Architectural Support for Low Overhead Memory Safety Checks

Mohamed Tarek Ibn Ziad, Miguel Arroyo, Evgeny Manzhosov, Ryan Piersma and Simha Sethumadhavan

Columbia Engineering
The Fu Foundation School of Engineering and Applied Science
06/16/2021
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.

Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder
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
Testing alone is **NOT** enough!
Testing alone is **NOT** enough!

Researchers: Beware of 10-Year-Old Linux Vulnerability

Qualys Says Flaw in Sudo Utility Could Grant Attackers Root Access

Akshaya Asokan (@asokan_akshaya) • January 28, 2021
Testing alone is **NOT** enough!

Researchers: Beware of 10-Year-Old Linux Vulnerability

Qualys Says Flaw in Sudo Utility Could Grant Attackers Root Access

Akshaya Asokan (@asokan_akshaya) · January 28, 2021

Microsoft patches critical 17-year-old DNS bug in Windows Server
Testing alone is **NOT** enough!

Researchers: Beware of 10-Year-Old Linux Vulnerability

Qualys Says Flaw in Sudo Utility Could Grant Attackers Root Access

Akshaya Asokan (@asokan_akshaya) • January 28, 2021

Microsoft patches critical 17-year-old DNS bug in Windows Server

Chrome: 70% of all security bugs are memory safety issues

Google software engineers are looking into ways of eliminating memory management-related bugs from Chrome.
It’s easy to make mistakes
It’s easy to make mistakes
What about memory safety vulnerabilities?
Prevalence of Memory Safety Vulns

Microsoft Product 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
Modern software design is useful for security
Modern software design is useful for security

Increasing adoption of binning allocators

- Maintains memory locality.
- Implicit lookup of allocation information.
Modern software design is useful for security

Increasing adoption of binning allocators

- Maintains memory locality.
- Implicit lookup of allocation information.
The benefits of No-FAT

- **Fuzz-Testing**: 10X speedup over ASan
- **Runtime Security**: 8% overheads for spatial & temporal memory safety
- **Resilience to Spectre-V1**: Bounds aware memory accesses
Binning Memory Allocators 101
Binning Memory Allocators

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

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

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

Memory is requested by the allocator.
Binning Memory Allocators

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

Memory is divided into bins.
Binning Memory Allocators

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

Each bin is associated with a size.

Bins

Sizes

<table>
<thead>
<tr>
<th>Bins</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>16B</td>
</tr>
<tr>
<td>B</td>
<td>32B</td>
</tr>
<tr>
<td>C</td>
<td>64B</td>
</tr>
</tbody>
</table>

Virtual Memory
Bin Memory Allocations

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

Virtual Memory

<table>
<thead>
<tr>
<th>Bins</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>16B</td>
</tr>
<tr>
<td>B</td>
<td>32B</td>
</tr>
<tr>
<td>C</td>
<td>64B</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Binning Memory Allocators

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

Virtual Memory

Bins

Sizes

A

16B

12B

B

32B

C

64B

...
Given any pointer, we can derive its allocation size and base address.
From Bins to Security
Spatial Memory Safety (Inter-Object)

The Problem

Bin A

Object 1

Object 2

Virtual Memory
Spatial Memory Safety (Inter-Object)

The Problem

Adjacent objects can overflow into each other.

Bin A

Object 1

Object 2

Virtual Memory
Spatial Memory Safety (Inter-Object)

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

```c
int main() {
    char* ptr = malloc(12);
    ptr[1] = 'A';  // store ptr[1], 'A'
    ...
}
```
Spatial Memory Safety (Inter-Object)

```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.
Spatial Memory Safety (Inter-Object)

40. int main() {
41.     char* ptr = malloc(12); \(\rightarrow\) ptr_{trusted\_base}
42.     ptr[1] = ‘A’; \(\rightarrow\) s_store \(ptr[1], ‘A’, ptr_{trusted\_base}\)
43.     ... 
50. }
Spatial Memory Safety (Inter-Object)

