HART: Hardware-assisted Modular Tracing on ARM

Yunlan Du\textsuperscript{1,*}, Zhenyu Ning\textsuperscript{2,*}, Jun Xu\textsuperscript{3}, Zhilong Wang\textsuperscript{4}, Yueh-Hsun Lin\textsuperscript{5}, Fengwei Zhang\textsuperscript{2}, Xinyu Xing\textsuperscript{4}, Bing Mao\textsuperscript{1}

\textsuperscript{1}Nanjing University
\textsuperscript{2}Southern University of Science and Technology
\textsuperscript{3}Stevens Institute of Technology
\textsuperscript{4}Pennsylvania State University
\textsuperscript{5}JD Silicon Valley R&D Center

*These authors contributed equally to this work

ESORICS, Sep 15, 2020
Outline

▶ Introduction
▶ Background
▶ HART: Hardware-Assisted Runtime Tracing framework
▶ HASAN: HART-based Address Sanitizer
▶ Evaluation
▶ Conclusion
Outline

- Introduction
- Background
- HART: Hardware-Assisted Runtime Tracing framework
- HASAN: HART-based Address Sanitizer
- Evaluation
- Conclusion
The vulnerabilities in kernel modules have been a serious threat for the security of the Linux kernel.

- Caused by lacking of code correctness and testing rigorousness

- CVE patches to kernel drivers comprise roughly of 19% commits from 2005 to 2017 [1, 2].

- In 2017, 41% of 660 collected bugs in Android ecosystem came from kernel components most of which were device drivers [3].
Introduction

- To solve the problem, many solutions have been proposed

<table>
<thead>
<tr>
<th>Approach Category</th>
<th>Representative Works (in the Order of Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Debugger</td>
<td>Slub_debug [4], Kmemeak [5], Kmemcheck [6], KASAN [7]</td>
</tr>
<tr>
<td>Integrity Protection</td>
<td>KOP [8], HyperSafe [9], HUKO [10], KCoFI [11], DFI for kernel [12]</td>
</tr>
<tr>
<td>Kernel Isolation</td>
<td>Nooks [13], SUD [14], Livewire [15], SafeDrive [16], SecVisor [17]</td>
</tr>
</tbody>
</table>

Table: Existing kernel protection works.
Introduction

- To solve the problem, many solutions have been proposed
- But, the problem is far from solved

<table>
<thead>
<tr>
<th>Approach Category</th>
<th>Binary -support</th>
<th>Non-intrusive</th>
<th>Low overhead</th>
<th>Representative Works (in the Order of Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Debugger</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Slub_debug [4], Kmemleak [5], Kmemcheck [6], KASAN [7]</td>
</tr>
<tr>
<td>Integrity Protection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>KOP [8], HyperSafe [9], HUKO [10], KCoFI [11], DFI for kernel [12]</td>
</tr>
<tr>
<td>Kernel Isolation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Nooks [13], SUD [14], Livewire [15], SafeDrive [16], SecVisor [17]</td>
</tr>
</tbody>
</table>

Table: Existing kernel protection works.
(✓ = yes, X = no, * = partially supported.)
Motivation: Build a high-performance tracing framework for unmodified kernel modules without module source code
Outline

▶ Introduction

▶ Background

▶ HART: Hardware-Assisted Runtime Tracing framework

▶ HASAN: HART-based Address Sanitizer

▶ Evaluation

▶ Conclusion
Embedded Trace Macrocell

Embedded Trace Macrocell (ETM) is a hardware component on Arm processors. It is able to tracing the instruction execution and memory access with negligible overhead.

Figure: A general hardware model of ETM.
Outline

- Introduction
- Background
- **HART: Hardware-Assisted Runtime Tracing framework**
- HASAN: HART-based Address Sanitizer
- Evaluation
- Conclusion
Hardware-Assisted Runtime Tracing framework

- HART, a **Hardware-Assisted Runtime Tracing** framework

**Figure:** Architecture of HART framework.
Selective Tracing

**Challenge 1: Selective Tracing**

- As a hardware component, ETM is lacking of OS semantics
  - Filters in ETM are limited
  - Hard to identify the trace of target module from the output
- Size of trace buffer is limited
  - Tracing the entire execution in the processor leads to frequent overflow
  - To trace the other components in the system is a waste of resource
Selective Tracing

Solution: Selective Tracing via hooking and wrapping

- Hook the entrances and exits during the module loading stage
  - Achieved by callbacks registered via trace-point, without intrusion to the kernel
- Replace entrances and exits with wrappers at relocation stage
  - Including code points in .data and .text segments

(a) Relocation of internal function references

(b) Relocation of external calls
Continuous Tracing

Challenge 2: Continuous Tracing

- The size of the trace buffer in SoCs are limited
  - According to our observation, normally 4k trace buffer is implemented
  - Could be fully occupied in milliseconds or seconds
- The overflow of the trace buffer leads to losing of trace
  - The trace buffer is a ring buffer
  - Older trace data will be overridden after overflow
Continuous Tracing

