Carnegie Mellon University
Research Showcase @ CMU

Department of Electrical and Computer Engineering

Carnegie Institute of Technology

2007

Creating Vulnerability Signatures Using Weakest Preconditions

David Brumley
Carnegie Mellon University

Hao Wang
University of Wisconsin Colleges

Somesh Somesh
University of Wisconsin Colleges

Dawn Song
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/ece

Part of the Electrical and Computer Engineering Commons

Recommended Citation
.

This Conference Proceeding is brought to you for free and open access by the Carnegie Institute of Technology at Research Showcase @ CMU. It has been accepted for inclusion in Department of Electrical and Computer Engineering by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

Creating Vulnerability Signatures Using Weakest Preconditions

David Brumley dbrumley@cs.cmu.edu
Hao Wang hbwang@cs.wisc.edu
Somesh Jha jha@cs.wisc.edu
Dawn Song dawnsong@cmu.edu

Abstract

Signature-based tools such as network intrusion detection systems are widely used to protect critical systems. Automatic signature generation techniques are needed to enable these tools due to the speed at which new vulnerabilities are discovered. In particular, we need automatic techniques which generate sound signatures — signatures which will not mistakenly block legitimate traffic or raise false alarms. In addition, we need signatures to have few false negatives and will catch many different exploit variants.

We investigate new techniques for automatically generating sound vulnerability signatures with fewer false negatives than previous research using program binary analysis. The key problem to reducing false negatives is to consider as many as possible different program paths an exploit may take. Previous work considered each possible program path an exploit may take separately, thus generating signatures that are exponential in the size of the number of branches considered. In the exact same scenario, we show how to reduce the overall signature size and the generation time from exponential to polynomial. We do this without requiring any additional assumptions, or relaxing any properties. This efficiency gain allows us to consider many more program paths, which results in reducing the false negatives of generated signatures. We achieve these results by creating algorithms for generating vulnerability signatures that are based on computing weakest preconditions (WP). The weakest precondition for a program path to a vulnerability is a function which matches all exploits that may exploit the vulnerability along that path.

We have implemented our techniques and generated signatures for several binary programs. Our results demonstrate that our WP-based algorithm generates more succinct signatures than previous approaches which were based on forward symbolic execution.

1 Introduction

A *vulnerability* is a software bug that can be used by an attacker to alter execution of a program to achieve harmful consequences, such as executing malicious code specified by the attacker. For a given program with a vulnerability, an *exploit* is an input to the program that triggers the vulnerability and results in a successful attack. One common approach for protecting a vulnerable program from being exploited by attackers is to generate signatures which will recognize exploits of the vulnerability. Signatures are widely used in network and host defense systems to filter out any input which may exploit programs.

Since manual generation of signatures is a slow, cumbersome, and error-prone process, there is great interest in automatic signature generation techniques. Previous algorithms for automatically generating signatures have focused on learning signatures from actual exploits [26, 27, 30, 33, 43, 50]. We call these *exploit-based signatures*. The main shortcoming of these exploit-based signatures is that they are based on specific exploit instances and may have both false positives and negatives [16, 34, 40]. For example, an exploit-based signature may have a high false-positive rate, and thus may block a large amount of legitimate traffic. Worse, it is usually impossible to know before deploying an exploit-based signature what the error rate is, thus the only choices for a user are bad: deploy the exploit-based signature and possibly block important legitimate traffic, or not deploy the exploit-based signature and possibly get compromised.

To remedy the shortcomings of exploit-based signatures, recently researchers have developed a new type of signature: *vulnerability signatures*, which are based on actual vulnerabilities instead of exploit instances, and will have a guaranteed zero false positive rate [12, 49]. Specifically, previous work demonstrated that automatically generating sound vulnerability signatures is possible via program binary analysis [12]. This approach uses forward symbolic execution to generate a separate signature for each program path an exploit may take through the program. Loops and other cyclic structures are explored (unrolled) a fixed num-

ber of times. The generated signature is sound, but exponential in size to the number of program paths in the unrolled program. As a result, this approach for generating vulnerability signatures is not scalable when there are many paths an exploit may take.

In this paper, we address the shortcomings of previous approaches and present an efficient and practical method for creating vulnerability signatures based upon binary program analysis. In particular, we explore how to generate vulnerability signatures with the same properties as previous research, but *reduce the overall signature size and generation time from exponential to quadratic* in the size of the unrolled program (see Section 2 for exact details). We do this without requiring any additional assumptions, or relaxing any properties. Intuitively, a signature which recognizes all exploits of a vulnerability should not be much larger than the program since the program itself accepts all exploits, and just needs to be modified to recognize them appropriately. Our approach is able to achieve these gains by more efficiently representing the many program paths an exploit may take. At a high level, our approach summarizes multiple program paths, while previous approaches enumerated them. Because our approach is more scalable, the signature can encompass more program paths, thus be more complete (i.e., have fewer false negatives).

At a high level, our vulnerability signature generation algorithm takes as input a program and a vulnerability, and first outputs a signature formula (f), which when evaluated on an input ($f(x)$) returns EXPLOIT if the input would exploit the program, and SAFE otherwise. If desired, the formula can be given to a solver such as a decision procedure to generate a regular expression signature.

We realize these gains by adopting the formal verification technique of *weakest preconditions (WP)* to signature generation. The novelty of our approach is that, by applying the weakest precondition to generating vulnerability signatures, we can efficiently characterize all vulnerable states in the unrolled program and produce signatures that are more succinct. In addition, unlike previous work, we can consider the case when we know program loop invariants. This is important since often loop invariants can be automatically generated (e.g., [22, 25, 38, 45]).

There are many challenges to adapting the weakest precondition computation to vulnerability signature generation. First, we generate signatures with only access to the program binary. Weakest precondition calculations are normally done on structured programs (i.e., source code), while a binary is unstructured and has only jumps. Second, the method used for calculating the weakest precondition matters: using the standard techniques will generate exponential size formulas, thus little benefit. We show how to adapt recent advances in weakest precondition research to binary analysis, and provide a succinct proof of correctness. This adaptation is necessary to achieve our results. The overall difference using our approach is quite astounding: we reduce vulnerability-signature size from previously best exponential to quadratic (Section 2).

In summary, this paper makes the following contributions:

- We show how to adapt the weakest precondition to create a vulnerability signature. This connection allows us to realize immense efficiency gains. These efficiency gains can be translated into better signatures with fewer false negatives, while still enjoying zero false positives, compared to signatures produced using techniques from previous approaches.
- We develop new methods for calculating the weakest precondition for binary programs, which are unstructured. Our method uses novel structural analysis to convert binaries into structured forms to facilitate weakest precondition computation. We also present a new proof that the resulting signature size is $O(n^2)$ where n is the number of instructions (for programs with loops unrolled).
- We have implemented a prototype system for automatically creating vulnerability signatures from program binaries to evaluate our approach. Our evaluation shows that previous methods generated signatures orders of magnitude larger than our approach.

