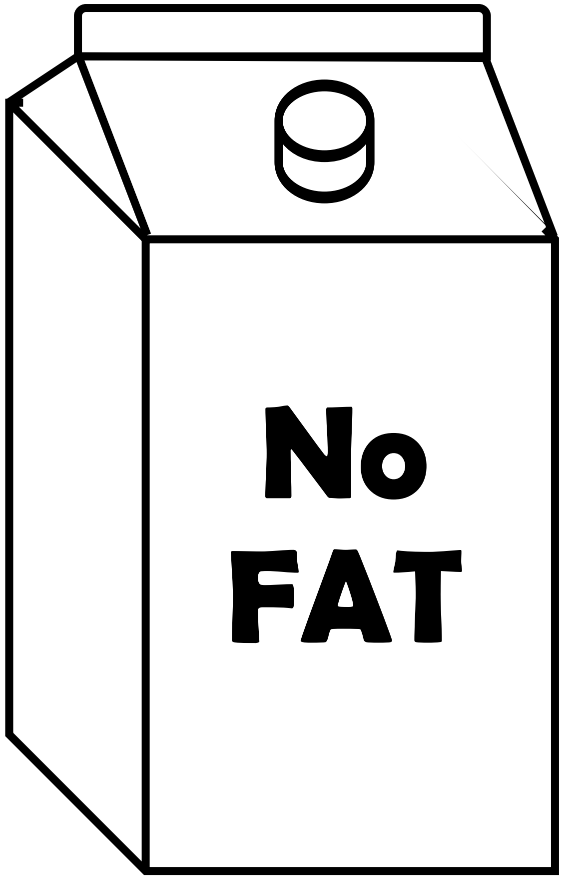


Architectural Support for Low Overhead Memory Safety Checks

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

CS@CU Computer Science

Columbia University

06/16/2021

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!

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.

EDITOR'S PICK | 42,742 views | Nov 21, 2018, 07:00am

Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder

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.

The New York Times

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

EDITOR'S PICK | 42,742 views | Nov 21, 2018, 07:00am

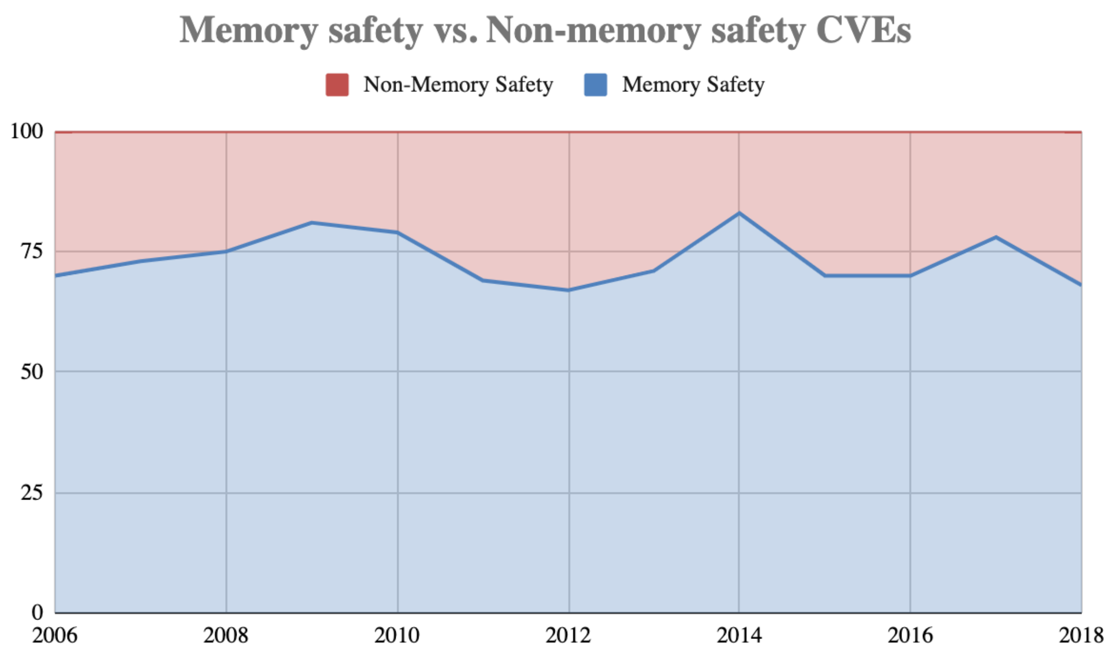
Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder

It’s easy to make mistakes

It’s easy to make mistakes

SEGFAULT!

