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

By: Edward J. Schwartz, Thanassis Avgerinos, and David Brumley from Carnegie Mellon University

Presented by: Aamir Mandviwalla and Sharmishtha Dutta
Introduction and Background
Overview

- Simple Intermediate Language (SimpIL)
- Dynamic Taint Analysis
- Dynamic Forward Symbolic Execution
- Common implementation strategies and challenges
Motivation

- The two most common dynamic program analyses used in security
  - Dynamic taint analysis
  - Dynamic Forward Symbolic Execution
- There existed a lack of -
  - Formal definition of these analyses
  - Critical issues that arise when implementing them in a security context
  - Techniques and trade-offs to deal with these issues
Prior related work

- Using operational semantics to define dynamic security mechanisms
  - “Active property checking” by Godefroid et al
  - “Enforceable security policies” by Schneider
  - “CCured: type-safe retrofitting of legacy code” by Necula et al

- Defining a programming language for analysis descriptions
  - “Dynamic test input generation for database applications” by Emmi et al
  - “DART: Directed automated random testing” by Godefroid et al
  - “Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks” by Xu et al
Key Contributions
SimplIL (Simple Intermediate Language)
Operational Semantics of SimpIL

Yuck!
Zooming in

<table>
<thead>
<tr>
<th>Context</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma$</td>
<td>Maps a statement number to a statement</td>
</tr>
<tr>
<td>$\mu$</td>
<td>Maps a memory address to the current value at that address</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>Maps a variable name to its value</td>
</tr>
<tr>
<td>$pc$</td>
<td>The program counter</td>
</tr>
<tr>
<td>$i$</td>
<td>The next instruction</td>
</tr>
</tbody>
</table>

\[
\text{computation} \\
\langle \text{current state} \rangle, \text{stmt} \rightsquigarrow \langle \text{end state} \rangle, \text{stmt}'
\]

\[
\begin{align*}
\mu, \Delta & \vdash e \downarrow v \\
\Delta' & = \Delta[\text{var} \leftarrow v] \\
i & = \Sigma[pc + 1]
\end{align*}
\]

\[
\Sigma, \mu, \Delta, pc, \text{var} := e \rightsquigarrow \Sigma, \mu, \Delta', pc + 1, i
\]

ASSIGN
SimpIL - Example

Just like the tree method we’ve been using in class!
Dynamic Taint Analysis
Dynamic Taint Analysis - Semantics

\[ t \text{aint } t \ ::= \ T \mid F \]
\[ \text{value } ::= \langle v, t \rangle \]
\[ \tau_\Delta ::= \text{Maps variables to taint status} \]
\[ \tau_\mu ::= \text{Maps addresses to taint status} \]

<table>
<thead>
<tr>
<th>Context</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Sigma )</td>
<td>Maps a statement number to a statement</td>
</tr>
<tr>
<td>( \mu )</td>
<td>Maps a memory address to the current value at that address</td>
</tr>
<tr>
<td>( \Delta )</td>
<td>Maps a variable name to its value</td>
</tr>
<tr>
<td>( pc )</td>
<td>The program counter</td>
</tr>
<tr>
<td>( t )</td>
<td>The next instruction</td>
</tr>
</tbody>
</table>

\[ \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t \rangle \quad \Delta' = \Delta[\text{var} \leftarrow v] \quad \tau'_\Delta = \tau_\Delta[\text{var} \leftarrow P_{\text{assign}}(t)] \quad \iota = \Sigma[pc + 1] \quad \text{T-ASSIGN} \]
Dynamic Taint Analysis - Taint Policy

- Inputs are always treated as tainted
- Tainted values propagate

<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>$P_{\text{condcheck}}(t_c, t_a)$</td>
<td>$\neg t_a$</td>
</tr>
<tr>
<td>$P_{\text{gotocheck}}(t_a)$</td>
<td>$\neg t_a$</td>
</tr>
</tbody>
</table>
Dynamic Taint Analysis - Example

- get_input(.) returns 20 and treats it as a tainted value
- Hence x is tainted
- Hence y is tainted
- Since y is tainted, the goto operation fails
Dynamic Taint Analysis - Challenges

- **Over- and under-tainting**
  - Not all inputs need to be identified as taint sources
  - But we also shouldn’t neglect any taint sources
  - Goal: precision

- **Tainted addresses**
  - Need to keep track of memory addresses of tainted values

- **Sanitization**
  - When can we ignore/remove taint?

