Laboratory for Innovative Software

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

Paper Summary

Giulio Paparelli

Introduction

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.

Dynamic Taint Analysis

Dynamic Taint Analysis

The purpose of dynamic taint analysis is to track information flow between sources and sinks.
Any program whose computation depends on data derived from a taint source is considered **tainted**.

A **taint policy P** determines exactly how taint flows as a program executes, what sorts of operations introduce a new taint, and what checks are performed on tainted values.

Taint Policy

A Taint Policy specifies three properties:

Taint Introduction: Taint introduction rules specify how taint is introduced into a system.

Taint Propagation: Taint propagation rules specify the taint status for data derived from tainted or untainted operands.

Taint Checking: Taint status values are often used to determine the runtime behavior of a program.

Dynamic T.A. 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.

To keep track of the taint status of each program value, we redefine values in our language to be tuples of the form $<v, \tau>$ where v is a value in the initial language, and τ is the taint status of v.

Dynamic T.A. Semantics

example: assignment semantic rule with taint analysis

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

- Δ: maps a variable name to its value
- Σ: maps a statement number to a statement
- μ: maps a memory address to the current value at that address
- pc: program counter
- ι: the next instruction

- τ_Δ: maps variables to taint status
- τ_μ: maps address to taint status

Dynamic T.A. Challenges

There are several challenges to using dynamic taint analysis correctly:

Taint Addresses: Distinguishing between memory addresses and cells is not always appropriate.

Undertainting: Dynamic taint analysis can miss the information flow from a source to a sink. In the attack detection scenario, undertainting means the system missed a real attack.

Overtainting: Deciding when to introduce taint is often easier than deciding when to remove taint.

ToD vs ToA: When used for attack detection, dynamic taint analysis may raise an alert too late.

Dynamic T.A. Opportunities

Taint Analysis have many uses in the area of software security.

Unknown Vulnerability Detection: By tracking the flow of data during program execution, dynamic taint analysis can identify tainted data that may come from untrusted sources. This can help detect security vulnerabilities such as injection attacks.

Malware Analysis: Taint analysis is used to analyze how information through a malware binary.

Test Case Generation: Taint analysis can be used to automatically generate inputs to test programs, and can generate inputs that cause two implementations of the same protocol to behave differently.

Forward Symbolic Execution

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.

F.S. Execution Semantics

When the function *get_input(.)* is evaluated symbolically, it returns a symbol instead of a value.

Expressions involving symbols cannot be fully evaluated to a concrete value: our language must be modified, allowing a value to be a partially evaluated symbolic expression.

F.S. Execution Semantics

example: conditional semantic rule with F.S. execution

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

- Δ: maps a variable name to its value
- Σ: maps a statement number to a statement
- μ: maps a memory address to the current value at that address
- pc: program counter
- ι: the next instruction

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

F.S. Execution Challenges

There are several challenges to using F.S Execution correctly:

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. How should we decide which branches to take?

Symbolic Memory: When executing symbolically we must decide what to do when a memory reference is an expression instead of a concrete number

F.S. Execution Opportunities

Forward Symbolic Execution have many uses in the area of software security.

Automatic Input Filter Generation: Forward symbolic execution can be used to automatically generate input filters that detect and remove exploits from the input stream.

Test Case Generation: Forward symbolic execution can be used to automatically generate inputs to test programs, and can generate inputs that cause two implementations of the same protocol to behave differently.