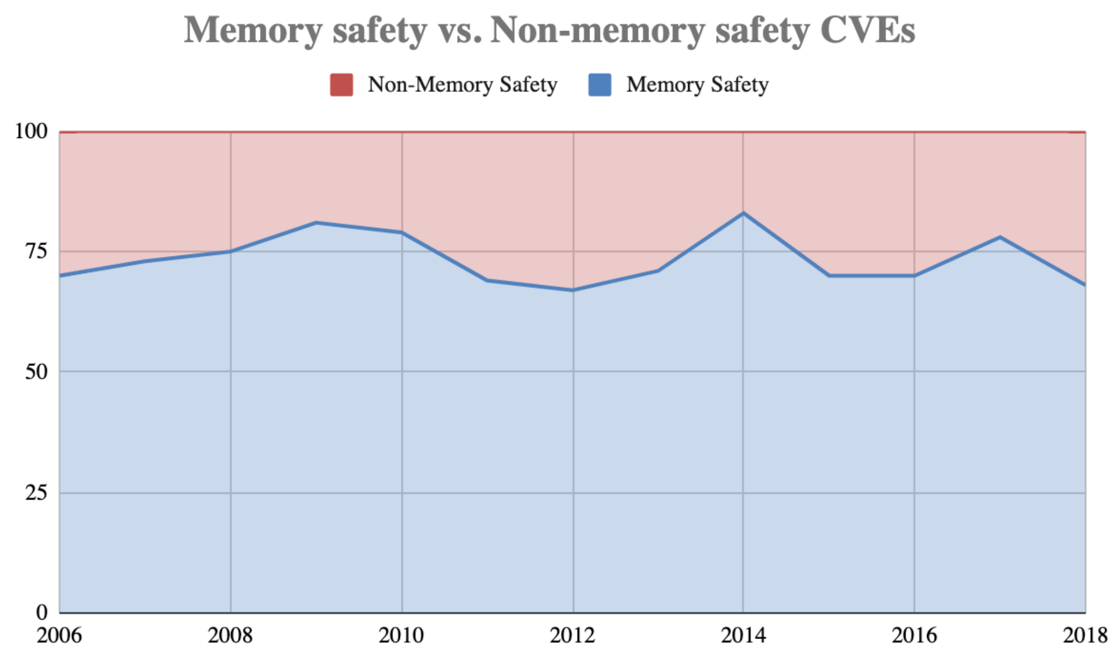
Prevalence of Memory Safety Vulns



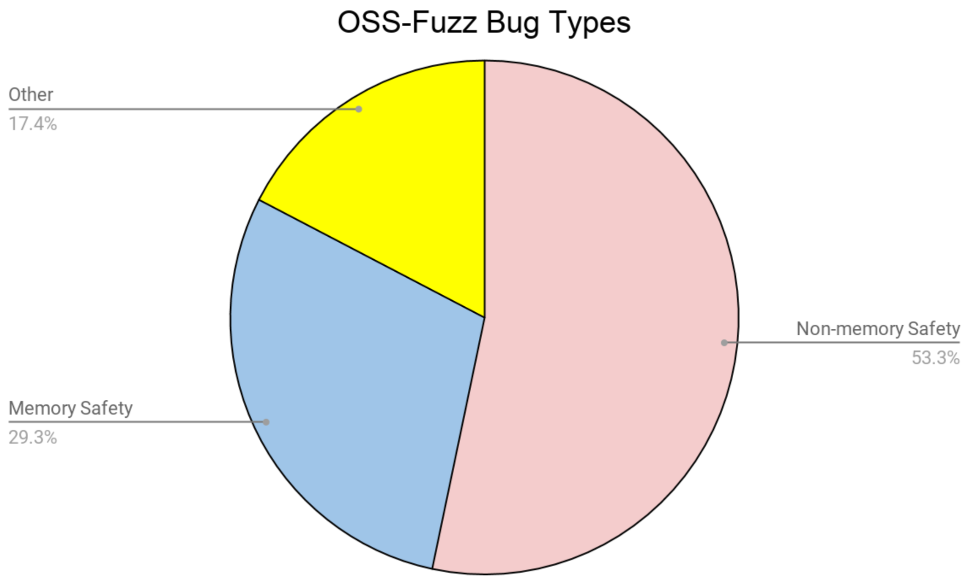
Microsoft Product CVEs