2 Overview and Intuitions

In this section, we first introduce our terminology for vulnerabilities and vulnerability signatures, and present a running example. Then we give the high level intuition behind our approach.

2.1 Vulnerability Definitions

Given a binary program P, a *vulnerability* is a bug in P where execution may "go wrong" and violate the intended semantics of the program. Common ways to "go wrong" are to dereference NULL pointers, overwrite critical data such as return addresses, and attempt to double-free memory. We call the point where execution may "go wrong" the *vulnerability point*, denoted by i_p. We call the conditions necessary for the program to "go wrong" at i_p the *vulnerability condition*, denoted as c. At a high level, the tuple $< P, i_p, c >$ completely describes exactly one vulnerability. In this work, we are concerned with a single vulnerability (multiple vulnerabilities are each handled independently). An *exploit* for a vulnerability is one input to the program (e.g., a network packet) which causes the program to satisfy the vulnerability condition at the vulnerability point, thus exploiting the vulnerability.

```
1  int buf[15];
2  if(x < 0)
3      x := (-x) % 16;
4  else
5      x := x % 16;
6  buf[x] := x;
```

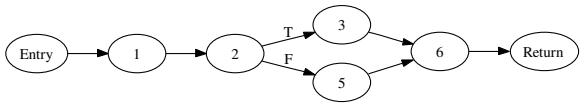

Figure 1. Our running example. The left side shows our example program. When $x = 15$ or -15, line 6 will write past the array bounds. The right side shows the corresponding control flow graph, where each line of the program corresponds to a basic block node.

Figure 1 shows our running example, which contains a off-by-one vulnerability. We show the vulnerability as source code for illustrative purposes: our algorithm and techniques are specifically designed to work on binary programs. In this example, x is the input and $x = 15$ or $x = -15$ will result in an out-of-bounds write on line 6. Thus, the vulnerability condition is $x \neq 15$ at the vulnerability point is line 6.

We generate signatures for known vulnerabilities, and take as input the description of the vulnerability and the vulnerable program. Note how the vulnerability is originally detected is orthogonal to our problem, and addressed by other research, which we do not duplicate here. An illustrative scenario in which the vulnerability description is automatically provided is: (1) a new vulnerability is discovered, (2) an attacker releases an exploit, (3) a detector detects the new exploit, and (4) the detector furnishes our algorithm with the vulnerability point and the detected vulnerability condition. For example, dynamic taint analysis can provide us with the required information about the most common types of vulnerabilities, including buffer overflows, heap overflows, format string bugs, and similar overwrite attacks [17, 35, 44].

2.2 Automatic Vulnerability Signature Generation from Binaries

Our job, given the vulnerability condition c, vulnerability point i_p, and the program binary P, is to generate a *vulnerability signature* $\mathcal{S}_{<P,c>}$ that will recognize subsequent exploits of the vulnerability (without needing to run the program itself). Abstractly, a signature is a Boolean function $\mathcal{S}_{<P,c>}$ which takes as input any input x from the program input domain $\mathcal{I}$, and returns either EXPLOIT or SAFE, i.e., generate: $\mathcal{S}_{<P,c>} : \mathcal{I} \rightarrow \{\text{EXPLOIT, SAFE}\}$. Common signature types include regular expressions [8, 14, 21, 36, 39, 46,47], protocol state machines [49], and Boolean functions which are evaluated on program inputs [12, 15]. In our setting, we focus primarily on generating signatures which are Boolean functions (also called symbolic signatures), though in Section 3.3.3 we explain how a Boolean function signature can be converted to a regular expression signature.

In this paper, we investigate how to generate sound vulnerability signature by binary program analysis. Binary program analysis is very challenging. First, binary programs are unstructured, while most techniques are designed to work for structured programs. Second, binary programs are often much larger than the corresponding source code since a single source code construct can stand for very complicated operations. Third, because our system deals with x86 binaries, it is challenging to accurately model the complexity of x86 instruction set.

However, the benefits of generating a signature from the binary are quite compelling: we can automatically generate signatures for vulnerable programs even when the source code is not available, and the generated signature is completely faithful to the program. Techniques based upon source code offer neither advantage, especially since the source code may not reflect the underlying vulnerability in the binary [5].

2.3 The High-Level Idea

Our approach calculates the *weakest precondition* on inputs to a binary program to exploit the vulnerability $< P, i_p, c >$. At a high level, we think of a vulnerability in terms of the vulnerability point i_p and the vulnerability condition c. Any possible exploit must reach the vulnerability point i_p and satisfy the vulnerability condition c. An exploit must also execute some instruction i_{p-1} just before i_p. The weakest precondition calculates a formula which is true iff executing i_{p-1} will lead to execute i_p and satisfy c. In this manner we recursively calculate a signature. This process is iterated until we reach the initial point of input(s) to the program. The net result of this process is depicted in Figure 2. Inductively, the generated formula will be true for inputs which exploit the given vulnerability.

More formally, let $P : \mathcal{I} \rightarrow \mathcal{I}$ be a program from states $\in \mathcal{I}$ to states $\in \mathcal{O}$ (e.g., a program with n variables is an n-dimensional state-space $\mathcal{I}$). The weakest precondition $wp(P, c)$ for a program P with respect to the vul-

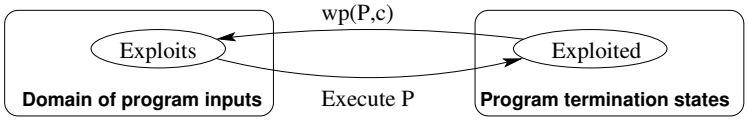

Figure 2. For a program P and a vulnerability c, the weakest precondition $wp(P, c)$ describes the inputs which, upon execution, result in an exploited state.

nerability condition c is a Boolean formula over the initial state which is true for all inputs which cause P to terminate in a final state satisfying c. Thus, if P is a sequence of instructions $\{i_0; i_1; ..., i_p\}$, where i_p is our vulnerability point, inductively we calculate $wp(i_p, c) = c_{p-1}$, then $wp(i_{p-1}, c_{p-1}) = c_{p-2}$, and so on. The resulting formula $\mathcal{S}_{<P,c>} = wp(P, c)$ is a predicate such that $\forall x \in \mathcal{I}$: $\mathcal{S}_{<P,c>}(x) =$ EXPLOIT if $P(x)$ would exploit the vulnerability.

Creating a compact representation for multiple paths an exploit may take is key for creating a succinct vulnerability signature. For instance, in a loop free program with b branches there are $O(2^b)$ program paths. Previous approach based on forward symbolic execution that calculates a separate formula for each path results in an exponential size formula. For example, suppose our signature is calculated over a loop-free program with n instructions and b conditional jumps. Then using forward symbolic execution will generate 2^b signatures—one for each branch—and the total signature $f = f_1 \wedge f_2 \wedge ...f_{2^b}$ will be $O(2^b)$ in size [12]. However, this is not a lower bound. Intuitively, the program itself is almost a signature since it accepts all exploits to i_p—it only fails because it does not output EXPLOIT at the vulnerability point. Thus, signatures that are not much larger than the program itself seem possible.

