Towards Practical Automatic Generation of Multipath Vulnerability Signatures

David Brumley, Zhenkai Liang, James Newsome, and Dawn Song

Original manuscript prepared April 19, 2007 [1]
CMU-CS-07-150

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Signature-based defense systems are one of the most popular architectures for defending against exploits of vulnerabilities. At the heart of a signature-based defense system is the signature generation mechanism. Since manual signature generation tends to be slow and error-prone, we need automatic signature generation techniques.

In this paper, we present the first practical approach for automatically creating vulnerability signatures which recognize different exploit variants of a vulnerability regardless of the execution path they take. Vulnerability signatures are based on the semantics of the vulnerability in the program itself, thus are more accurate than other types of signatures. A key limitation of previous vulnerability signature generation approaches is that they were only able to demonstrate signature generation for a single program path that an exploit may take to exploit a vulnerability. However, there may be multiple program paths which an exploit can take to the vulnerability, resulting in unacceptably many false negatives if only one path is covered by the signature. We address this shortcoming by presenting and implementing techniques for automatically generating practical vulnerability signatures which cover multiple paths. By covering multiple paths, our signatures have lower false negatives than previous approaches, while still guaranteeing zero false positives.

[1] This paper was originally submitted to CCS 2007, and is currently in draft form. Please contact the authors for later versions.

Keywords: vulnerability signature, ternary signature, multi-path signature, error-free signature

1 Introduction

New vulnerabilities are constantly being discovered and utilized by attackers to compromise systems. Today, it is not uncommon for a new vulnerability to be discovered and exploited in the wild before the vendor and the public learns about it. One of the most popular and effective defense mechanisms against attacks is signature-based input filtering (a.k.a. content-based filtering). Signature-based input filtering matches program inputs against a signature, where an input that matches the signature is considered as an exploit, and an input that does not match the signature is considered as benign by the signature-based filter.

At a high level, the heart of a signature-based defense system is signature generation. As manual signature generation is usually too slow and error prone to be effective, we need techniques for *automatic* signature generation. The challenge is to create automatic signature generation techniques that can *guarantee* the accuracy of their generated signatures. Signature accuracy is typically measured in terms of false positives, which are benign inputs that the signature indicates are exploits, and false negatives, which are exploit inputs that the signature indicates are benign. Signatures with high false negatives are undesirable since they may not block a sufficient number of different exploits for the same vulnerability. Signatures with false positives block legitimate traffic, which in some scenarios may be worse than missing exploits. In addition, the signature generation algorithm should be fast to reduce the window of vulnerability, and to defend against fast-propagating worm outbreaks.

Although numerous signature-based defense systems have been proposed [7,11,12,14–16,22,24,26,28], signature generation is not a solved problem. Most previous signature generation work can generally be categorized as either pattern-extraction-based or vulnerability-based. The pattern-extraction-based approach infers a signature by extracting out common patterns in a set of (potential) exploit samples [11,12,14,16,22]. Since this approach relies on training data provided by a malicious attacker, and typically leverages little or no semantic knowledge about the vulnerability, signatures generated using this approach cannot provide useful accuracy guarantees; hence they may generate signatures with arbitrarily high false positive and false negative rates. Indeed, recent research has shown that an attacker can often fool these approaches into generating highly inaccurate signatures [17, 19].

Vulnerability-based signature generation has been recently proposed to address the deficiencies of the pattern-extraction-based approach [5–7]. Vulnerability-based signature generation generates signatures based on the vulnerability itself (instead of exploit samples). By basing the signature generation on the vulnerability, we can make stronger guarantees about signature accuracy. For example, previous work shows how to generate vulnerability signatures which have zero false positives, i.e., guarantee *soundness* [5–7]. However, previously implemented vulnerability-signature generation methods only detected attacks that caused the vulnerable program to follow a particular program path [5, 7], or were of theoretical interest [6]. As a result, exploit variations that cause the vulnerable program to follow even a slightly different program path would evade the generated signature. This is a severe shortcoming, as many vulnerabilities can be exploited via several program paths.

For example, consider a vulnerable URL decoding procedure in a web server. This procedure may be called in a variety of contexts, e.g., to decode URLs as part of an HTTP "POST" procedure and decoding URLs using the HTTP "GET" procedure. A vulnerability in the URL decoding could then be exploited along two different paths: one going from "POST" to the URL decoding, and one from "GET" to the URL decoding. Considering either path leads to a sound signature, but may miss exploits along the other code path. Worse, for many vulnerable programs, many exploit variations that cause the server to follow different program paths can be generated by simply varying the number or length of protocol fields, thus changing the number of times that a particular loop executes.

One may think that handling multiple paths to a vulnerability is a straight-forward extension of the single path case. However, this is not the case: in real programs the presence of loops often results in an infinite number of paths to the vulnerability point. As a result, enumerating each path one-by-one is impossible. Brumley *et al.* [5] propose that a vulnerability signature can be generated by modeling the vulnerability as a whole, in a language with the same expressiveness as the language of the vulnerability, hence allowing the signature to contain loops. While this approach could be used to generate signatures that never have false positives or false negatives, the authors describe this approach as only of theoretical interest as a basis for measuring the accuracy of other signature classes. The reason this is of theoretic interest is such a signature could recognize all exploits, but would not necessarily return for benign inputs. Thus, handling multiple paths in a practical way was previously an unsolved problem.

Overall, creating multiple-path signatures poses many challenges which previous work does not sufficiently address, including:

- No previously implemented approach can guarantee that an input is *safe*; *i.e.* that it will *not* exploit the vulnerability. The only previously *proposed* approach [5] that may do so may not return SAFE until the exiting the entire program. This solution is impractical since some programs are not designed to terminate, such as network servers.
- Whether a vulnerability is exploited often depends not only on network input, but also on additional state external to the vulnerable program, such as configuration files. While this seems obvious, to our knowledge no previous work in sound signature generation addresses this issue.
- Much of the processing in the vulnerable program is irrelevant to whether or not a vulnerability is exploited, and does not need to be considered in generated VSM signatures. We show that even with limited static analysis, VSM signatures can be generated that can be evaluated very efficiently. There are several more sophisticated static analysis techniques that could be used to greatly improve the performance of generated VSM signatures.