**Solution:** Continuous Tracing via timely interrupts

- Leverage PMU to issue an interrupt before overflow
  - In general, at most 6 byte trace data per instruction
  - We make 670 instructions as the threshold, and issue an interrupt after every 670 instructions are executed
- During the interrupt, validate and extract the trace with careful designed algorithm
High-performance Tracing

**Challenge 3: High-performance Tracing**

- The overhead of ETM tracing is negligible
- But, it takes performance to handle the trace
  - Extracting data from the trace buffer
  - Decoding the trace data
High-performance Tracing

**Solution:** High-performance Tracing via elastic decoding

- A dedicated decoding thread

- Yielding CPU based on the workload of the decoding thread
  - Calculating extracted data size

- To yield according to the data size
HASAN: a HART-based address sanitizer, reusing the scheme of AddressSanitizer [18]

- Redzones for out of bound detection
  - Wrapping objects with redzones
  - Accessing the redzones leads to fault

- Shadow memory for memory tags
  - 0xbf000000 to 0xffffffff as kernel space in our system
  - Allocate 130M continuous virtual space as shadow memory
HART-based Address Sanitizer

**HASAN**: a HART-based address sanitizer.

- *With module source code*:
  - Both HASAN and KASAN can achieve heap & stack protection

- *Without module source code*:
  - HASAN achieves heap protection
  - KASAN would not work at all
HART-based Address Sanitizer

Heap protection without module source code

- Achieved by hooking the slab interfaces for memory management

<table>
<thead>
<tr>
<th>Category</th>
<th>Allocation</th>
<th>De-allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kmem_cache</td>
<td>kmem_cache_alloc, kmem_cache_create</td>
<td>kmem_cache_free, kmem_cache_destroy</td>
</tr>
<tr>
<td>Kmalloc</td>
<td>kmalloc, krealloc, kzalloc, kcalloc</td>
<td>kfree</td>
</tr>
<tr>
<td>Page operations</td>
<td>alloc_pages, __get_free_pages</td>
<td>free_pages, __free_pages</td>
</tr>
</tbody>
</table>

**Table:** Memory management interfaces HASAN hooked.
Outline

▶ Introduction
▶ Background
▶ HART: Hardware-Assisted Runtime Tracing framework
▶ HASAN: HART-based Address Sanitizer
▶ Evaluation
▶ Conclusion
Evaluation

Experiment setup:

- Freescale i.MX53 Quick Start Board

- Raspberry Pi 3+ for KASAN
  - We implement HASAN in 32-bit i.MX53 QSB, but KASAN only support 64-bit systems

- lmbench, and 6 widely-used kernel modules with standard benchmarks
Overhead to the main kernel

<table>
<thead>
<tr>
<th>Func.</th>
<th>Setting</th>
<th>Native</th>
<th>KASAN</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes (ms)</td>
<td>stat</td>
<td>3.08</td>
<td>16.4</td>
<td>5.3</td>
</tr>
<tr>
<td></td>
<td>open clos</td>
<td>8.33</td>
<td>36.7</td>
<td>4.4</td>
</tr>
<tr>
<td></td>
<td>sig hndl</td>
<td>6.06</td>
<td>20.4</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td>fork proc</td>
<td>472</td>
<td>1940</td>
<td>4.1</td>
</tr>
<tr>
<td>Local Comm.</td>
<td>Pipe</td>
<td>18.9</td>
<td>45.8</td>
<td>2.4</td>
</tr>
<tr>
<td>latency (ms)</td>
<td>AF UNIX</td>
<td>26.6</td>
<td>97.9</td>
<td>3.7</td>
</tr>
<tr>
<td></td>
<td>UDP</td>
<td>41.4</td>
<td>127.6</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td>53.4</td>
<td>176.4</td>
<td>3.3</td>
</tr>
<tr>
<td>File &amp; VM</td>
<td>0K File Create</td>
<td>44.0</td>
<td>136.1</td>
<td>3.1</td>
</tr>
<tr>
<td>system</td>
<td>0K File Delete</td>
<td>35.2</td>
<td>227.1</td>
<td>6.5</td>
</tr>
<tr>
<td>latency (ms)</td>
<td>10K File Create</td>
<td>99.9</td>
<td>370.2</td>
<td>3.7</td>
</tr>
<tr>
<td></td>
<td>10K File Delete</td>
<td>64.2</td>
<td>204.7</td>
<td>3.2</td>
</tr>
<tr>
<td></td>
<td>Mmap Latency</td>
<td>188000</td>
<td>385000</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>Prot Fault</td>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>Page Fault</td>
<td>1.5</td>
<td>2.3</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>100fd selct</td>
<td>6.6</td>
<td>13.7</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Table: Performance evaluation on KASAN with lmbench. HART and HASAN introduce no overhead to the main kernel, so the results are omitted here.

