Efficient Pointer Integrity For Securing Embedded Systems

Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Vasileios P. Kemerlis, and Simha Sethumadhavan

Columbia University
Brown University
09/21/2021
Embedded systems are everywhere!
Embedded systems are dominated by 32-bit.
Why embedded system security is important?

Software has become increasingly complex.
Why embedded system security is important?

Software has become increasingly complex.

Why embedded system security is important?

Software has become increasingly complex. Heavily utilized software is predominantly written in unsafe languages.

Why Memory Safety?

It is the predominant source of vulnerabilities (ie. CVEs).

Source: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019
Why Memory Safety?

Memory Safety CVEs are heavily exploited.

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

CALL <Foo>
STORE
RET

Program

Memory
Return Address Integrity

```
CALL <Foo>
STORE
RET
```

Program

Memory

Return Address
Return Address Integrity

Program
CALL <Foo>
STORE
RET

Memory
Return Address
Return Address Integrity
Return Address Integrity

CALL <Foo>
STORE
RET

Program

Memory

Return Address
Return Address Integrity

CALL <Foo>
STORE
RET

Program

Memory

Return Address
Return Address Integrity

EPI uses advisory exceptions to avoid crashing when under attack.
Code Pointer Integrity

Program

CPtrST
...
CPtrLD

Memory

Function Pointer
Code Pointer Integrity

Program

Memory

CPtrST
...
CPtrLD

Function Pointer
Code Pointer Integrity
Data Pointer Integrity

Works in the same way as Code Pointer Integrity but for data pointers!
Cache Line Formats
Cache Line Formats

Normal
Cache Line Formats

![Diagram of cache line formats with fields A, B, C, D, E, and pointers]

Normal
Cache Line Formats

bit-vector

Normal

Pointers
Cache Line Formats

Format Encoding Table

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

bit-vector

Pointers

A B C D E

Normal
# Cache Line Formats

## Format Encoding Table

<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>

- **bit-vector**
- **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><strong>Normal</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Pointers**
# Cache Line Formats

## Format Encoding Table

<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>

![Diagram showing bit-vector and pointers](image-url)

- **A**, **B**, **C**, **D**, **E**: Normal
- **bit-vector**: Pointers
Cache Line Formats

Format Encoding Table

<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>

\[bit-vector\] Pointers

Normal
## Cache Line Formats

### Format Encoding Table

<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>

This introduces a 6.25% area overhead.
Cache Line Formats

Using a bit-vector throughout the memory hierarchy is inefficient!
Cache Line Formats

With EPI, we encode metadata within unused pointer bits.
Cache Line Formats

With EPI, we encode metadata within unused pointer bits.

Pointers

Normal

Encoded

Header
With EPI, we encode metadata within unused pointer bits.
Cache Line Formats

With EPI, we encode metadata **within** unused pointer bits.

### Pointers

<table>
<thead>
<tr>
<th>Normal</th>
<th>Encoded</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D E</td>
<td>Is Ret?</td>
<td>Is Ptr?</td>
</tr>
<tr>
<td>N Y</td>
<td>Header</td>
<td>A B C D E</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>Is Ret?</td>
<td>Is Ptr?</td>
</tr>
<tr>
<td>N N</td>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>
Cache Line Formats

With EPI, we encode metadata within unused pointer bits.

Extra bits add 0.39% area overhead.
Cache Line Formats

A novel variant of ZeRØ & Califorms

Pointers

Normal

A B C D E

Normal

0 1 2 3 4 5 6 7

Is Ret? Is Ptr?

Normal

Header

Encoded

Is Ret? Is Ptr?

Normal

0 1 2 3 4 5 6 7

ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks

ISCA 2021

Practical Byte-Granular Memory Blacklisting using Califorms

MICRO 2019
Cache Line Formats

With EPI, we encode metadata within unused pointer bits.

