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:
  - **D**ynamic **T**aint **A**nalysis (DTA)
  - **F**orward **S**ymbolic **E**xecution (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: **Simp**le **I**ntermediate **L**anguage
  - 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 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

program	::=	*stmt**
stmt s	::=	*var* := *exp* | store(*exp*, *exp*)
		| goto *exp* | assert *exp*
		| if *exp* then goto *exp* else goto *exp*
exp e	::=	load(*exp*) | *exp* $\lozenge_b$ *exp* | $\lozenge_u$ *exp*
		| *var* | get_input(*src*) | v
$\lozenge_b$	::=	typical binary operators
$\lozenge_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Figure 1: SimpIL syntax

Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

Figure 2: SimpIL execution context variables

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

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)

taint t	::=	$\mathbf{T} \mid \mathbf{F}$
value	::=	$\langle v, t \rangle$
τ_Δ	::=	Maps variables to taint status
τ_μ	::=	Maps addresses to taint status

Figure 3: SimpIL extensions for DTA

- DTA is considered *precise* if there is no overtainting or undertainting
- Taint status is tracked for both variables and memory cells (i.e. arrays)

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

Component	Policy Check
$P_{\mathbf{input}}(\cdot)$, $P_{\mathbf{bincheck}}(\cdot)$, $P_{\mathbf{memcheck}}(\cdot)$	**T**
$P_{\mathbf{const}}()$	**F**
$P_{\mathbf{unop}}(t)$, $P_{\mathbf{assign}}(t)$	t
$P_{\mathbf{binop}}(t_1, t_2)$	$t_1 \vee t_2$
$P_{\mathbf{mem}}(t_a, t_v)$	t_v
$P_{\mathbf{condcheck}}(t_e, t_a)$	$\neg t_a$
$P_{\mathbf{gotocheck}}(t_a)$	$\neg t_a$

Figure 4: Tainted jump policy

DTA - Example

- Below example shows the taint calculations for an example program
  - Recall: Δ maps variables to their values and τ_Δ 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*

Line #	Statement	Δ	τ_Δ	Rule	pc
	start	$\{\}$	$\{\}$		1
1	x := 2*get_input(·)	$\{x \rightarrow 40\}$	$\{x \rightarrow \mathbf{T}\}$	T-ASSIGN	2
2	y := 5 + x	$\{x \rightarrow 40, y \rightarrow 45\}$	$\{x \rightarrow \mathbf{T}, y \rightarrow \mathbf{T}\}$	T-ASSIGN	3
3	goto y	$\{x \rightarrow 40, y \rightarrow 45\}$	$\{x \rightarrow \mathbf{T}, y \rightarrow \mathbf{T}\}$	T-GOTO	*error*

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

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

Figure 6: SimpIL extensions for FSE

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

- Branches create constraints based on the path executed

FSE - Example

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

Statement	Δ	Π	Rule	pc
start	$\{\}$	$true$		1
x := 2*get_input(·)	$\{x \to 2 * s\}$	$true$	S-ASSIGN	2
if x-5 == 14 goto 3 else goto 4	$\{x \to 2 * s\}$	$[(2 * s) - 5 == 14]$	S-TCOND	3
if x-5 == 14 goto 3 else goto 4	$\{x \to 2 * s\}$	$\neg[(2 * s) - 5 == 14]$	S-FCOND	4

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?