In this work, we design and implement a method for automatically generating error-free vulnerability signatures that are practical and cover multiple execution paths, using only a vulnerable binary program and a vulnerability specification. We call these signatures *Vulnerability State Machine signatures* (VSM signatures). Our techniques address the challenges and issues left unresolved by previous work. For example, since deciding whether a signature with loops will halt is undecidable, we develop techniques based on the principle that *the next best thing to always being right is knowing when to admit that you don't know.* Hence, rather than simply matching or not matching an input as is done by a traditional signature, a VSM signature may return EXPLOIT, SAFE, or UNKNOWN. Differentiating between inputs that must be safe, and those that cannot be accurately classified enables signature defenses to enact comprehensive and flexible security policies. For example, a filtering application may allow inputs classified as SAFE to pass, but apply further analysis to inputs classified as UNKNOWN. We call any signature type that provides this type of guarantee an *error-free ternary signature*.

Contributions In particular, our contributions are as follows:

- We propose *error-free ternary signatures*, which are the first kind of signature that are guaranteed to never return an incorrect answer, *and* to always return an answer. At a high level, ternary signatures guarantee the meaning of both inputs matching the signature (i.e., exploits) and also inputs *not* matching the signature, i.e., whether they are really benign vs. the signature cannot guarantee it is benign or malicious. Binary signatures typically do not typically make such guarantees for both safe and malicious traffic.
- We design and implement a method for automatically generating Vulnerability State Machine signatures (VSM signatures), a type of error-free ternary signature, given only a vulnerable binary program

and a vulnerability specification. In addition to the theoretical guarantees provided by all error-free ternary signatures, we show that in practice VSM signatures have the following additional properties:

  - Rarely return UNKNOWN for exploit inputs for many vulnerabilities.
  - Rarely return UNKNOWN for safe inputs for many vulnerabilities.
  - Take very little time to process and classify each input.

- We have implemented the first system which generates sound vulnerability signatures over multiple paths. We perform end-to-end tests on three vulnerabilities, and macro-benchmarks on over 90 programs.

2 Problem Definition

In this section we present definitions for terms used in the remainder of the paper.

2.1 Error-free Ternary Signature Definition

A traditional vulnerability (or exploit) signature takes an input, and produces a binary output; typically the signature "matches", signifying that the input is an exploit, or doesn't match.

Most previous approaches provide no accuracy guarantees, making the output of the signature a hint at best. Some previous approaches guarantee *soundness*, meaning that when the signature matches an input, that input is guaranteed to be an exploit. However, no previously implemented approach provides any guarantee about the meaning of the signature not matching an input; *i.e.*, an input that does not match may still be an exploit.

In this work, we propose signatures that can classify an input as an exploit with guaranteed accuracy, and unlike previous approaches, can classify an input as *safe* with guaranteed accuracy.

For such a signature to be possible, we must address two fundamental issues. First, the problem of deciding whether an input will exploit a particular vulnerability is reducible to the halting problem [5], and is hence undecidable [10]. As a result, all previous signatures either may classify non-exploits as exploits, may classify exploits as non-exploits, or may not halt. We address this problem by instead allowing the signature to return a third classification: UNKNOWN. While one can trivially create a signature that meets these requirements, *e.g.* by returning UNKNOWN for all inputs, the challenge is to create signatures that rarely return UNKNOWN.

The second issue is that a particular input may be an exploit for a vulnerable program running on a host with one state, but be *safe* when running on a host with a different state. For example, an exploit may only be effective against a server with a particular configuration option enabled. Therefore, it is only possible to accurately classify an input as an exploit or as safe *with respect to a particular external state*. We address this problem by parameterizing the signature by the external state. That is, the signature takes both an input, and an external state, and determines whether the input is an exploit for a host running the vulnerable program with the given state.

Definition: Vulnerability specification The *vulnerability specification* (v_p, v_c) is a concise specification of a vulnerability [5, 6] consisting of a distinguished *vulnerability point* v_p where the vulnerable program P may "go wrong", and *vulnerability condition* v_c that specifies what "going wrong" is. We say that a program with vulnerability (v_p, v_c) is *exploited* when it reaches v_p in a state that satisfies the predicate v_c.

The vulnerability specification is what determines a single vulnerability in a program.

Definition: Error-free ternary signature An error-free ternary signature $\mathcal{S}$ is a function

$$\mathcal{S}(i, \Sigma) \to \{\text{EXPLOIT}, \text{SAFE}, \text{UNKNOWN}\}$$

where i is an input, and Σ is an external state. An error-free ternary signature returns EXPLOIT only when the vulnerable program *must* be exploited if executing given i and Σ; returns SAFE only when the vulnerable program *cannot* be exploited if executing given i and Σ; and always returns an answer in finite time (*i.e.*, halts).

2.2 The Signature Generation Problem

In this work, we address the problem of how to *automatically* generate error-free ternary signatures. As with previous work, we assume we are given the program P and the vulnerability specification (v_p, v_c) [5–7]. One way to obtain this information is from an exploit detector; in many cases (v_p, v_c) is easily identified directly from the security violation detected by such detectors.

Formally, an error-free ternary signature generation algorithm is given as input a vulnerable program P, and a vulnerability specification (v_p, v_c), and produces an error-free ternary signature $\mathcal{S}$.

In addition to the formal guarantees provided by error-free ternary signatures, there are several additional goals:

- They should be efficient to evaluate.
- They should return EXPLOIT for most inputs that would exploit the vulnerability, including polymorphic variations.
- They should return SAFE for most inputs that cannot exploit the vulnerability.

3 Our signature generation approach

Our approach consists of building the signature $\mathcal{S}$ by constructing a model of the vulnerability in the program using program analysis techniques. We call the signatures we generate *Vulnerability State Machine signatures* (VSM signatures). The high-level idea of this approach is that the signature models the program's vulnerability directly, thus we can guarantee our signature's accuracy with respect to the vulnerability. The beauty of the approach is that we need not have semantic information about the program; we can use semantic-preserving transformations on the program itself to automatically extract the model of the specific vulnerability, given the vulnerability specification. This approach allows us to generate guaranteed *error-free* signatures: if $\mathcal{S}$ returns EXPLOIT for an input, then the input would exploit the vulnerability, and if $\mathcal{S}$ return SAFE, then the input is benign.

The signature faithfully replicates the behavior of the vulnerability by faithfully modeling the parts of the program that are relevant to the vulnerability. The states in the signatures model are derived from the program itself, e.g., if the program evaluates an instruction on which the vulnerability is dependent, the signature will have a corresponding statement which replicates the effects of that instruction. At the vulnerability point, $\mathcal{S}$ checks if the vulnerability condition would be satisfied, and if so, transition to a special EXPLOIT state. At each place in the model where it can guaranteed that the exploit state is unreachable (i.e., corresponding to a state in the program at which the program could not possibly be exploited), the model transitions to the SAFE state. This correspondence between the signatures model and the program's vulnerability is what makes our signatures error-free. Finally, we ensure that the signature always returns an answer by constructing it to return UNKNOWN in cases where it may not be possible to return SAFE or EXPLOIT. Therefore, VSM signatures satisfy the criteria of an error-free ternary signature.

