All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)

NOT USELESS

IMPORTANT

CRUCIAL

Edward J. Schwartz, Thanassis Avgerinos, David Brumley
Carnegie Mellon University
Pittsburgh, PA
{edmcman, thanassis, dbrumley}@cmu.edu

***Abstract*—Dynamic taint analysis and forward symbolic execution are quickly becoming staple techniques in security analyses. Example applications of dynamic taint analysis and forward symbolic execution include malware analysis, input filter generation, test case generation, and vulnerability discovery. Despite the widespread usage of these two techniques, there has been little effort to formally define the algorithms and summarize the critical issues that arise when these techniques are used in typical security contexts.**

The contributions of this paper are two-fold. First, we precisely describe the algorithms for dynamic taint analysis and forward symbolic execution as extensions to the run-time semantics of a general language. Second, we highlight important implementation choices, common pitfalls, and considerations when using these techniques in a security context.

***Keywords*-taint analysis, symbolic execution, dynamic analysis**

I. Introduction

Dynamic analysis — the ability to monitor code as it executes — has become a fundamental tool in computer security research. Dynamic analysis is attractive because it allows us to reason about actual executions, and thus can perform precise security analysis based upon run-time information. Further, dynamic analysis is simple: we need only consider facts about a single execution at a time.

Two of the most commonly employed dynamic analysis techniques in security research are dynamic taint analysis and forward symbolic execution. Dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources such as user input. Dynamic forward symbolic execution automatically builds a logical formula describing a program execution path, which reduces the problem of reasoning about the execution to the domain of logic. The two analyses can be used in conjunction to build formulas representing only the parts of an execution that depend upon tainted values.

The number of security applications utilizing these two techniques is enormous. Example security research areas employing either dynamic taint analysis, forward symbolic execution, or a mix of the two, are:

1) **Unknown Vulnerability Detection.** Dynamic taint analysis can look for misuses of user input during an execution. For example, dynamic taint analysis can be used to prevent code injection attacks by monitoring whether user input is executed [22–24, 49, 58].
2) **Automatic Input Filter Generation.** Forward symbolic execution can be used to automatically generate input filters that detect and remove exploits from the input stream [13, 21, 22]. Filters generated in response to actual executions are attractive because they provide strong accuracy guarantees [13].
3) **Malware Analysis.** Taint analysis and forward symbolic execution are used to analyze how information flows through a malware binary [5, 6, 64], explore trigger-based behavior [11, 44], and detect emulators [57].
4) **Test Case Generation.** Taint analysis and forward symbolic execution are used to automatically generate inputs to test programs [16, 18, 35, 56], and can generate inputs that cause two implementations of the same protocol to behave differently [9, 16].

Given the large number and variety of application domains, one would imagine that implementing dynamic taint analysis and forward symbolic execution would be a textbook problem. Unfortunately this is not the case. Previous work has focused on how these techniques can be applied to solve security problems, but has left it as out of scope to give exact algorithms, implementation choices and pitfalls. As a result, researchers seeking to use these techniques often rediscover the same limitations, implementation tricks, and trade-offs.

The goals and contributions of this paper are two-fold. First, we formalize dynamic taint analysis and forward symbolic execution as found in the security domain. Our formalization rests on the intuition that run-time analyses can precisely and naturally be described in terms of the formal run-time semantics of the language. This formalization provides a concise and precise way to define each analysis, and suggests a straightforward implementation. We

program	::=	*stmt**
stmt s	::=	*var* := *exp* | store(*exp*, *exp*) | goto *exp* | assert *exp* | if *exp* then goto *exp* else goto *exp*
exp e	::=	load(*exp*) | *exp* $\Diamond_b$ *exp* | $\Diamond_u$ *exp* | *var* | get_input(src) | v
$\Diamond_b$	::=	typical binary operators
$\Diamond_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Table I: A simple intermediate language (SIMPIL).

then show how our formalization can be used to tease out and describe common implementation details, caveats, and choices as found in various security applications.

II. First Steps: A General Language

A. Overview

A precise definition of dynamic taint analysis or forward symbolic execution must target a specific language. For the purposes of this paper, we use SIMPIL: a **S**imple **I**ntermediate **L**anguage. The grammar of SIMPIL is presented in Table I. Although the language is simple, it is powerful enough to express typical languages as varied as Java [30] and assembly code [7**?**]. Indeed, the language is representative of internal representations used by compilers for a variety of programming languages [1].

A program in our language consists of a sequence of numbered statements. Statements in our language consist of assignments, assertions, jumps, and conditional jumps. Expressions in SIMPIL are side-effect free (i.e., they do not change the program state). We use "$\Diamond_b$" to represent typical binary operators, e.g., you can fill in the box with operators such as addition, subtraction, etc. Similarly, $\Diamond_u$ represents unary operators such as logical negation. The statement get_input(src) returns input from source src. We use a dot ($\cdot$) to denote an argument that is ignored, e.g., we will write get_input($\cdot$) when the exact input source is not relevant. For simplicity, we consider only expressions (constants, variables, etc.) that evaluate to 32-bit integer values; extending the language and rules to additional types is straightforward.

For the sake of simplicity, we omit the type-checking semantics of our language and assume things are well-typed in the obvious way, e.g., that binary operands are integers or variables, not memories, and so on.

B. Operational Semantics

The operational semantics of a language specify unambiguously how to execute a program written in that language.

Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

Figure 2: The meta-syntactic variables used in the execution context.

Because dynamic program analyses are defined in terms of actual program executions, operational semantics also provide a natural way to define a dynamic analysis. However, before we can specify program analyses, we must first define the base operational semantics.

The complete operational semantics for SIMPIL are shown in Figure 1. Each statement rule is of the form:

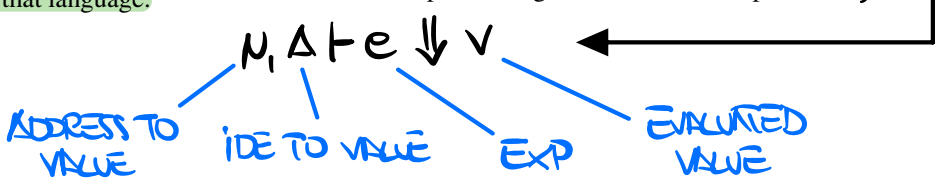
$$\frac{\text{computation}}{\langle\text{current state}\rangle, \text{stmt} \rightsquigarrow \langle\text{end state}\rangle, \text{stmt'}}$$

Rules are read bottom to top, left to right. Given a statement, we pattern-match the statement to find the applicable rule, e.g., given the statement $x := e$, we match to the ASSIGN rule. We then apply the computation given in the top of the rule, and if successful, transition to the end state. If no rule matches (or the computation in the premise fails), then the machine halts abnormally. For instance, jumping to an address not in the domain of Σ would cause abnormal termination.

The execution context is described by five parameters: the list of program statements (Σ), the current memory state (μ), the current value for variables (Δ), the program counter (pc), and the current statement (ι). The Σ, μ, and Δ contexts are maps, e.g., $\Delta[x]$ denotes the current value of variable x. We denote updating a context variable x with value v as $x \leftarrow v$, e.g., $\Delta[x \leftarrow 10]$ denotes setting the value of variable x to the value 10 in context Δ. A summary of the five meta-syntactic variables is shown in Figure 2.