We show that using the weakest precondition, we can in fact generate a vulnerability signature at most $O(n^2)$ in size under the same set of assumptions, regardless of the number of (acyclic) branches, while previous work was exponential (Section 3.3). Further, the time to create this signature is polynomial in the size of the program, while previous approaches are exponential. We achieve this efficiency because the weakest precondition summarizes multiple program paths to the vulnerability point. In comparison, previous work is based on forward symbolic execution which essentially forks on each conditional jump separately, hence produces exponential growth in signature size and time by considering each path independently. By using weakest precondition, we need not consider each path independently: we can efficiently calculate a signature based on the number of statements.

However, it is not straightforward to apply the weakest precondition to the problem of generating vulnerability signatures. Weakest precondition calculations are usually performed over a structured programming language, while we are working with unstructured binaries. We show how to overcome this challenge by using Guarded Command Language (GCL) (Section 3.2). To the best of our knowledge, we are the first to perform a weakest precondition calculation over real binary programs.

The main advantages of our approach are: (a) we can more efficiently reason about the many paths that lead to the vulnerability point, and (b) the generated signatures are more concise. We do not require any additional assumption, such as source code, and we do not lose any accuracy. Therefore, our work is the first practical approach to creating sound vulnerability signatures which cover many different paths to a vulnerability.

2.4 Automatically Generating Signatures for Programs with Loops

Because automatic analysis of any non-trivial property in programs with loops in undecidable [24], we consider two cases: one for generating signatures on programs with loops, and one generating signatures on loop-free programs. Note previous work only considered analyzing loop-free programs. In the loop-free case, we can always perform completely automatic analysis. In the presence of loops, we may need help from the user by providing loop invariants.

When completely automatic analysis is necessary, we take as input, in addition to the program, the vulnerability condition, and the vulnerability point, a upper bound on the number of times to unroll loops. For example, the upper bound may be the same as the number of times a loop is executed in a known exploit sample. This is the same scenario as previously proposed for signature generation [12], and is common for automatic bound checking [9]. Loops (and other recursive elements) are analyzed up to the specified number of times.

To reason about loops, our algorithm requires, in addition to the vulnerability inputs, a set of loop invariants. Loop invariants are necessary because otherwise even the most basic analysis, halting, is undecidable. Note that although we take loop invariants as input, in many cases they can also automatically be generated (Section 3.4). One advantage of our weakest precondition approach is the semantics are well-defined even for programs with loops.

3 Calculating a Vulnerability Signature Using the Weakest Preconditions

In this section we formalize the process of computing a vulnerability signature via the weakest precondition. We take as input a binary program P, the vulnerability condition c, the vulnerability point i_p, calculate the weakest precondition $\mathcal{S}_{<P,c>} = wp(P, c)$ with respect to the vulnerability point, and output $\mathcal{S}_{<P,c>}$ as our signature. Our approach has three steps:

1. **Pre-process the program.** We convert a binary program P to an intermediate representation (IR), and create the control flow graph for the program. In this step, we also create a control flow graph G of P, and then compute a chop G' of G which contains only the part of the program relevant to the vulnerability. If completely automatic analysis is desired, we also unroll loops in G'. The output of this step is the (chopped and possibly unrolled) control flow graph G'.
2. **Convert to GCL.** This step takes as input the CFG G' and the program P and outputs an equivalent program P_g in the guarded command language (GCL). This step is crucial to our approach because the weakest precondition is designed for structured programs (i.e., source code), while our work deals with binaries, which are unstructured. By converting a binary into GCL form, we abstract the program into a structured form on which we can then apply analysis and compute the weakest precondition.
3. **Compute the weakest precondition.** We compute the weakest precondition $\mathcal{S}_{<P,c>} = wp(P_g, c)$ over the GCL in a syntax directed manner. $\mathcal{S}_{<P,c>}$ is our signature, which can further be refined. For example, we can remove any non-input variables (Section 3.2) or convert it to a regular expression (Section 3.3.3).

The weakest precondition $\mathcal{S}_{<P,c>}$ is our signature because $\mathcal{S}_{<P,c>}(x)$ is true iff $P(x)$ would exploit the vulnerability. In this section, we detail the above steps.

3.1 Step 1: Pre-processing the Program

Translating x86 to the IR. To compute a vulnerability signature on an x86 program binary, we must know how to accurately model any instruction that could be executed. This is a challenging task: x86 is a CISC instruction set with hundreds of instructions, many with implicit side-effects (e.g., overflow in addition sets to `eflags` register), an instruction may behave differently with different operands (e.g., shifting by 0 does not set the `eflags` register, while any other value will), and there are even single instruction loops (e.g., the `rep` family of instructions).

Therefore, our first step is to translate x86 instructions into a form amenable to analysis. In our approach, we convert x86 instructions into the RISC-like IR shown in Table 1. System calls can be translated as assignments to special variables, e.g., the `read` call creates new variables for new inputs (see Section 3.3.2 for further discussion). A vulnerability signature calculated over the IR language is semantically equivalent to a signature over the original x86 instructions.

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

Creating a control flow graph and compute the chop. We next build a control flow graph (CFG) over the IR form of the program. In this step, we also compute a chop of the graph which includes only those program paths which may reach the vulnerability point i_p. When completely automatic analysis is desired, we also unroll loops as necessary.

A CFG is a directed graph $G = (V, E)$ where $v_i \in V$ represents an instruction i, and has edge $(v_{i_1}, v_{i_2}) \in E$ if control can pass from instruction i_1 to i_2. There is a distinguish node called v_{entry} which the unique entry point for the program, and v_{exit} which represents the program terminating. Control flow is straight-forward for all statements except indirect jumps, where the CFG has an edge for any potential successor blocks. The possible targets of an indirect jump can be found with assembly register value analysis [4, 11].

Because in the remaining steps we will reason about all program statements in the output graph from this step, we wish the graph to reflect only those paths which can reach the vulnerability point. Let v_p be the vertex for our vulnerability point i_p. We compute the chop of the graph from v_{entry} to v_p. The chop is computed by (a) adding an edge (v_p, v_{entry}) to the graph; (b) computing the strongly connected component; (c) creating an output graph G' which contains only those nodes in the same component as i_p.

Unrolling Loops. As previously mentioned, automatic reasoning about any non-trivial property in a program with loops is undecidable [24]. Therefore, in the worst case, completely automatic analysis requires us to remove any loops from the graph by unrolling them, i.e., duplicating the loop body a fixed number of times. In some cases we can infer the number of times to an unroll a loop by static analysis, e.g., by analyzing loop induction variables [3,31]. If we cannot automatically infer the maximum number of times a loop may be executed, we may ask the user to provide

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

Table 1. Our RISC-like IR. We convert all x86 assembly instructions into this IR.

