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

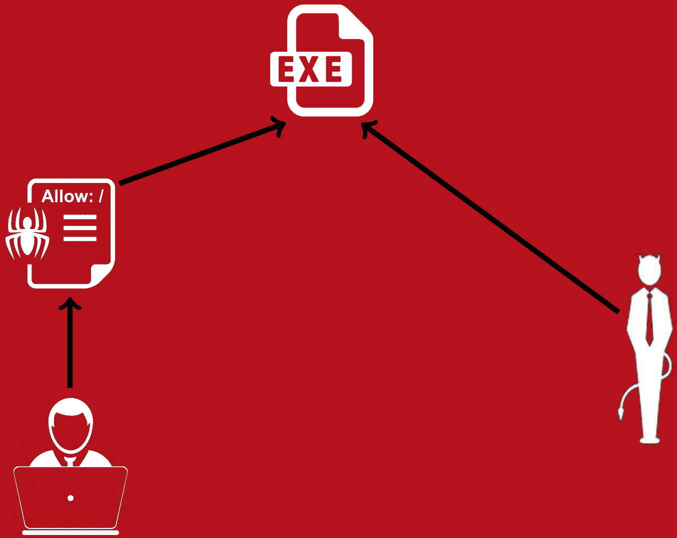
The language

Dynamic Taint Analysis

Forward Symbolic Execution

Conclusions

Of who or what do we trust?



Input Analysis

There are two essential questions about the input analysis:

Input 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

Input 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

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

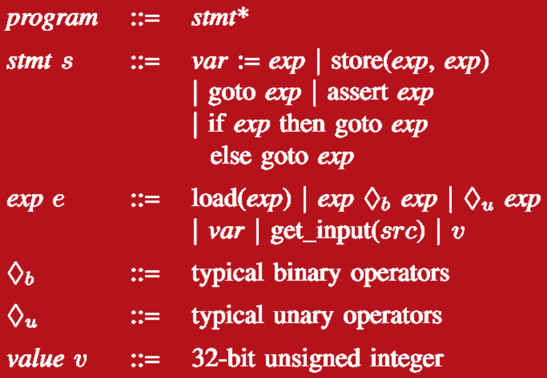
Use cases

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

Designed to demonstrate the critical aspects of this analysis.



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

SimpIL Grammar

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(😈)
z := 42
y := x + z
goto y
```

Untainted

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
```

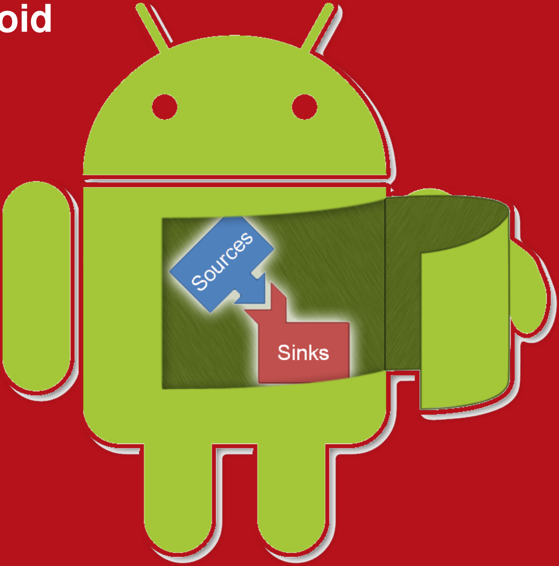
Is y taited?

It depends on the selected policy

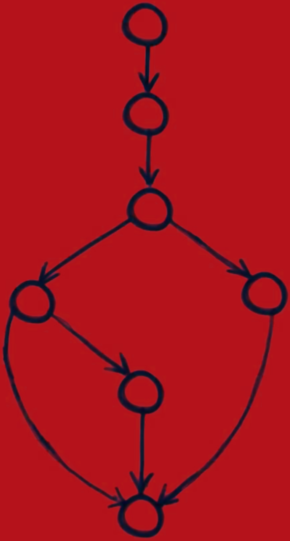
What's a policy?

- A taint policy specifies three properties:
  - **Taint Introduction**
    - How is taint introduced into a system?
  - **Taint Propagation**
    - How does taint propagate into a system?
  - **Taint Checking**
    - Is the current operation secure?
- **Undertainting** vs **Overtainting**

Can we go further? TaintDroid



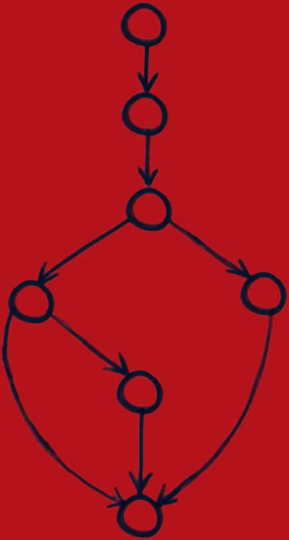
Limitations



- Undertainting and overtainting are nearly **unavoidable**!
  - **Time of detection** vs **Time of attack**
- **Sanitization problem**
- Pure dynamic taint analysis considers **data flows**...