In our evaluation rules, the program context Σ does not change between transitions. The implication is that our operational semantics do not allow programs with dynamically generated code. However, adding support for dynamically generated code is straightforward. We discuss how SIMPIL can be augmented to support dynamically generated code and other higher-level language features in Section II-C.

The evaluation rules for expressions use a similar notation. We denote by $\mu, \Delta \vdash e \Downarrow v$ evaluating an expression e to a value v in the current state given by μ and Δ. The expression e is evaluated by matching e to an expression evaluation rule and performing the attached computation.



$$\frac{v \text{ is input from } src}{\mu, \Delta \vdash \text{get\_input}(src) \Downarrow v}\ \textsc{Input} \qquad \frac{\mu, \Delta \vdash e \Downarrow v_1 \quad v = \mu[v_1]}{\mu, \Delta \vdash \text{load } e \Downarrow v}\ \textsc{Load} \qquad \frac{}{\mu, \Delta \vdash var \Downarrow \Delta[var]}\ \textsc{Var}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v' = \Diamond_u v}{\mu, \Delta \vdash \Diamond_u e \Downarrow v'}\ \textsc{Unop} \qquad \frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \Diamond_b v_2}{\mu, \Delta \vdash e_1 \Diamond_b e_2 \Downarrow v'}\ \textsc{Binop} \qquad \frac{}{\mu, \Delta \vdash v \Downarrow v}\ \textsc{Const}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad \Delta' = \Delta[var \leftarrow v] \quad \iota = \Sigma[pc+1]}{\Sigma, \mu, \Delta, pc, var := e \rightsquigarrow \Sigma, \mu, \Delta', pc+1, \iota}\ \textsc{Assign} \qquad \frac{\mu, \Delta \vdash e \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \Sigma, \mu, \Delta, v_1, \iota}\ \textsc{Goto}$$

$$\frac{\mu, \Delta \vdash e \Downarrow 1 \quad \Delta \vdash e_1 \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu, \Delta, v_1, \iota}\ \textsc{TCond}$$

$$\frac{\mu, \Delta, \vdash e \Downarrow 0 \quad \Delta \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[v_2]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu, \Delta, v_2, \iota}\ \textsc{FCond}$$

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[pc+1] \quad \mu' = \mu[v_1 \leftarrow v_2]}{\Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \rightsquigarrow \Sigma, \mu', \Delta, pc+1, \iota}\ \textsc{Store}$$

$$\frac{\mu, \Delta \vdash e \Downarrow 1 \quad \iota = \Sigma[pc+1]}{\Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \Sigma, \mu, \Delta, pc+1, \iota}\ \textsc{Assert}$$

Figure 1: Operational semantics of SIMPIL.

Most of the evaluation rules break the expression down into simpler expressions, evaluate the subexpressions, and then combine the resulting evaluations.

Example 1. *Consider evaluating the following program:*

```
1  x := 2 * get_input(·)
```

The evaluation for this program is shown in Figure 3 for the input of 20. Notice that since the ASSIGN *rule requires the expression* e *in* var := e *to be evaluated, we had to recurse to other rules (*BINOP, INPUT, CONST*) to evaluate the expression* 2∗*get_input(·) to the value 40.*

C. Language Discussion

We have designed our language to demonstrate the critical aspects of dynamic taint analysis and forward symbolic execution. We do not include some high-level language constructs such as functions or scopes for simplicity and space reasons. This omission does not fundamentally limit the capability of our language or our results. Adding such constructs is straightforward. For example, two approaches are:

1) Compile missing high-level language constructs down to our language. For instance, functions, buffers and user-level abstractions can be compiled down to SIMPIL statements instead of assembly-level instructions. Tools such as BAP [**?**] and BitBlaze [7] already use a variant of SIMPIL to perform analyses. BAP is freely available at http://bap.ece.cmu.edu.

   Example 2. *Function calls in high-level code can be compiled down to* SIMPIL *by storing the return address and transferring control flow. The following code calls and returns from the function at line 9.*

```
1  /* Caller function */
2  esp := esp + 4
3  store(esp, 6) /* retaddr is 6 */
4  goto 9
5  /* The call will return here */
6  halt
7
8  /* Callee function */
9  ...
10 goto load(esp)
```

   We assume this choice throughout the paper since previous dynamic analysis work has already demonstrated that such languages can be used to reason about programs written in any language.
2) Add higher-level constructs to SIMPIL. For instance, it might be useful for our language to provide direct support for functions or dynamically generated code. This could slightly enhance our analyses (e.g., allowing us to reason about function arguments), while requiring only small changes to our semantics and analyses. Figure 4 presents the CALL and RET rules that need to be added to the semantics of SIMPIL to provide support for call-by-value function calls. Note that several new contexts were introduced to support functions, including a stack context (λ) to store return addresses, a scope context (ζ) to store function-local variable contexts and a map from function names to addresses (ϕ).
   In a similar manner we can enhance SIMPIL to support

$$\dfrac{\dfrac{}{\mu, \Delta \vdash 2 \Downarrow 2}\ \textsc{Const} \quad \dfrac{20 \text{ is input}}{\mu, \Delta \vdash \text{get\_input}(\cdot) \Downarrow 20}\ \textsc{Input} \quad v' = 2 * 20}{\mu, \Delta \vdash 2\text{*get\_input}(\cdot) \Downarrow 40}\ \textsc{Binop} \quad \Delta' = \Delta[x \leftarrow 40] \quad \iota = \Sigma[pc+1]}{\Sigma, \mu, \Delta, pc, x := 2\text{*get\_input}(\cdot) \rightsquigarrow \Sigma, \mu, \Delta', pc+1, \iota}\ \textsc{Assign}$$

Figure 3: Evaluation of the program in Listing 1.

dynamically generated code. We redefine the abstract machine transition to allow updates to the program context ($\Sigma \rightsquigarrow \Sigma'$) and provide the rules for adding generated code to Σ. An example GENCODE rule is shown in Figure 4.

III. Dynamic Taint Analysis

The purpose of dynamic taint analysis is to track information flow between sources and sinks. Any program value whose computation depends on data derived from a taint source is considered *tainted* (denoted **T**). Any other value is considered *untainted* (denoted **F**). A *taint policy* P determines exactly how taint flows as a program executes, what sorts of operations introduce new taint, and what checks are performed on tainted values. While the specifics of the taint policy may differ depending upon the taint analysis application, e.g., taint tracking policies for unpacking malware may be different than attack detection, the fundamental concepts stay the same.

Two types of errors can occur in dynamic taint analysis. First, dynamic taint analysis can mark a value as tainted when it is not derived from a taint source. We say that such a value is *overtainted*. For example, in an attack detection application overtainting will typically result in reporting an attack when no attack occurred. Second, dynamic taint analysis can miss the information flow from a source to a sink, which we call *undertainting*. In the attack detection scenario, undertainting means the system missed a real attack. A dynamic taint analysis system is *precise* if no undertainting or overtainting occurs.

In this section we first describe how dynamic taint analysis is implemented by monitoring the execution of a program. We then describe various taint analysis policies and trade-offs. Finally, we describe important issues and caveats that often result in dynamic taint analysis systems that overtaint, undertaint, or both.