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

Table 2. The guarded command language (GCL) fragment we use.

a upper bounds on the number of times each loop can be executed. Unrolling is generally well accepted technique, especially by the model checking community [9]. We discuss scenarios in which loops do not need to be unrolled further in Section 3.4.

For example, if the user specifies the loop `while (a[i] != NULL)`{`i++;`} should be be unrolled 1 time, we generate:

```
if(a[i] != NULL){
  i++;
  if(a[i] != NULL) assert(false); }
```

The `assert(false)` is necessary so that we do not mistakenly reason about additional iterations in the remaining steps.

3.2 Step 2: Calculating the GCL

The weakest precondition is calculated over the guarded command language (GCL), shown in Table 2 (we show only the fragment relevant to our work). Although this language looks simple, it powerful enough to reason about general purpose programming languages [18, 19]. A program written in GCL may either terminate normally, or it may "go wrong". Statements s in the language are assignments of expressions to l-values (e.g., registers and memory cells), "**assert** e" which checks that expression e is true and fails if it is false, "**assume** e" which adds an assumption that e is true, sequences of statements, and the choice statement "$s_1 \Box s_2$" which executes either s_1 or s_2. A program written in GCL terminates normally iff none of the assertions fail.

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

In order to work on binaries, we develop an algorithm which converts an unstructured binary program into the GCL. Our algorithm is a type of structural analysis where we create an appropriate GCL based upon the structure of the CFG. The intuition behind our approach is that although a program may use jumps, their behavior can still be replicated with the GCL. For example, a diamond-shaped subgraph in a CFG, such as with nodes 2,3,5,and 6 in Figure 1, represent an `if-then-else` choice between two branches, and are translated into the GCL as choice statements.

Algorithm 1 Algorithm to calculate a GCL in terms of the CFG.

1: Γ: $V \rightarrow S$ //Mapping from vertex to CFG GCL
2: $pred$: $V \rightarrow V$ // Map vertex to predecessor set
3: **for all** v in topological order **do**
4: **if** $|pred(v)| = 1$ **then**
5: Γ[v] := $\Gamma(pred(v))$;v
6: **else**
7: $s := p_i \in pred(v)$: $\Gamma(p_i)$
8: **for all** $p_i \in pred(v)$ **do**
9: (prefix, v_b, s_1, s_2) := split_prefix($\Gamma(p_i)$, s)
10: s := prefix; (**assume** v_b; s_1) $\Box$ (**assume** $\neg v_b$; s_2);
11: **end for**
12: Γ[v] := s;v
13: **end if**
14: **end for**

For now, we assume the graph is acyclic. We discuss cyclic graphs in Section 3.4. Algorithm 1 works by constructing a GCL program in terms of the CFG vertices. A post-processing phase converts the vertex numbers in the GCL program to actual GCL statements. We maintain two maps: Γ, which maps a vertex to the GCL statement up to that point in the program, and $pred$, which maps a vertex to its predecessor set. Γ is initialized so that each vertex v maps to GCL statement corresponding to executing v.

Our structural analysis distinguishes whether a vertex v

has one or many predecessors. If there is only one predecessor v_i for a vertex v, then the instructions for these two vertices must be a sequence, thus we generate the GCL program $v_i; v$, as shown on line 5.

If there is more than one predecessor, say v_i and v_j, then because the graph is rooted at v_{entry}, there must be a least common predecessor v_b to both v_i and v_j. The function `split_prefix` takes in two GCL programs, and outputs the largest common prefix `prefix`, the least common predecessor v_b, and the remaining statements as s_1 and s_2 for v_i and v_j, respectively. The least common predecessor reflects a choice between two branches: one which goes through v_i and one which goes through v_j. This intuition is reflected in lines 7-12. For a vertex v, we first set our current GCL statement s to be one path to v on line 7. Then, for each predecessor, we calculate the greatest common path prefix `prefix`, the choice point v_b and the two GCL program for the two paths after the choice point s_1 and s_2.

For the CFG of our running example (Figure 1), our algorithm would generate the GCL program:

$$1; (\textbf{assume } 2; 3;) \,\square\, (\textbf{assume } \neg\, 2; 5;); 6;$$

An extended example of this algorithm is provided in the Appendix A.2.

The output of Algorithm 1 is a GCL program P_g where each atomic statement is a CFG vertex ID. The GCL program P_g corresponding to the vulnerability point will be that for all paths from entry to v_p. We then post-process P_g GCL program, replacing statement ID's with the corresponding instruction, and output the results. The final GCL program P_g produced for our running example is:

$$\text{skip}; (\textbf{assume } x < 0; x := (-x)\%16)\square$$
$$(\textbf{assume } \neg(x < 0); x := x\%16); \text{buf}[x] := x;$$

Analysis. The running time of the algorithm includes a topological sort, which can be done in $O(|V| + |E|)$. Because each node is visited at most once, the running time is linear in the size of the graph.

To see correctness, note the important case to consider is when we split a common prefix and generate a choice ($\square$) statement. This happens when two nodes i_j and i_k have a common prefix $i_1, ..., i_b$, then all paths from the entry of the CFG to i_j and i_k most go through i_b, i.e., i_b dominates i_j and i_k. Because i_b is the least common predecessor, i_b must be a branch point. Therefore, we create the GCL program $i_1, ..., i_{b-1}; (\textbf{assume } i_b; i_j \square \textbf{assume } \neg i_b; i_k)$, indicating that any path must first go through i_b, then there is a choice on whether the follow the path to i_j or i_k. Once we find such a structure, we can collapse it into the mentioned GCL program, and iterate. Eventually, all nodes (reachable from the entry) will be collapsed.

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

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

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

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

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

Table 3. Algorithm for calculating the weakest pre-condition.

3.3 Generating Vulnerability Signatures with the Weakest Precondition

In this section, we first describe how to efficiently calculate the weakest precondition. We then describe its application to vulnerability signature generation.

3.3.1 Calculating the Weakest Precondition

The weakest precondition $wp(P, c)$ is a Boolean function which is true iff for all assignments of values to variables x in which $P(x)$ halts in a state satisfying c. The weakest precondition is calculated in a syntax-directed manner from the guarded command language (GCL). Dijkstra's proposed the rules shown in Table 3 for calculating the weakest precondition $wp(P, Q)$. In this section, we use Q to refer to any Boolean predicate, and c to refer specifically to the vulnerability condition predicate. These rules can be read as an algorithm where the ":" separates the inputs from the outputs. For example, given $wp(s_1 \square s_2, Q)$, we first calculate $wp(s_1, Q)$ to generate Q_1, then compute $wp(s_2, Q)$ to generate Q_2, and the resulting predicate is $Q_1 \wedge Q_2$. We give a small further example in Appendix A.1. However, calculation done using Dijkstra's algorithm may result in a formula exponential in the size of the program. Flanagan and Saxe noticed that there are two reasons for the exponential explosion: assignment statements and duplicating the post-condition on both premises of choice statement [23]. We explain the source of the explosion, propose solutions, and prove the correctness of our approach.

