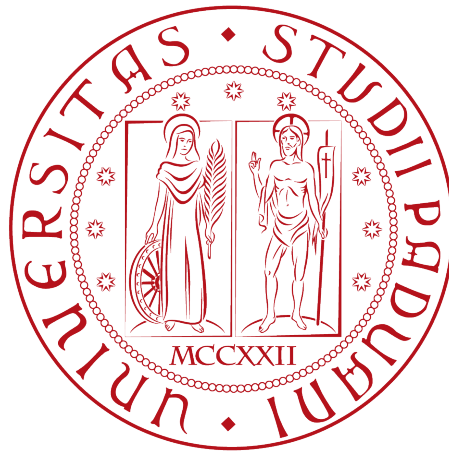


Università degli Studi di Padova

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	2
1.1	Static and Dynamic Analysis	2
1.2	Questions about user input	2
1.2.1	Is the final value affected by user input? \Rightarrow Dynamic Taint Analysis	2
1.2.2	What input will make execution reach this line of code? \Rightarrow Forward Symbolic Execution	2
1.3	Applications	2
2	SimpIL: The language	4
2.1	The basics	4
3	Dynamic Taint Analysis	5
3.1	Policies	5
	Bibliography	7

1 Introduction

1.1 Static and Dynamic Analysis

Dynamic analysis is the analysis of the properties of a running program. In contrast to static analysis, which examines a program's text to derive properties that hold for all executions, dynamic analysis derives properties that hold for one or more executions by examination of the running program. While dynamic analysis cannot prove that a program satisfies a particular property, it can detect violations of properties as well as provide useful information to programmers about the behavior of their programs. Although dynamic analysis provides a powerful mechanism for relating program inputs, this dependence from the inputs makes it incomplete. Viewed in this light, dynamic and static analysis might be better termed "input-centric" and "program-centric" analysis respectively. [1]

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.

1.2 Questions about user input

The two analyses can be used in conjunction to build formulas representing only the parts of an execution that depend upon tainted values.

1.2.1 Is the final value affected by user input? \Rightarrow 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? \Rightarrow 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

2.1 The basics

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.

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`. We use a dot (`.`) to denote an argument that is ignored, e.g., we will write `get_input(.)` when the exact input source is not relevant. For simplicity, we consider only expressions (constants, variables, etc.) that evaluate to 32-bit integer values.

Because dynamic program analyses are defined in terms of actual program executions, operational semantics also provide a natural way to define a dynamic analysis. Each statement rule is of the form:

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

Rules are deterministic and read bottom to top, left to right. Given a statement, we pattern-match the statement to find the applicable rule.

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.

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(.)` 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.

Bibliography

- [1] Thomas Ball. “The concept of dynamic analysis”. In: *Software Engineering—ESEC/FSE’99*. Springer. 1999, p. 216 (cit. on p. 2).