What unused pointer bits?
Harvesting Unused Pointer Bits

Common software properties allow us harvest extra bits from pointers on 32-bit architectures.
Harvesting Unused Pointer Bits

Regular Data

4 Bytes

[0] ... [31]
Harvesting Unused Pointer Bits

- **Regular Data**
  - 4 Bytes
  - [0] ... [31]

- **Return Address**
  - 4 Bytes
  - [0] ... [31]
Harvesting Unused Pointer Bits

Regular Data

4 Bytes

[0] ... [31]

Return Address

[0] [1] ... [31]

Fixed-width instructions on RISC architectures allow us to harvest the 2 LSBs.
Harvesting Unused Pointer Bits

Aligning functions (e.g. \texttt{-falign-functions}) allows to harvest the 4 LSBs.
Harvesting Unused Pointer Bits

Compacting the code address space allows us to harvest 2 MSBs.
Harvesting Unused Pointer Bits

Inserting padding bytes allows us to store a per-pointer ID.

Regular Data

Return Address

Function Pointer

Data Pointer

4 Bytes

[0] ... [31]

[0] [1] ... [30] [31]

Padding Padding [0] [1] [2] [3] ... [30] [31]

Padding Padding ...
EPI

Performance
EPI Performance Overheads

Hardware Modifications
EPI Performance Overheads

**Hardware Modifications**
Our hardware measurements show minimal latency/area/power overheads.
EPI Performance Overheads

Hardware Modifications
Our hardware measurements show minimal latency/area/power overheads.

Software Modifications
- Our special load/stores do not change the binary size.
EPI Performance Overheads

Hardware Modifications
Our hardware measurements show minimal latency/area/power overheads.

Software Modifications
- Our special load/stores do not change the binary size.
- The ClearMeta instructions are only called on memory deallocation.
EPI Performance Overheads

Hardware Modifications
Our hardware measurements show minimal latency/area/power overheads.

Software Modifications
- Our special load/stores do not change the binary size.
- The ClearMeta instructions are only called on memory deallocation.
- Padding bytes are added to pointers only.
Performance Results

Experimental Setup
We use emulate EPI on x86_64 by modifying LLVM to emit new instructions.
- ClearMeta is emulated using dummy stores.
- Padding bytes & necessary LD/ST emulate extra memory utilization.
Performance Results

0.47%
Performance Results

0.88%
Performance Results

![Performance Results Chart]

- **EPI-Return**
- **EPI-Full**
- **PAC-Return**

4% performance improvement
Performance Results

![Graph showing performance results for different benchmarks.](image)

- **Average Norm. Perf.:** 0.0
- **EPI-Return:** 0.0
- **EPI-Full:** 0.0
- **PAC-Return:** 0.0
- **PAC-Full:** 0.0

The overall average performance improvement is **8.5%**.
Performance Results

PAC’s overheads are attributed to the extra QARMA encryption invocations upon pointer:
- loads/stores
- usages
Performance Results

EPI reduces the average runtime overheads of pointer integrity from 8.5% to 0.88%!
EPI does not compromise on security

No Pointer Manipulation
Protects against all known pointer manipulation attacks (e.g. ROP, JOP/COP, COOP, DOP).
Handling Security Violations

Advisory Exceptions
- Skip faulty instructions.
- Do NOT crash the running process.
Handling Security Violations

Advisory Exceptions
- Skip faulty instructions.
- Do NOT crash the running process.

Permit List
- Initialized during program startup
Handling Security Violations

Advisory Exceptions
• Skip faulty instructions.
• Do NOT crash the running process.

Permit List
• Initialized during program startup
• Avoid false alarms for non-type aware functions (e.g., `memcpy` and `memmove`)
Handling Third Party Code

We can pick from the following options:
Handling Third Party Code

We can pick from the following options:

1. Compile with EPI
   Compile third party code with EPI support.
Handling Third Party Code

We can pick from the following options:

1. **Compile with EPI**
   Compile third party code with EPI support.

2. **Add to Permit List**
   Add to a permit list during program initialization.
Handling Third Party Code

We can pick from the following options:

1. Compile with EPI
   Compile third party code with EPI support.

2. Add to Permit List
   Add to a permit list during program initialization.

3. Invoke ClearMeta
   ClearMeta is inserted before passing pointers to external libraries.
Limitations

Non-pointer Data Corruption
These attacks require a full memory safety solution.
An efficient pointer integrity mechanism

Specifically tailored for 32-bit embedded systems.

✓ Offers Robust Security
✓ Easy to Implement
✓ Minimal Runtime Overheads
✓ Low Power
✓ Increased Reliability