Blowup from assignments To see why assignment statements can cause exponential explosion, consider calculation $wp(b = a + a; c = b + b; d = c + c, d < 5)$. Using Dijkstra's semantics shown in Table 3, we generate the formula

$a + a + a + a + a + a < 5$, i.e., the wp calculation generates an exponential number of a's. Flanagan and Saxe propose a method termed *passification* to remedy this problem where the program is transformed into a semantically equivalent form in which all program variables are assigned only once. This condition is easily met by (acyclic) programs by converting the program into SSA form [1]. Assignments are then replaced with **assume**s, e.g., `x := e` is replaced with the logically equivalent "**assume** x = e". Note that this transformation is correct because each variable in the program is assigned once, therefore we can simply bind the variable "x" to the expression "e" throughout the program. A program that has undergone the above transformation is called passified. Our previous calculation on the passified program would be: $wp($**assume** b $= a + a;$ **assume** c $= b + b;$ **assume** d $= c + c, d < 5)$, and the weakest precondition is: $b \Rightarrow (a + a \Rightarrow c \Rightarrow (b + b \Rightarrow (d \Rightarrow c + c \Rightarrow d < 5)))$. Passification will increase the size of the program quadratically, but in most cases, only linearly [23, 29].

Unnecessary duplication of the post-condition The other source of exponential formula growth is with the WP-CHOICE rule. This rule duplicates the post-condition on each branch, thus the formula size potentially doubles at each branch point. This is very similar to what causes the problem with forward symbolic execution: when a branch is encountered the formula is "forked" and two identical copies continue executing each branch [12]. Leino was the first to realize that the problem could be averted if we calculate slightly differently using the *weakest liberal precondition* (wlp) [23, 29]. The weakest liberal precondition is the weakest condition which guarantees that the post-condition is met if the program terminates, i.e., is the same as the weakest precondition, except the program may not terminate. The inference rules for the weakest liberal precondition are the same as for the weakest precondition except for:

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

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

$$wp(P, Q) \Leftrightarrow wp(P, \text{true}) \wedge wlp(P, Q) \qquad (1)$$

Equation 1 (proposed by Dijkstra in [19]) can be read as is that the weakest precondition for a program to terminate in a state satisfying Q ($wp(P, Q)$) is the same as for *if* the program terminates it satisfies Q (the $wlp(P, Q)$ term) *and* it terminates (the $wp(P, \text{true})$ term).

An essential insight is that passified programs do not change state, i.e., since assignments are removed everything is an expression. Therefore, if a passified program starts in a state satisfying Q and nothing goes wrong ($wlp(P, Q \equiv \text{false})$), it will end in a state satisfying Q. This is expressed in the following identity which is true for all assignment free programs:

$$wlp(P, Q) \Leftrightarrow wlp(P, \text{false}) \vee Q \qquad (2)$$

The reason this identity is important is that the post-condition Q does not appear in the weakest precondition calculation, thus is not duplicated along branches in WP-CHOICE. If during sub-derivation we get a new post-condition for WP-CHOICE that is not a constant (i.e., through WP-SEQ), we just apply the transformation again. As a result, the signature size is at most twice the size of the passified program. Because the passified program is at most quadratic in size, the total signature size with be at most $O(n^2)$.

Putting together equation 1 and equation 2, we get:

$$wp(P, Q) \Leftrightarrow wp(P, \text{true}) \wedge (wlp(P, \text{false}) \vee Q) \qquad (3)$$

Correctness Proof. The argument above holds for assignment-free, acyclic programs, assuming that Equation 2 holds (since Equation 1 follows from the definition of wp and wlp [19]). We prove the following generalization where we can swap the position of any two predicate Q_a and Q_b:

Lemma 3.1 *For all assignment-free acyclic programs P, $\forall Q_a, Q_b | wlp(P, Q_a) \vee Q_b : Q \Leftrightarrow wlp(P, Q_b) \vee Q_a : Q$.*

Equation 2 is a specific case of Lemma 3.1 with $Q = Q_a$ and $Q_b = \text{false}$, i.e., $wlp(P, Q) \equiv wlp(P, Q) \vee \text{false} \Leftrightarrow wlp(P, \text{false}) \vee Q$.

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

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

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

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

[1] A dynamic static assignment (DSA) form is required for programs with loops, which is equivalent to SSA for loop-free programs.

Case: $\mathcal{D} = \dfrac{wlp(s_1, Q_a) \vee Q_b : Q_1 \quad wlp(s_2, Q_a) \vee Q_b : Q_2}{wlp(s_1 \Box s_2, Q_a) \vee Q_b : Q_1 \wedge Q_2}$
where $P = s_1 \Box s_2$ and $Q = Q_1 \wedge Q_2$. We show $wlp(s_1 \Box s_2, Q_b) \vee Q_a : Q_1 \wedge Q_2$.

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

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

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

□

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

3.3.2 Generating Vulnerability Signatures

Our calculation for the weakest precondition takes in the vulnerability condition c, and the GCL version of the program P_g from the previous step, and generates $\mathcal{S}_{<P,c>} = wp(P_g, c)$. The output signature $\mathcal{S}_{<P,c>}$ is true for input x iff $P(x)$ would exploit the vulnerability. Our algorithm is shown in Figure 3.

INPUT: A (chopped) program P_g and a vulnerability condition c. Let $Var(P)$ and $I(P)$ be the set of variables and input variables in program P_g, respectively.

1. Compute the weakest precondition $\mathcal{S}_{<P,c>} = wp(P_g, c)$.
2. Eliminate non-input variables ($Var(P) - I(P)$) from $\mathcal{S}_{<P,c>}$. are called non-input variables.
3. If needed, convert the predicate $\mathcal{S}_{<P,c>}$ to an appropriate signature representation, such as a regular expression signature (Section 3.3.3).

Figure 3. Creating vulnerability signatures from weakest preconditions.

We calculate $wp(P_g, c)$ as in equation 3:

$$\mathcal{S}_{<P,c>} = wp(P_g, \text{true}) \wedge (wlp(P_g, \text{false}) \vee c)$$

The total size of the vulnerability signature is at most $O(n^2)$ (from passification) where n is the number of instructions in P_g.

Let $Var(P)$ and $I(P)$ be the set of variables and input variables in program P, respectively. For programs where all variables can be defined in terms of the input, $wp(P_g, Q)$ can be used as a signature: any input satisfying the resulting predicate will exploit the program. However, the weakest precondition is not quite a signature for programs with non-input variables $Var(\mathcal{P}) - I(\mathcal{P})$. One way to remove such variables is to assign them values, say values as they appear in an exploit sample trace. For example, configuration options are often non-input variables, and thus using the sample trace values may be an appropriate action. If we cannot provide a value for a variable, and we cannot write it in terms of an input variable, then we can existentially quantify them out. Existential quantification may result in an imprecise signature: when the signature is satisfied, there there is some assignment of values to non-input variables that makes the predicate true, but perhaps not the actual values the real program would compute. In our setting, we assume all non-input variables can be provided values, e.g., the same values as provided by the initial detection of the vulnerability. As a result of the elimination step we augment $\mathcal{S}_{<P,c>}$ such that free variables are all input variables.

