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

Edward J. Schwartz, Thanassis Avgerinos, David Brumley
Carnegie Mellon University
IEEE Symposium on Security and Privacy, 2010

Collin Jones
Yarden Ne’eman
Program Analysis
Spring 2019
Introduction

- There is a need to monitor the flow of user input in a program.
- This highlights the parts of a program that can be affected by outside input.
- Potential Applications: Security, filters, & test cases
- Two algorithms described in this paper:
  - **Dynamic Taint Analysis (DTA)**
  - **Forward Symbolic Execution (FSE)**
Motivation & Problem Statement

- A program has various sources of input that affect execution
- Mal-intended users can exploit security vulnerabilities & run malicious outside code
- Some code chunks may lead to fatal errors or crashes
- These techniques (namely FSE) can generate preconditions or postconditions
- There was a lack of formalization in these two algorithms
Related Work

- Representative applications of DTA and FSE include:
  - Automatic test case generation (FSE)
  - Automatic filter generation (FSE)
  - Automatic network protocol understanding (DTA)
  - Malware analysis (FSE, DTA)
  - Web applications (DTA)
  - Taint performance & frameworks (DTA)
  - Extensions to taint analysis (DTA)
Key Contributions

▪ SimpIL: **Simple Intermediate Language**
  ▫ An intermediate representation that allows for easy extension to formalize DTA & FSE semantics.

▪ Definition of operational semantics for DTA and FSE
  ▫ Including formalization of taint policies for DTA

▪ Discussion of challenges and opportunities with this and other implementations of DTA and FSE
SimpIL: Simple Intermediate Language

- The goal is to create an easily-parsed intermediate representation powerful enough to encapsulate a variety of languages.
- Can express anything from Java to Assembly with the same meaning.
- Makes some assumptions for simplicity, namely that programs are well-typed and that operands are applied to the proper types.
- Does not include high-level constructs (buffers, etc) but making this extension to SimpIL is trivial.
**SimpIL: Syntax & Contexts**

- A program is a sequence of statements
- Support for both binary and unary operators
- Very simple types (only includes integers)
- Various contexts for mapping during compilation & runtime analysis

![Figure 1: SimpIL syntax](image1)

![Figure 2: SimpIL execution context variables](image2)
DTA - Definitions & Semantics

- Tainted values are denoted by $T$, untainted values are denoted by $F$
- A value can be overtainted (false positive) or undertainted (false negative)
- DTA is considered *precise* if there is no overtainting or undertainting
- Taint status is tracked for both variables and memory cells (i.e. arrays)

Figure 3: SimpIL extensions for DTA

<table>
<thead>
<tr>
<th>taint $t$</th>
<th>::= $T \mid F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>::= $\langle v, t \rangle$</td>
</tr>
<tr>
<td>$\tau_{\Delta}$</td>
<td>::= Maps variables to taint status</td>
</tr>
<tr>
<td>$\tau_{\mu}$</td>
<td>::= Maps addresses to taint status</td>
</tr>
</tbody>
</table>

Figures from *All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution*
DTA - Policies

- A taint policy is defined by three properties
  - Taint introduction: how taint is introduced into a system
  - Taint propagation: how taint is derived for operation arguments
  - Taint checking: how taint is checked during execution
- Different policies are defined for different applications and contexts
- Tainted jump policy focuses on detecting control flow hijacking attacks

<table>
<thead>
<tr>
<th>Component</th>
<th>Policy Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{\text{input}}(\cdot)$, $P_{\text{bincheck}}(\cdot)$, $P_{\text{memcheck}}(\cdot)$</td>
<td>T</td>
</tr>
<tr>
<td>$P_{\text{const}}()$</td>
<td>F</td>
</tr>
<tr>
<td>$P_{\text{unop}}(t)$, $P_{\text{assign}}(t)$</td>
<td>$t$</td>
</tr>
<tr>
<td>$P_{\text{binop}}(t_1, t_2)$</td>
<td>$t_1 \lor t_2$</td>
</tr>
<tr>
<td>$P_{\text{mem}}(t_a, t_v)$</td>
<td>$t_v$</td>
</tr>
<tr>
<td>$P_{\text{condcheck}}(t_e, t_a)$</td>
<td>$\neg t_a$</td>
</tr>
<tr>
<td>$P_{\text{gotocheck}}(t_a)$</td>
<td>$\neg t_a$</td>
</tr>
</tbody>
</table>