HART: Hardware-assisted Modular Tracing on ARM, ESORICS 2020
# Performance evaluation

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Benchmark</th>
<th>Setting</th>
<th>Native img +</th>
<th>KASAN img +</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Native module</td>
<td>KASAN module</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>HASAN module</td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>HSTCP [19]</td>
<td>iperf [20]</td>
<td>Local Comm.</td>
<td>1.00</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>TCPW [21]</td>
<td>iperf [20]</td>
<td>Local Comm.</td>
<td>0.92</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>H-TCP [22]</td>
<td>iperf [20]</td>
<td>Local Comm.</td>
<td>0.94</td>
<td>0.26</td>
</tr>
<tr>
<td>File System</td>
<td>HFS+ [23]</td>
<td>IOZONE [24]</td>
<td>Wr/fs=4048K/reclen=64</td>
<td>1.00</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wr/fs=4048K/reclen=512</td>
<td>0.88</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd/fs=4048K/reclen=64</td>
<td>0.92</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd/fs=4048K/reclen=512</td>
<td>0.90</td>
<td>0.99</td>
</tr>
<tr>
<td>File System</td>
<td>UDF [25]</td>
<td>IOZONE [24]</td>
<td>Wr/fs=4048K/reclen=64</td>
<td>0.95</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wr/fs=4048K/reclen=512</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd/fs=4048K/reclen=64</td>
<td>0.98</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd/fs=4048K/reclen=512</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>Driver</td>
<td>USB_STORAGE[26]</td>
<td>dd [27]</td>
<td>Wr/bs=1M/count=1024</td>
<td>1.00</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wr/bs=4M/count=256</td>
<td>1.00</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd/bs=1M/count=1024</td>
<td>0.99</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rd/bs=4M/count=256</td>
<td>1.00</td>
<td>0.76</td>
</tr>
<tr>
<td>Avg.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.95</td>
<td>0.85</td>
</tr>
</tbody>
</table>

Table: Performance evaluation with kernel modules and benchmarks.
## Tracing evaluation

<table>
<thead>
<tr>
<th>Module</th>
<th>Retrieving times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Name</td>
</tr>
<tr>
<td>Network</td>
<td>HSTCP</td>
</tr>
<tr>
<td></td>
<td>TCP-W</td>
</tr>
<tr>
<td></td>
<td>H-TCP</td>
</tr>
<tr>
<td>File</td>
<td>HFS+</td>
</tr>
<tr>
<td>Driver</td>
<td>USB_STORAGE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module</th>
<th>Max size(Byte)</th>
<th>Min size(Byte)</th>
<th>Average size(Byte)</th>
<th>Full ETB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Name</td>
<td>HART</td>
<td>HASAN</td>
<td>HART</td>
</tr>
<tr>
<td>Network</td>
<td>HSTCP</td>
<td>1100</td>
<td>1196</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>TCP-W</td>
<td>1460</td>
<td>1456</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>H-TCP</td>
<td>1292</td>
<td>1304</td>
<td>20</td>
</tr>
<tr>
<td>File</td>
<td>HFS+</td>
<td>1652</td>
<td>1756</td>
<td>20</td>
</tr>
<tr>
<td>System</td>
<td>UDF</td>
<td>2424</td>
<td>2848</td>
<td>20</td>
</tr>
<tr>
<td>Driver</td>
<td>USB_STORAGE</td>
<td>1544</td>
<td>1692</td>
<td>20</td>
</tr>
</tbody>
</table>

**Table:** Tracing evaluation of HART and HASAN.
## Effectiveness evaluation

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Type</th>
<th>PoC</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2016-0728</td>
<td>Use-after-free</td>
<td>REFCOUNT overflow [28]</td>
<td>HASAN: Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>KASAN: Y</td>
</tr>
<tr>
<td>CVE-2016-6187</td>
<td>Out-of-bound</td>
<td>Heap off-by-one [29]</td>
<td>HASAN: Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>KASAN: Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>KASAN: Y</td>
</tr>
<tr>
<td>CVE-2017-8824</td>
<td>Use-after-free</td>
<td>dccp_disconnect [31]</td>
<td>HASAN: Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>KASAN: Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>KASAN: Y</td>
</tr>
<tr>
<td>CVE-2018-12929</td>
<td>Use-after-free</td>
<td>ntfs_read_locked_inode [33]</td>
<td>HASAN: Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>KASAN: Y</td>
</tr>
</tbody>
</table>

**Table:** Effectiveness evaluation on HASAN.
Outline

▶ Introduction

▶ Background

▶ HART: Hardware-Assisted Runtime Tracing framework

▶ HASAN: HART-based Address Sanitizer

▶ Evaluation

▶ Conclusion
Conclusion

- We present HART, a hardware-based high-performance tracing framework specially for kernel modules

- Based on the HART, we build a modular security solution, HASAN, to effectively detect memory corruptions without requiring the source code of the module

- The evaluation result shows that HASAN can achieve the detection with only 5%-6% performance overhead, which is significantly superior to the state-of-the-art solution KASAN
References


References II


References III


2019.


https://github.com/SecWiki/linux-kernel-exploits/blob/master/2016/CVE-2016-0728/cve-2016-0728.c,
2017.


[33] S. Schumilo, “Multiple memory corruption issues in ntfs.ko (linux 4.15.0-15.16),”
Thank you!

Questions?

{duyunlan}@smail.nju.edu.cn