A. Dynamic Taint Analysis Semantics

Since dynamic taint analysis is performed on code at runtime, it is natural to express dynamic taint analysis in terms of the operational semantics of the language. Taint policy actions, whether it be taint propagation, introduction, or checking, are added to the operational semantics rules. To keep track of the taint status of each program value, we redefine values in our language to be tuples of the form $\langle v, \tau \rangle$, where v is a value in the initial language, and τ is the taint status of v. A summary of the necessary changes to SIMPIL is provided in Table II.

taint t	::=	**T** | **F**
value	::=	$\langle v, t \rangle$
τ_Δ	::=	Maps variables to taint status
τ_μ	::=	Maps addresses to taint status

Table II: Additional changes to SIMPIL to enable dynamic taint analysis.

Figure 5 shows how a taint analysis policy P is added to SIMPIL. The semantics show *where* the taint policy is used; the semantics are independent of the policy itself. In order to support taint policies, the semantics introduce two new contexts: τ_Δ and τ_μ. τ_Δ keeps track of the taint status of scalar variables. τ_μ keeps track of the taint status of memory cells. τ_Δ and τ_μ are initialized so that all values are marked untainted. Together, τ_Δ and τ_μ keep the taint status for all variables and memory cells, and are used to derive the taint status for all values during execution.

B. Dynamic Taint Policies

A taint policy specifies three properties: how new taint is introduced to a program, how taint propagates as instructions execute, and how taint is checked during execution.

Taint Introduction. Taint introduction rules specify how taint is introduced into a system. The typical convention is to initialize all variables, memory cells, etc. as untainted. In SIMPIL, we only have a single source of user input: the get_input(·) call. In a real implementation, get_input(·) represents values returned from a system call, return values from a library call, etc. A taint policy will also typically distinguish between different input sources. For example, an internet-facing network input source may always introduce taint, while a file descriptor that reads from a trusted configuration file may not [7, 49, 64]. Further, specific taint sources can be tracked independently, e.g., τ_Δ can map not just the bit indicating taint status, but also the source.

Taint Propagation. Taint propagation rules specify the taint status for data derived from tainted or untainted operands. Since taint is a bit, propositional logic is usually used to express the propagation policy, e.g., $t_1 \vee t_2$ indicates the result is tainted if t_1 is tainted or t_2 is tainted.

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \ldots \quad \mu, \Delta \vdash e_i \Downarrow v_i \quad \Delta' = \Delta[x_1 \leftarrow v_1, \ldots, x_i \leftarrow v_i] \quad pc' = \phi[f] \quad \iota = \Sigma[pc']}{\lambda, \Sigma, \phi, \mu, \Delta, \zeta, pc, \text{call f}(e_1,\ldots,e_i) \rightsquigarrow (pc+1) :: \lambda, \Sigma, \phi, \mu, \Delta', \Delta :: \zeta, pc', \iota} \text{ CALL}$$

$$\frac{\iota = \Sigma[pc']}{pc' :: \lambda', \Sigma, \phi, \mu, \Delta, \Delta' :: \zeta', pc, \text{return} \rightsquigarrow \lambda', \Sigma, \phi, \mu, \Delta', \zeta', pc', \iota} \text{ RET}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v \notin dom(\Sigma) \quad s = \text{disassemble}(\mu[\text{v}]) \quad \Sigma' = \Sigma[v \leftarrow s] \quad \iota = \Sigma'[v]}{\Sigma, \mu, \Delta, pc, \text{jmp } e \rightsquigarrow \Sigma', \mu, \Delta, v, \iota} \text{ GENCODE}$$

Figure 4: Example operational semantics for adding support for call-by-value function calls and dynamically generated code.

Component	Policy Check
$P_{\textbf{input}}(\cdot)$, $P_{\textbf{bincheck}}(\cdot)$, $P_{\textbf{memcheck}}(\cdot)$	**T**
$P_{\textbf{const}}()$	**F**
$P_{\textbf{unop}}(t)$, $P_{\textbf{assign}}(t)$	t
$P_{\textbf{binop}}(t_1, t_2)$	$t_1 \vee t_2$
$P_{\textbf{mem}}(t_a, t_v)$ (a: ADDRESS, V: VALUE)	t_v
$P_{\textbf{condcheck}}(t_e, t_a)$	$\neg t_a$
$P_{\textbf{gotocheck}}(t_a)$	$\neg t_a$

Table III: A typical tainted jump target policy for detecting attacks. A dot $(\cdot)$ denotes an argument that is ignored. A taint status is converted to a boolean value in the natural way, e.g., **T** maps to true, and **F** maps to false.

Taint Checking. Taint status values are often used to determine the runtime behavior of a program, e.g., an attack detector may halt execution if a jump target address is tainted. In SIMPIL, we perform checking by adding the policy to the premise of the operational semantics. For instance, the T-GOTO rule uses the $P_{\textbf{gotocheck}}(t)$ policy. $P_{\textbf{gotocheck}}(t)$ returns **T** if it is safe to perform a jump operation when the target address has taint value t, and returns **F** otherwise. If **F** is returned, the premise for the rule is not met and the machine terminates abnormally (signifying an exception).

C. A Typical Taint Policy

A prototypical application of dynamic taint analysis is attack detection. Table III shows a typical attack detection policy which we call the *tainted jump policy*. In order to be concrete when discussing the challenges and opportunities in taint analysis, we often contrast implementation choices with respect to this policy. We stress that although the policy is designed to detect attacks, other applications of taint analysis are typically very similar.

The goal of the tainted jump policy is to protect a potentially vulnerable program from control flow hijacking attacks. The main idea in the policy is that an input-derived value will never overwrite a control-flow value such as a return address or function pointer. A control flow exploit, however, will overwrite jump targets (e.g., return addresses) with input-derived values. The tainted jump policy ensures safety against such attacks by making sure tainted jump targets are never used.

The policy introduces taint into the system by marking all values returned by get_input(·) as tainted. Taint is then propagated through the program in a straightforward manner, e.g., the result of a binary operation is tainted if either operand is tainted, an assigned variable is tainted if the right-hand side value is tainted, and so on.

Example 3. *Table IV shows the taint calculations at each step of the execution for the following program:*

```
1  x := 2*get_input(·)
2  y := 5 + x
3  goto y
```

On line 1, the executing program receives input, assumed to be 20, and multiplies by 2. Since all input is marked as tainted, $2 * get\_input(\cdot)$ is also tainted via T-BINOP, *and x is marked in τ_Δ as tainted via* T-ASSIGN. *On line 2, x (tainted) is added to y (untainted). Since one operand is tainted, y is marked as tainted in τ_Δ. On line 3, the program jumps to y. Since y is tainted, the* T-GOTO *premise for P is not satisfied, and the machine halts abnormally.*

Different Policies for Different Applications. Different applications of taint analysis can use different policy decisions. As we will see in the next section, the typical taint policy described in Table III is not appropriate for all application domains, since it does not consider whether memory addresses are tainted. Thus, it may miss some attacks. We discuss alternatives to this policy in the next section.

D. Dynamic Taint Analysis Challenges and Opportunities

There are several challenges to using dynamic taint analysis correctly, including:

- **Tainted Addresses.** Distinguishing between memory addresses and cells is not always appropriate.
- **Undertainting.** Dynamic taint analysis does not properly handle some types of information flow.

