Loop-Extended Symbolic Execution on Binary Programs

Prateek Saxena*  Pongsin Poosankam‡*

Stephen McCamant*  Dawn Song*

* UC Berkeley
‡ Carnegie Mellon University
Authors

• **Prateek Saxena**
  - A graduate student at UC Berkeley. Advised by Dawn Song. Areas of interest are program analysis, system security, web security and operating systems.

• Recent Papers
  
  - **Efficient fine-grained binary instrumentation with applications to taint-tracking** in *International Symposium on Code Generation and Optimization (CGO)*, April 2008
  
Authors

- **Pongsin Poosankam**

- **Recent Papers**
Authors

- **Stephen McCamant**
  - A postdoc at UC Berkeley. Now working with BitBlaze research group

- **Recent Papers**
Conference

- **ISSTA 2009**

  - Related Papers
    - Precise pointer reasoning for dynamic test generation
      
      *Bassem Elkarablieh, Patrice Godefroid, Michael Y. Levin*
    
    - Penumbra: automatically identifying failure-relevant inputs using dynamic tainting
      
      *James Clause, Alessandro Orso*
Dynamic Symbolic Execution

• Combines concrete execution with symbolic execution
• Has important applications
  • Program Testing and Analysis
    – Automatic test case generation (DART, PEX, EXE, KLEE, SAGE)
    – Given an initial test case, find a variant that executes a different path
  • Computer Security
    – Vulnerability Discovery & Exploit Generation
      – Given an initial benign test case, find a variant that triggers a bug
    – Vulnerability Diagnosis & Signature Generation
      – Given an initial exploit for a vulnerability, find a set of conditions necessary to trigger it
Limitations of Previous Approach

Single-Path Symbolic Execution (SPSE)

Ineffective for loops!

Concrete Execution

Symbolic Execution
Contributions of Our Work

• **Loop-Extended Symbolic Execution** (LESE)
  - Generalizes symbolic reasoning to loops

  - Applicable directly to binaries
  - Demonstrate its effectiveness in an important security application
    - Buffer overflow diagnosis & discovery
    - Show scalability for practical real-world examples
Talk Outline

• Motivation
• Overview of LESE
• Technical Approach
• Results and Applications
• Conclusion & Related Work
Motivation: A HTTP Server Example

- Input
  
  ```
  GET /index.html HTTP/1.1
  ```

  - CMD
  - URL
  - VERSION

  ```
  void process_request (char* input) {
    char URL [1024];
    ...
    for (ptr = 4; input[ptr] != ' '; ptr++)
      urlLen ++;
    ...
    for (i = 0, p = 4; i < urlLen; i++) {
      URL[i] = input[p++];
    }
  }
  ```

  - Calculating length
  - Copying URL to buffer
Motivation: A HTTP Server Example

• Goal: Check if the buffer can be overflowed

```c
void process_request (char* input) {
    char URL [1024];
    ... 
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    ... 
    for (i = 0, p = 4; i < urlLen; i++)    {
        ASSERT (i < 1024);  
        URL [i] = input [p++];
    }
}
```
Motivation: A HTTP Server Example

void process_request (char* input) {
    char URL [1024];
    ... 
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    ... 
    for (i = 0, p = 4; i < urlLen; i++) {
        ASSERT (i < 1024);
        URL [i] = input [p++];
    }
}
Overview

• LESE: Finds an exploit for the example in 1 step
• Key Point: Summarize loop effects

• Intuition: Why was ‘i’ not symbolic?
  – SPSE only tracks data dependencies
  – ‘i’ was loop dependent

• Model loop-dependencies in addition to data dependencies
Practical Requirements

• Applicable to binary code
  – Works for closed-sourced security-critical applications

• Should be simple
  – Scale to real-world programs

• Should not require manual function summaries
  – Need to reason about custom input processing
  – Hard to write summaries for certain functions (like sprintf)
Approach

• Introduce a symbolic “trip count” for each loop
  – Symbolic variable representing the number of times a loop executes

• LESE has 2 steps
  – STEP 1: Derive relationship between program variables and trip counts
    » Linear Relationships
  – STEP 2: Relate trip counts to inputs
Introducing Symbolic Trip Counts

- Introduces symbolic loop *trip counts*

```c
void process_request (char* input) {
    char URL [1024];
    ...
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    ...
    for (i = 0, p = 4; i < urlLen; i++) {
        ASSERT (i < 1024);
        URL [i] = input [p++];
    }
}
```
Step 1: Relating program variables to TCs

- Links trip counts to program variables

```c
void process_request (char* input) {
    char URL [1024];
    ...
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    ...
    for (i = 0, p = 4; i < urlLen; i++)
        {
            ASSERT (i < 1024);
            URL [i] = input [p++];
        }
}
```