3.3.3 Converting a Vulnerability Signature to a Regular Expression

The signature computed in the previous step can be converted to a regular expression if desired. The regular expression is calculated from the signature by enumerating satisfying inputs which would cause the signature to return EXPLOIT. Since our algorithm produces one compact formula for the entire vulnerability, we need not enumerate each path independently in order to generate a regular expression [12].

A regular expression signature is calculated from $\mathcal{S}_{<P,c>} = wp(P_g, c)$ via the following algorithm:

1. The initial signature is the empty regular expression $S_{re} = \epsilon$
2. Ask the decision procedure for a satisfying answer to $\mathcal{S}_{<P,c>}$, which will be a set of values $x \in I(P)$
3. Set the signature to $S_{re} := S_{re}|x$ where | is a regular-expression "or"
4. Set $\mathcal{S}_{<P,c>} := \mathcal{S}_{<P,c>} \wedge (\neg x)$
5. Repeat step 1.

Note that the formulas we generate using the optimized weakest pre-condition calculate are not only more compact, but also easier for a decision procedure to reason about [23]. The intuition is that forward execution will duplicate the vulnerability condition c along each path, resulting in formulas of the form (path 1 $\wedge c$) $\vee$ (path 2 $\wedge c$). The decision

procedure will, in many cases, reason about c twice. Our generation technique exposes the commonality in the formula, e.g., $(\text{path } 1 \vee \text{path } 2) \wedge c$, in which case the decision procedure only reasons once about c. This is an additional benefit to our approach.

3.4 Loop Invariants

Previous methods did not reason about loops, even when loop invariants are known; instead they executed loops a fixed number of times, which is equivalent to unrolling. However, our approach using weakest preconditions naturally extends to loops. We can add to the GCL language:

$$\text{s ::= ... } | \textbf{ do } e \Rightarrow s \textbf{ od}$$

in which we execute s repeatedly while e is true.

Our algorithm for constructing the appropriate GCL program given a control flow graph with loops is a straightforward change to Algorithm 1. We identify each loop using standard techniques [3, 31], and calculate the GCL program P for the loop body using Algorithm 1. Note that irreducible graphs can be made reducible [3], and extending the CFG to GCL algorithm to reducible graphs is trivial. We then identify the loop condition e, and create **do** $e \Rightarrow s$ **od**. Note this algorithm assumes the graph is reducible; irreducible graphs can be made reducible via node-splitting [31]. Certain types of "improper regions" in the CFG are not amenable to our analysis [31]. However, standard algorithms are available to identify such regions. Our analysis is still applicable to the remaining "proper" regions. Further, since most binary programs are compiled from a high level language where such odd behavior does not exist, thus it will likely not exist at the binary level.

In order to calculate the weakest precondition of this construct we need to identify a loop invariant [19] which, among other things, is needed because deciding whether a loop terminates is undecidable. Loop invariants can in some cases be automatically generated [22, 25, 38, 45]. In some cases, however, loop invariants may need to be supplied manually. Once the loop invariant is supplied, we can calculate the weakest precondition using standard algorithms, which do to space we do not duplicate here.

4 Implementation and Evaluation

We have implemented a prototype system that performs the steps described in Section 3: we disassemble the binary, translate assembly into the IR, and calculate the weakest precondition formula. Our implementation is written primarily in C++ and OCaml, and consists of about 29,000 lines of code. About 20,000 of those lines are the translation from x86 to the IR. Signatures are generated by converting the WP formula into a C program that returns *true* iff an input string satisfies the calculated weakest precondition. We have also implemented the approach from Section 3.3.3 for converting a WP generated formula to a regular expression. We use CVC-Lite [7] as our decision procedure for generating regular expression signatures. We chose CVC-Lite because it allows us to easily model bit-vector operations and memory operations.

Prototype limitations Our implementation is a prototype geared at understanding the trade-offs for vulnerability signature creation. As a result, we do not perform unnecessary though potentially useful analysis. We currently do not support indirect jumps. The main problem is without additional analysis, we must assume an indirect jump could transfer control to any address. Others have researched resolving possible indirect jump targets, such as the work on Control Flow Integrity [2], value-set analysis [4], and assembly alias analysis [11]. We need not resolve jump targets exactly: our techniques can deal with many possible targets. We manually verify this limitation does not affect our results. We also do not perform alias analysis. Our implementation hands off the problem of discovering aliases to the decision procedure. Providing alias analysis would make it easier to prove formulas [11]. Finally, computing data dependencies would result in a much more refined view of which statements are important, and further reduce the formula size. Therefore, the numbers provided here are an upper bound: subsequent analysis may reduce them further.

Currently, we unroll all loops. In the future, we plan to improve our implementation by incorporating an automated technique to identify loop invariants, and only using loop unrolling as the fall-back mechanism when the automated technique fails to infer the invariants.

4.1 Evaluation

We have evaluated our system using four different vulnerable programs: *iwconfig* [20], *atphttpd* [41], *Bind* [48], and *Samba* [42]. Note atphttpd and bind were part of the evaluation suite for of [12]. Each of these programs contain previously known buffer overflows (our approach works equally well with other types of vulnerabilities). In our experiments, we consider all program paths that may lead from reading an input to the vulnerability point.

Our experiments show:

- The optimized weakest pre-condition formula results in smaller signatures than previously proposed methods. The signature using previous methods for the same number of paths as our wp-based approach wrapped a 64-bit counter, while our signatures were slightly larger than the program itself.

- Creating a signature that covers all program paths to the vulnerability is possible.
- The source of much of the complexity is due to library functions. Providing summary functions for library calls significantly reduces the size of the generated signatures.

4.1.1 WP vs. Forward Symbolic Execution Formula Size

We compare the size of formulas generated via forward symbolic execution against the ones produced using our weakest-precondition formulation from Section 3.3. Previous work uses forward symbolic-execution to create vulnerability signatures [12], where multiple program paths are the logical OR of each individual path. We use the SSA form of the IR in all experiments for consistency.

We first measured the formula size for all paths from when input (e.g., the exploit) is read to the vulnerability point. The formula size is simply a count of the number of terms. We determine viable program paths via control flow analysis as described in [12] where any path from the initial read to the vulnerability point is considered. Note that we do not count the size of the desired post-condition in the formula, which only adds a constant term in both cases.

Table 4 shows the size of the generated signatures for both methods for 4 programs, with and without summary functions for glibc. Our experiments show that the size of the weakest precondition formula is only slightly larger than the number of IR statements, while the forward symbolic-execution signatures were often too big for a 2^{64}-bit counter. The average signature size using weakest pre-condition is about 16% larger than the program without summaries and 15% larger than the program with summaries (i.e., not counting glibc). The forward symbolic execution signature with summaries is about 3.13×10^{11} times larger than the program, while unmeasurable without summaries.

