 16^{th} *international conference on parallel and distributed computing*, 2003.

[50] R. A. Uhlig and T. Mudge, “Trace-driven memory simulation: a survey,” *ACM Computing Surveys*, vol. 29, 1997.

[51] Y. Ramin, “ATPhttpd,” http://miyazaki.ece.cmu.edu/~jnewsome/atphttpd.

[52] r code, “ATPhttpd exploit,” http://miyazaki.ece.cmu.edu/~jnewsome/atphttpd.

[53] M. E. Locasto, K. Wang, A. D. Keromytis, and S. J. Stolfo, “FLIPS: Hybrid adaptive intrustion prevention,” in *Proc. of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, 2005.

[54] P. Vixie, O. Gudmundsson, D. Eastlake, and B. Wellington, “RFC 2845: Secret key transaction authentication for dns (TSIG),” http://www.ietf.org/rfc/rfc2845.txt.

[55] CERT/CC, “ISC BIND 8 contains buffer overflow in transaction signature TSIG handling code,” http://www.kb.cert.org/vuls/id/196945.

[56] US-CERT, “Vulnerability note vu#196945 - isc bind 8 contains buffer overflow in transaction signature (tsig) handling code,” http://www.kb.cert.org/vuls/id/196945.

[57] M. Cost, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, “Vigilante: End-to-end containment of internet worms,” in 20^{th} *ACM Symposium on Operating System Principles (SOSP 2005)*, 2005.

[58] J. Newsome, D. Brumley, and D. Song, “Vulnerability-specific execution filtering for exploit prevention on commodity software,” in *Proc. of the* 13^{th} *Annual Network and Distributed System Security Symposium (NDSS)*, 2006.

[59] M. Mohri and M.-J. Nederhof, *Robustness in Language and Speech Technology*. Kluwer Academic Publishers, 2001.

[60] J. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, pp. 386–394, 1976.

[61] D. Detlefs, G. Nelson, and J. Saxe, “Simplify: A theorem prover for program checking,” HP Labs, Tech. Rep. HPL-2003-148, 2003.

[62] G. Kildall, “A unified approach to global program optimization,” in *1st ACM Symposium on Principles of Programming Languages (POPL)*, 1973.

[63] M. Weiser, “Programmers use slices when debugging,” *Communications of the ACM*, vol. 25, pp. 446–452, 1982.

[64] A. S. Christensen, A. Møller, and M. I. Schwartzbach, “Precise analysis of string expressions,” in *Proc. 10th International Static Analysis Symposium, SAS ’03*, ser. LNCS, vol. 2694. Springer-Verlag, June 2003, pp. 1–18.

[65] M. Christodorescu, N. Kidd, and W.-H. Goh, “String analysis in x86 binaries,” in *Proc. of the* 6^{th} *ACM Workshop on Program Analysis for Software Tools and Engineering PASTE*, 2005.

.1 Regular expression notation

Signatures are given as regular expressions enclosed between quotes, i.e. "ω" is the regular expression ω. We use the following common meta-characters and notation:

. (dot) - matches any character.

$c\{x, y\}$ - The character c repeated at least x times, and at most y times. $c\{x,\}$ means c is repeated at least x times, with no upper bound.

$c*$ - The character c is repeated zero or more times.

$c+$ - The character c is repeated one or more times.

$[xy]$ - Matches either x or y. Example: "[gG]" matches either 'g' or 'G'.