- **Control-flow taint**
  - Dynamic analysis cannot analyze blocks that do not run

- **Time of detection vs. Time of Attack**
  - What if I detect an error too late?
  - Integer overflow attacks
Dynamic Taint Analysis - Solutions

- **Over- and under-tainting**
  - Treat only inputs from untrusted channels as taint sources

- **Sanitization**
  - Treat constant functions like \( b = a \ xor \ a \) as untainted
  - Treat cryptographically secure functions as untainted

- **Control-flow taint**
  - Combine with static analysis

- **Tainted addresses, integer overflow**:

<table>
<thead>
<tr>
<th>Policy</th>
<th>Substitutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tainted Value</td>
<td>( P_{\text{mem}}(t_a, t_v) \equiv t_v )</td>
</tr>
<tr>
<td>Tainted Addresses</td>
<td>( P_{\text{mem}}(t_a, t_v) \equiv t_a \lor t_v )</td>
</tr>
<tr>
<td>Tainted Overflow</td>
<td>( P_{\text{bincheck}}(t_1, t_2, v_1, v_2, \Diamond_b) \equiv (t_1 \lor t_2) \Rightarrow \neg \text{overflows}(v_1 \Diamond_b v_2) )</td>
</tr>
</tbody>
</table>
Dynamic Forward Symbolic Execution
Dynamic forward symbolic execution - Semantics

\[ \text{value } v \quad ::= \quad \text{32-bit unsigned integer } | \exp \]

\[ \Pi \quad ::= \quad \text{Contains the current constraints on symbolic variables due to path choices} \]

<table>
<thead>
<tr>
<th>Context</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Sigma )</td>
<td>Maps a statement number to a statement</td>
</tr>
<tr>
<td>( \mu )</td>
<td>Maps a memory address to the current value at that address</td>
</tr>
<tr>
<td>( \Delta )</td>
<td>Maps a variable name to its value</td>
</tr>
<tr>
<td>( pc )</td>
<td>The program counter</td>
</tr>
<tr>
<td>( \nu )</td>
<td>The next instruction</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\mu, \Delta \vdash e \Downarrow e' & \quad \Delta \vdash e_1 \Downarrow v_1 \\
\Pi' = \Pi \land (e' = 1) & \quad \nu = \Sigma[v_1] \quad \text{S-TCOND}
\end{align*}
\]

\[ \Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \sim \Pi', \Sigma, \mu, \Delta, v_1, \nu \]
Dynamic forward symbolic execution - Example

1. \( X \) is stored as an expression in \( \Delta \)
2. If the condition is true, a constraint is added to \( \Pi \), \( \Delta \) remains unchanged
3. If the condition is false, a different constraint is added to \( \Pi \), \( \Delta \) remains same

<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><code>{}</code></td>
<td><code>true</code></td>
<td>S-ASSIGN</td>
<td>1</td>
</tr>
<tr>
<td>( x := 2 \times \text{get_input}() )</td>
<td>( {x \to 2 \times s} )</td>
<td><code>true</code></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>if ( x-5 == 14 ) goto 3 else goto 4</td>
<td>( {x \to 2 \times s} )</td>
<td>( (2 \times s) - 5 == 14 )</td>
<td>S-TCOND</td>
<td>3</td>
</tr>
<tr>
<td>if ( x-5 == 14 ) goto 3 else goto 4</td>
<td>( {x \to 2 \times s} )</td>
<td>( -(2 \times s) - 5 == 14 )</td>
<td>S-FCOND</td>
<td>4</td>
</tr>
</tbody>
</table>
Dynamic forward symbolic execution - Challenges

- **Symbolic Memory**
  - What should be done when the analysis uses the context — whose index must be a non-negative integer — with a symbolic index?

- **System and Library Calls**
  - How should an analysis deal with external interfaces such as system calls?