HERE THE TAINTNESS IS DISTINGUISHED BETWEEN ADDRESSES AND ACTUAL VALUES (THE CONTENT OF THE MEM CELLS WITH THAT ADDRESSES)

$$\frac{v \text{ is input from } src}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{get\_input}(src) \Downarrow \langle v, P_{\textbf{input}}(\text{src})\rangle}\ \text{T-INPUT} \qquad \frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash v \Downarrow \langle v, P_{\textbf{const}}()\rangle}\ \text{T-CONST}$$

$$\frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash var \Downarrow \langle \Delta[var], \tau_\Delta[var]\rangle}\ \text{T-VAR} \qquad \frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{load } e \Downarrow \langle \mu[v], P_{\textbf{mem}}(t, \tau_\mu[v])\rangle}\ \text{T-LOAD}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \lozenge_u e \Downarrow \langle \lozenge_u v, P_{\textbf{unop}}(t)\rangle}\ \text{T-UNOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad P_{\textbf{bincheck}}(t_1, t_2, v_1, v_2, \lozenge_b) = \mathbf{T}}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \lozenge_b e_2 \Downarrow \langle v_1 \lozenge_b v_2, P_{\textbf{binop}}(t_1, t_2)\rangle}\ \text{T-BINOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle \quad \Delta' = \Delta[var \leftarrow v] \quad \tau'_\Delta = \tau_\Delta[var \leftarrow P_{\textbf{assign}}(t)] \quad \iota = \Sigma[pc+1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, var := e \rightsquigarrow \tau_\mu, \tau'_\Delta, \Sigma, \mu, \Delta', pc+1, \iota}\ \text{T-ASSIGN}$$

$$\frac{\begin{array}{c}\iota = \Sigma[pc+1] \quad P_{\textbf{memcheck}}(t_1, t_2) = \mathbf{T} \\ \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad \mu' = \mu[v_1 \leftarrow v_2] \quad \tau'_\mu = \tau_\mu[v_1 \leftarrow P_{\textbf{mem}}(t_1, t_2)]\end{array}}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \rightsquigarrow \tau'_\mu, \tau_\Delta, \Sigma, \mu', \Delta, pc+1, \iota}\ \text{T-STORE}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t\rangle \quad \iota = \Sigma[pc+1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc+1, \iota}\ \text{T-ASSERT}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_2\rangle \quad P_{\textbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota}\ \text{T-TCOND}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 0, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad P_{\textbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_2]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_2, \iota}\ \text{T-FCOND}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v_1, t\rangle \quad P_{\textbf{gotocheck}}(t) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota}\ \text{T-GOTO}$$

Figure 5: Modified operational semantics of SIMPIL that enforce a taint policy P. **T** denotes true.

Line #	Statement	Δ	τ_Δ	Rule	pc
	start	$\{\}$	$\{\}$		1
1	x := 2*get_input(·)	$\{x \to 40\}$	$\{x \to \mathbf{T}\}$	T-ASSIGN	2
2	y := 5 + x	$\{x \to 40, y \to 45\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	T-ASSIGN	3
3	goto y	$\{x \to 40, y \to 45\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	T-GOTO	*error*

Table IV: Taint calculations for example program. **T** denotes tainted.

- **Overtainting.** Deciding when to introduce taint is often easier than deciding when to remove taint.
- **Time of Detection vs. Time of Attack.** When used for attack detection, dynamic taint analysis may raise an alert too late.

Table V summarizes the alternate policies proposed for addressing some of these challenges in particular scenarios. In the remainder of the section we discuss the advantages and disadvantages of these policy choices, and detail common implementation details and pitfalls.

Tainted Addresses. Memory operations involve two values: the address of the memory cell being referenced, and the value stored in that cell. The tainted jump policy in Table III independently tracks the taint status of addresses and memory cells separately. This policy is akin to the idea that the taint status of a pointer (in this case, an address) and the object pointed to (in this case, the memory cell) are independent [31].

Example 4. *Given the tainted jump policy, consider the*

Policy	Substitutions
Tainted Value	$P_{\mathbf{mem}}(t_a, t_v) \equiv t_v$
Tainted Addresses	$P_{\mathbf{mem}}(t_a, t_v) \equiv t_a \vee t_v$
Control Dependent	Not possible
Tainted Overflow	$P_{\mathbf{bincheck}}(t_1, t_2, v_1, v_2, \Diamond_b) \equiv (t_1 \vee t_2) \Rightarrow \neg \mathit{overflows}(v_1 \Diamond_b v_2)$

Table V: Alternate taint analysis policy choices.

following program:

```
1  x := get_input(·)
2  y := load(z + x)
3  goto y
```

The user provides input to the program that is used as a table index. The result of the table lookup is then used as the target address for a jump. Assuming addresses are of some fixed-width (say 32-bits), the attacker can pick an appropriate value of x to address any memory cell she wishes. As a result, the attacker can jump to any value in memory that is untainted. In many programs this would allow the user to violate the intended control flow of the program, thus creating a security violation.

The tainted jump policy applied to the above program still allows an attacker to jump to untainted, yet attacker-determined locations. This is an example of undertaint by the policy. This means that the tainted jump policy may miss an attack.

One possible fix is to use the *tainted addresses policy* shown in Table V. Using this policy, a memory cell is tainted if either the memory cell value or the memory address is tainted. TaintCheck [49], a dynamic taint analysis engine for binary code, offers such an option.

The tainted address policy, however, also has issues. For example, the tcpdump program has legitimate code similar to the program above. In tcpdump, a network packet is first read in. The first byte of the packet is used as an index into a function pointer table to print the packet type, e.g., if byte 0 of the packet is 4, the IPv4 printer is selected and then called. In the above code z represents the base address of the function call table, and x is the first byte of the packet. Thus, the tainted address modification would cause *every* non-trivial run of tcpdump to raise a taint error. Other code constructs, such as switch statements, can cause similar table lookup problems.

The tainted address policy may find additional taint flows, but may also overtaint. On the other hand, the tainted jump policy can lead to undertaint. In security applications, such as attack detection, this dichotomy means that the attack detector either misses some exploits (i.e., false negatives) or reports safe executions as bad (i.e., false positives).

Control-flow taint. Dynamic taint analysis tracks data flow taint. However, information flow can also occur through control dependencies.

Informally, a statement s_2 is control-dependent on statement s_1 if s_1 controls whether or not s_2 will execute. A more precise definition of control-dependency that uses post-dominators can be found in [29]. In SIMPIL, only indirect and conditional jumps can cause control dependencies.

Example 5. *Consider the following program:*

```
1  x := get_input(·)
2  if x = 1 then goto 3 else goto 4
3    y := 1
4  z := 42
```

The assignment to y is control-dependent on line 2, since the branching outcome determines whether or not line 3 is executed. The assignment to z is not control-dependent on line 2, since z will be assigned the value 42 regardless of which branch is taken.

If you do not compute control dependencies, you cannot determine control-flow based taint, and the overall analysis may undertaint. Unfortunately, pure dynamic taint analysis cannot compute control dependencies, thus cannot accurately determine control-flow-based taint. The reason is simple: reasoning about control dependencies requires reasoning about multiple paths, and dynamic analysis executes on a single path at a time. In the above example, any single execution will not be able to tell that the value of y is control-dependent and z is not.