Glibc increases the complexity of formulas. We investigated where the complexity and size of the resulting formula comes from, and found (to our surprise) it was almost completely due to glibc. This phenomena has not been observed in previous work, most likely because only single program paths were considered (e.g., in [12]. This experiment indicates that our WP-based approach allowed us to learn something new about vulnerability signatures.

Table 5 shows sizes of some of the larger glibc functions. The problem is glibc functions have deep callgraphs. For instance, `vsprintf` calls dozens of functions, many of which in turn call `malloc`, which surprisingly calls `vsprintf` recursively. This experiment indicates that providing summary functions, even for only a few glibc functions, can significantly reduce the total signature size.

4.1.2 Final Generated Signature Size

Our tool automatically converts the final WP formula into C code, which can then be compiled and evaluated on inputs. We tested our tool on *atphttpd*, and the tool generated about 4,000 lines of C code for the corresponding WP formulas. The compiled signature takes in an input string, and returns either 1 for exploit, or 0 for safe.

5 Related Work

Brumley *et al.* [12] were the first to propose techniques for automatically create vulnerability signatures for software. They broke down the problem of signature creation into two dimensions: the amount of code covered by the signature, and how the signature is represented. We focus on the problem of how to efficiently cover many vulnerable program paths.

Forward Symbolic Execution. Our approach is likely applicable to other security scenarios which use forward symbolic execution to reason about programs. The EXE project finds security errors deep in programs through mixed forward and symbolic execution [1, 13] of source code. In their project, they instrument the source code such that the symbolic formulas are built up as the code executes. Brumley et al uses mixed execution of binaries to automatically detecting trigger-based behavior such as time-bombs in software [10]. Kruegel uses symbolic execution to automate mimicry attacks [28]. His approach iteratively considers each possible path for a mimicry attack. Since each formula f generated represents an execution down one code path π, it corresponds to the formula $wp(\pi, \text{true})$. Therefore, our technique is applicable and a) may reduce the size of the formulas generated b) make it easier for decision procedures to reason about the formula in each of these settings.

Weakest precondition Dijkstra extensively explored weakest preconditions for formal program verification using the guarded command language [19]. (The guards that gave the language its name have been replaced by **assume** statements.) Flanagan and Saxe were the first to derive techniques for reducing the resulting formula from exponential to quadratic size [23]. They use two auxiliary functions "N.S and "W.S". Leino was the first to spotlight it was the weakest liberal precondition property that made their techniques work [29], eliminating the need for "N.S" and "W.S". Leino proved equation 3 using a logic-based argument. However, we believe our inductive proof is of value

Program	No Summaries			glibc summaries		
	# Stmts	WP	Forward Exec.	# Stmts	WP	Forward Exec.
atphttpd	4.16×10^{11}	4.32×10^{11}	5.75×10^{18}	15834	16316	96658
bind	2.91×10^{6}	3.06×10^{6}	3.27×10^{16}	182196	191220	5.27×10^{13}
iwconfig	1.97×10^{14}	2.13×10^{14}	$> 2^{64}$	45634	46735	7.19×10^{13}
samba	2.78×10^{7}	3.00×10^{7}	$> 2^{64}$	2.40×10^{7}	2.72×10^{7}	7.72×10^{18}

Table 4. Symbolic signature size for full path converge using weakest preconditions and forward symbolic execution as in [12]. We consider both with and without summary functions for common glibc functions.

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

Table 5. Measurements for how much complexity various glibc functions add to the overall signature.

since it closely follows the natural recursive nature of a WP calculation.

Although our primary interest is to generate sound vulnerability signatures, our techniques can be applied to enable classical Hoare-style program verification of binary programs. Others have also explored calculating weakest preconditions on assembly programs. Previous approaches for creating weakest preconditions on assembly programs have tried introducing auxiliary terms [6], explicitly model the control flow counter [37]. Our approach is much more straight-forward: we perform structural analysis to transform the "unstructuredness" of the assembly program into the corresponding form of a structured program. The advantage of our approach is that we do not need to invent new machinery in order to reason about binary programs: the structural analysis reduces reasoning about unstructured programs to the structured case. Our analysis should not fail for acyclic programs, however certain cyclic programs may cause problem. Note to the best of our knowledge previous work also considered only acyclic programs. To the best of our knowledge, our work on sound application dialog replay [32] was the first to compute the WP on binary programs. However, the algorithm implemented in that work was exponential; our work here would bring the formula sizes down in that work to quadratic.

6 Conclusion

We have shown how to formulate the problem of automatically creating a vulnerability signature from a program binary by using weakest preconditions. Our approach required us to develop new algorithms to adapt weakest preconditions to binary programs, which may be of independent interest. The result of our approach is we can reduce the size of a vulnerability signatures from exponential to quadratic in the size of the (acyclic) program, and we eliminate the need to consider each of the exponential paths separately. We also ran experiments using our analysis on real binaries. Our measurements indicate the signatures are orders of magnitude smaller than previous approaches. Thus, our methods are the first practical approach for creating a sound vulnerability signature over multiple program paths.

7 Acknowledgments

The authors thank Ivan Jager, James Newsome, Vyas Sekar, and the anonymous reviewers for their many helpful comments and suggestions while preparing this paper. This material is based upon work partially supported through the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, the International Technology Alliance, the Department of Energy under grant W-7405-ENG-36, and the National Science Foundation under grant numbers CCF-0524051, 0311808, 0433540, 0448452,

and 067511. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*, 2006.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of the ACM Conference on Computer and Communication Security*, 2005.

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

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. Int. Conf. on Compiler Construction*, 2004.

[5] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. In *Proc. IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, 2005.

[6] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, 2005.

[7] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *CAV*, Lecture Notes in Computer Science. Springer, 2004.

[8] S. P. Berry. Shoki intrusion detection system. `shoki.sf.net`.

[9] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 2003.

[10] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Towards automatically identifying trigger-based behavior in malware using symbolic execution and binary analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University School of Computer Science, January 2007.

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

[12] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[13] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. In *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*, 2006.

[14] CISCO. Cisco secure intrusion detection system director - string matching signatures. `http://www.cisco.com/univercd/cc/td/doc/product/iaabu/csids/csids5/csidscog/sigs.htm#wp1015446`.

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

[16] 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.

[17] J. R. Crandall and F. Chong. Minos: Architectural support for software security through control data integrity. In *To appear in International Symposium on Microarchitecture*, December 2004.

[18] D. Detlefs, K. R. M. Leino, G. Nelson, and J. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, December 1998.

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

[20] H. Doli. iwconfig vulnerability (wireless tools v.26). `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0947`, 2003.

[21] Enterasys. Dragon intrusion detection system. `http://www.enterasys.com/products/ids/`.

[22] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM Press, 2002.

[23] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM Symposium on the Principles of Programming Languages (POPL)*, 2001.

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

[25] A. Ireland and J. Stark. the automatic discovery of loop invariants. In *Fourth NASA Langley formal methods workshop*, 1997.

