What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices

Marius Muench\textsuperscript{1} \quad Jan Stijohann\textsuperscript{2,3} \\
Frank Kargl\textsuperscript{3} \quad Aurélien Francillon\textsuperscript{1} \quad Davide Balzarotti\textsuperscript{1}

\textsuperscript{1}EURECOM \quad \textsuperscript{2}Siemens AG \quad \textsuperscript{3}Ulm University
Introduction

• Embedded devices are becoming increasingly more important
• Vulnerabilities go beyond misconfigurations, weak authentication, hard-coded keys, etc.
• Fuzz testing is a popular and effective method for uncovering *programming errors*
  • A variety of work improves input generation and fault detection for fuzzing
How efficient are we at fuzzing embedded devices?
Can we do it better?
Fuzzing, Corruptions & Crashes
Corruption $\neq$ Crash
Embedded Devices: A minimalistic classification

**Type-I:**
General purpose OS-based

**Type-II:**
Embedded OS-based

**Type-III:**
No OS-Abstraction
Challenge #1: Fault Detection

- Lack of basic features, such as:
  - Memory Management Unit (MMU)
  - Heap consistency checks
  - Canaries

- Often only solution: Basic liveness checks
Challenge #2: Performance & Scalability

- Fuzzing greatly benefits from parallelization
  - This would mean 1 device per instance
- Frequent restarts are required
  - Fast for software, slow for full systems
Challenge #3: Instrumentation

- Hard to retrieve coverage information
- Tools for turning *silent* corruptions into observable ones rarely available
  - Unsupported instruction set architecturess
  - Operation tied to OS-specific features
Measuring the effect of memory corruptions

- Five common types of memory corruptions
- Insertion of artificial bugs in two popular open source programs
  - Expat
  - mbedTLS
- Trigger condition inspired by LAVA [1]
- Vulnerable programs are compiled for four different devices

## Effects of Corruptions across different systems

<table>
<thead>
<tr>
<th>Platform</th>
<th>Desktop</th>
<th>Type-I</th>
<th>Type-II</th>
<th>Type-III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format String</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Stack-based buffer overflow</td>
<td>✓</td>
<td>✓</td>
<td>✓ (opaque)</td>
<td>! (hang)</td>
</tr>
<tr>
<td>Heap-based buffer overflow</td>
<td>✓</td>
<td>! (late crash)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Double Free</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓ (malfunc.)</td>
</tr>
<tr>
<td>Null Pointer Dereference</td>
<td>✓</td>
<td>✓</td>
<td>✓ (reboot)</td>
<td>(malfunc.)</td>
</tr>
</tbody>
</table>
Possible Directions for Improvement

- Static Instrumentation
- Binary Rewriting
- Physical Re-Hosting
- Full Emulation
- Partial Emulation
- Hardware-Supported Instrumentation
Possible Directions for Improvement

- Static Instrumentation
- Binary Rewriting
- Physical Re-Hosting
- Full Emulation
- Partial Emulation
- Hardware-Supported Instrumentation
Leveraging (partial) emulation to improve fuzz testing
Set-up: Overview

Figure 1: Setup for fuzzing utilizing partial emulation

Code will be available at: https://github.com/avatartwo/ndss18_wycinwyc
Set-up: Target

- The vulnerable expat program, as seen in the last part
- Focus on a Type-III device
- Fuzzed in four different configurations
Set-up: Native

1. Native (NAT)
2. Partial Emulation with Memory Forwarding (PE/MF)
3. Partial Emulation with Peripheral Modeling (PE/PM)
4. Full Emulation (FE)
1. Native (NAT)
2. Partial Emulation with Memory Forwarding (PE/MF)
3. Partial Emulation with Peripheral Modeling (PE/PM)
4. Full Emulation (FE)
Set-up: PE/PM

1. Native (NAT)
2. Partial Emulation with Memory Forwarding (PE/MF)
3. Partial Emulation with Peripheral Modeling (PE/PM)
4. Full Emulation (FE)
1. Native (NAT)
2. Partial Emulation with Memory Forwarding (PE/MF)
3. Partial Emulation with Peripheral Modeling (PE/PM)
4. Full Emulation (FE)
Set-up: Fuzzer

- `boofuzz [2]`, a python-based fuzzer based on Sulley
- Configured to trigger the corruptions with different ratios
- Used for 100 fuzzing sessions over one hour each

Set-up: Corruption Detection

1. Native (NAT)
2. Partial Emulation with Memory Forwarding (PE/MF)
3. Partial Emulation with Peripheral Modeling (PE/PM)
4. Full Emulation (FE)
Set-up: Corruption Detection

1. Native (NAT)
2. Partial Emulation with Memory Forwarding (PE/MF)
3. Partial Emulation with Peripheral Modeling (PE/PM)
4. Full Emulation (FE)
Set-up: Corruption detection

• 6 simple heuristics, monitoring the execution:
  1. Segment Tracking
  2. Format Specifier Tracking
  3. Heap Object Tracking
  4. Call Stack Tracking
  5. Call Frame Tracking
  6. Stack Object Tracking
Measuring Fuzzing Throughput

No Heuristics:
- Native
- Partial Emulation/Memory Forwarding
- Partial Emulation/Peripheral Modeling
- Full Emulation

Combined Heuristics:
- Partial Emulation/Memory Forwarding'
- Partial Emulation/Peripheral Modeling'
- Full Emulation'

Graph showing the relationship between Corruption Ratio [%] and #Inputs for different heuristics.
Discussion, Future Work & Conclusion
Insights from the experiments

- Liveness checks only is a poor strategy
- Full emulation is good - but rarely possible
- Partial emulation can already help
  - But introduces significant performance overhead
Limitations and Future Work

- We focused on improving fault detection
  - Other challenges of fuzzing (e.g., input generation) not addressed in this work
- Our experiments focused on artificial vulnerabilities
  - Good for improving our initial understanding
- We investigated solutions based on partial emulation
  - Other approaches still open for research
Conclusion

- Fuzzing embedded devices requires a paradigm shift
- (Partial) emulation can improve fault detection
  - We need good emulators
- Fuzzing of embedded devices needs more investigation