There are several possible approaches to detecting control-dependent taint:

1) Supplement dynamic analysis with static analysis. Static analysis can compute control dependencies, and thus can be used to compute control-dependent taint [1, 20, 52]. Static analysis can be applied over the entire program, or over a collection of dynamic analysis runs.
2) Use heuristics, making an application-specific choice whether to overtaint or undertaint depending upon the scenario [20, 49, 63].

Sanitization. Dynamic taint analysis as described only adds taint; it never removes it. This leads to the problem of *taint spread*: as the program executes, more and more values become tainted, often with less and less taint precision.

A significant challenge in taint analysis is to identify when taint can be removed from a value. We call this the *taint sanitization problem*. One common example where we wish

EXAMPLE 4 EXPLAINED

1) $X := \text{GET\_INPUT}(\cdot)$

THE RULE T-ASSIGN IS APPLIED (WE DO RULE INSTANCING)

$$\frac{\tau_N, \tau_\Delta, N, \Delta \vdash \text{GET\_INPUT} \Downarrow \langle v, t \rangle \quad \Delta' = \Delta[X \leftarrow v] \quad \tau'_\Delta = \tau_\Delta[X \leftarrow P_{ASSIGN}(t)] \quad \iota = \Sigma[PC+1]}{\tau_N, \tau_\Delta, N, \Delta, PC, X = \text{GET\_INPUT}(\cdot) \rightsquigarrow \tau_N, \tau'_\Delta, \Sigma, N, \Delta', PC+1}$$

HERE T-INPUT IS APPLIED

$$\frac{V \text{ IS TAKEN FROM INPUT}}{\tau_N, \tau_\Delta, N, \Delta \vdash \text{GET\_INPUT}(\cdot), \Downarrow \langle v, \underbrace{P_{INPUT}(\cdot)}_{T \text{ BY DEF OF } P_{INPUT}} \rangle}$$

— THIS IS t

HENCE, SAID THAT THE VALUE FROM THE INPUT IS v, THE PREV RULE T-ASSIGN GIVES THE FOLLOWING:

- τ_N IS INVARIATED (ADDRESSES TO TAINT STATUS INVARIATED)
- $\tau'_\Delta = \tau_\Delta[X \leftarrow P_{ASSIGN}(t)]$; $t = T$ AND $P_{ASSIGN}(t) = t$
  HENCE $\tau'_\Delta(X) = \text{TRUE}$; τ_Δ MAPS VARIABLES TO TAINT STATUS, AND IN FACT X IS TAINTED SINCE IS TAKEN FROM EXTERNAL INPUT
- Δ' ; $\Delta'(X) = v$
- N IS INVARIATED

2) $V = \text{LOAD}(X + Z)$

WE APPLY ONCE AGAIN THE ASSIGN RULE BUT STEP BY STEP

a) T-BINOP TO HANDLE THE SUM $X+Z$, THAT PRODUCES $\langle v+z, P_{BINOP}(t_x, t_z) \rangle$

WHERE t_1 AND t_2 ARE THE TAINTNESS OF X AND Z. HENCE BY DEF OF P_{BINOP} WE HAVE $P_{BINOP}(t_1, t_2) = T$, AS $t_1 = T$ BECAUSE X IS TAINTED (IF X IS TAINTED ALSO X+Y IS TAINTED)

b) NOW WE USE THE RULE T-LOAD

$$\frac{\tau_N, \tau_\Delta, N, \Delta \vdash x+z \Downarrow \langle v, t \rangle}{\tau_N, \tau_\Delta, N, \Delta \vdash \text{LOAD}(x+z) \Downarrow \langle N[v], P_{MEM}(t, \tau_N[v]) \rangle}$$

THIS IS FROM BEFORE! IT IS TRUE BECAUSE OF X+Z

THIS IS FROM BEFORE! $v = v + z$

N: ADDRESS TO VALUES WE LOAD THE VALUE AT ADDRESS V

BY DEF: $P_{MEM}(t_a, t_v) = t_v$ AND $\tau_N[v]$ IS THE VALUE AT ADDRESS V: IT MATTERS NOT IF V IS TAINTED! IF THE VALUE AT THE "TAINTED ADDRESS" IS TAINTED, AS LONG THE VALUE IN THE MEMORY IS NOT TAINTED THE POLICY IS RESPECTED AND THIS MIGHT BE A PROBLEM

A FIRST EXAMPLE OF (NEEDED) SANITIFICATION

to sanitize is when the program computes constant functions. A typical example in x86 code is $b = a \oplus a$. Since b will always equal zero, the value of b does not depend upon a. x86 programs often use this construct to zero out registers. A default taint analysis policy, however, will identify b as tainted whenever a is tainted. Some taint analysis engines check for well-known constant functions, e.g., TEMU [7] and TaintCheck [49] can recognize the above xor case.

EXAMPLE 2

The output of a constant function is completely independent of user input. However, some functions allow users to affect their output without allowing them to choose an arbitrary output value. For example, it is computationally hard to find inputs that will cause a cryptographically secure hash function to output an arbitrary value. Thus, in some application domains, we can treat the output of functions like cryptographic hash functions as untainted. Newsome *et al.* have explored how to automatically recognize such cases by quantifying how much control users can exert on a function's output [48].

EXAMPLE 3

Finally, there may be application-dependent sanitization. For example, an attack detector may want to untaint values if the program logic performs sanitization itself. For example, if the application logic checks that an index to an array is within the array size, the result of the table lookup could be considered untainted.

Time of Detection vs Time of Attack. Dynamic taint analysis be used to flag an alert when tainted values are used in an unsafe way. However, there is no guarantee that the program integrity has not been violated before this point.

One example of this problem is the *time of detection/time of attack gap* that occurs when taint analysis is used for attack detection. Consider a typical return address overwrite exploit. In such attacks, the user can provide an exploit that overwrites a function return address with the address of attacker-supplied shellcode. The tainted jump policy will catch such attacks because the return address will become tainted during overwrite. The tainted jump policy is frequently used to detect such attacks against potentially unknown vulnerabilities. [20–22, 49, 63]

Note, however, that the tainted jump policy does not raise an error when the return address is first overwritten; only when it is later used as a jump target. Thus, the exploit will not be reported until the function returns. Arbitrary effects could happen between the time when the return address is first overwritten and when the attack is detected, e.g., any calls made by the vulnerable function will still be made before an alarm is raised. If these calls have side effects, e.g., include file manipulation or networking functions, the effects can persist even after the program is aborted.

The problem is that dynamic taint analysis alone keeps track of too little information. In a return overwrite attack, the abstract machine would need to keep track of where return addresses are and verify that they are not overwritten. In binary code settings, this is difficult.

value v	::=	32-bit unsigned integer | *exp*
Π	::=	Contains the current constraints on symbolic variables due to path choices

Table VI: Changes to SIMPIL to allow forward symbolic execution.

Another example of the time of detection/time of attack gap is detecting integer overflow attacks. Taint analysis alone does not check for overflow: it just marks which values are derived from taint sources. An attack detector would need to add additional logic beyond taint analysis to find such problems. For example, the *tainted integer overflow policy* shown in Table V is the composition of a taint analysis check and an integer overflow policy.