```c
40. int main() {
41.   char* ptr = malloc(12);
42.   ptr[1] = 'A';  
43.   ...
44. }
```

The compiler propagates the allocation base address.
Spatial Memory Safety (Inter-Object)

40. int main() {
41.   char* ptr = malloc(12);
42.   ptr[1] = ‘A’;  \textcolor{gray}{\texttt{s_store ptr[1],’A’,ptr}}
43.   ...
50. }
Spatial Memory Safety (Inter-Object)

s_store ptr[1], ‘A’, ptr\text{trusted_base}
Spatial Memory Safety (Inter-Object)

\[ \text{\texttt{s\_store ptr[1],'A',ptr\_trusted\_base}} \]

\[ \begin{align*}
\text{ptr[1]} & \quad \text{ptr\_trusted\_base}
\end{align*} \]
Spatial Memory Safety (Inter-Object)

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

\[
\text{offset} = ptr[1] - ptr_{\text{trusted\_base}}
\]
Spatial Memory Safety (Inter-Object)

```c
s_store ptr[1], 'A', ptr\text{trusted_base}
```

\[
\begin{align*}
\text{offset} & \equiv \quad \text{ptr[1]} \quad \text{ptr}_{\text{trusted_base}} \\
\text{size} & \equiv \quad \text{getSize(} \text{ptr}_{\text{trusted_base}} \text{)}
\end{align*}
\]
Spatial Memory Safety (Inter-Object)

```
\texttt{s}\_\texttt{store} \texttt{ptr[1]},’A’,\texttt{ptr}_{\texttt{trusted\_base}}
```

```
\texttt{offset} \texttt{=} \texttt{ptr[1]} \texttt{=} \texttt{ptr}_{\texttt{trusted\_base}}
```

```
\texttt{size} \texttt{=} \texttt{getSize(} \texttt{ptr}_{\texttt{trusted\_base}} \texttt{)}
```

 Bounds Check

```
\texttt{offset} \texttt{<} \texttt{size} \texttt{?}
```
Spatial Memory Safety (Inter-Object)

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

\[
\text{size} \quad \equiv \quad \text{getSize}(\text{ptr}_{\text{trusted base}})
\]

Bounds Check

\[
\text{offset} \quad \text{size} \quad \text{?}
\]
Spatial Memory Safety (Inter-Object)

```c
int main() {
    char* ptr = malloc(12);
    ptr[1] = 'A';
    s_store ptr[1], 'A', ptr
    ...
}
```
Spatial Memory Safety (Inter-Object)

Let's pass the pointer to another context (e.g., foo).
Spatial Memory Safety (Inter-Object)

```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';
    ...
}
```
Spatial Memory Safety (Inter-Object)

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. `    s_store xptr[7], 'A', xptr_trusted_base`
50. `    ...`
51. `}`
Spatial Memory Safety (Inter-Object)

```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'; // s_store xptr[7], 'A', xptr_trusted_base
    ...
}
```

How do we get this?
```c
40. int main() {
41.     char* ptr = malloc(12);               /*ptr_trusted_base*/
42.     ptr[1] = 'A';                        /*s_store ptr[1],‘A’,ptr_trusted_base*/
43.     ...
49.     foo(ptr);
50. }
51. void Foo (char* xptr){
52.     ...
53.     xptr[7] = 'B';                      /*s_store xptr[7],‘A’,xptr_trusted_base*/
54.     ...
60. }
```
Spatial Memory Safety (Inter-Object)

\[ \texttt{xptr_{\text{trusted base}} \leftarrow \text{compBase(xptr[7])}} \]
Spatial Memory Safety (Inter-Object)

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

Bin \[ \equiv \text{xptr} \gg \log_2(S) \]

where S is the size of the bins.
**Spatial Memory Safety** (Inter-Object)

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

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

size \( \equiv \) getSize(Bin)
Spatial Memory Safety (Inter-Object)

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

\[
\begin{align*}
\text{Bin} & \equiv xptr \gg \log_2(S) \\
\text{size} & \equiv \text{getSize(Bin)} \\
\end{align*}
\]

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

where S is the size of the bins.
**Spatial Memory Safety** (Inter-Object)

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

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

\[
\text{size} = \text{getSize(Bin)}
\]

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

Base pointer is implicitly derived!
int main() {
    char* ptr = malloc(12);
    ptr[1] = 'A';
    ... 
    foo(ptr);
}
Spatial Memory Safety (Inter-Object)