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*

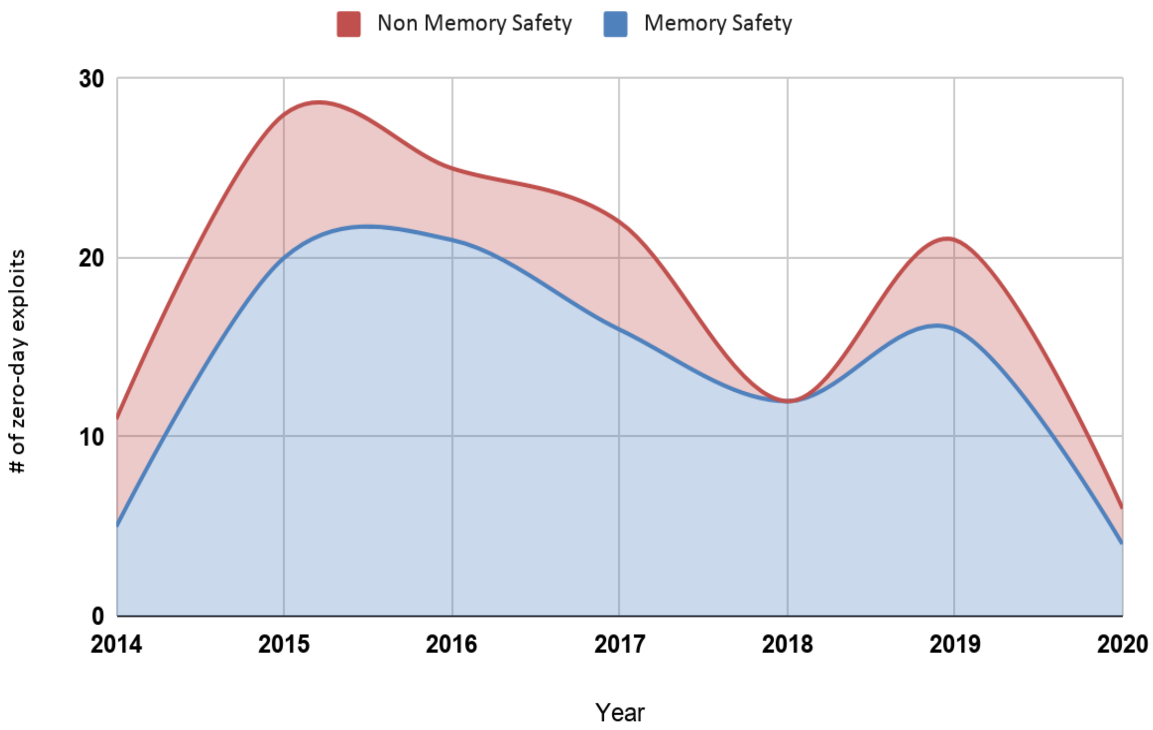
Source: *https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html*