Current taint-based attack detectors [7, 20, 49, 63] typically exhibit time of detection to time of attack gaps. BitBlaze [7] provides a set of tools for performing a post hoc instruction trace analysis on execution traces produced with their taint infrastructure for post hoc analysis. Post hoc trace analysis, however, negates some advantages of having a purely dynamic analysis environment.

IV. Forward Symbolic Execution

Forward symbolic execution allows us to reason about the behavior of a program on many different inputs at one time by building a logical formula that represents a program execution. Thus, reasoning about the behavior of the program can be reduced to the domain of logic.

A. Applications and Advantages

Multiple inputs. One of the advantages of forward symbolic execution is that it can be used to reason about more than one input at once. For instance, consider the program in Example 6 — only one out of 2^{32} possible inputs will cause the program to take the true branch. Forward symbolic execution can reason about the program by considering two different input classes — inputs that take the true branch, and those that take the false branch.

Example 6. *Consider the following program:*

```
1  x := 2*get_input(·) + 1
2  if x−5 == 14 then goto 3 else goto 4
3  // catastrophic failure
4  // normal behavior
```

Only one input will trigger the failure.

B. Semantics of Forward Symbolic Execution

The primary difference between forward symbolic execution and regular execution is that when get_input(·) is evaluated symbolically, it returns a symbol instead of a concrete value. When a new symbol is first returned, there are no

constraints on its value; it represents any possible value. As a result, expressions involving symbols cannot be fully evaluated to a concrete value (e.g., $s+5$ can not be reduced further). Thus, our language must be modified, allowing a value to be a partially evaluated symbolic expression. The changes to SIMPIL to allow forward symbolic execution are shown in Table VI.

Branches constrain the values of symbolic variables to the set of values that would execute the path. The updated rules for branch statements are given as S-TCOND and S-FCOND in Figure 6. For example, if the execution of the program follows the true branch of "if $x > 2$ then goto e_1 else goto e_2", then x must contain a value greater than 2. If execution instead takes the false branch, then x must contain a value that is not greater than 2. Similarly, after an assertion statement, the values of symbols must be constrained such that they satisfy the asserted expression.

We represent these constraints on symbol assignments in our operational semantics with the path predicate Π. We show how Π is updated by the language constructs in Figure 6. At every symbolic execution step, Π contains the constraints on the symbolic variables.

C. Forward Symbolic Execution Example

The symbolic execution of Example 6 is shown in Table VII. On Line 1, get_input(·) evaluates to a fresh symbol s, which initially represents any possible user input. s is doubled and then assigned to x. This is reflected in the updated Δ.

When forward symbolic execution reaches a branch, as in Line 2, it must choose which path to take. The strategy used for choosing paths can significantly impact the quality of the analysis; we discuss this later in this section. Table VII shows the program contexts after symbolic execution takes both paths (denoted by the use of the S-TCOND and S-FCOND rules). Notice that the path predicate Π depends on the path taken through the program.

D. Forward Symbolic Execution Challenges and Opportunities

Creating a forward symbolic execution engine is conceptually a very simple process: take the operational semantics of the language and change the definition of a value to include symbolic expressions. However, by examining our formal definition of this intuition, we can find several instances where our analysis breaks down. For instance:

- **Symbolic Memory.** What should we do when the analysis uses the μ context — whose index must be a non-negative integer — with a symbolic index?
- **System Calls.** How should our analysis deal with external interfaces such as system calls?
- **Path Selection.** Each conditional represents a branch in the program execution space. How should we decide which branches to take?

We address these issues and more below.

Symbolic Memory Addresses. The LOAD and STORE rules evaluate the expression representing the memory address to a value, and then get or set the corresponding value at that address in the memory context μ. When executing concretely, that value will be an integer that references a particular memory cell.

When executing symbolically, however, we must decide what to do when a memory reference is an expression instead of a concrete number. The *symbolic memory address* problem arises whenever an address referenced in a load or store operation is an expression derived from user input instead of a concrete value.

When we load from a symbolic expression, a sound strategy is to consider it a load from any possible satisfying assignment for the expression. Similarly, a store to a symbolic address could overwrite any value for a satisfying assignment to the expression. Symbolic addresses are common in real programs, e.g., in the form of table lookups dependent on user input.

Symbolic memory addresses can lead to aliasing issues even along a single execution path. A potential address alias occurs when two memory operations refer to the same address.

Example 7. *Consider the following program:*

```
1  store(addr1, v)
2  z = load(addr2)
```

If $addr1 = addr2$*, then* $addr1$ *and* $addr2$ *are aliased and the value loaded will be the value* v*. If* $addr1 \neq addr2$*, then* v *will not be loaded. In the worst case,* $addr1$ *and* $addr2$ *are expressions that are sometimes aliased and sometimes not.*

There are several approaches to dealing with symbolic references:

- One approach is to make unsound assumptions for removing symbolic addresses from programs. For example, Vine [7] can optionally rewrite all memory addresses as scalars based on name, e.g., Example 7 would be rewritten as:

```
1  mem_addr1 = v
2  z = mem_addr2
```

  The appropriateness of such unsound assumptions varies depending on the overall application domain.
- Let subsequent analysis steps deal with them. For example, many application domains pass the generated formulas to a SMT solver [4, 32]. In such domains we can let the SMT solver reason about all possible

MAYBE TO DO LATER ON

$$\frac{v \text{ is a fresh symbol}}{\mu, \Delta \vdash \text{get\_input}(\cdot) \Downarrow v} \text{ S-INPUT}$$

$$\frac{\mu, \Delta \vdash e \Downarrow e' \quad \Pi' = \Pi \wedge e' \quad \iota = \Sigma[pc+1]}{\Pi, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \Pi', \Sigma, \mu, \Delta, pc+1, \iota} \text{ S-ASSERT}$$

$$\frac{\mu, \Delta \vdash e \Downarrow e' \quad \Delta \vdash e_1 \Downarrow v_1 \quad \Pi' = \Pi \wedge (e' = 1) \quad \iota = \Sigma[v_1]}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Pi', \Sigma, \mu, \Delta, v_1, \iota} \text{ S-TCOND}$$

$$\frac{\mu, \Delta, \vdash e \Downarrow e' \quad \Delta \vdash e_2 \Downarrow v_2 \quad \Pi' = \Pi \wedge (e' = 0) \quad \iota = \Sigma[v_2]}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Pi', \Sigma, \mu, \Delta, v_2, \iota} \text{ S-FCOND}$$

Figure 6: Operational semantics of the language for forward symbolic execution.

Statement	Δ	Π	Rule	pc
start	$\{\}$	$true$		1
x := 2*get_input(·)	$\{x \rightarrow 2 * s\}$	$true$	S-ASSIGN	2
if x-5 == 14 goto 3 else goto 4	$\{x \rightarrow 2 * s\}$	$[(2 * s) - 5 == 14]$	S-TCOND	3
if x-5 == 14 goto 3 else goto 4	$\{x \rightarrow 2 * s\}$	$\neg[(2 * s) - 5 == 14]$	S-FCOND	4

Table VII: Simulation of forward symbolic execution.

aliasing relationships. In order to logically encode symbolic addresses, we must explicitly name each memory update. Example 7 can be encoded as:

$$mem_1 = (mem_0 \text{ with } mem_0[addr_1] = v) \wedge z = mem_1[addr_2]$$

The above formula should be read as mem_1 is the same as mem_0 except at index $addr_1$, where the value is v. Subsequent reads are performed on mem_1.