- **Path Selection**
  - Needs a strategy for choosing which state to explore next

- **Symbolic jumps**
  - The jump target may be an expression instead of a concrete location

- **Performance**
  - a running time exponential in the number of program branches
  - an exponential number of formulas
  - an exponentially-sized formula per branch
Dynamic forward symbolic execution - Solutions

- **Symbolic Memory**
  - Remove symbolic addresses from programs
    - Appropriateness varies depending on the overall application domain
  - Use an SMT solver on the generated formulas
  - Perform alias analysis
    - Offline/static

- **System and Library Calls**
  - Create summaries of their side effects
    - Summaries abstract only those details necessary for the application domain at hand
    - Need to be generated manually
  - Using concolic execution
    - Use values returned from system calls on previous concrete execution
    - Simple, easy to implement
    - Some calls do not return the same result given the same input e.g., `gettimeofday()`
Dynamic forward symbolic execution - Solutions

- Path Selection
  - Depth-First Search
    - can get stuck in loops with symbolic conditions if no maximum depth is specified
  - Concolic Testing
  - Random Paths
  - Heuristics

- Symbolic Jumps
  - Use concrete and symbolic (concolic) analysis
    - perform symbolic execution of the concrete path
    - code coverage can suffer
  - Use an SMT solver
    - Does not take advantage of program structure
  - Use static analysis
    - can reason about the entire program to locate possible jump targets
Dynamic forward symbolic execution - Solutions

- **Performance**
  - Use more and faster hardware
  - Assign each variable a unique name
  - Identify redundancies between formulas and make them more compact
    - Use heuristics
    - Use post-hoc analysis
  - Identify independent subformulas and implement caching in SMT solver
  - Use Weakest Precondition to calculate the formulas

- **In general**
  - Practice Mixed execution - Allow some inputs to be concrete and others symbolic
Critique
Critique - SimpIL

“We consider only expressions (constants, variables, etc.) that evaluate to 32-bit integer values…”

“We do not include some high-level language constructs such as functions or scopes…”

“The implication is that our operational semantics do not allow programs with dynamically generated code…”

“We omit the type-checking semantics of our language and assume things are well-typed…”
Critique - Dynamic Forward Symbolic Execution

- Positive
  - Presents a good framework for the analysis
  - Does a good job of describing the challenges and some solutions
  - Cites relevant existing systems and mention how they handle these issues
Critique - Dynamic Forward Symbolic Execution

- **Negative**
  - Does not mention
    - the usefulness of the operational semantics based formulation of the forward symbolic execution of SimpIL
    - an evaluation the operational semantics of SimpIL compared to the rest
    - how security analysis lead to some unique challenges in this area
  - It is unclear how the paper adds to
    - the understanding of the use of forward symbolic execution for security analysis
  - SimpIL does not allow
    - Mixed execution
    - Mechanism for path selection
Critique - Dynamic Taint Analysis

● Positive
  ○ Simple! Easy to describe while also being logically formalized
  ○ Highly adaptable to different situations - can make small modifications to fix many problems

● Negative
  ○ Incomplete - Needs aforementioned modifications to deal with common problems
  ○ Not all problems discussed in the paper have solutions provided:
    ■ Time of detection vs. Time of attack
    ■ Over- and under-tainting
  ○ Some solutions are lacking SimpIL definitions:
    ■ Sanitization - removal of taint
    ■ Control-flow taint
Critique - Symbol Choices

- Sigma (Σ) is usually used for summations
- Delta (Δ) is usually used for... deltas (difference/net change)
- Mu (µ) is the prefix for micro - 10^{-6}
- The authors of the paper decided to use all three of these as data structures... why?!?
Impact

- Published in 2010 - 708 citations

- Efficient and Safe Control Flow Recovery Using a Restricted Intermediate Language
  - Used SimpIL to model control-flow taint and present a solution

- A Semantics-Based Approach to Concept Assignment in Assembly Code
  - Used SimpIL to formalize another topic
Impact

- A Critical Review of Dynamic Taint Analysis and Forward Symbolic Execution
  - Used the Paul Elder critical thinking framework to analyze the paper, and the framework itself

- Enhancing Symbolic Execution of Heap-based Programs with Separation Logic for Test Input Generation
  - Used SimPL to build a symbolic execution engine
References

- https://www.usenix.org/legacy/event/sec06/tech/full_papers/xu/xu_html/
- https://www.researchgate.net/publication/323881042_A_Semantics-Based_Approach_to_Concept_Assignment_in_Assembly_Code
Thank You :)}