**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.

# 2 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 know 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 may not.

But what's a policy?

## 2.1 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. 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;
- **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.

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.

## 2.2 TaintDroid

TaintDroid is an extension to the Android OS that tracks the flow of privacy sensitive data through third-party applications. TaintDroid assumes that downloaded, third-party applications are not trusted, and monitors in real time how these applications access and manipulate users' personal data. The primary goals are to detect when sensitive data leaves the system via untrusted applications and to facilitate analysis of applications by phone users or external security services.

TaintDroid automatically labels (taints) data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and interprocess messages. When tainted data are transmitted over the network, or otherwise leave the system, TaintDroid logs the data's labels, the application responsible for transmitting the data, and the data's destination. Such real time feedback gives users and security services greater insight into what mobile applications are doing, and can potentially identify misbehaving applications.

Tracking incurs a runtime overhead of less than 14% for a CPU-bound microbenchmark. More importantly, interactive third-party applications can be monitored with negligible perceived latency.

## 2.3 Malware Detection

Pointer tainting has been used for two main purposes: detection of privacy-breaching malware (e.g., trojan keyloggers obtaining the characters typed by a user), and detection of memory corruption attacks against non-control data (e.g., a buffer overflow that modifies a user's privilege level). In both of these cases the attacker does not modify control data such as stored branch targets, so the control flow of the target program does not change. Phrased differently, in terms of instructions executed, the program behaves "normally". As a result, these attacks are exceedingly difficult to detect. Pointer tainting is considered one of the only methods for detecting them in unmodified binaries. Unfortunately, almost all of the incarnations of pointer tainting are flawed.

Pointer tainting could be defined as a kind of dynamic information flow tracking which marks the origin of data by way of a taint bit in a shadow memory that is inaccessible to software. By tracking the propagation of tainted data through the system (e.g., when tainted data is copied, but also when tainted pointers are dereferenced), we see whether any value derived

from data from a tainted origin ends up in places where it should never be stored.

Pointer tainting is very popular because:

- it can be applied to unmodified software without recompilation;
- according to its advocates, it incurs hardly (if any) false positives
- it is assumed to be one of the only (if not the only) reliable techniques capable of detecting both control-diverting and non-control-diverting attacks without requiring recompilation.

Conceptually is a simple process. We try to detect whether a new application $X$ is malicious or not, by running it in a system and analyzing it using pointer tainting. Sensitive data, such as keystrokes that are unrelated to $X$ (e.g., a password you type in when logging to a remote machine) are tagged tainted. If at some point, any byte in the address space of $X$ is tainted, it means that the sensitive data has leaked into $X$, which should not happen. Thus, the program must be malicious and is probably a keylogger.

However, for privacy-breaching malware detection, the method is flawed.

## 2.4 Limitations

### 2.4.1 Sanitization Problem

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. This is why in real world pointer tainting for malware detection is almost unusable.

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 to sanitize is when the program computes constant functions, like `b := a` $\oplus$ `a`. Since `b` will always equal zero, its value does not depend upon `a`. A default taint analysis policy, however, will identify `b` as tainted whenever `a` is tainted.

### 2.4.2 Control Flow

Dynamic taint analysis tracks data flow taint. However, information flow can also occur through control dependencies. Informally, a statement `s2` is control-dependent on statement `s1` if `s1` controls whether or not `s2` will execute. Consider the following example:

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

If we 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.

Another thing that we would discover is whether sensible information, like user-provided passwords, flow to insecure output channel. Generally we want that any secret information does not flow to a public output channel, but we might be interested to allow only a specific channel to receive only a specific type of secret information, and we would discover every other violation: in brief, we want to specify declassification policies, but with dynamic analysis we can't, or it's very difficult to do so. Thus, in such cases, static analysis is required.

# 3 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.

```
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.

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.

## 3.1 Challenges and Opportunities

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

### 3.1.1 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.

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

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.

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 suffer;
  - **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;
- **Identify independent subformulas**. By identifying logically independent subformulas and analyze them using SMT solvers;
- **Identify redundancies** between formulas and make them more compact.

# 4 Conclusions

## 4.1 Comparison

In my opinion, symbolic execution uses a kind of taint analysis to construct path constraints. Symbolic execution also employs an SMT solver to generate concrete values for variables and/or inputs, such that a certain path constraint is satisfied. Since taint analysis does not employ an SMT solver, I would say it is not a kind of symbolic execution.

Furthermore, dynamic analysis can also reason about *feasible* path, while symbolic execution builds constraint and reasons about the whole program by reducing it in the domain of logic.

Thus, in my opinion, taint analysis is a **subset** of symbolic execution.

## 4.2 Last things

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;
- **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;
- **Automatic Network Protocol Understanding**: Dynamic taint analysis has been used to automatically understand the behavior of network protocols.

However, recalling the limitations and the problems with both analysis I would say that they have to be used carefully. Their effectiveness depends on the specific application.

# Bibliography


[1] 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.

[2] Bruno P. S. Rocha et al. "Towards Static Flow-Based Declassification for Legacy and Untrusted Programs." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010, pp. 93–108. ISBN: 978-0-7695-4035-1. URL: http://dblp.uni-trier.de/db/conf/sp/sp2010.html#RochaBHWE10.

[3] William Enck et al. "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 393–407. URL: http://dl.acm.org/citation.cfm?id=1924943.1924971.

[4] Asia Slowinska and Herbert Bos. "Pointless Tainting? Evaluating the Practicality of Pointer Tainting". In: *Proceedings of the 4th EuroSys Conference*. Nuremberg, Germany, Apr. 2009.

[5] Asankhaya Sharma. *A critical review of dynamic taint analysis and forward symbolic execution*. Tech. rep. 2012.