UNIVERSITÀ DEGLI STUDI DI PADOVA

Limitations



- Undertainting and overtainting are nearly **unavoidable**!
  - **Time of detection** vs **Time of attack**
- **Sanitization problem**
- Pure dynamic taint analysis considers **data flows...**
- ...but it ignores **control-flows**
  - What about different security policies for different I/O channels?
    → **Static analysis**

Forward Symbolic Execution

- We can reason about the behavior of a program using the logic...
- ... and it 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

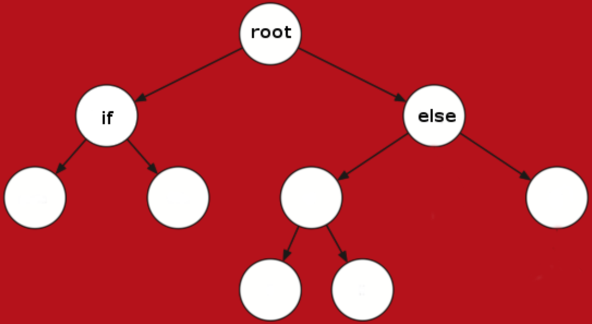
- We can reason about the behavior of a program using the logic...
- ... and it is **conceptually** a very simple process

```
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
- But now expressions **cannot** be fully evaluated to a concrete value

Path Selection and Performance

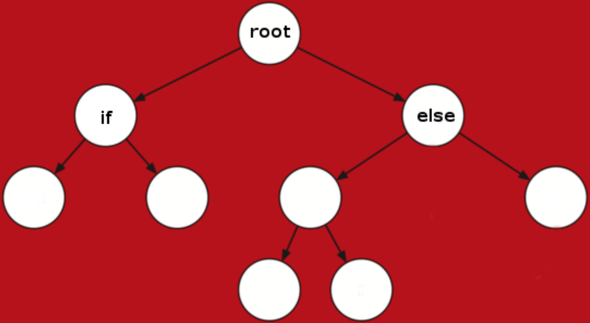
- Every conditional jump we must decide what path to follow first
  - But some path may never terminate
- Exponential blowup due to branches



Path Selection and Performance

- Every conditional jump we must decide what path to follow first
  - But some path may never terminate

    `while` $(3^n + 4^n == 5^n)$ `{n++; ...}`
- Exponential blowup due to branches



Path Selection and Performance

- Every conditional jump we must decide what path to follow first
  - But some path may never terminate

    `while` $(3^n + 4^n == 5^n)$ `{n++; ...}`
- Exponential blowup due to branches
- Solutions
  - **Path Selection Heuristic**
    - Concolic Testing
    - Depth-First or Random Search
  - More and faster **hardware**
  - Identify **redundancies** between formulas
  - Identify **independent subformulas**

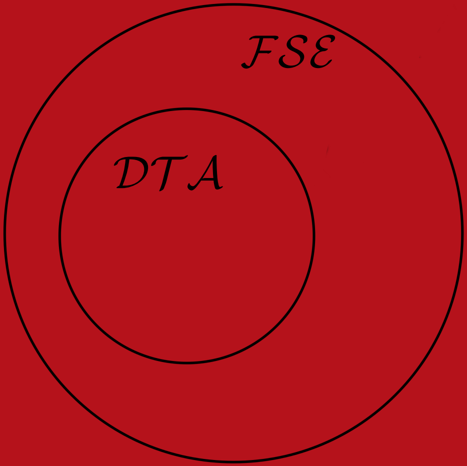
Memory Address Problems

- What are we supposed to do if a referenced address is derived from user input?
  - `LOAD`, `STORE` → **Symbolic Memory Address**
  - `GOTO` → **Symbolic Jumps**
- Solutions
  - Concolic testing
  - SMT (**S**atisfiability **M**odulo **T**heories) solvers
    - $\mathcal{NP}$**-Complete** problem!
  - Static and alias analysis

A small comparison

Dynamic Taint Analysis $\overset{?}{\underset{\supseteq}{\subseteq}}$ Forward Symbolic Execution

A small comparison



Dynamic taint analysis analyzes only ***feasible*** paths

Conclusions

- *v* Conceptually simple methods of analysis
- *v* There are a lot of possible use cases
  - Malware detection and analysis
  - Automatic testing
  - Automatic programs understanding

- *x* Usable with some care
- *x* The effectiveness depends on the application

Thank you for allowing me to taint your time!

Questions?