The Core Challenges There are several core challenges in creating a practical error-free ternary signature generation algorithm that produces useful VSM signature.

The core challenges are as follows:

- Correctly replicate the relevant semantics of the original program to ensure that the signature is error-free.
- Ensure that the signature returns an answer, and does so as quickly as possible, while minimizing how often the signature returns UNKNOWN.
- Ensure that the signature is of finite, reasonable size, while minimizing how often the signature returns UNKNOWN.

Generation steps. Our approach for signature generation consists of the following steps:

1. Model the program P to an *execution state model* M, which models the semantics of P.
2. Augment M to return EXPLOIT for cases when the vulnerability (v_p, v_c) would be exploited by an input.
3. Augment M to return SAFE for cases when the vulnerability (v_p, v_c) cannot be exploited by an input.
4. Compile M into a VSM signature $\mathcal{S}$ that is guaranteed to halt, and which can be efficiently evaluated.

Given an input i and external state Σ, we can then evaluate the signature $\mathcal{S}(i, \Sigma)$ on an input i to determine whether i is an exploit, benign, or cannot be accurately classified by $\mathcal{S}$ under the external state Σ.

We describe each of the signature generation steps below.

3.1 Translate program to execution state machine

We first build an *execution state model* (ESM) M that preserves and models the semantics of the program P. We write the ESM in our signature language (Section 4.1). The execution state machine is represented as a graph $g = (V, E)$, where each node in V is a statement in the signature language, which replicates the effects of an instruction, and each edge $E(n_1, n_2)$ is labeled with a predicate c. The predicate c must be true to make the transition from n_1 to n_2. For example, if n_1 is a conditional jump based on condition c with true jump target n_2 and false jump target n_3, then there will be an edge $E(n_1, n_2)$ with label c and $E(n_1, n_3)$ with label $\neg c$. Note system calls and calls to library functions are translated as calls to externally defined functions at this stage.

To handle cases where we cannot or do not represent every possible (x86) instruction of P in our signature language, or cannot accurately represent the control flow in M (such as with indirect jumps), we introduce the unknown state $n_{unknown}$. We add edges to M such that whenever control reaches such an instruction, we transition to $n_{unknown}$.

Note that this step does not depend on the vulnerability specification, and hence may be done as a preprocessing step, before any vulnerability is known. At this point, M models the semantics of the program P.

3.2 Determine when to return EXPLOIT

Next, we augment M by adding an EXPLOIT state represented by $n_{exploit}$, and at a high level, add edges such that if the vulnerability would be exploited on an input, M transitions to $n_{exploit}$.

To augment M, we first locate the node in M that corresponds to the vulnerability point v_p in the original program P. Let n_p represent this node. For each edge that transitions to the vulnerability point, $E(n_i, n_p)$ with predicate label c, we replace the label c with $c \wedge \neg v_c$. This step ensures that only inputs which do not satisfy the vulnerability condition transition to n_p. We then add a transition $E(n_i, n_{exploit})$ with predicate $c \wedge v_c$. The final result is M such that M transitions to $n_{exploit}$ when the vulnerability condition is satisfied.

At this point M recognizes exploits of the vulnerability, but cannot yet determine when an input is benign.

3.3 Determine when to return SAFE

Next, we augment M to transition to SAFE for inputs which cannot exploit the vulnerability. We add a node n_{safe} corresponding to the safe state. We then add edges $E(n_i, n_{safe})$ to transfer control to n_{safe} when we can determine that it is impossible for the execution on the given input to reach the vulnerability point v_p such that the vulnerability condition v_c is satisfied.

Adding these edges in such a way that M is sound, while reaching n_{safe} for as many non-exploit inputs as possible, as quickly as possible, is one of the challenging issues that we address in this work. We leave the details of our approach to Section 4.3.

3.4 Generating the Final Signature

The last step is to generate the final VSM signature from the augmented execution state model M. We first ensure that M will terminate on all inputs by analyzing all loops in the program, and adding transitions to transfer control to the unknown state $n_{unknown}$ in cases where it may otherwise never terminate. We then perform optimizations on M, and compile M into $\mathcal{S}$, which can be evaluated directly on inputs. We compile by translating from the signature language to an executable language, and by linking external function calls to actual implementations.

Note that we do not link against external functions or system calls directly; we link against *stub functions* which take appropriate action by either calling the real function, or simulating external behavior the signature should not replicate (see Section 4.4.4).

The output of this step is an executable VSM signature $\mathcal{S}$.

3.5 Signature Evaluation

Signature evaluation $\mathcal{S}(i, \Sigma)$ consists of running $\mathcal{S}$ on the input i, and an external state Σ. Note it is important to distinguish between the original program and the signature: evaluating the signature corresponds to *simulating* part of the execution of the program. For example, in our signature language we model memory as a hash-table: given a 32-bit integer, we return an 8-bit integer, which corresponds in the program to an 8-bit read from memory at a 32-bit address location. Thus, a memory write in the original program is simply a hash-table update in the signature. As a result, an unknown buffer-overflow vulnerability in the original program may cause data (i.e., a hash-table entry) in the *simulated* memory to be overwritten, but does not and cannot overwrite data outside of the hash-table itself.

Informally [1], the operational semantics of the signature language are as follows. The evaluation state of a signature at any point is described by $\phi(n, \Delta)$, where n specifies a node in the graph (i.e., is like a program counter), and Δ specifies the current values of all memory and variables. We begin at an initial state $\phi(n_0, \Delta_0)$ (e.g., all variables and memories in Δ_0 are initialized to zero). Optionally, we need not begin at n_0, as discussed in Section 4.5.2. The machine transitions from $\phi(n_1, \Delta_1) \rightarrow \phi(n_2, \Delta_2)$ if executing the statement at node n_1 with values from Δ_1 satisfied the edge predicate label c for $E(n_1, n_2)$. For example, if n_1 is the statement `if` $v_1 = 0$ `then jmp(`n_2`) else jmp(`n_3`)`, we lookup the current value of v_1 in Δ_1, and transition to n_2 if the value is zero, else transition to n_3. The new state of values in variables and memories after executing n_1 is in Δ_2.