Figure 4: Tainted jump policy
DTA - Example

- Below example shows the taint calculations for an example program
  - Recall: $\Delta$ maps variables to their values and $\tau_\Delta$ maps variables to their taint status
- The rules T-ASSIGN and T-GOTO are defined by the operational semantics for SimpIL, modified to enforce a given taint policy $P$

![Table showing taint calculations for a program](image)

Figure 5: Example taint calculations for a program
DTA - Challenges & Opportunities

- Tainted Addresses: User input modifying memory addresses or the data at that address
  - Example: arrays, pointers, etc.
  - Included in tainted jump analysis

- Overtaint & Undertaint
  - Creating precise policies can prove to be challenging

- Time of Detection vs. Time of Attack
  - There is often a delay between the time a value is marked tainted and the time an error is actually raised
FSE - Semantics

- An advantage of FSE is that it can reason about multiple inputs at a time
  - Inputs are grouped into two different classes, those that take the true branch and those that take the false branch

- Getting the input returns a symbol instead of a concrete value

- Expressions involving symbols can’t be fully evaluated to a concrete value

- Branches create constraints based on the path executed

\[
\begin{align*}
  \text{value } v & \::= \text{ 32-bit unsigned integer } \mid \text{ expr} \\
  \Pi & \::= \text{Contains the current constraints on symbolic variables due to path choices}
\end{align*}
\]

Figure 6: SimPL extensions for FSE
FSE - Example

- Below example shows the program contexts after forward symbolic execution
  - Recall: $\Delta$ maps variables to their values and $\Pi$ keeps track of the current constraints on symbolic variables
  - $\Pi$ depends on the path taken through the program

<table>
<thead>
<tr>
<th>Statement</th>
<th>$\Delta$</th>
<th>$\Pi$</th>
<th>Rule</th>
<th>$pc$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>${}$</td>
<td>true</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$x := 2*\text{get_input}()$</td>
<td>${x \rightarrow 2 \times s}$</td>
<td>true</td>
<td>S-ASSIGN</td>
<td>2</td>
</tr>
<tr>
<td>if $x-5 == 14$ goto 3 else goto 4</td>
<td>${x \rightarrow 2 \times s}$</td>
<td>$(2 \times s) - 5 == 14$</td>
<td>S-COND</td>
<td>3</td>
</tr>
<tr>
<td>if $x-5 == 14$ goto 3 else goto 4</td>
<td>${x \rightarrow 2 \times s}$</td>
<td>$-(2 \times s) - 5 == 14$</td>
<td>S-COND</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 7: Simulation of forward symbolic execution
FSE - Challenges & Opportunities

- Symbolic Memory Address Problem
  - Analysis breaks down when memory references are symbolic expressions instead of concrete values

- Path Selection Problem
  - Execution must determine which branch to follow first, but certain choices can lead to infinite loops

- Symbolic Jump Problem
  - A jump target may be an expression instead of a concrete location during execution
FSE - Performance Considerations

- Generic implementation will be exponential in the number of program branches
- Option to use faster hardware and parallelize the solving of formulas
- Option to compact redundancies in formulas and identify independent subformulas
- Alternative to FSE is to use the weakest precondition to calculate the formula
Critique

- Thorough & clear definitions for semantics
- No formal semantic for raising flags / marking operations to raise errors
- SimpIL is missing syntax / semantics for output operations
- Disorganized figures & tables
Extensions

- Output operations
  - Formal separation between different forms of output in SimpIL
- SimpIL type checking
- Addition of high-level constructs to SimpIL
- Semantics to raise an alert based on marked operations
  - If tainted data reaches a marked operation, raise flag or stop execution
Conclusions

- Dynamic analyses are becoming more popular, especially in security contexts

- An intermediate representation, SimpIL, has been defined to target the building blocks necessary for DTA and FSE
  - Including syntax & operational semantics

- Extended operational semantics of SimpIL to define DTA and FSE

- Highlighted some challenges that come from both algorithms
Thanks!

Questions?