ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks

Mohamed Tarek, Miguel Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan
Columbia University
04/15/2021
About Me

Mohamed Tarek

4th year PhD Candidate

@M_TarekIbnZiad

https://cs.columbia.edu/~mtarek
Memory Safety is a serious problem!

Computing Sep 6

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google’s analysis of the novel hacking operation.
Memory Safety is a serious problem!

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google's analysis of the novel hacking operation.
Memory Safety is a serious problem!

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google's analysis of the novel hacking operation.

---

WhatsApp Rushes to Fix Security Flaw Exposed in Hacking of Lawyer’s Phone

---

Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder
It’s easy to make mistakes
It’s easy to make mistakes

SEGFAULT!
Prevalence of Memory Safety Vuln

Memory safety vs. Non-memory safety CVEs

Source: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019
Prevalence of Memory Safety Vulns

Microsoft Product CVEs

Google OSS-Fuzz bugs from 2016-2018.

Source: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019

ATTACKERS

MEMORY SAFETY
Attackers Prefer Memory Safety Vulns

Zero-day “in the wild” exploits from 2014-2020

Source: Google Project Zero, 0day "In the Wild" spreadsheet. Last updated: April 2020
How To Fix Memory (Un)Safety?
How To Fix Memory (Un)Safety?

Memory Safe Languages
How To Fix Memory (Un)Safety?

Memory Safe Languages

- Performance?
- Legacy Code?
How To Fix Memory (Un)safety?

Memory Safe Languages
- Performance?
- Legacy Code?

Pre-deployment Testing
How To Fix Memory (Un)Safety?

Memory Safe Languages
- Performance?
- Legacy Code?

Pre-deployment Testing
- Time?
- Scalability?
How To Fix Memory (Un)safety?

Memory Safe Languages
• Performance?
• Legacy Code?

Pre-deployment Testing
• Time?
• Scalability?

Post-deployment Mitigations
Overview
ZeRØ: Overview

Program

CALL <Foo>

Memory

Return Address

Return Address Protection
ZeRØ: Overview

Program

CALL <Foo>

RET

\ldots\]

Memory

Return Address

Return Address Protection
ZeRØ: Overview

Program

CALL <Foo>
STORE
RET...

Memory

Return Address

Return Address Protection
ZeRØ: Overview

ZeRØ uses 1-bit tag per pointer to protect return addresses.
ZeRØ: Overview

ZeRØ rejects any regular store that accesses a tagged return address.
ZeRØ: Overview

ZeRØ uses advisory exceptions to avoid crashing the running process under attack.
ZeRØ: Overview

ZeRØ protects return addresses, code pointers, and data pointers.
ZeRØ: Overview

ISA extensions: CPtrLD & CPtrST

ZeRØ protects return addresses, code pointers, and data pointers.
ZeRØ: Overview

ISA extensions: CPtrLD & CPtrST
DPtrLD & DPtrST

ZeRØ protects return addresses, code pointers, and data pointers.
ZeRØ: Overview

How can ZeRØ efficiently identify if a memory word is a return address, code pointer, data pointer, or regular data?
Cache Line Formats
ZeRØ: Cache Line Formats

Normal

1  2  3  4  5  6  7  8
ZeRØ: Cache Line Formats

Program
Pointers

Normal

bit-vector
ZeRØ: Cache Line Formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return address</td>
<td>01</td>
</tr>
</tbody>
</table>

- **Type**: Program Pointers
- **Normal**
- **bit-vector**
ZeRØ: Cache Line Formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return address</td>
<td>01</td>
</tr>
<tr>
<td>Function pointer</td>
<td>10</td>
</tr>
</tbody>
</table>
# ZeRØ: Cache Line Formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return address</td>
<td>01</td>
</tr>
<tr>
<td>Function pointer</td>
<td>10</td>
</tr>
<tr>
<td>Data pointer</td>
<td>11</td>
</tr>
</tbody>
</table>

The diagram illustrates the cache line formats with a 'bit-vector' indicating the valid bits for different types of pointers. The program pointers are shown in dark gray.
ZeRØ: Cache Line Formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular data</td>
<td>00</td>
</tr>
<tr>
<td>Return address</td>
<td>01</td>
</tr>
<tr>
<td>Function pointer</td>
<td>10</td>
</tr>
<tr>
<td>Data pointer</td>
<td>11</td>
</tr>
</tbody>
</table>
ZeRØ: Cache Line Formats