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

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

[28] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, Aug. 2005.

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

[30] 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.

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

[32] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the* 13th *ACM Conference on Computer and and Communications Security (CCS)*, Oct. 2006.

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

[34] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *Rapid Advances in Intrusion Detection (RAID)*, 2006.

[35] J. Newsome and D. 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.

[36] NFR. Network flight recorder intrusion detection system. `http://www.nfr.com`.

[37] T. Norvell. In *Machine code programs are predicates too*, 1994.

[38] L. I. on Demand. K rustan leino and francesco logozzo. In *The Third Asian Symposium on Programming Languages and Systems*, 2005.

[39] V. Paxson. Bro: A system for detecting network intruders in real-time. *computer networks*, 31, December 1999.

[40] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symposium on Security and Privacy*, 2006.

[41] r-code. ATPhttpd exploit. `http://www.cotse.com/mailing-lists/todays/att-0003/01-atphttp0x06.c`.

[42] J. Rafail. Samba contains buffer overflow in smb/cifs packet fragment reassembly code. `http://www.kb.cert.org/vuls/id/298233`, 2003.

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

[44] G. E. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of ASPLOS*, 2004.

[45] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*. ACM Press, 1977.

[46] K. Takeda. Pakemon intrusion detection system. `http://www.inas.mag.keio.ac.jp/ids/pakemon/index.html`. referenced by http://www.whitehats.com/ids/.

[47] The Snort Project. Snort, the open-source network intrusion detection system. http://www.snort.org/.

[48] 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`.

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

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

A Additional Figures

We provide a few more illustrative examples in this appendix.

A.1 Small WP calculation

We give here a sample derivation using the weakest precondition rules from Table 2. Given the program:

$$\textbf{assume } \mathrm{e}; x := y \Box \textbf{assume } \mathrm{e}; x := z \tag{4}$$

We would passified the program, and calculate the weakest precondition with respect $wlp(P, \text{true}) \vee Q$ as shown in Figure 4.

A.2 Extended Example of CFG to GCL

Figure 5 shows an extended example of how Algorithm 1 calculates the GCL over vertices in a graph. Our structural analysis algorithm iteratively considers each node in topological order, essentially collapsing nodes as they are processed. Note confluence points indicate a choice in the graph, at which point the GCL program builds an appropriate choice statement.

$$\frac{\dfrac{wlp(\textbf{assume } \text{x =y}, \text{true}) : (x = y) \Rightarrow \text{true} \quad wlp(\textbf{ assume } \text{e}, (x = y) \Rightarrow \text{true}) : e \Rightarrow ((x = y) \Rightarrow \text{true})}{wlp(\textbf{assume } e; \textbf{assume } \text{x = y}; , \text{true}) : e \Rightarrow ((x = y) \Rightarrow \text{true})} \quad \dfrac{\ldots}{wlp(\textbf{assume } \neg e; \textbf{assume } \text{x = z}; , true) : Q_1}}{wlp(\textbf{assume } e; \textbf{assume } \text{x = y}; \Box \textbf{assume } \neg e; \textbf{assume } \text{x = z}; , \text{true}) \vee Q : (e \Rightarrow ((x = y) \Rightarrow \text{true}) \wedge Q_1) \vee Q}$$

Figure 4. Calculate $wlp(P, \text{true}) \vee Q$ for passified program P = program 4

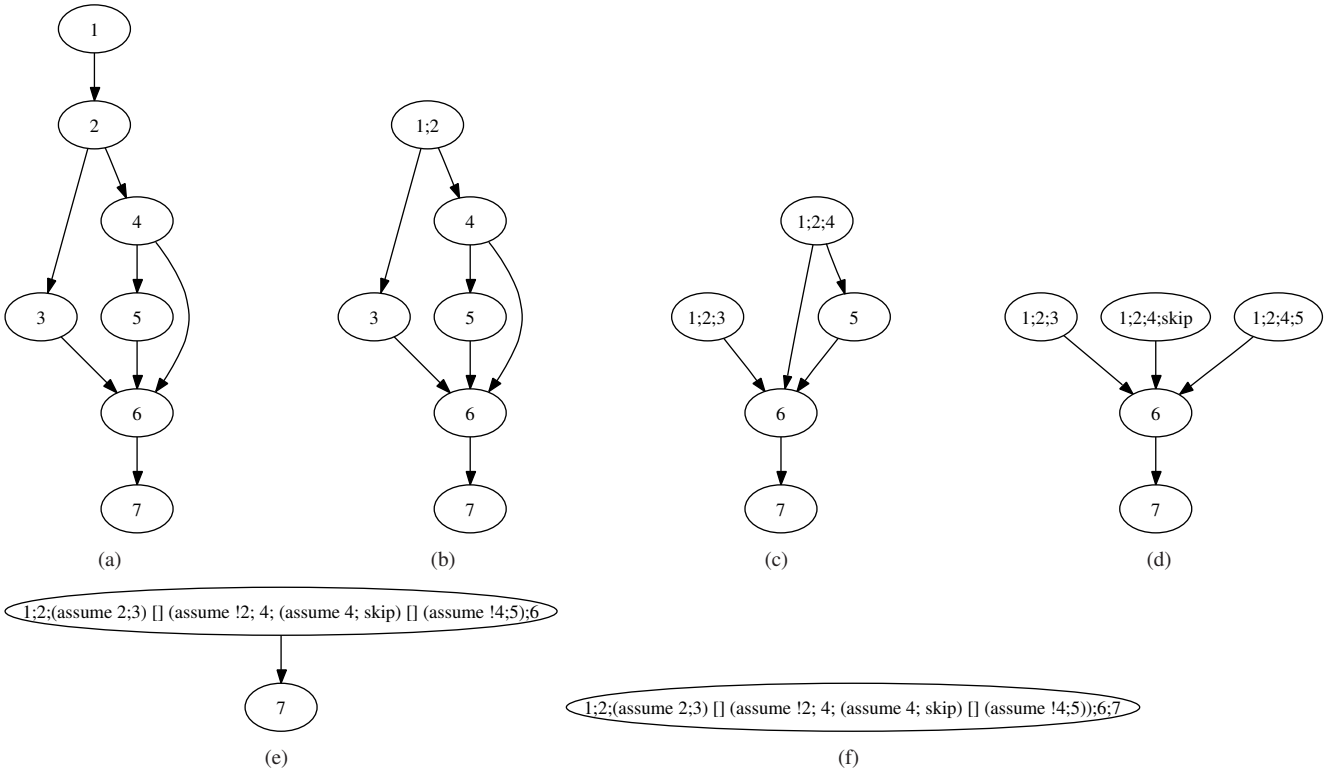

Figure 5. An example of our algorithm for creating the GCL from a CFG. Note that at each step we calculate the current GCL formula at each node in topological order via node collapsing. In the transition from d to e corresponds to lines 7-12 of the algorithm, where we compute a common prefix and update the GCL to correspond to the choice $(\Box)$ between the various branches that could have been taken to reach node 6.