- Perform alias analysis. One could try to reason about whether two references are pointing to the same address by performing alias analysis. Alias analysis, however, is a static or offline analysis. In many application domains, such as recent work in automated test-case generation [8, 16–18, 28, 33, 34, 56], fuzzing [35], and malware analysis [10, 44], part of the allure of forward symbolic execution is that it can be done at run-time. In such scenarios, adding a static analysis component may be unattractive.

Unfortunately, most previous work does not specifically address the problem of symbolic addresses. KLEE and its predecessors [16, 18] perform a mix of alias analyses and letting the SMT solver worry about aliasing. DART [35] and CUTE [56] only handle formulas that are linear constraints and therefore cannot handle general symbolic references. However, when a symbolic memory access is a linear address, they can solve the system of linear equations to see if they may be aliased. To the best of our knowledge, previous work in malware analysis has not addressed the issue. Thus, malware authors could intentionally create malware that includes symbolic memory operations to thwart analysis.

Path Selection. When forward symbolic execution encounters a branch, it must decide which branch to follow first. We call this the *path selection problem*.

We can think of a forward symbolic execution of an entire program as a tree in which every node represents a particular instance of the abstract machine (e.g., $\Pi, \Sigma, \mu, \Delta, pc, \iota$). The analysis begins with only a root node in the tree. However, every time the analysis must fork, such as when a conditional jump is encountered, it adds as children all possible forked states to the current node. We can further explore any leaf node in the tree that has not terminated. Thus, forward symbolic execution needs a strategy for choosing which state to explore next. This choice is important, because loops with symbolic conditions may never terminate. If an analysis tries to explore such a loop in a naïve manner, it might never explore other branches in the state tree.

Loops can cause trees of infinite depth. Thus, the handling of loops are an integral component in the path-selection strategy. For example, suppose n is input in:

```
while (3^n + 4^n == 5^n) { n++; ... }
```

Exploring all paths in this program is infeasible. Although we know mathematically there is no satisfying answer to the branch guard other than 2, the forward symbolic execution algorithm does not. The formula for one loop iteration will include the branch guard $3^n + 4^n = 5^n$, the second iteration will have the branch guard $3^{n+1} + 4^{n+1} = 5^{n+1}$, and so on.

Typically, forward symbolic execution will provide an upper bound on loop iterations to consider in order to keep it from getting "stuck" in such potentially infinite or long-running loops.

Approaches to the path selection problem include:

1) **Depth-First Search.** DFS employs the standard depth-first search algorithm on the state tree. The primary disadvantage of DFS is that it can get stuck in non-terminating loops with symbolic conditions if no maximum depth is specified. If this happens, then no other branches will be explored and code coverage will be low. KLEE [16] and EXE [18] can implement a DFS search with a configurable maximum depth for cyclic paths to prevent infinite loops.
2) **Concolic Testing.** Concolic testing [28, 36, 56] uses concrete execution to produce a trace of a program execution. Forward symbolic execution then follows the same path as the concrete execution. The analysis can optionally generate concrete inputs that will force the execution down another path by choosing a conditional and negating the constraints corresponding to that conditional statement.
   Since forward symbolic execution can be magnitudes slower than concrete execution, one variant of concolic testing uses a single symbolic execution to generate many concrete testing inputs. This search strategy is called *generational search* [36].
3) **Random Paths.** A random path strategy is also implemented by KLEE [16] where the forward symbolic execution engine selects states by randomly traversing the state tree from the root until it reaches a leaf node. The random path strategy gives a higher weight to shallow states. This prevents executions from getting stuck in loops with symbolic conditions.
4) **Heuristics.** Additional heuristics can help select states that are likely to reach uncovered code. Sample heuristics include the distance from the current point of execution to an uncovered instruction, and how recently the state reached uncovered code in the past.

Symbolic Jumps. The premise of the GOTO rule requires the address expression to evaluate to a concrete value, similar to the LOAD and STORE rules. However, during forward symbolic execution the jump target may be an expression instead of a concrete location. We call this the *symbolic jump* problem. One common cause of symbolic jumps are jump tables, which are commonly used to implement switch statements.

A significant amount of previous work in forward symbolic execution does not directly address the symbolic jump problem [8, 16–18, 28, 35, 36, 56]. In some domains, such as automated test-case generation, leaving symbolic jumps out-of-scope simply means a lower success rate. In other domains, such as in malware analysis, widespread use of symbolic jumps would pose a challenge to current automated malware reverse engineering [10, 11, 44].

Three standard ways to handle symbolic jumps are:

1) Use concrete and symbolic (concolic) analysis [56] to run the program and observe an indirect jump target. Once the jump target is taken in the concrete execution, we can perform symbolic execution of the concrete path. One drawback is that it becomes more difficult to explore the full-state space of the program because we only explore known jump targets. Thus, code coverage can suffer.
2) Use a SMT solver. When we reach a symbolic jump to e with path predicate Π, we can ask the SMT solver for a satisfying answer to $\Pi \wedge e$. A satisfying answer includes an assignment of values to variables in e, which is a concrete jump target. If we are interested in more satisfying answers, we add to the query to return values different from those previously seen. For example, if the first satisfying answer is n, we query for $\Pi \wedge e \wedge \neg n$. Although querying a SMT solver is a perfectly valid solution, it may not be as efficient as other options that take advantage of program structure, such as static analysis.
3) Use static analysis. Static analysis can reason about the entire program to locate possible jump targets. In practice, source-level indirect jump analyses typically take the form of pointer analyses. Binary-level jump static analyses reason about what values may be referenced in jump target expressions [2]. For example, function pointer tables are typically implemented as a table of possible jump targets.

Example 8. *Consider the following program:*

```
1  bytes := get_input(·)
2  p := load(functable + bytes)
3  goto p
```

Since `functable` *is statically known, and the size of the table is fixed, a static analysis can determine that the range of targets is load*$(functable+x)$ *where* $\{x|\, 0 \leq x \leq k\}$ *and* k *is the size of the table.*

Handling System and Library Calls. In concrete execution, system calls introduce input values to a program. Our language models such calls as get_input(·). We refer to calls that are used as input sources as system-level calls. For example, in a C program system-level calls may correspond to calling library functions such as `read`. In a binary program, system-level calls may correspond to issuing an interrupt.

Some system-level calls introduce fresh symbolic variables. However, they can also have additional side effects. For example, `read` returns fresh symbolic input and updates an internal pointer to the current read file position. A subsequent call to `read` should not return the same input.

One approach to handling system-level calls is to create summaries of their side effects [12, 16, 18]. The summaries are models that describe the side effects that occur whenever the respective code is called concretely. The advantage of summaries is that they can abstract only those details necessary for the application domain at hand. However, they typically need to be generated manually.

A different approach when using concolic execution [56] is to use values returned from system calls on previous concrete executions in symbolic execution. For example, if during a concrete execution `sys_call()` returns 10, we use 10 during forward symbolic execution of the corresponding `sys_call()`. The central advantages of a concolic-based approach is it is simple, easy to implement, and sidesteps the problem of reasoning about how a program interacts with its environment. Any analysis that uses concrete values will not, by definition, provide a complete analysis with respect to system calls. In addition, the analysis may not be sound, as some calls do not always return the same result even when given the same input. For example, `gettimeofday` returns a different time for each call.