3.125% area overhead
ZeRØ: Cache Line Formats

**Our Metadata:** Encoded within unused pointer bits.

- Program Pointers

![Diagram showing Normal and Protected cache line formats with metadata encoded within unused pointer bits.](image-url)
ZeRØ: Cache Line Formats

**Our Metadata:** Encoded within unused pointer bits.

Program Pointers

<table>
<thead>
<tr>
<th>Normal</th>
<th>Protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D E</td>
<td>Y Header A B C D E</td>
</tr>
</tbody>
</table>

Is Protected?
ZeRØ: Cache Line Formats

Our Metadata: Encoded within unused pointer bits.

Program Pointers

Normal

A B C D E

Protected

Is Protected?

Y

Header

A B C D E

Normal

Is Protected?

N

1 2 3 4 5 6 7 8

Program Pointers

Normal

1 2 3 4 5 6 7 8
ZeRØ: Cache Line Formats

Our Metadata: Encoded within unused pointer bits.

Program Pointers

Normal

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Protected

<table>
<thead>
<tr>
<th>Y</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is Protected?

<table>
<thead>
<tr>
<th>N</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0.2% area overhead
ZeRØ: Microarchitectural Overview
ZeRØ: Microarchitectural Overview

CPU → L1-D → L2 → DRAM

Bit vector format:
0 1 2 3 4 5 ... 12 13 14 15
ZeRØ: Microarchitectural Overview
ZeRØ: Microarchitectural Overview
ZeRØ: Microarchitectural Overview
ZeRØ: Microarchitectural Overview
ISA Extensions
ZeRØ: ISA Extensions

CPtrST/CPtrLD Address, Value

DPtrST/DPtrLD Address, Value
ZeRØ: ISA Extensions

- CPtrST/CPtrLD Address, Value
- DPtrST/DPtrLD Address, Value

Same Layout as regular Loads/Stores
ZeRØ: ISA Extensions

- CPtrST/CPtrLD Address, Value
- DPtrST/DPtrLD Address, Value
- ClearMeta Address, Mask

Only invoked upon free() or delete()
Performance
ZeRØ: Performance Overheads

- Hardware Overheads.
- Software Overheads.
ZeRØ: Performance Overheads

- Hardware Overheads.
  - Our hardware measurements show that ZeRØ has minimal latency/area/power overheads.

- Software Overheads.
ZeRØ: Performance Overheads

- **Hardware Overheads.**
  - Our hardware measurements show that ZeRØ has minimal latency/area/power overheads.

- **Software Overheads.**
  - Our special loads/stores do not change binary size.
ZeRØ: Performance Overheads

➢ Hardware Overheads.
  • Our hardware measurements show that ZeRØ has minimal latency/area/power overheads.

➢ Software Overheads.
  • Our special loads/stores do not change binary size.
  • The ClearMeta instructions are only called upon memory deallocation.
ZeRØ: Performance Overheads

- **Hardware Overheads.**
  - Our hardware measurements show that ZeRØ has minimal latency/area/power overheads.

- **Software Overheads.**
  - Our special loads/stores do not change binary size.
  - The ClearMeta instructions are only called upon memory deallocation.

*The ClearMeta instructions are emulated on x86_64 using dummy stores*
ZeRØ: Performance Results on x86_64

ZeRØ
ZeRØ: Performance Results on x86_64

![Performance Results](image-url)
ZeRØ: Performance Results on x86_64

![Performance Results Graph]

6%
ZeRØ: Performance Results on x86_64

ARM PAC’s overheads are attributed to the extra QARMA invocations upon pointer loads/stores
ZeRØ: Performance Results on x86_64
ZeRØ: Performance Results on x86_64

ZeRØ reduces the average runtime overheads of pointer integrity from 14% to 0%
Limitations
ZeRØ: Limitations

- Non-pointer data corruption attacks.
  - Require a full memory safety solution.
ZeRØ: Limitations

- Non-pointer data corruption attacks.
  - Require a full memory safety solution.

- Third-party code.
  - Clear the metadata bits before passing pointers to shared libraries.
Conclusion

ZeRØ provides an efficient pointer integrity mechanism:
• Is easy to implement.
• Has no runtime overheads.
• Offers robust security.

ZeRØ can be applied to a wide variety of systems:
• Ranging from servers to mobile devices.