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

Matteo Di Pirro

BSc in Computer Science
Department of Mathematics

University of Padova

December 7, 2016

UNIVERSITÀ DEGLI STUDI DI PADOVA

Outline

Introduction

The language

Dynamic Taint Analysis

Forward Symbolic Execution

Static and Dynamic Analysis

- **Static Analysis**
  - Examines a program's text to derive properties that hold for all executions
  - Program-centric analysis
- **Dynamic Analysis**
  - Examines the running program to derive properties hold for one or more executions
  - Detect violations of stated properties
  - Provide useful information about the behavior of the program
  - Input-centric analysis

Dynamic Analysis

There are two essential questions about the input analysis:

Dynamic Analysis

There are two essential questions about the input analysis:

1. **Is the final value affected by user input?**
    - **Dynamic Taint Analysis**!
    - Tracks information flow between sources and sinks

Dynamic Analysis

There are two essential questions about the input analysis:

1. **Is the final value affected by user input?**
   - **Dynamic Taint Analysis**!
   - Tracks information flow between sources and sinks
2. **What input will make execution reach this line of code?**
   - **Forward Symbolic Execution**
   - Allows us to reason about the behavior of a program on many different inputs

Use cases

The number of security applications utilizing these two techniques is enormous:

Use cases

Università degli Studi di Padova

The number of security applications utilizing these two techniques is enormous:

1. **Unknown Vulnerability Detection**: monitor whether user input is executed

Use cases

The number of security applications utilizing these two techniques is enormous:

1. **Unknown Vulnerability Detection**: monitor whether user input is executed
2. **Automatic Input Filter Generation**: detect and remove exploits from the input stream

Use cases

The number of security applications utilizing these two techniques is enormous:

1. **Unknown Vulnerability Detection**: monitor whether user input is executed
2. **Automatic Input Filter Generation**: detect and remove exploits from the input stream
3. **Forward Symbolic Execution**: analyze how information flows through a malware binary

Use cases

The number of security applications utilizing these two techniques is enormous:

1. **Unknown Vulnerability Detection**: monitor whether user input is executed
2. **Automatic Input Filter Generation**: detect and remove exploits from the input stream
3. **Forward Symbolic Execution**: analyze how information flows through a malware binary
4. **Test Case Generation**: automatically generate inputs to test programs

SimpIL

Università degli Studi di Padova

Designed to demonstrate the critical aspects of this analysis.

$$
\begin{array}{lcl}
\textit{program} & ::= & \textit{stmt*} \\
\textit{stmt } s & ::= & \textit{var} := \textit{exp} \mid \text{store}(\textit{exp}, \textit{exp}) \\
 & & \mid \text{goto } \textit{exp} \mid \text{assert } \textit{exp} \\
 & & \mid \text{if } \textit{exp} \text{ then goto } \textit{exp} \\
 & & \quad \text{else goto } \textit{exp} \\
\textit{exp } e & ::= & \text{load}(\textit{exp}) \mid \textit{exp} \Diamond_b \textit{exp} \mid \Diamond_u \textit{exp} \\
 & & \mid \textit{var} \mid \text{get\_input}(\textit{src}) \mid v \\
\Diamond_b & ::= & \text{typical binary operators} \\
\Diamond_u & ::= & \text{typical unary operators} \\
\textit{value } v & ::= & \text{32-bit unsigned integer}
\end{array}
$$

SimpIL Grammar

SimpIL

Designed to demonstrate the critical aspects of this analysis.

- **Each** statement rule of the operational semantic is like:

$$\frac{\text{computation}}{<\text{current state}>,\ \text{stmt} \rightarrow <\text{end state}>,\ \text{stmt}}$$

- The `state` is composed of:
  - Program statements ($\sum$)
  - Current memory state (μ)
  - Current values for variables (Δ)
  - Program counter (***pc***)
  - Current statement (***i***)

Dynamic Taint Analysis

`x := get_input(``)`

`z := 42`

`y := x + z`

`goto y`

Dynamic Taint Analysis

Tainted

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

x is derived from a tainted source

Dynamic Taint Analysis

```
x := get_input( )
```

Untainted

```
z := 42
y := x + z
goto y
```

z is a ”static” constant

Dynamic Taint Analysis

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

Is y taited?

Dynamic Taint Analysis


```
x := get_input(
```


```
)
z := 42
y := x + z
goto y
```

Is y taited?

It depends on the selected policy

What's a policy?

- A taint policy specifies three properties:
  - **Taint Introduction**
    - specifies how taint is introduced into a system
    - typically distinguishes between different input sources
  - **Taint Propagation**
    - specifies the taint status for data derived from tainted or untainted operands
  - **Taint Checking**
    - is used to determine the runtime behavior of a program
- **Undertainting** vs **Overtainting**

Forward Symbolic Execution

- Reasoning about the behavior of the program can be reduced to the domain of logic!
- Creating a forward symbolic execution engine is **conceptually** a very simple process

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

Forward Symbolic Execution

- Reasoning about the behavior of the program can be reduced to the domain of logic!
- Creating a forward symbolic execution engine is **conceptually** a very simple process

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

- `get_input(src)` now returns a **symbol** instead of a concrete value
- Expressions involving symbols **cannot** be fully evaluated to a concrete value

Symbolic Memory Addresses

- The `LOAD` and `STORE` rules evaluate the expression representing the memory address to a value
  - that value must be a **non-negative integer** that references a particular memory cell
- What are we supposed to do if the address referenced operation is an expression derived from user input?
  - **Symbolic Memory Address problem**
- **Sound** strategy: consider the instruction for any possible satisfying assignment

Symbolic Memory Addresses

- The `LOAD` and `STORE` rules evaluate the expression representing the memory address to a value
  - that value must be a **non-negative integer** that references a particular memory cell
- What are we supposed to do if the address referenced operation is an expression derived from user input?
  - **Symbolic Memory Address problem**
- **Sound** strategy: consider the instruction for any possible satisfying assignment
- There's even worse: **aliasing**

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

Symbolic Memory Addresses Possible Solutions

-

UNIVERSITÀ DEGLI STUDI DI PADOVA

THANK YOU FOR ALLOWING ME TO TAINT YOUR PRECIOUS TIME!

Questions?

DIPARTIMENTO MATEMATICA