```c
int main() {
    char* ptr = malloc(12);
    ptr[1] = 'A';
    ptr = ptr + 100;
    ... foo(ptr);
}
```

Pointer arithmetic can push the pointer out-of-bounds before calling `foo`!
Spatial Memory Safety (Inter-Object)

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

Verify the bounds of all pointers that escape to memory (or another function).
Spatial Memory Safety (Intra-Object)

The Problem
**Spatial Memory Safety** *(Intra-Object)*

The Problem

typedef struct {
    char a;
    double b;
    char c[3];
    void (*fp)();
} A_t;

Adjacent fields can be overflowed into.
Spatial Memory Safety (Intra-Object)

The **Buf2Ptr** transformation promotes intra-allocation buffers to standalone allocations.
Temporal Memory Safety
Temporal Memory Safety

The Problem

malloc
#1

0x00004000

A

Time
Temporal Memory Safety

The Problem

malloc #1

```
0x00004000
```

`A`

`free`

Time
Temporal Memory Safety

The Problem

malloc #1

malloc #2
Temporal Memory Safety

The Problem

Different allocations reuse old memory!
Temporal Memory Safety

No-FAT adds a 16-bit random tag is added to every pointer.
ISA Extensions

No FAT
ISA Extensions

1. s_store Addr, Dest, BaseAddr

2. s_load Addr, Src, BaseAddr
ISA Extensions

1. \texttt{s\_store \, Addr, Dest, BaseAddr}
2. \texttt{s\_load \, Addr, Src, BaseAddr}
3. \texttt{verifyBounds \, Addr, BaseAddr}
ISA Extensions

1. \texttt{s\_store \textit{Addr}, \textit{Dest}, \textit{BaseAddr}}
2. \texttt{s\_load \textit{Addr}, \textit{Src}, \textit{BaseAddr}}
3. \texttt{verifyBounds \textit{Addr}, \textit{BaseAddr}}

Exceptions are thrown in the case the target memory address does not match \texttt{BaseAddr}.
ISA Extensions

1. $s_{\text{store}} \text{ Addr, Dest, BaseAddr}$
2. $s_{\text{load}} \text{ Addr, Src, BaseAddr}$
3. $\text{verifyBounds Addr, BaseAddr}$
4. $\text{compBase Addr, Dest}$

Exceptions are thrown in the case the target memory address does not match $\text{BaseAddr}$. 
Microarchitectural Overview
Microarchitectural Overview

CPU → L1-D → L2 → DRAM
Microarchitectural Overview

Diagram:
- CPU
- L1-D
- L2
- DRAM

Annotations:
- Dedicated Register File
Microarchitectural Overview

1-KiB Memory Allocation Sizes Table (MAST)
Microarchitectural Overview

No changes to the memory subsystems!
Resilience to Common Exploits
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot corrupt memory.
Resilience to Common Exploits

1. **Buffer Over-/Under-flows**
   - Cannot corrupt memory.

Original: Buffer A | Buffer B

Buffer A

Buffer B
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot corrupt memory.

Original:
Buffer A  Buffer B

No-FAT:
Buffer A  Buffer B

Exception!
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot corrupt memory.

2. Use-after-free
   Each allocation instance is tagged randomly.

Tag is propagated with the allocation base address.
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot corrupt memory.

2. Use-after-free
   Each allocation instance is tagged randomly.

3. Spectre-V1

// mispredicted branch
if (i < sizeof(a)) {
    secret = a[i];
    // secret is leaked
    val = b[64 * secret];
}
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot corrupt memory.

2. Use-after-free
   Each allocation instance is tagged randomly.

3. Spectre-V1
   Speculative loads are aware of the legitimate allocation-bounds.

```c
// mispredicted branch
if (i < sizeof(a)) {
    secret = a[i];
    // secret is leaked
    val = b[64 * secret];
}
```
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot corrupt memory.

2. Use-after-free
   Each allocation instance is tagged randomly.

3. Spectre-V1
   Speculative loads are aware of the legitimate allocation-bounds.

```c
// mispredicted branch
if (i < sizeof(a)) {
    secret = a[i];
    // secret is leaked
    val = b[64 * secret];
}
```

- Speculative out-of-bounds loads are not allowed to change the cache state or forward values to dependent instructions.
Resilience to Common Exploits

1. Buffer Over-/Under-flows
   Cannot corrupt memory.

2. Use-after-free
   Each allocation instance is tagged randomly.

3. Spectre-V1
   Speculative loads are aware of the legitimate allocation-bounds.