Performance. A straightforward implementation of forward symbolic execution will lead to a) a running time exponential in the number of program branches, b) an exponential number of formulas, and c) an exponentially-sized formula per branch.

The running time is exponential in the number of branches because a new interpreter is forked off at each branch point. The exponential number of formulas directly follows, as there is a separate formula at each branch point.

Example 9. *Consider the following program:*

```
1  x := get_input(·)
2  x := x + x
3  x := x + x
4  x := x + x
5  if e then S1 else S2
6  if e2 then S3 else S4
7  if e3 then S5 else S6
8  assert(x < 10);
```

S_i are statements executed in the branches. There are 8 paths through this program, so there will be 8 runs of the interpreter and 8 path predicates.

The size of a formula even for a single program path may be exponential in size due to substitution. During both concrete and symbolic evaluation of an expression e, we substitute all variables in e with their value. However, unlike concrete evaluation, the result of evaluating e is not of constant size. Example 9 demonstrates the problem with x. If during forward symbolic execution get_input(·) returns s, after executing the three assignments Δ will map $x \rightarrow s+s+s+s+s+s+s+s$.

In practice, we can mitigate these problems in a number of ways:

- Use more and faster hardware. Exploring multiple paths and solving formulas for each path is inherently parallelizable.
- Exponential blowup due to substitution can be handled by giving each variable assignment a unique name, and then using the name instead of performing substitution. For example, the assignments to x can be written as:

$$x_1 = x_0 + x_0 \wedge x_2 = x_1 + x_1 \wedge x_3 = x_2 + x_2$$

- Identify redundancies between formulas and make them more compact. In the above example, the path predicates for all formulas will include the first four statements. Bouncer [21] uses heuristics to identify commonalities in the formulas during signature generation. Godefroid *et al.* [36] perform post hoc optimizations of formulas to reduce their size.
- Identify independent subformulas. Cadar *et al.* identify logically independent subformulas, and query each subformula separately in EXE and KLEE [16, 18]. They also implement caching on the SMT solver such that if the same formula is queried multiple times they can use the cached value instead of solving it again. For example, all path predicates for Example 9 contain as a prefix the assignments to x. If these assignments are independent of other parts of the path predicate, KLEE's cache will solve the subformula once, and then use the same returned value on the other 8 paths. Cadar *et al.* found caching instrumental in scaling forward symbolic execution [18].
- One alternative to forward symbolic execution is to use the weakest precondition [26] to calculate the formula. Formulas generated with weakest preconditions require only $O(n^2)$ time and will be at most $O(n^2)$ in size, for a program of size n [14, 30, 42]. Unlike forward symbolic execution, weakest preconditions normally process statements from last to first. Thus, weakest preconditions are implemented as a static analysis. However, a recent algorithm for efficiently computing the weakest precondition in any direction can be used as a replacement for applications that build formulas using symbolic execution [40]. The program must be converted to dynamic single assignment form before using this new algorithm.

Mixed Execution. Depending on the application domain and the type of program, it may be appropriate to limit symbolic input to only certain forms of input. For instance, in automated test generation of a network daemon, it may not make sense to consider the server configuration file symbolically — in many cases, a potential attacker will not have access to this file. Instead, it is more important to handle network packets symbolically, since these are the primary interface of the program. Allowing some inputs to be concrete and others symbolic is called *mixed execution*.

Our language can be extended to allow mixed execution by concretizing the argument of the get_input(·) expression, e.g., get_input(file), get_input(network), etc.

Besides appropriately limiting the scope of the analysis, mixed execution enables calculations involving concrete values to be done on the processor. This allows portions of the program that do not rely on user input to potentially run at the speed of concrete execution.

V. Related Work

A. Formalization and Systematization

The use of operational semantics to define dynamic security mechanisms is not new [37, 45]. Other formal mechanisms for defining such policies exist as well [54]. Despite these tools, prior work has largely avoided formalizing dynamic taint analysis and forward symbolic execution. Some analysis descriptions define a programming language similar to ours, but only informally discuss the semantics of the analyses [28, 35, 63]. Such informal descriptions of semantics can lead to ambiguities in subtle corner cases.

B. Applications

In the remainder of this section, we discuss applications of dynamic taint analysis and forward symbolic execution. Due to the scope of related work, we cite the most representative work.

Automatic Test-case Generation. Forward symbolic execution has been used extensively to achieve high code-coverage in automatic test-case generation [16–18, 28, 35, 36, 56]. Many of these tools also automatically find well-defined bugs, such as assertion errors, divisions by zero, NULL pointer dereferences, etc.

Automatic Filter Generation. Intrusion prevention/detection systems use input filters to block inputs that trigger known bugs and vulnerabilities. Recent work has shown that forward symbolic execution path predicates can serve as accurate input filters for such systems [12–14, 21, 22, 43, 46, 47].

Automatic Network Protocol Understanding. Dynamic taint analysis has been used to automatically understand the behavior of network protocols [15, 62] when given an implementation of the protocol.

Malware Analysis. Automatic reverse-engineering techniques for malware have used forward symbolic execution [10, 11, 44] and dynamic taint analysis [5, 6, 27, 57, 64] to analyze malware behavior. Taint analysis has been used to track when code unpacking is used in malware [64].

Web Applications. Many analyses of Web applications utilize dynamic taint analysis to detect common attacks such as SQL injections [3, 38, 39, 50, 55, 61] and cross-site scripting attacks [53, 55, 60]. Some researchers have also combined dynamic taint analysis with static analysis to find bugs in Web applications [3, 61]. Sekar [55], introduced taint inference, a technique that applies syntax and taint-aware policies to block injection attacks.

Taint Performance & Frameworks. The ever-growing need for more efficient dynamic taint analyses was initially met by binary instrumentation frameworks [20, 51]. Due to the high overhead of binary instrumentation techniques, more efficient compiler-based [41, 63] and hardware-based [24, 25, 58, 59] approaches were later proposed. Recent results show that a dynamic software-based approach, augmented by static analysis introduce minimal overhead, and thus can be practical [19].

Extensions to Taint Analysis. Our rules assume data is either tainted or not. For example, Newsome *et al.* have proposed a generalization of taint analysis that quantifies the influence that an input has on a particular program statement based on channel capacity [48].

VI. Conclusion

Dynamic program analyses have become increasingly popular in security. The two most common — dynamic taint analysis and forward symbolic execution — are used in a variety of application domains. However, despite their widespread usage, there has been little effort to formally define these analyses and summarize the critical issues that arise when implementing them in a security context.

In this paper, we introduced a language for demonstrating the critical aspects of dynamic taint analysis and forward symbolic execution. We defined the operational semantics for our language, and leveraged these semantics to formally define dynamic taint analysis and forward symbolic execution. We used our formalisms to highlight challenges, techniques and tradeoffs when using these techniques in a security setting.

VII. Acknowledgements

We would like to thank Dawn Song and the BitBlaze team for their useful ideas and advice on dynamic taint analysis and forward symbolic execution. We would also like to thank our shepherd Andrei Sabelfeld, JongHyup Lee, Ivan Jager, and our anonymous reviewers for their useful comments and suggestions. This work is supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. The views expressed herein are those of the authors and do not necessarily represent the views of our sponsors.

References

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

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

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications.