Hardware-Software Co-design For Practical Memory Safety

Mohamed Tarek
Ph.D. Defense - April 11th, 2022
Hardware-Software Co-design For Practical Memory Safety

Mohamed Tarek
Ph.D. Defense - April 11th, 2022
What is Memory Safety?

A program property that guarantees memory objects can only be accessed:
What is Memory Safety?

A program property that guarantees **memory objects** can only be accessed:

- Between their intended bounds,
What is Memory Safety?

A program property that guarantees memory objects can only be accessed:

• Between their intended bounds,

• During their lifetime, and
What is Memory Safety?

A program property that guarantees memory objects can only be accessed:

- Between their intended bounds,
- During their lifetime, and
- Given their original (or compatible) type.
Memory Attacks Taxonomy

Root cause

Memory safety vulnerability

Spatial
Memory Attacks Taxonomy

Memory safety vulnerability

- Spatial
  - Non-contiguous
  - Contiguous

Root cause
Memory Attacks Taxonomy

Root cause

Memory safety vulnerability

Spatial

Non-contiguous

Contiguous
Memory Attacks Taxonomy

Root cause

Memory safety vulnerability

Spatial

Non-contiguous

Contiguous
Memory Attacks Taxonomy

Root cause

Memory safety vulnerability

- Spatial
  - Non-contiguous
  - Contiguous
- Temporal
Memory Attacks Taxonomy

Root cause

Memory safety vulnerability

- Spatial
  - Non-contiguous
  - Contiguous
- Temporal
  - Use-after-free
  - Uninitialized read
Memory Attacks Taxonomy

Root cause

Memory safety vulnerability

- Spatial
  - Non-contiguous
  - Contiguous

- Temporal
  - Use-after-free
  - Uninitialized read
Memory Attacks Taxonomy

Root cause

Memory safety vulnerability

Spatial
- Non-contiguous
- Contiguous

Temporal
- Use-after-free
- Uninitialized read
# Memory Attacks Taxonomy

## Root cause

<table>
<thead>
<tr>
<th>Spatial</th>
<th>Temporal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-contiguous</td>
<td>Use-after-free</td>
</tr>
<tr>
<td>Contiguous</td>
<td>Uninitialized read</td>
</tr>
</tbody>
</table>

## Asset

<table>
<thead>
<tr>
<th>Program code</th>
<th>Return address</th>
<th>Function pointer</th>
<th>Data pointer</th>
<th>Non-pointer data</th>
</tr>
</thead>
</table>

Memory safety vulnerability
Memory Attacks Taxonomy

Root cause

- Non-contiguous
- Contiguous
- Use-after-free
- Uninitialized read

Temporal

Asset

- Program code
- Return address
- Function pointer
- Data pointer
- Non-pointer data

Result

- Code corruption
- Control-flow hijacking
- Data-flow hijacking
- Data corruption
Memory Attacks Taxonomy

Root cause

Memory safety vulnerability

Spatial

Non-contiguous

Contiguous

Temporal

Use-after-free

Uninitialized read

Asset

Program code

Return address

Function pointer

Data pointer

Non-pointer data

Result

Code corruption

Control-flow hijacking

Data-flow hijacking

Data corruption
Memory Attacks Taxonomy

Memory safety vulnerability

Root cause

Spatial

Non-contiguous

Contiguous

Temporal

Use-after-free

Uninitialized read

Asset

Program code

Return address

Function pointer

Data pointer

Non-pointer data

Result

Code corruption

Control-flow hijacking

Data-flow hijacking

Data corruption
Why is memory safety a concern?
Memory Safety is a serious problem!
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.
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
Prevalence of Memory Safety Vulns

Microsoft Product CVEs between 2006-2018

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

Microsoft Product CVEs between 2006-2018

Chromium high severity security bugs between 2015-2020

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

Source: https://www.chromium.org/Home/chromium-security/memory-safety/
ATTACKERS

MEMORY SAFETY
Attackers prefer Memory Safety Vulns

% of Zero-day “in the wild” exploits from 2014-2021

Source: Google Project Zero, 0day “In the Wild” spreadsheet. Last updated: January 2022
C/C++ is here to stay!
C/C++ is here to stay!
C/C++ is here to stay!
C/C++ is here to stay!
C/C++ is here to stay!
How to fix C/C++ memory (un)safety?
How to fix C/C++ memory (un)safety?

- Memory Blocklisting
- Memory Permitlisting
- Exploit Mitigation
How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

Ptr → Memory Object (A)
How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

Memory Object (A)

Ptr

✓
How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

e.g., Google’s Address Sanitizer
How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

e.g., Google’s Address Sanitizer
How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Object (A)

Ptr

Memory Permitlisting

Size

Base

Ptr

Memory Object (A)

Exploit Mitigation

e.g., Google’s Address Sanitizer

e.g., Intel’s MPX and CHERI
How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

Enforcing strict memory safety rules comes with non-negligible performance costs!

e.g., Google’s Address Sanitizer

e.g., Intel’s MPX and CHERI
How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

e.g., Google’s Address Sanitizer

e.g., Intel’s MPX and CHERI
How to fix C/C++ memory (un)safety?

**Memory Blocklisting**

- **Ptr** → **Memory Object (A)**
- e.g., Google’s Address Sanitizer

**Memory Permitlisting**

- **Size** → **Memory Object (A)**
- **Base** → **Ptr** → **Memory Object (A)**
- e.g., Intel’s MPX and CHERI

**Exploit Mitigation**

- **Ptr** → **Memory Object (A)**
- **Ptr** → **Memory Object (B)**
How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

e.g., Google’s Address Sanitizer
e.g., Intel’s MPX and CHERI
How to fix C/C++ memory (un)safety?

**Memory Blocklisting**

- e.g., Google’s Address Sanitizer

**Memory Permitlisting**

- Size
- Base
- Ptr

- e.g., Intel’s MPX and CHERI

**Exploit Mitigation**

- Ptr

- e.g., ARM’s PAC
How to fix C/C++ memory (un)safety?

Memory Blocklisting
Memory Permitlisting
Exploit Mitigation

All prior approaches share a common theme:

- e.g., Google’s Address Sanitizer
- e.g., Intel’s MPX and CHERI
- e.g., ARM’s PAC
How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

All prior approaches share a common theme:
Adding more features to a program to make it secure

e.g., Google’s Address Sanitizer

e.g., Intel’s MPX and CHERI

e.g., ARM’s PAC
My solutions for C/C++ memory (un)safety

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

My research work turns existent program features into security primitives to save on performance.

e.g., Google’s Address Sanitizer

e.g., Intel’s MPX and CHERI

e.g., ARM’s PAC
My solutions for C/C++ memory (un)safety

Thesis Statement

Leveraging common software trends and rethinking computer microarchitectures can efficiently circumvent the problems of traditional memory safety solutions for C and C++.
My solutions for C/C++ memory (un)safety

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

Uses dead bytes in program memory

Practical byte-granular memory blacklisting using Califorms. [MUSC 2019]
My solutions for C/C++ memory (un)safety

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

Leverages modern software trends

Architectural Support for Low Overhead Memory Safety Checks. [ISCA 2021]
My solutions for C/C++ memory (un)safety

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks. [ISCA 2021]
My solutions for C/C++ memory (un)safety

- Memory Blocklisting
- Memory Permitlisting
- Exploit Mitigation

[MICRO 2019] [ISCA 2021] [ISCA 2021]
Cache Line Formats


[MICRO 2019] [IEEE Micro Top Picks Honorable Mention]
CaLiForms Memory Blocklisting

This is program data.

A blocklisted location.

Challenge
How to efficiently track the state of memory locations?
CaLiForms Memory Blocklisting

Program Memory

Shadow memory

Disjoint Memory
CaLiForms Memory Blocklisting

- Program Memory

- Disjoint Memory

Shadow memory

~ 2X runtime overheads!
~ 3X memory overheads!
CaLiForms Memory Blocklisting

Memory Tagging
$n$ bits per cache line
CaLiForms Memory Blocklisting

Memory Tagging
\( n \) bits per cache line

Limited entropy!
CaLiForms Memory Blocklisting

CaLiForms
1 bit per cache line
CaLiForms Memory Blocklisting

- CaLiForms
  - 1 bit per cache line

- Metadata

- Program Memory

- 0.2% memory overhead!
- 2-14% runtime overhead!
CaLiForms Memory Blocklisting

The key insight is to change how data is stored in cache lines!

Metadata

0.2% memory overhead!
2-14% runtime overhead!
CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.
CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

Blocklisted Location

Normal
CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

Blocklisted Location

Normal

bit-vector
CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

Blocklisted Location

Normal

12.5% memory overhead

bit-vector
**CaLiForms Cache Line Formats**

**Our Metadata:** Encoded within unused data.

- **Blocklisted**
- **Location**

### Normal

A | B | C | D | E

### Califorms

- **Header**
- A | B | C | D | E
CaLiForms Cache Line Formats

**Our Metadata:** Encoded within unused data.

Blocklisted
Location

<table>
<thead>
<tr>
<th>Normal</th>
<th>Califorms</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 Califormed?
**CaLiForms Cache Line Formats**

**Our Metadata:** Encoded within unused data.

- Blocklisted
- Location

**Normal**

```
A B C D E
```

**Is Califormed?**

```
Y
```

**Califorms**

```
Header A B C D E
```

**Normal**

```
1 2 3 4 5 6 7 8
```

**Is Califormed?**

```
N
```

**Normal**

```
1 2 3 4 5 6 7 8
```
CaLiForms Microarchitectural Overview
CaLiForms Microarchitectural Overview

Bit vector format

1 2 3 4 5 6 ... 61 62 63 64

Core

L1-D

C

C

L2

DRAM
CaLiForms Microarchitectural Overview
CaLiForms Microarchitectural Overview
CaLiForms Microarchitectural Overview
CaLiForms Microarchitectural Overview
CaLiForms Performance Overheads

Hardware Modifications
Our measurements show no impact on the cache access latency.
CaLiForms Performance Overheads

Hardware Modifications
Our measurements show no impact on the cache access latency.

Software Modifications
• We evaluate three different insertion policies using Clang/LLVM.
struct A_opportunistic {
    char c;
    char tripwire[3];
    int i;
    char buf[64];
    void (*fp)();
}

(1) Opportunistic
CaLiForms Insertion Polices

```c
struct A_opportunistic {
    char c;
    char tripwire[3];
    int i;
    char buf[64];
    void (*fp)();
}
```

(1) Opportunistic

```c
struct A_full {
    char tripwire[2];
    char c;
    char tripwire[1];
    int i;
    char tripwire[3];
    char buf[64];
    char tripwire[2];
    void (*fp)();
    char tripwire[1];
}
```

(2) Full
CaLiForms Insertion Polices

(1) Opportunistic

```
struct A_opportunistic {
    char c;
    char tripwire[3];
    int i;
    char buf[64];
    void (*fp)();
}
```

(2) Full

```
struct A_full {
    char tripwire[2];
    char c;
    char tripwire[1];
    int i;
    char tripwire[3];
    char buf[64];
    char tripwire[2];
    void (*fp)();
    char tripwire[1];
}
```

(3) Intelligent

```
struct A_intelligent {
    char c;
    int i;
    char tripwire[3];
    char buf[64];
    char tripwire[2];
    void (*fp)();
    char tripwire[3];
}
```
CaLiForms Performance Overheads

Hardware Modifications
Our measurements show no impact on the cache access latency.

Software Modifications
- We evaluate three different insertion policies using Clang/LLVM.
CaLiForms Performance Overheads

Hardware Modifications
Our measurements show no impact on the cache access latency.

Software Modifications
• We evaluate three different insertion policies using Clang/LLVM.
• We emulate the overheads of BLOC instructions that are used during malloc/free to mark the blocklisted locations per cacheline.
CaLiForms Performance Results (x86_64)

Opportunistic (BLOC)

Slowdown

-10.0% 0.0% 10.0% 20.0% 30.0% 40.0% 50.0% 60.0% 70.0% 80.0% 90.0%

astar bzip2 gobmk h264ref hammer lbm lbquantum mcf milc namd perlbench povray sjeng soplex sphinx3 xalancbmk

AMean

7.9%

80.3%
CaLiForms Performance Results (x86_64)

- Sloppy Delay
  - Opportunistic (BLOC): 7.9%
  - Full (BLOC): 13.9%

- AMean
  - Opportunistic (BLOC): 0.0%
  - Full (BLOC): 5.0%
  - AMean: 7.9%

- Slowdown Distribution
  - astar: -10.0%
  - bzip2: 0.0%
  - gromacs: 10.0%
  - h264ref: 20.0%
  - hminer: 30.0%
  - lbm: 40.0%
  - libquantum: 50.0%
  - mcf: 60.0%
  - mkl: 70.0%
  - namd: 80.0%
  - perlbench: 90.0%
  - povray: 100.0%
  - sjeng: -10.0%
  - soplex: 0.0%
  - sphinx3: 10.0%
  - xalancbmk: 20.0%
CaLiForms Performance Results (x86_64)

- Performance Results:
  - Astar: 85.2%
  - Bzip2: 80.3%
  - Gobmk: 7.9%
  - H264ref: -10.0%
  - Hmmer: 0.0%
  - Lbm: 10.0%
  - Libquantum: 20.0%
  - Mcf: 30.0%
  - Milc: 40.0%
  - Namd: 50.0%
  - Perlbench: 13.9%
  - Povray: 1.5%
  - Sjeng: 0.0%
  - Soplex: 5.0%
  - Sphinx3: 10.0%
  - Xalancbmk: 15.0%

- Slowdown:
  - Opportunistic (BLOC): 80.3%
  - Full (BLOC): 85.2%
  - Intelligent (BLOC): 88.8%
The intelligent policy provides the best performance-security tradeoff.

---

(1) Opportunistic

```
struct A_opportunistic {
    char c;
    char tripwire[3];
    int i;
    char buf[64];
    void (*fp)();
}
```

(2) Full

```
struct A_full {
    char tripwire[2];
    char c;
    char tripwire[3];
    char buf[64];
    char tripwire[2];
    void (*fp)();
    char tripwire[3];
}
```

(3) Intelligent

```
struct A_intelligent {
    char c;
    int i;
    char tripwire[3];
    char buf[64];
    char tripwire[2];
    void (*fp)();
    char tripwire[3];
}
```
Memory Attacks Taxonomy

Root cause

Memory safety vulnerability

Spatial
Non-contiguous
Contiguous
Temporal
Use-after-free
Uninitialized read

Asset

Program code
Return address
Function pointer
Data pointer
Non-pointer data

Result

Code corruption
Control-flow hijacking
Data-flow hijacking
Data corruption

Detect violation

Prevent exploitation

CaLiForms (+1.5% runtime) (+0.2% memory) *probabilistic
Mohamed Tarek Ibn Ziad, Miguel A. Arroyo Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan, Architectural Support for Low Overhead Memory Safety Checks. [ISCA 2021]
No-FAT: Key Observation

Current software trends can be used to enhance systems security
No-FAT: Key Observation

Current software trends can be used to enhance systems security

Increasing adoption of binning allocators
No-FAT: Key Observation

Current software trends can be used to enhance systems security

Increasing adoption of binning allocators

- Maintains memory locality.
- Implicit lookup of allocation information.
No-FAT: Key Observation

Current software trends can be used to enhance systems security

Increasing adoption of binning allocators

- Maintains memory locality.
- Implicit lookup of allocation information.
Binning Memory Allocators

```c
40. int main() {
41.     char* ptr = malloc(12);
42.     ...
50. }
```
```c
int main() {
    char* ptr = malloc(12);
    ...
}
```
Binning Memory Allocators

```c
40. int main() {
41.     char* ptr = malloc(12);
42.     ...
50. }
```

Memory is requested by the allocator.
Binning Memory Allocators

40. int main() {
41.     char* ptr = malloc(12);
42.     ...
50. }

Memory is divided into bins.

Bins

A
B
C

Virtual Memory
Binning Memory Allocators

```c
40. int main() {
41.     char* ptr = malloc(12);
42.     ...
50. }
```

Each bin is associated with a size.
Binning Memory Allocators

```c
40. int main() {
41.     char* ptr = malloc(12);
42.     ...
50. }
```

![Binning Memory Allocators Diagram]
Binno Memory Allocators

```c
40. int main() {
41.     char* ptr = malloc(12);
42.     ...
50. }
```
Binning Memory Allocators

Given any pointer, we can derive its allocation size and base address.

```c
40. int main() {
41.     char* ptr = malloc(12);
42.     ...
50. }
```
int main() {
    char* ptr = malloc(12);
    ptr[1] = 'A';
    ...
}

How No-FAT Provides Memory Safety
How No-FAT Provides Memory Safety

```c
40. int main() {
41.     char* ptr = malloc(12);
42.     ptr[1] = 'A';  \textbf{store ptr[1], 'A'}
43.     ...
50. }
```
How No-FAT Provides Memory Safety

```c
40. int main() {
41.     char* ptr = malloc(12);
42.     ptr[1] = 'A';  // s_store ptr[1], 'A'  ptr
43.     ...
50. }
```

We add one extra operand for loads/stores.
How No-FAT Provides Memory Safety

```c
40. int main() {
41.   char* ptr = malloc(12);
42.   ptr[1] = 'A';
43.   ...  
50. }
```

The compiler propagates the allocation base address.
How No-FAT Provides Memory Safety

40. int main() {
41. char* ptr = malloc(12);
42. ptr[1] = ‘A’;
43. ...
50. }

s_store ptr[1],‘A’,ptr_{trusted base}
How No-FAT Provides Memory Safety

\[ s\_store\; ptr[1], 'A', ptr_{trusted\_base} \]
How No-FAT Provides Memory Safety

\[ s\_store\{ptr[1]\}, 'A', ptr_{trusted\_base} \]

\[
\text{offset} = ptr[1] - ptr_{trusted\_base}
\]
How No-FAT Provides Memory Safety

\[ \text{s\_store } \text{ptr}[1], 'A', \text{ptr}_{\text{trusted\_base}} \]

\[
\begin{align*}
\text{offset} & = \text{ptr}[1] - \text{ptr}_{\text{trusted\_base}} \\
\text{size} & = \text{getSize}(\text{ptr}_{\text{trusted\_base}})
\end{align*}
\]

108
How No-FAT Provides Memory Safety

\[
\text{s\_store } \text{ptr}[1], 'A', \text{ptr}_{\text{trusted\_base}}
\]

\[
\begin{align*}
\text{offset} & = \text{ptr}[1] - \text{ptr}_{\text{trusted\_base}} \\
\text{size} & = \text{getSize}(\text{ptr}_{\text{trusted\_base}})
\end{align*}
\]

Bounds Check

\[
\text{offset} < \text{size} \quad ?
\]
How No-FAT Provides Memory Safety

\[
s_{\text{store}} \quad \text{ptr}[1], 'A', \text{ptr}_{\text{trusted_base}}
\]

\[
\text{offset} \quad = \quad \text{ptr}[1] - \text{ptr}_{\text{trusted_base}}
\]

\[
\text{size} \quad = \quad \text{getSize}(\text{ptr}_{\text{trusted_base}})
\]

Bounds Check

\[
\text{offset} < \text{size} \quad ?
\]

Temporal Check

\[
\text{ptr}[1] [63:48] = \text{ptr}_{\text{trusted_base}} [63:48] \quad ?
\]
How No-FAT Provides Memory Safety

The allocation size information is made available to the hardware to verify memory accesses.

- Temporal Check: $\text{ptr}[1][63:48] = \text{ptr}_{\text{trusted_base}}[63:48]$?

- Bounds Check: $\text{offset} < \text{size}$?
How No-FAT Provides Memory Safety

```c
40. int main() {
41.     char* ptr = malloc(12);
42.     ptr[1] = 'A';
43.     ...
50. }
```
How No-FAT Provides Memory Safety

```c
int main() {
    char* ptr = malloc(12);
    ptr[1] = ‘A’;
    foo(ptr);
}
```

Let’s pass the pointer to another context (e.g., foo).
How No-FAT Provides Memory Safety

```c
int main() {
    char* ptr = malloc(12);  // ptr_trusted_base
    ptr[1] = 'A';            // s_store ptr[1], 'A', ptr_trusted_base
    ... foo(ptr);
}

void Foo (char* xptr){
    ... xptr[7] = 'B';
    ... }
```
How No-FAT Provides Memory Safety

40. `int main() {`
41. `char* ptr = malloc(12);`
42. `ptr[1] = 'A';`
43. `...`
44. `foo(ptr);`
45. `}`
46. `void Foo (char* xptr){`
47. `...`
48. `xptr[7] = 'B';`
49. `}`
How No-FAT Provides Memory Safety

```c
40. int main() {
41.     char* ptr = malloc(12);          \[ptr\text{\textsubscript{trusted_base}}\]
42.     ptr[1] = ‘A’;                    \[s\_store\ \text{ptr}[1],\text{‘A’},\text{ptr\textsubscript{trusted_base}}\]
43.     ...
49.     foo(ptr);
50. }                                    \[\text{foo}(\text{ptr})\]
51. void Foo (char* xptr){
52.     ...
53.     xptr[7] = ‘B’;                \[s\_store\ \text{xptr}[7],\text{‘B’},\text{xptr\textsubscript{trusted_base}}\]
54.     ...
60. }
```

How do we get this?
How No-FAT Provides Memory Safety

40. `int main() {`
41. `char* ptr = malloc(12);`
42. `ptr[1] = 'A';`
43. `...`
49. `foo(ptr);`
50. `}
51. `void Foo (char* xptr){`
52. `...`
53. `xptr[7] = 'B';`
54. `...`
60. `}`
How No-FAT Provides Memory Safety

\[
\text{xptr}_{\text{trusted base}} \leftarrow \text{compBase}(\text{xptr}[7])
\]
How No-FAT Provides Memory Safety

\[ x_{\text{ptr, trusted base}} \leftarrow \text{compBase}(x_{\text{ptr}[7]}) \]

\[ \text{Bin} = x_{\text{ptr}} \gg \log_2(S) \] where \( S \) is the size of the bins.
How No-FAT Provides Memory Safety

\[
xptr_{\text{trusted base}} \leftarrow \text{compBase}(xptr[7])
\]

\[
\text{Bin} \equiv xptr \gg \log_2(S) \quad \text{where } S \text{ is the size of the bins.}
\]

\[
\text{size} \equiv \text{getSize(Bin)}
\]
How No-FAT Provides Memory Safety

$\text{xptr}_{\text{trusted base}} \leftarrow \text{compBase}(\text{xptr}[7])$

$\text{Bin} \equiv \text{xptr} \gg \log_2(S)$, where $S$ is the size of the bins.

$\text{size} \equiv \text{getSize(} \text{Bin} \text{)}$

$\text{xptr}_{\text{trusted base}} \equiv [\text{xptr} \times (1/\text{size})] \times \text{size}$
How No-FAT Provides Memory Safety

\[
xptr_{\text{trusted base}} \leftarrow \text{compBase}(xptr[7])
\]

Bin \(\equiv xptr \gg \log_2(S)\) where \(S\) is the size of the bins.

size \(\equiv \text{getSize(Bin)}\)

\[
xptr_{\text{trusted base}} \equiv [xptr \times (1/\text{size})] \times \text{size}
\]

Base pointer is implicitly derived!
How No-FAT Provides Memory Safety

```c
int main() {
    char* ptr = malloc(12);
    ptr[1] = 'A';
    ... foo(ptr);
}
```
How No-FAT Provides Memory Safety

```c
int main() {
    char* ptr = malloc(12);  // ptr_{trusted_base}
    ptr[1] = 'A';            // s_store ptr[1], 'A', ptr_{trusted_base}
    ptr = ptr + 100;         // Pointer arithmetic can push the pointer out-of-bounds before calling foo!
    …
    foo(ptr);
}
```
How No-FAT Provides Memory Safety

```c
int main() {
    char* ptr = malloc(12);  // ptr
    ptr[1] = ‘A’;            // s_store ptr[1], ‘A’, ptr
    ptr = ptr + 100;         // verifyBounds ptr, ptr
    verifyBounds ptr, ptr
    foo(ptr);
}
```

Verify the bounds of all pointers that escape to memory (or another function).
No-FAT Microarchitectural Overview

Core → L1-D → L2 → DRAM
No-FAT Microarchitectural Overview

No changes to the memory subsystems!
No-FAT Microarchitectural Overview
No-FAT Microarchitectural Overview

1-KiB Memory Allocation Sizes Table (MAST)
No-FAT Microarchitectural Overview
No-FAT Microarchitectural Overview
No-FAT Performance Results (x86_64)
No-FAT Performance Results (x86_64)
Most of No-FAT’s overheads are attributed to:
• The binning memory allocator, and
No-FAT Performance Results (x86_64)

Most of No-FAT’s overheads are attributed to:
- The binning memory allocator, and
- The back-to-back MULs during base address computation
No-FAT Performance Results (x86_64)

Most of No-FAT’s overheads are eliminated with:
- A performant binning memory allocator (e.g., MiMalloc), and
No-FAT Performance Results (x86_64)

Most of No-FAT’s overheads are eliminated with:
- A performant binning memory allocator (e.g., MiMalloc), and
- A base address cache for derived pointers.
My solutions for C/C++ memory (un)safety

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

[MICRO 2019] [ISCA 2021]
Comparison with prior work
## Comparison with prior work

<table>
<thead>
<tr>
<th>Memory Tagging</th>
<th>Metadata</th>
<th>Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>N bits per pointer &amp; allocation</td>
<td>Spatial &amp; temporal safety limited by tag width</td>
<td></td>
</tr>
</tbody>
</table>
## Comparison with prior work

<table>
<thead>
<tr>
<th></th>
<th>Metadata</th>
<th>Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Tagging</td>
<td>N bits per pointer &amp; allocation</td>
<td>Spatial &amp; temporal safety limited by tag width</td>
</tr>
<tr>
<td>Tripwires</td>
<td>N bits per allocation</td>
<td>Susceptible to non-adjacent overflows</td>
</tr>
</tbody>
</table>
## Comparison with prior work

<table>
<thead>
<tr>
<th></th>
<th>Metadata</th>
<th>Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Tagging</td>
<td>N bits per pointer &amp; allocation</td>
<td>Spatial &amp; temporal safety limited by tag width</td>
</tr>
<tr>
<td>Tripwires</td>
<td>N bits per allocation</td>
<td>Susceptible to non-adjacent overflows</td>
</tr>
<tr>
<td>CaLiForms</td>
<td>1 bit per cache line</td>
<td>Provides probabilistic guarantees</td>
</tr>
</tbody>
</table>
## Comparison with prior work

<table>
<thead>
<tr>
<th>Method</th>
<th>Metadata</th>
<th>Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Tagging</td>
<td>N bits per pointer &amp; allocation</td>
<td>Spatial &amp; temporal safety limited by tag width</td>
</tr>
<tr>
<td>Tripwires</td>
<td>N bits per allocation</td>
<td>Susceptible to non-adjacent overflows</td>
</tr>
<tr>
<td>CaLiForms</td>
<td>1 bit per cache line</td>
<td>Provides probabilistic guarantees</td>
</tr>
<tr>
<td>Explicit Base &amp; Bounds</td>
<td>N bits per pointer or allocation</td>
<td>Breaks compatibility with the rest of the system (eg. unprotected libraries).</td>
</tr>
</tbody>
</table>
## Comparison with prior work

<table>
<thead>
<tr>
<th>Method</th>
<th>Metadata</th>
<th>Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Tagging</td>
<td>N bits per pointer &amp; allocation</td>
<td>Spatial &amp; temporal safety limited by tag width</td>
</tr>
<tr>
<td>Tripwires</td>
<td>N bits per allocation</td>
<td>Susceptible to non-adjacent overflows</td>
</tr>
<tr>
<td>CaLiForms</td>
<td>1 bit per cache line</td>
<td>Provides probabilistic guarantees</td>
</tr>
<tr>
<td>Explicit Base &amp; Bounds</td>
<td>N bits per pointer or allocation</td>
<td>Breaks compatibility with the rest of the system (eg. unprotected libraries).</td>
</tr>
<tr>
<td>No-FAT</td>
<td>Fixed (1K) bits per process</td>
<td>Requires binning allocator</td>
</tr>
</tbody>
</table>
My solutions for C/C++ memory (un)safety

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

[MICRO 2019]

[ISCA 2021]

[ISCA 2021]
Return Address Protection with ZeRØ
Return Address Protection with ZeRØ
Return Address Protection with ZeRØ

CALL <Foo>
STORE
RET

Program

Return Address

Memory
Return Address Protection with ZeRØ
Return Address Protection with ZeRØ

Program

Memory

CALL <Foo>
STORE
RET
Return Address Protection with ZeRØ

Program

Memory

CALL <Foo>
STORE
RET

Return Address
Return Address Protection with ZeRØ

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

Program

CPtrST
...
CPtrLD

Memory

Function Pointer
Code Pointer Integrity with ZeRØ
Data Pointer Integrity with ZeRØ

Works in the same way as Code Pointer Integrity but for data pointers!
How can we keep track of ZeRØ bits?
Efficiently Tracking Metadata

In ZeRØ, we encode metadata within unused pointer bits.
Efficiently Tracking Metadata

We use a novel variant of CaLiForms

Pointers

Normal

A B C D E

Has Pointers?
Y

Encoded

Header A B C D E

Normal

0 1 2 3 4 5 6 7

Has Pointers?
N

0 1 2 3 4 5 6 7
ZeRØ Performance Overheads

**Hardware Modifications**

Our measurements show no impact on the cache access latency.
ZeRØ Performance Overheads

Hardware Modifications
Our measurements show no impact on the cache access latency.

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

Hardware Modifications
Our measurements show no impact on the cache access latency.

Software Modifications
- Our special load/stores do not change the binary size.
- The ClearMeta instructions are only called on memory deletion.
ZeRØ Performance Results (x86_64)
ZeRØ Performance Results (x86_64)
ZeRØ Performance Results (x86_64)
ZeRØ Performance Results (x86_64)
ZeRØ Performance Results (x86_64)

PAC’s overheads are attributed to the extra QARMA encryption invocations upon pointer:
- loads/stores
- usages
ZeRØ Performance Results (x86_64)

ZeRØ reduces the average runtime overheads of pointer integrity from 14% to 0%!
An efficient pointer integrity mechanism

An ideal candidate for end-user deployment.

- Easy to Implement
- No Runtime Overheads
- Provides Strong Security

A drop-in replacement for ARM’s PAC
My solutions for C/C++ memory (un)safety

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

[MICRO 2019]

[ISCA 2021]

[ISCA 2021]
Memory Attacks Taxonomy

- **Root cause**: Spatial, Temporal
  - Spatial: Non-contiguous, Contiguous
  - Temporal: Use-after-free, Uninitialized read

- **Asset**: Program code, Return address, Function pointer, Data pointer, Non-pointer data
  - Program code: Code corruption
  - Return address: Control-flow hijacking
  - Function pointer: Data-flow hijacking
  - Data pointer: Uninitialized read
  - Non-pointer data: Data corruption

- **Result**: Detect violation, Prevent exploitation
  - Detect violation: CaliForms (+1.5% runtime) (+0.2% memory) *probabilistic
  - Prevent exploitation:

---

*Notes:*
- CaliForms: Memory safety vulnerability detection tool.
- +1.5% runtime: Improvement in runtime efficiency.
- +0.2% memory: Memory usage improvement.
- *probabilistic: Indicates the nature of the detection method.*
Memory Attacks Taxonomy

Root cause
- Non-contiguous
- Contiguous
- Use-after-free
- Uninitialized read

Memory safety vulnerability
- Spatial
- Temporal

Asset
- Program code
- Return address
- Function pointer
- Data pointer
- Non-pointer data

Result
- Code corruption
- Control-flow hijacking
- Data-flow hijacking
- Data corruption

No-FAT
(<1% runtime)
(+0% memory)
*deterministic

CaLiForms
(+1.5% runtime)
(+0.2% memory)
*probabilistic

Detect violation

Prevent exploitation

Memory safety vulnerability

Memory Attacks Taxonomy

Root cause
- Non-contiguous
- Contiguous
- Use-after-free
- Uninitialized read

Memory safety vulnerability
- Spatial
- Temporal

Asset
- Program code
- Return address
- Function pointer
- Data pointer
- Non-pointer data

Result
- Code corruption
- Control-flow hijacking
- Data-flow hijacking
- Data corruption

No-FAT
(<1% runtime)
(+0% memory)
*deterministic

CaLiForms
(+1.5% runtime)
(+0.2% memory)
*probabilistic

Detect violation

Prevent exploitation

Memory Attacks Taxonomy

Root cause
- Non-contiguous
- Contiguous
- Use-after-free
- Uninitialized read

Memory safety vulnerability
- Spatial
- Temporal

Asset
- Program code
- Return address
- Function pointer
- Data pointer
- Non-pointer data

Result
- Code corruption
- Control-flow hijacking
- Data-flow hijacking
- Data corruption

No-FAT
(<1% runtime)
(+0% memory)
*deterministic

CaLiForms
(+1.5% runtime)
(+0.2% memory)
*probabilistic

Detect violation

Prevent exploitation

Memory Attacks Taxonomy

Root cause
- Non-contiguous
- Contiguous
- Use-after-free
- Uninitialized read

Memory safety vulnerability
- Spatial
- Temporal

Asset
- Program code
- Return address
- Function pointer
- Data pointer
- Non-pointer data

Result
- Code corruption
- Control-flow hijacking
- Data-flow hijacking
- Data corruption

No-FAT
(<1% runtime)
(+0% memory)
*deterministic

CaLiForms
(+1.5% runtime)
(+0.2% memory)
*probabilistic

Detect violation

Prevent exploitation

Memory Attacks Taxonomy

Root cause
- Non-contiguous
- Contiguous
- Use-after-free
- Uninitialized read

Memory safety vulnerability
- Spatial
- Temporal

Asset
- Program code
- Return address
- Function pointer
- Data pointer
- Non-pointer data

Result
- Code corruption
- Control-flow hijacking
- Data-flow hijacking
- Data corruption

No-FAT
(<1% runtime)
(+0% memory)
*deterministic

CaLiForms
(+1.5% runtime)
(+0.2% memory)
*probabilistic

Detect violation

Prevent exploitation
Memory Attacks Taxonomy

Root cause

- Program code corruption
- Return address hijacking
- Function pointer hijacking
- Use-after-free
- Uninitialized read

Asset

- Code
- Control-flow
- Data
- Non-pointer

Result

- No-FAT (<1% runtime) (+0% memory) *deterministic
- CaliForms (+1.5% runtime) (+0.2% memory) *probabilistic
- ZeRO (+0% runtime) (+0.2% memory) *deterministic

Detect violation

Prevent exploitation
Acknowledgement

Simha Sethumadhavan
Columbia University

Miguel A. Arroyo
Columbia University

Evgeny Manzhosov
Columbia University

Vasileios P. Kemerlis
Brown University

Kanad Sinha
Columbia University

Koustubha Bhat
Vrije Universiteit Amsterdam

Ryan Piersma
Columbia University

Hiroshi Sasaki
Tokyo Institute of Technology
My solutions for C/C++ memory (un)safety

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

[MICRO 2019]

[ISCA 2021]

[ISCA 2021]