```c
// mispredicted branch
if (i < sizeof(a)) {
    secret = a[i];
    // secret is leaked
    val = b[64 * secret];
}
```

- Speculative out-of-bounds loads are not allowed to change the cache state or forward values to dependent instructions.
No FAT

Performance
Performance Overheads

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

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

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

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

**Software Modifications**
- Our special load/stores do not change the binary size.
- We verify pointer bounds before storing them to memory.
Performance Overheads

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

Software Modifications
- Our special load/stores do not change the binary size.
- We verify pointer bounds before storing them to memory.
- We compute the allocation base address of arbitrary pointers when they are loaded from memory.
Performance Results (x86_64)

Experimental Setup
We use emulate NO-FAT on x86_64 by modifying LLVM to emit new instructions.
- CompBase is emulated using two multiplications followed by a store.
- VerifyBounds is emulated using dummy stores.
Performance Results (x86_64)
Performance Results (x86_64)

- 500.perlbench_r
- 502.gcc_r
- 505.mcf_r
- 508.namd_r
- 510.parest_r
- 511.povray_r
- 519.lbm_r
- 520.omnetpp_r
- 523.xalancbmk_r
- 525.x264_r
- 526.blender_r
- 531.deepsjeng_r
- 538.imagick_r
- 541.leela_r
- 544.nab_r
- 557.xz_r

Norm. Exec:

- Binning-Malloc
- No-FAT

4%
Performance Results (x86_64)
Performance Results (x86_64)
Performance Results (x86_64)

![Graph showing performance results with normalized execution times for different benchmarks and optimizations.]

- Intel MPX
- Address Sanitizer
- Software-EBB
- Binning-Malloc
- No-FAT

The graph shows normalized execution times for various benchmarks with different optimizations applied. The metrics include gMean and other performance indicators.
We reduce the average runtime overheads of full memory safety from 100% to 8%!
No FAT

Related Work
## Related Work

<table>
<thead>
<tr>
<th>Technique</th>
<th>Metadata</th>
<th>Security Coverage</th>
</tr>
</thead>
</table>
## Related Work

<table>
<thead>
<tr>
<th>Technique</th>
<th>Metadata</th>
<th>Security Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Base &amp; Bounds</td>
<td>N-bits per pointer or allocation</td>
<td>Complete</td>
</tr>
</tbody>
</table>
## Related Work

<table>
<thead>
<tr>
<th>Technique</th>
<th>Metadata</th>
<th>Security Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Base &amp; Bounds</td>
<td>N-bits per pointer or allocation</td>
<td>Complete</td>
</tr>
<tr>
<td>Memory Tagging</td>
<td>N-bits per pointer &amp; allocation</td>
<td>Limited by tag width</td>
</tr>
</tbody>
</table>
# Related Work

<table>
<thead>
<tr>
<th>Technique</th>
<th>Metadata</th>
<th>Security Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Base &amp; Bounds</td>
<td>N-bits per pointer or allocation</td>
<td>Complete</td>
</tr>
<tr>
<td>Memory Tagging</td>
<td>N-bits per pointer &amp; allocation</td>
<td>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>
# Related Work

<table>
<thead>
<tr>
<th>Technique</th>
<th>Metadata</th>
<th>Security Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Base &amp; Bounds</td>
<td>N-bits per pointer or allocation</td>
<td>Complete</td>
</tr>
<tr>
<td>Memory Tagging</td>
<td>N-bits per pointer &amp; allocation</td>
<td>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>No-FAT</td>
<td>Fixed (1K) bits per process</td>
<td>Complete</td>
</tr>
</tbody>
</table>
Takeaways

Having no metadata

✓ Improves Fuzzing
✓ Improves Runtime Security
✓ Improves Resilience to Spectre-V1
Takeaways

Having no metadata

✓ Improves Fuzzing
✓ Improves Runtime Security
✓ Improves Resilience to Spectre-V1

Checkout ZeRO for end-user deployment!
https://isca21.arroyo.me
Takeaways

Having no metadata

✓ Improves Fuzzing
✓ Improves Runtime Security
✓ Improves Resilience to Spectre-V1

The benefits of having allocation sizes as an architectural feature can go well beyond memory safety!
Backup Slides