### Symbolic Constraints

- `ptr = 4 + TC_{L1}`
- `urlLen = 0 + TC_{L1}`
- `(i < urlLen)`
- `i = -1 + TC_{L2}`
- `p = 4 + TC_{L2}`
Step 2: Relating Trip Counts to Input

- **Inputs**
  - Initial Concrete Test Case
  - A Grammar
    - Fields
    - Delimiters

- Implicitly models symbolic attributes for fields
  - Lengths of fields
  - Counts of repeated elements

- Available from off-the-shelf tools
  - Network application grammars in Wireshark, GAPA
  - Media file formats in Hachoir, GAPA
  - Can even be automatically inferred [CCS07,S&P09]
Step 2: Link trip counts to input

- Link trip counts to the input grammar

```c
void process_request (char* input) {
    char URL [1024];
    ...
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    ...
    for (i = 0, p = 4; i < urlLen; i++) {
        ASSERT (i < 1024);
        URL [i] = input [p++];
    }
}
```

Symbolic Constraints

\[
\begin{align*}
(F_{\text{uri}} [0] \neq ' ') & \quad \& \quad (F_{\text{uri}} [1] \neq ' ') & \quad \& \quad 
\ldots & \quad (F_{\text{uri}} [12] == ' ') \\
\end{align*}
\]

\[\mathcal{G}\]

\[\text{Len}(F_{\text{URL}}) == TC_{L1}\]
Solve using a decision procedure

• Link trip counts to the input grammar

```c
void process_request (char* input) {
    char URL [1024];
    ...
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    ...
    for (i = 0, p = 4; i < urlLen; i++)    {
        ASSERT (i < 1024);
        URL [i] = input [p++];
    }
}
```
Solution: HTTP Server Example

• Solve constraints

```c
void process_request (char* input) {
    char URL [1024];
    ...
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    ...
    for (i = 0, p = 4; i < urlLen; i++)    {
        ASSERT (i < 1024);
        URL [i] = input [p++];
    }
}
```

Exploit Condition

\[ \text{Len} (F_{URL}) > 1024 \]

GET aaa...
(1025 times)
Challenges

• Problems:
  – Identifying loop dependencies on binaries
    » Syntactic induction variable analysis insufficient
  – Capturing the inter-dependence between two loops
    » An induction variable of may influence trip counts of subsequent loops

• Our Solution
  – Dynamic abstract interpretation of x86 machine code
  – Symbolic memory model for buffer pointers only
Experimental Setup

Initial Test Case
Program

LESE

Decision Procedure (STP)

No Error

Validation

Candidate Exploits
Results (I): Vulnerability Discovery

- On 14 benchmark applications (MIT Lincoln Labs)
  - Created from historic buffer overflows (BIND, sendmail, wuftp)
- Found 1 or more vulnerabilities in each benchmark
  - 1 new exploit location in sendmail 7 benchmark
Results (II): Real-world Vulnerabilities

• Diagnosis and Discovery 3 Real-world Case Studies
  – SQL Server Resolution [Slammer Worm 2003]
  – GDI Windows Library [MS07-046]
  – Gaztek HTTP web Server

• Diagnosis Results
  – Results precise and field level

• Discovery Results: Found 4 buffer overflows in 6 candidates
  – 1 new exploit location for Gaztek HTTP server

<table>
<thead>
<tr>
<th>Program</th>
<th>Buffer size (bytes)</th>
<th>Condition for overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHttpd (1)</td>
<td>220</td>
<td>URI.len &gt; 172</td>
</tr>
<tr>
<td>GHttpd (2)</td>
<td>208</td>
<td>URI.len &gt; 133</td>
</tr>
<tr>
<td>SQL Server</td>
<td>128</td>
<td>DBName.len &gt; 64</td>
</tr>
<tr>
<td>GDI</td>
<td>4096</td>
<td>(2*INP[19:18]) ≫ 2 &lt; 0</td>
</tr>
</tbody>
</table>
Results (III): Loop statistics

- Provides new symbolic conditions
  - Loop Conditions
Conclusion

• LESE is a generalization of SPSE
  – Captures effect of program inputs on loops
  – Summarizes the effect of loops on program variables

• Works for real-world Windows and Linux binaries

• Key enabler for several applications
  – Buffer overflow discovery and diagnosis
    » Capable of finding new bugs
    » Does not require manual function summaries
Reference

-- J. Caballero, H. Yin, Z. Liang, and D. Song.
   **Polyglot: Automatic extraction of protocol message format using dynamic binary analysis.**

-- D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song..
   **Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation.**
   *In USENIX Security Symposium, Aug. 2007.*

-- D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha..
   **Towards automatic generation of vulnerability-based signatures.**

-- V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek
   **Buffer overrun detection using linear programming and static analysis.**

-- R.-G. Xu, P. Godefroid, and R. Majumdar.
   **Testing for buffer overflows with length abstraction.**
   *In International Symposium on Software Testing and Analysis, July 2008.*
Thanks…
Any Questions, comments, or feedback?