Università degli Studi di Padova

Department of Mathematics

Master Degree in Computer Science

All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution

(but might have been afraid to ask)

Matteo Di Pirro
1154231

Academic Year 2016-2017

Contents

1 Introduction

1.1 Of who or what do we trust?

In an ideal world users enter the right inputs and programs are written in the right way, without bugs or vulnerabilities. Unfortunately the real world is bad, very bad. In this world users often enter wrong inputs, both intentionally or not, and programmers write programs in a lazy and vulnerable way. They often don't care about user inputs, and this leads to a lot of software vulnerabilities, like buffer overflow or code injection.

This means that we can't neither trust user nor programmers and we should assume that *everything* is vulnerable. This is why we analyze programs: to discover vulnerabilities. Over the last few years, dynamic analysis has become increasingly popular in security. We run programs and we observe their behavior: that's all.

1.2 Questions about user input

Talking about users' inputs, two main questions can arise:

1.2.1 Is the final value affected by user input? ⇒ Dynamic Taint Analysis

Dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources such as user input. Any program value whose computation depends on data derived from a taint source is considered *tainted*. Any other value is considered untainted.

1.2.2 What input will make execution reach this line of code? ⇒ Forward Symbolic Execution

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.

1.3 Applications

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 the following:

- **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;

- **Automatic Input Filter Generation**: Forward symbolic execution can be used to automatically generate input filters that detect and remove exploits from the input stream. Filters generated in response to actual executions are attractive because they provide strong accuracy guarantees;
- **Malware Analysis**: Taint analysis and forward symbolic execution are used to analyze how information flows through a malware binary, explore trigger-based behavior and detect emulators;
- **Test Case Generation**: Taint analysis and forward symbolic execution are used to automatically generate inputs to test programs, and can generate inputs that cause two implementations of the same protocol to behave differently.

2 SimpIL: The language

A precise definition of dynamic taint analysis or forward symbolic execution must target a specific language: we use SimpIL, a Simple Intermediate Language. Although the language is simple, it is powerful enough to express typical languages as varied as Java and assembly code. Indeed, the language is representative of internal representations used by compilers for a variety of programming languages. A program in this language consists of a sequence of numbered statements. Statements in our language consist of assignments, assertions, jumps, and conditional jumps.

I don't want to bore you with the details of the operational semantic, but spending a few minutes talking about the grammar is necessary. Expressions are side-effect free, and we use $\Diamond$`b` to represent typical binary operators, e.g., addition, subtraction, etc. Similarly, $\Diamond$`u` represents unary operators such as logical negation. The statement `get_input(src)` returns input from source `src`. For simplicity, we consider only expressions (constants, variables, etc.) that evaluate to 32-bit integer values.

We do not include some high-level language constructs such as functions or scopes for simplicity reasons. This omission does not fundamentally limit the capability of our language or our results. Adding such constructs is trivial, and we can do both by compiling missing high-level language constructs down to the language or by adding them in a native way. Furthermore we don't really care about types: designing a sound and strong type system is possible, but we are not interested in that: remember that the dynamic analysis is dynamic, while a sound type checking should be static.

Figure 1: SimpIL Grammar

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

3 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**). We can represent this taint marker with a single bit: in this way we can perform a lot of logical operation over the taint mark in an efficient way.

```
x := get_input(devil)
z := 42
y := x + z
goto y
```

On one hand `x` contains a tainted value because it comes from outside: we don't really care from where for now, is enough to that is something external. It might come from a user, the network, a database or everything else. On the other hand `z` contains a perfectly legal value, a constant. Thus `z` is not tainted. But what about `y`? Well, actually there isn't a right answer: `y` can be tainted or not, accordingly to the selected policy. With a basic and simple policy, which always propagates the taint, `y` will be tainted, but with different policies it might not.

But what's a policy?

3.1 Policies

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. 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.

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($\cdot$) call. 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;

- **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., t1 $\vee$ t2 indicates the result is tainted if t1 is tainted or t2 is tainted;

- **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. We can perform this check adding the policy to the operational semantics. Thus, we compute a rule iff the status is correct according to the policy.

Two types of errors can occur in dynamic taint analysis, according to the selected policy. 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.

4 Farward 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.

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, in this

program, 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.

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.

```
x := 2 * get_input(src)
if x - 5 == 14 then goto 3 else goto 4
// catastrophic failure
// normal behaviour
```

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. Thus, our language must be modified, allowing a value to be a partially evaluated symbolic expression.

Branches constrain the values of symbolic variables to the set of values that would execute the path. For example, in the program above, if the execution follows the `true` branch, then `x` must contain 19. Otherwise `x` contains a value which is not 19. Every `if then else` statement creates two partitions of the input domain. The strategy used for choosing paths can significantly impact the quality of the analysis.

4.1 Challenges and Opportunities

Unfortunately, by examining our formal definition of this intuition, we can find several instances where our analysis breaks down.

4.1.1 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. 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 naive 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.

$$\texttt{while } (3^n \texttt{ + } 4^n \texttt{ == } 5^n) \texttt{ \{n++; ...\}}$$

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.

Furthermore, a straightforward implementation of forward symbolic execution will lead to:

- a running time exponential in the number of program branches, because a new interpreter is forked off at each branch point;
- an exponential number of formulas, which directly follows from the previous, as there is a separate formula at each branch point;
- an exponentially-sized formula per branch 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.

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

- **Path selection heuristics**:
  - **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 nonterminating 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;
  - **Random Paths**: this kinds of strategies traverse the state tree from the root until they reach 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;
  - **Concolic Testing**: concolic testing 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.

- **Use more and faster hardware**. Exploring multiple paths and solving formulas for each path is inherently parallelizable; item **Identify independent subformulas**. By identifying logically independent subformulas and analyze them using SMT solvers. Eventually we can implement caching so that if the same formula is queried multiple times we can use the cached value instead of solving it again;

- **Identify redundancies** between formulas and make them more compact.

4.1.2 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.

There are several approaches to dealing with symbolic references:

- Make unsound assumptions for removing symbolic addresses from programs. The appropriateness of such unsound assumptions varies depending on the overall application domain;

- Let the SMT solver reason about all possible aliasing relationships. In order to logically encode symbolic addresses, we must explicitly name each memory update;

- 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, fuzzing, and malware analysis, 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.

The premise of the `GOTO` rule requires the address expression to evaluate to a concrete value. 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. We can solve this problem in a number of ways:

- Use concrete and symbolic (concolic) analysis 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;
- Using an SMT solver. 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;
- Use static analysis. Static analysis can reason about the entire program to locate possible jump targets.

Bibliography

[1] Thomas Ball. "The concept of dynamic analysis". In: *Software Engineering—ESEC/FSE'99*. Springer. 1999, p. 216.

[2] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331. ISBN: 978-0-7695-4035-1. URL: `https://edmcman.github.io/papers/oakland10.pdf`.