Calls to external functions are redirected to the appropriate stub function. The stub function has access both to the external state Σ and the current internal state Δ. For example, the stub function for a web server vulnerability may contain which files exist on the server, while the internal state contains the current

[1] The full signature language and formal operational semantics are specified in [2].

$*(v_1) := v_2 | v_1 := *(v_2) | v := c | v := v_1 \diamond v_2$
$| \; v := \neg v_1 \; | v := !v_1 \; |$ `label` l_i | nop |halt
| `jmp` ℓ | `ijmp` v | `if` v `jmp` ℓ_1 `else jmp` ℓ_2

Figure 1: A representative part of the signature language.

request. A stub function will need both pieces of information in order to return the appropriate value, e.g., to determine whether a file specified in Δ exists in Σ.

4 Design and Implementation

We now describe the design and implementation details of each step from Section 3.

4.1 Translate program to execution state machine

The first step is to model the given program P to a semantically equivalent execution state model M. In our system, the given program is an IA-32 binary. IA-32 programs have a number of complexities that make them difficult to analyze, including single instruction loops, instructions that implicitly read and set registers not directly referenced, and instructions whose semantics depend upon the operand values.

We address this challenge by first translating the program into a semantically equivalent program in our signature language, summarized in Figure 1. The language is simple, making analysis such as rewriting jump targets in the model, yet expressive enough so that we can easily translate x86 assembly into the language. In Figure 1, v denotes variables, c denotes constants, and $\diamond$ is a binary operator $\in +, -, *, \mathrm{mod}, <<, >>$. The language has assignment, stores ($*(r_1) = r_2$), loads ($r_1 = *(r_2)$), and control flow either as a direct jump to a known label ℓ, or to a computed location via `ijmp`.

We create the nodes of M by first using an off-the-shelf disassembler to parse the binary program P into code segments, and to disassemble each statement. We then translate each disassembled x86 instruction into the semantically equivalent sequence of statements in our signature language. Each statement then becomes a node in M.

The next challenge we must address is to add edges in M to accurately represent the control flow of P. While most instructions simply transfer control to the next instruction, or to a statically determined address, x86 instructions may transfer control to an address that is computed dynamically at run time.

The most common type of indirect jump in IA-32 programs is the `ret` instruction, which uses a return address previously stored on the stack, usually by the `call` instruction that called the function containing the `ret`. In our implementation, we assume that the program obeys the normal stack discipline in which functions return to their caller. We use standard techniques to identify functions in the IA-32 binary program (e.g., use IDA-Pro's built-in feature to extract function boundaries, look for function prologues, etc), and resolve the possible destinations of the `ret` instructions accordingly.

For other types of indirect jumps, such as those which arise from using a function pointer, we can potentially resolve targets using register value analysis [3, 4, 23]. Indirect jump targets which cannot be resolved will go to UNKNOWN. Our experiments show the number of indirect jumps (other than `ret`) can be small (Section 5.5.2).

4.2 Determine when to return EXPLOIT

We assume we are given the vulnerability point v_p expressed as an instruction address in P, and the vulnerability condition v_c expressed as a predicate in our signature language. We create the exploit node $n_{exploit}$, and add the corresponding edges to M such that whenever the program would have transferred to the vulnerability point v_p in a state such that the vulnerability condition v_c would be satisfied, M transfers control to the exploit node $n_{exploit}$, adding the appropriate edge predicate labels as described in Section 3.2.

4.3 Return SAFE when unexploitable

We next find states in M from which $n_{exploit}$ (the EXPLOIT state) is unreachable, and rewrite M to transfer to n_{safe} (SAFE state) or $n_{unknown}$ (UNKNOWN state). Our goal is for most non-exploit inputs to drive execution of M to the SAFE state rather than the UNKNOWN state, and for M to transfer to the SAFE state with as little processing as possible for non-exploit inputs. We accomplish these goals by performing reachability analysis, and modifying M to transfer control to the SAFE state when it is no longer possible to reach the EXPLOIT or UNKNOWN states.

Reachability analysis computes all paths which start at some node n_i and terminate at a node n_j. Consider this analysis for computing all paths which can reach the EXPLOIT state $n_{exploit}$ from the start node n_0 of M. Reachability analysis is done via the following algorithm: First, we create a back-edge from $n_{exploit}$ to the start node n_0. Adding this back-edge creates a cycle in M. Assuming $n_{exploit}$ is reachable at all, then there is a path from n_0 to $n_{exploit}$; the back-edge completes the cycle. The case where there is no path from n_0 to $n_{exploit}$ is degenerative, since it signifies that no input could ever exploit the vulnerability in M. Second, we compute the strongly connected component (SCC) subgraph containing $n_{exploit}$. The SCC by definition contains all nodes reachable from n_0 to $n_{exploit}$, thus for all (reachable) nodes n_k not in the SCC, there is no path from n_k to $n_{exploit}$ (else there would be a path from n_0 to n_k to $n_{exploit}$, and n_k would be in the SCC).

We call the set of nodes and edges contained inside an SCC is called the *chop*. We compute the union of the chops from M's start state n_0 to $n_{exploit}$ (the EXPLOIT state), and from n_0 to $n_{unknown}$ (the UNKNOWN state). Since the EXPLOIT and UNKNOWN states are unreachable from any node not in this chop, we rewrite the destination of every edge $E(n_i, n_j)$ where n_j is not in the chop and n_i is as $E(n_i, n_{safe})$. Edges $E(n_j, n_i)$ when n_j is not in the chop and n_i is can be removed from M all together (since there must not be a path from n_0 to n_j, else n_j would have been in the chop, n_j is unreachable).

The above algorithm was first proposed for signature creation by Brumley *et al.* [5]. Unfortunately, this algorithm leaves out many details relevant to creating practical signatures, such as exactly what graph of M the chop is performed on. Up to now, we have not described precisely how functions in P are represented in M. As it turns out, how functions are represented in M can significantly affect the overall signature generation and evaluation time.

Each function f in the program induces a subgraph in M, i.e., the control flow graph of f is a subgraph of M. Each subgraph CFG $\in M$ has a distinguished entry point for the function. In addition, we can make it so all functions have exactly one canonical exit point. The most obvious thing to do is create a "super-graph" which link up call sites of f with the callee's entry points, and link up the return exit sites of the callee with the next instruction after the call site. For example, if `bar()` with entry node 10 and exit node 11 is called by `foo()` on line 4 in the program, we add an edge $(4, 10)$ and $(11, 5)$ to M.

On the surface, this solutions looks good since it links up callers to callee's. The main problem is that it is not *context-sensitive*. The lack of context-sensitivity results in including more in the chop than necessary. To see this, suppose a function `irrelevant()` also calls `bar()`. Then the chop cycle contains `bar()`, `foo()`, and `irrelevant()`. This is a well-known problem in compiler research where the lack of context-sensitivity results in unnecessary pollution in the analysis.

The fix is to add to the chop algorithm context sensitivity so that we can distinguish between different call sites. A calling context can be uniquely represented by the entire call stack. The string of call sites on a stack is referred to as a *call string* [1]. In the previous example, we have two call strings: `foo,bar` and `irrelevant,bar`. We build up the call strings for the program by symbolically executing call/returns in the program, up to recursion.

We compute M's graph in a context sensitive manner (up to recursion) based upon the call strings. If a function is called in 3 different contexts, there will be three copies of the function in M. Although duplicating nodes does increase the initial model size, in our experience it usually reduces the chop size because we do not include impossible call/return relationships in the chop.

At this point we have a context-sensitive graph in M, with an edge to EXPLOIT for transitions that must exploit the original program, to SAFE for transitions that cannot exploit the original program, and UNKNOWN for transitions from which we cannot determine for certain.

4.4 Generate final signature

The last step is to compile M to an executable signature $\mathcal{S}$, which we ultimately evaluate inputs on to determine if they are SAFE, EXPLOIT, or UNKNOWN.

4.4.1 Ensure termination

The original program P cannot be guaranteed to terminate. Indeed, for network servers, non-termination is the norm rather than the exception. Many network applications operate in a loop that indefinitely accepts and processes connections. We implement two techniques to ensure that M terminates on all inputs: *bounded iteration*, and *subprogram analysis*.

Bounded iteration We modify our model M to guarantee that it terminates on all inputs by performing *bounded iteration*. Using this technique, we first identify loops that may not terminate. For each back-edge $E(n_i, n_{loop})$, we set a limit l_i for the number of times that the loop may iterate. We then update the model to increment the iteration count for that loop every time the back-edge is traversed, and to transfer control to UNKNOWN if the limit is exceeded.

This is a fairly straight-forward process. The only question left is how many times to allow loops to iterate before returning UNKNOWN. If we select too few, the VSM signature may return UNKNOWN on inputs that it would have eventually classified successfully. If we select too many, the signature will waste time looping on inputs that it will never be able to classify successfully.

We have found the only common non-terminating loops that we have encountered in practice are server request-processing loops. We currently identify such loops manually, and limit the number of times it may iterate to *one*. We expect that this type of loop could usually be found automatically with some simple analysis.

For other loops, we try to find an upper bound on the number of iterations. If we cannot determine the maximum number automatically, a default threshold is used. In practice, we have found we did not need to analyze other loops. The reason is most loops happen inside library functions (e.g., `strcpy`, `sprintf`, etc.), which terminate by convention, and are not modeled in the signature itself. If subsequent common idioms using infinite loops are identified, we may again use heuristics to identify those idioms, and bound the number of iterations appropriately.

Subprogram analysis While we have found bounded iteration to be sufficient to ensure that the VSM signature returns an answer in a timely manner, it has the unfortunate drawback of returning UNKNOWN for non-exploit inputs for applications that implement infinite accept loops, when the vulnerability point is reachable from inside the loop.

We observe that as a practical consideration, we typically want to determine whether the vulnerable program will be exploited *while processing the given input*, not whether it can *ever* be exploited. Additionally, we observe that a program is typically done processing an input when it loops to accept a new connection.

Therefore, we propose an alternative solution of creating the VSM signature with respect to the *subprogram* of interest. In this case, the subprogram of interest is a *single* iteration of the accept loop. Since no loop exists in this subprogram, EXPLOIT is unreachable after processing the input, at which point the VSM signature returns SAFE. Subprogram analysis is performed simply by deleting edges in the execution state machine before performing the reachability analysis described in Section 4.3.

This solution is not perfect, as some vulnerabilities may exist in which a malicious input is not fully processed until some later point after processing other connections in the accept loop. In this case, an input that is safe with respect to a single iteration of the loop may not be safe with respect to the entire program. Such cases may be detected by performing dependence analysis to detect whether the input has further effect on the program after the first iteration of the loop.

4.4.2 Optimizations: Dead-code elimination

We can simplify the VSM signature generated so that is smaller by removing *dead code* code that has no effect on the final output of EXPLOIT, SAFE, or UNKNOWN. Smaller models both evaluate faster, and are faster to compile in the next step. Much of the processing in the original program P may be dead code in M. For example, P may spend some processing determining its response to a network request, which has no effect on whether the given vulnerability is exploited or not. This code may be safely removed from M.

We currently perform dead-code elimination on scalar variables, which removes some unnecessary processing. We expect that much more dead-code could be eliminated by performing dead-code elimination on memory, which is more challenging due to the need for pointer alias analysis. We may also be able to eliminate calls to external functions whose output does not affect the vulnerability, which has the additional benefit of the VSM signature not requesting irrelevant external state. To do this, we would also need summaries for functions that take pointer arguments, specifying the memory range that they may read from and write to.

4.4.3 Compile to native code

When creating the initial VSM signature, we translate the IA-32 program to our own signature language, as described in Section 4.1. We provide a compiler for the signature language which takes as input a model, and outputs a C program. We can then leverage existing C compilers to perform additional optimizations and produce the final executable.

Translating our language into C is straight-forward. The only complication is how memory access in the original IA-32 program are converted from our signature language to C programs such that all access are valid. As mentioned, we treat memory as a hashtable where given an address, we return a value. This memory representation is incompatible with that which external functions expect. To solve this problem, we provide stubs which translate between the signature language's memory representation and native code, and link against these stubs. However, one advantage of our approach is we need not worry about unknown buffer overflows in the original program, since they are not translated as real buffer overflows in the signature. In future work, we hope to eliminate the translation at compile time while still retaining the additional safety and abstraction advantages.

4.4.4 Resolve external function calls

During linking, we link calls to external functions to our stubs. If the called external function does not operate on the external state, then our stubs in turn calls the actual external function, i.e., the signature calls the stub (which performs any necessary translations from our memory scheme), then the stub calls the external function.

In order to prevent unnecessary side-effects such as writing and reading files, function stubs for system calls and for functions that operate on input or other external state must instead replicate the semantics of the original call while operating on the input i or external state Σ provided at evaluation time. As a concrete example, our function stub `stat` which returns a value specified in Σ. Note that these function stubs are not specific to a signature or vulnerability, and hence need only be written once for a given library function or system call.

In our current implementation, stubs that require input or external state that is not present block execution of M, and notify the caller of what input or state is necessary to continue. This allows us to supply the input and external state in an on-demand fashion. This is convenient for a number of reasons. In particular M can begin executing on the first packet of a network connection. It may reach a decision based on the first packet alone, or may notify the caller that it needs more data; *i.e.*, it cannot make a classification without processing the next packet.

4.5 Signature Evaluation

Once compiled, the signature is an executable piece of code that takes an input i, an external state Σ, and accurately returns either EXPLOIT, SAFE, or UNKNOWN.

A final step is to obtain the appropriate external state. Many vulnerabilities are independent of any state external to the vulnerable program, in which case it doesn't matter what external state is used when evaluating the signature. In Section 4.4.2 we describe how we may be able to detect such cases automatically, and remove the VSM signatures requests for external state.

However, there are some vulnerabilities for which the state is relevant. For example, a server may only be vulnerable if a particular option is enabled in a configuration file, or if a particular file exists. In these cases a VSM signature must be evaluated in the context of a specific external state.

4.5.1 Selecting and obtaining external state

Unlike previous signature approaches, our VSM signatures take not only an input i, but also an external state Σ. What external state to use depends on the precise question that the VSM signature user wishes to answer. If they wish to determine whether an input will exploit *some* host running a vulnerable program, they can use an external state that is known to be vulnerable. If they wish to determine whether an input will exploit a *specific* host, or any host with a particular configuration (as in a managed network), they can use the corresponding external state.

In either case, the external state can generally be obtained automatically, either as a preprocessing step or on-demand at run time, e.g., for our experiments, we obtained the necessary state from our host systems.

4.5.2 Additional Improvements: Preprocessing

As described so far, the VSM signature may perform an arbitrary amount of processing before actually reading the input. However, given some fixed external state (as described in Section 4.5.1), the VSM signature will always reach the same node n where input is initially read, with the same register and memory state Δ.

Rather than re-performing this processing for every input i to be evaluated, we take a snapshot of the state Δ of the actual program P at the point corresponding to node n, while running in a vulnerable configuration. When evaluating the VSM signature, we then begin execution at node n, with state VSM signature, rather than at the actual program start.

This optimization may be performed after the VSM signature has been created, using a hook to specify an alternative starting node n and state Δ. As an additional optimization, we perform this step *before* translating the program P to the execution state machine M. As a result, there is no need to incorporate any part of P into M that cannot execute after node n. This optimization not only improves matching time of the resulting signature, but also improves the signature size and generation time.

5 Implementation and Evaluation

5.1 Implementation

Our implementation is divided into two components. The first component is a translation engine which interfaces directly with the x86 binary and is responsible for parsing the binary (we currently support Windows PE and Linux ELF), reading in the assembly, and lifting the assembly to our signature language. This component is written in C/C++, and is about 16,500 lines of code, with about 1/3 of the lines of code geared towards parsing the binary and interfacing with the disassemblers, and the other 2/3 translating to the signature language. The second component is an analysis engine, which is written in OCaml. It performs the model creation, the compiler from the signature language to C, and the other analyses and optimizations discussed in Section 4. The total code size for the analysis component is about 28,000 of code.

5.2 Evaluation

We implemented and evaluated our approach on Linux operating system. Our evaluation was performed on a 3.2GHz Pentium 4 computer with 3GB memory. We measure three evaluation metrics: VSM signature accuracy, VSM signature matching speed, and VSM signature generation speed.

We perform in-depth evaluation on three vulnerable servers: ATPhttpd (web server), ghttpd (web server), and passlogd (syslog message sniffer). We then performed macro-benchmarks to measure overall model generation time on 98 additional binaries from the Linux coreutils-6.9 package.

5.3 In-Depth Analysis

We perform an in-depth evaluation of our techniques using three vulnerable servers: ATPhttpd, ghttpd, and passlogd.

We use the subprogram technique described in Section 4.4.1 to build the VSM signatures with respect to the accept loop, and the preprocessing technique described in Section 4.5.2 to begin evaluating the VSM signature at the point where the input is read. External state was unnecessary for all but the ATPhttpd vulnerability, for which we used external state from a known-vulnerable host.

For each vulnerable server, we provide a qualitative description of the vulnerability and the generated VSM signature, and evaluate the accuracy of our VSM signature on both exploit inputs and safe inputs. For completeness, we also include the time to generate each VSM signature and to match each VSM signature against exploit and safe inputs. These results are tabulated and further explained in Sections 5.3.4 and 5.4 respectively.

5.3.1 ATPhttpd

ATPhttpd is a small web server written in C [21]. ATPhttpd version 0.4b has a stack-smashing vulnerability. When the file specified in the URL of a 'GET' request is not found, ATPhttpd generates an error message including the requested URL, using an incorrect bounds check. This vulnerability can be exploited by requesting a file name that does not exist on the server, to trigger the error-handling routine, which is long enough to overflow the error-message buffer.

For this vulnerability, we defined the vulnerability specification as follows. The vulnerability point is the call to `sprintf` from the function `http_send_error`. The vulnerability condition is that the parameters supplied to `sprintf` will result in a long enough error message to overwrite the vulnerable stack frame's frame pointer.

Our techniques generated a VSM signature in 1.585 seconds. The accept loop contains no unanalyzed indirect jumps, and no unbounded loops. As a result, it contains no UNKNOWN node, and is hence guaranteed to return SAFE or EXPLOIT for every input.

We evaluated the VSM signature on both malicious and benign inputs. The VSM signature requires the result of a `stat` system call to determine whether the requested file exists. We resolved this external state by using a known-vulnerable configuration in which no requested files exist.

We evaluated the VSM signature using attacks from the vulnerability disclosure [20], for which the VSM signature returned EXPLOIT. We also manually varied the attack payload to confirm that the VSM signature also returns EXPLOIT for other attack variants. The VSM signature took an average of .00113 seconds to return an answer for each EXPLOIT input.

We also evaluated the VSM signature on synthetically generated benign HTTP requests. The VSM signature returned SAFE for such requests. The VSM signature took an average of .00015 seconds to return an answer for each benign input.

5.3.2 ghttpd

Ghttpd, a web-server written in C, contains a stack-based overflow in version 1.4.3 [25]. Similarly to ATPhttpd, ghttpd is vulnerable to excessively long 'GET' requests. It calls a logging function `Log` on every request, which fails to check the boundary of its local buffer when creating the log message using `vsprintf`. The vulnerability can be exploited using completely separate *multiple* execution paths, due to some error checking that occurs before calling `Log`.

The vulnerability specification for this vulnerability has a similar form to that of `ATPhttpd`. The vulnerability point is the call to `vsprintf` inside `Log`. The vulnerability condition is that the parameters passed to `vsprintf` will cause it to generate a message long enough to overwrite `Log`'s stack frame pointer.

Our techniques generated a VSM signature in 1.193 seconds. Again, the generated VSM signature contains no loops nor unresolved indirect jumps. As a result it contains no transitions to the UNKNOWN node, and hence returns EXPLOIT or SAFE for all inputs.

Attack	Compute VSM signature	VSM signature to C	Compile Signature	Total Time
ATPhttpd	0.676 s	0.029 s	0.88 s	1.585 s
ghttpd	0.549 s	0.034 s	0.61 s	1.193 s
passlogd	0.273 s	0.008 s	0.05 s	0.332 s

Table 1: Signature generation performance

We evaluated the VSM signature using attacks generated by the exploit program from [25]. We modified the exploit program to create attacks that have varying lengths, have randomly generated shell code, and cause different execution paths to be followed in the vulnerability. The VSM signature successfully returned EXPLOIT for all of these attack variations, with an average evaluation time of .00025 seconds. Note that previous vulnerability signature approaches that model only a single execution path would not have been able to detect these variations.

We also evaluated the VSM signature on synthetically generated benign HTTP requests. The VSM signature returned SAFE for all such requests, with an average evaluation time of .000165 seconds.

5.3.3 passlogd

Passlogd is an all purpose sniffer used to capture syslog messages. The vulnerability in passlogd is a stack-based buffer overflow. When copying a log level field in the function `sl_parse`, passlogd fails to check the buffer boundary when searching for a delimiting '>', resulting in a stack overflow [9].

The vulnerability point for this program is an assignment statement inside `sl_parse`: `level[j] = pkt[i]`. The vulnerability condition is that the address calculated by the expression `level[j]` is the address of `sl_parse`'s frame pointer.

Our techniques generated a VSM signature in 0.332 seconds. Unlike the previous vulnerabilities, the VSM signature contains several loops whose maximum iterations cannot be statically determined, including the copy loop where the vulnerability is located. As a result, the VSM signature *does* contain transitions to the UNKNOWN state, due to the bounded iteration transformation described in Section 4.4.1.

We evaluated the VSM signature using exploit code from [9], and again generated variations by randomizing the shell code and by altering the length of the input. The VSM signature returned EXPLOIT for all such variations, with an average evaluation time of .00270 seconds.

We evaluated the VSM signature on benign inputs obtained by converting messages from `/var/log/messages` on our desktop machines to syslog format. The VSM signature returned SAFE for all such inputs, with an evaluation time of .00220 seconds. The benign inputs were converted from the file `/var/log/messages`. We converted the log messages back to syslog format and evaluated them with our signature. Again, the signature had perfect accuracy with no transitions to the UNKNOWN state.

5.3.4 VSM signature Generation Time

We measured the time to generate VSM signatures for our three vulnerable programs. Figure 1 shows our results. We use CPU time (user time and system time) as the metric of performance evaluation. The column "Compute VSM signature" shows the time to compute the VSM signature from the binary; the column "VSM signature to C" shows the time to convert the VSM signature signature to C code; the column "Compile Signature" shows the time to compile C signature into native code. The results show signature generation is fast.

	Matching Benign (sec)	Matching Attacks (sec)
ATPhttpd	0.00015	0.00113
ghttpd	0.000165	0.00025
passlogd	0.00220	0.00270
passlogd*	0.00003	0.00051

Table 2: Signature matching performance

5.4 VSM signature Matching Time

We measured the matching performance of our signature both on malicious inputs and on benign inputs. In each case, for input i we execute $\mathcal{S}$ (i, Σ), and verified the correct result (either EXPLOIT or SAFE) was returned. The external state Σ was obtained from a vulnerable configuration for ATPhttpd, and is unneeded for the other vulnerabilities. We report the average of 1000 runs on each input. The results are shown in Table 2. The "Matching attacks" shows the matching time on attack requests, which were generated from publicly available exploits [9, 20, 25] and hand-crafted variants. The column "Matching benign" shows the matching time on benign requests for synthetically generated traffic (HTTP requests for the web-servers, and syslog messages from passlogd). When possible, we sent both malicious and benign requests which exercised multiple code paths.

Signature matching is overall reasonably fast. However, we were surprised passlogd took longer than previous examples, since it was a less complex program overall. We manually inspected our VSM signature to determine why passlogd is slower than ATPhttpd and ghttpd. It turns out that passlogd initializes about 5,000 bytes to zero in a loop, and the memory dereferences from this loop were resulting in worse performance due to our treatment of memory accesses. We provided an external state with all bytes already zeroed, and began executing the program after initialization, as described in Section 4.5.2. We show the performance numbers as passlogd*.

One interesting thing we observe is matching benign traffic takes significantly less time than attack traffic. On average, evaluating a benign request takes only about 55% as much time as attack requests. The reason for this is that the generated VSM signature returns SAFE as soon as the vulnerability point is unreachable, as described in Section 4.3, i.e., in general, the longest execution path is to the EXPLOIT state. Hence, safe inputs often require significantly less processing than exploit inputs.

5.5 Macro-benchmarks

In order to get a better idea how our techniques work on many different programs, we performed measurements on the 98 Linux binaries from the coreutils package. We report these numbers to give a feel of what our overall signature generation time is on other types of binaries.

5.5.1 Initial Model Creation Time

Since creating the end-to-end measurements indicate that creating the initial model dominates the total signature generation time, we performed a benchmark to see how long creating the context-sensitive VSM signature of the program in our signature language on a large number of other programs. Note that we are not creating a signature for a vulnerability, but measuring how long it takes to compute the initial model. This benchmark gives a rough idea how our signature generation system scales with binary size.

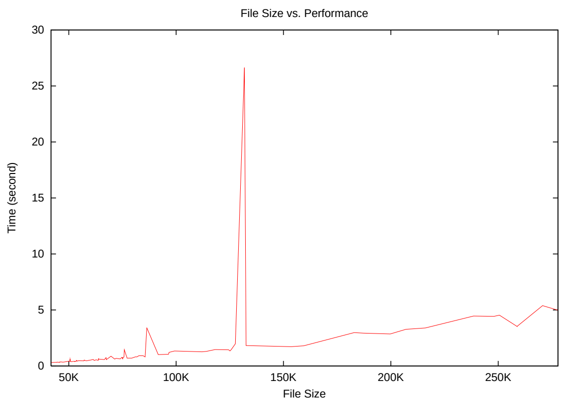

Figure 2: Macro-benchmark showing the time to create an initial model for 98 programs of varying size.

Figure 2 shows our results. We were able to analyze all 98 programs. Each took less than 30 seconds to analyze. Two programs, sha512sum and sha384sum, took the most times with about 26 seconds, mainly because one function, `digest_file`, was called in many different contexts, increasing the cost of the callstring analysis described in Section 4.3. If we remove these two outliers, the generation time for the remaining coreutils binaries ranges from .311 to 4.99 seconds, with an average 1.18 seconds.

5.5.2 Indirect Jump Analysis

We also analyzed the number of indirect jumps in the initial models created for the coreutils programs. Indirect jumps can arise from the use of function pointers or dynamically generated code. When an indirect jump is encountered during signature evaluation (not counting `ret` instructions), we return UNKNOWN, thus the number of indirect jumps is a metric of how many potential paths may classify an input as UNKNOWN. (Recall during our end-to-end test, no indirect jumps were evaluated, and we never returned UNKNOWN.)

We found on average 2.39 indirect calls (not counting the `ret` instruction) per program. The small number of indirect jumps supports the idea that there are very few control flow edges we cannot account for precisely and result in UNKNOWN being returned. Integrating alias analysis such as [3, 4, 23] to resolve these few remaining indirect jumps into our infrastructure remains future work.

6 Limitations and Future Work

While we are currently able to accurately build signatures for many programs. However, there are several things we do not address, and leave as future work:

- We currently only consider sequential execution. We do not currently model asynchronous events such as threads executing in the same address space, callbacks invoked asynchronously by the operating system, etc. As a result, we do not focus on vulnerabilities arising from asynchronous events such as deadlock. Generating accurate signatures for such code remains an open problem.
- We only model code that is present statically in the given program, *i.e.*, we do not model dynamically generated code. VSM signatures generated from programs that dynamically generate code will still be correct, but will return UNKNOWN if and when control transfers to the dynamically generated code.
- Applications which rely heavily on modifying external state, e.g., databases, pose unique challenges. For example, how should we model external state updates without duplicating the side effects? We

leave this as future work.

- We manually generate stub code at this time. In many cases we could automatically generate stub code by analyzing the actual library binary, or provide more assistance to the user who writes the stubs. Note that the stubs only need to be written once; e.g., in our experiments we used the same stubs for all vulnerable applications.
- We currently generate signatures that determine whether an input will exploit a vulnerable program in the context of a particular external state. Another interesting problem is to determine whether *an external state exists* such that a given input will exploit a vulnerable program. It may be possible to answer this question using our generated VSM signatures by using model-checking techniques to execute on a symbolic external state.

7 Related Work

Signature Generation Several automatic signature generation mechanisms have been proposed which take several examples of an exploit, and extract common patterns to use as a signature [11, 12, 14, 16, 22]. While these work well for some attacks, they may be defeated by polymorphism [8, 16] or by direct attacks against the algorithms [17, 19]. Hamsa [14] provides a false negative bound over the exploit inputs in its training data, but not over the program vulnerability itself.

Other approaches select which patterns to use in the signature by applying some heuristics to available semantic information [15, 18, 28], which helps reduce the risk of false negatives.

Several researchers have proposed signatures that model a vulnerability to provide accuracy guarantees [5, 6, 26]. These approaches have either required manually specifying protocols [26], handled only a single code path [5, 7], or were more theoretical and did not establish how to distinguish SAFE and EXPLOIT in practice [5, 6].

Binary Analysis We perform binary analysis of executables to create signatures. In particular, we assume the binary can be disassembled correctly and, at a minimum, can create a control flow graph. Previous work has shown these assumptions to be reasonable [13]. We use a variant of traditional program slicing techniques [27], which would benefit from data dependency analysis. Dependency analysis for assembly in our experience needs good memory access analysis to be useful. In future work, we plan on incorporating memory access analysis such as in [3, 4, 23].

8 Conclusion

We presented the first practical approach for creating vulnerability signatures which capture multiple program paths an exploit may take. Our signatures are guaranteed to be error-free, i.e., when the signature returns SAFE, the input is safe, when the signature returns EXPLOIT, the input is a real exploit. We also show that adding a third state, UNKNOWN, is useful for making vulnerability signatures practical for real programs. Previous techniques which covered multiple paths were of theoretic interest, and did not guarantee even basic properties like termination for SAFE inputs. We also implemented our techniques, and measured the performance of real end-to-end studies for vulnerable programs, as well as performed larger-scale macro-benchmarks. Our experiments indicate our signatures are efficient to create (creation takes about a second) and evaluate, can recognize many different exploits, even those which would take different code paths to reach the original vulnerability.

References

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

[2] Anonymous. Entry removed for submission anonymity.

[3] G Balakrishnan and T Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23. Springer-Verilag, 2004.

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

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

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

[7] Manuel Cost, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In 20^{th} *ACM Symposium on Operating System Principles (SOSP 2005)*, 2005.

[8] Jedidiah Crandall, Zhendong Su, S. Felix Wu, and Frederic 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.

[9] dong-h0un U. Passlog daemon sl_parse remote buffer overflow vulnerability. `http://www.securityfocus.com` Bugtraq ID 7261.

[10] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to automata theory, langauges, and computation*. Addison-Wesley, 2001.

[11] Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[12] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[13] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

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

[15] Zhenkai 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.

[16] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[17] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *Proceedings of the* 9^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

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

[19] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symposium on Security and Privacy*, may 2006.

[20] r code. Atphttpd remote get request buffer overrun vulnerability. `http://www.securityfocus.com` Bugtraq ID 8709.

[21] Yann Ramin. Atphttpd. `ttp://www.redsift.com/~yramin/atp/atphttpd/`.

[22] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[23] S.K.Debray, R Muth, and M Weippert. Alias analysis of executable code. In *Proceedings of the 1988 Principles of Programming Languages Conference (POPL)*, pages 12–24, 1988.

[24] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *Proceedings of 2007 EuroSys Conference*, 2007. `http://www.cs.cmu.edu/~dbrumley/`.

[25] `pyramid-rp\@hushmail.com`. ghttpd log() function buffer overflow vulnerability. `http://www.securityfocus.com` Bugtraq ID 5960.

[26] Helen J Wang, Chuanxiong Guo, Daniel Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proc. of the 2004 ACM SIGCOMM Conference*, August 2004.

[27] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25:446–452, 1982.

[28] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proc. of the* 12^{th} *ACM Conference on Computer and Communication Security (CCS)*, 2005.