TEEN COMIX
BRANSON '15
NO FEAR
WHAT ABOUT MEMORY SAFETY VULNERABILITIES?
ONE FEAR

ATTACKERS

MEMORY SAFETY

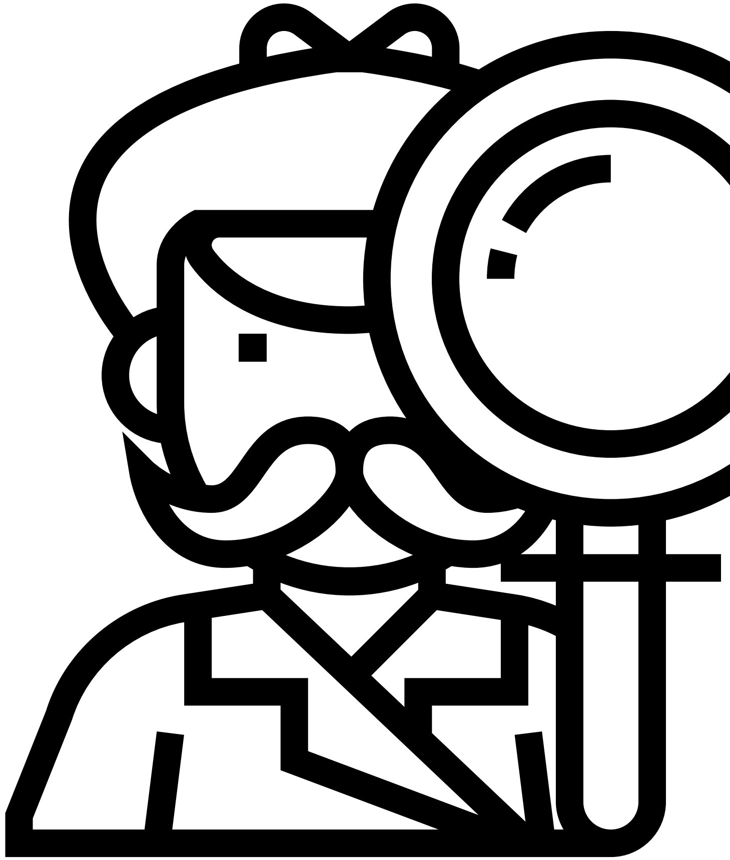
Attackers prefer Memory Safety Vulns



Zero-day "in the wild" exploits
from 2014-2020

Source: *Google Project Zero, 0day "In the Wild" spreadsheet. Last updated: April 2020*

ALL HOPE IS NOT LOST!



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.

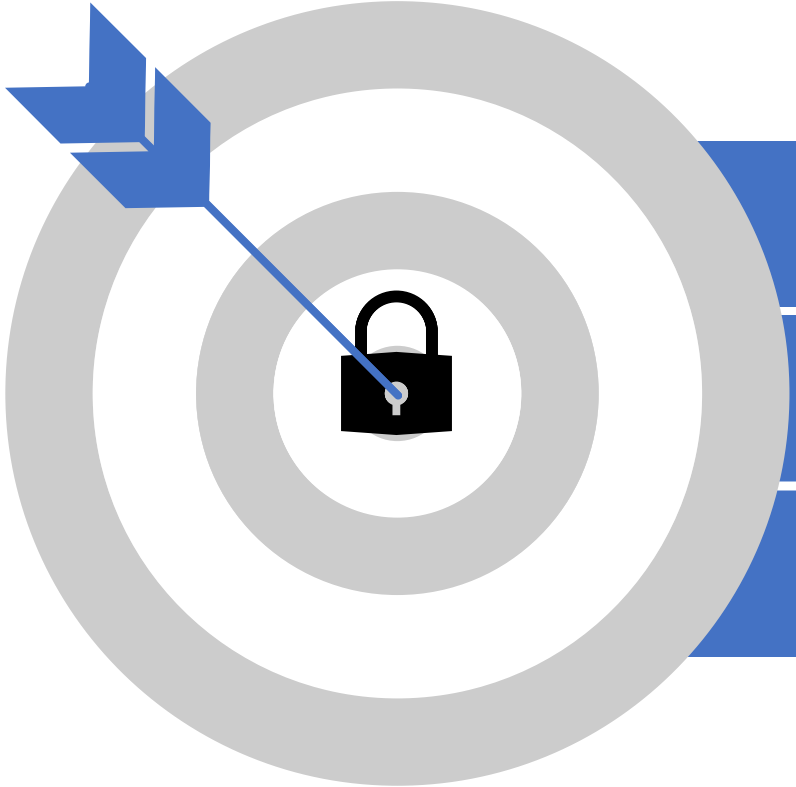
Modern software design is useful for security

Increasing adoption of binning allocators

- Maintains memory locality.
- Implicit lookup of allocation information.

tcMalloc

The benefits of No-FAT



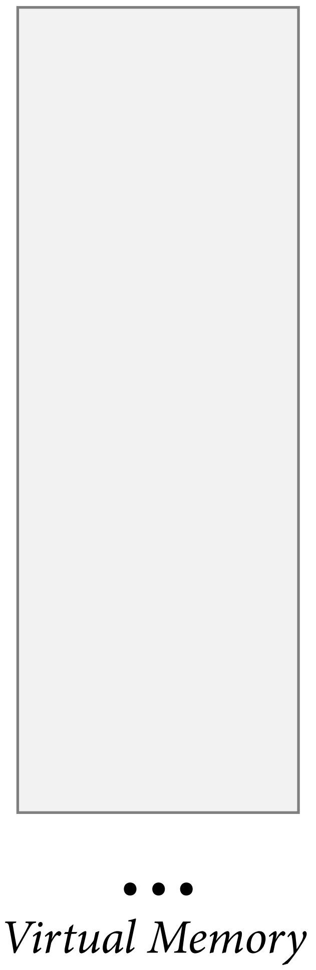
Benefit	Result
Fuzz-Testing	**10X speedup over ASan**
Runtime Security	**8% overheads for spatial & temporal memory safety**
Resilience to Spectre-V1	**Bounds aware memory accesses**

Binning Memory Allocators

101

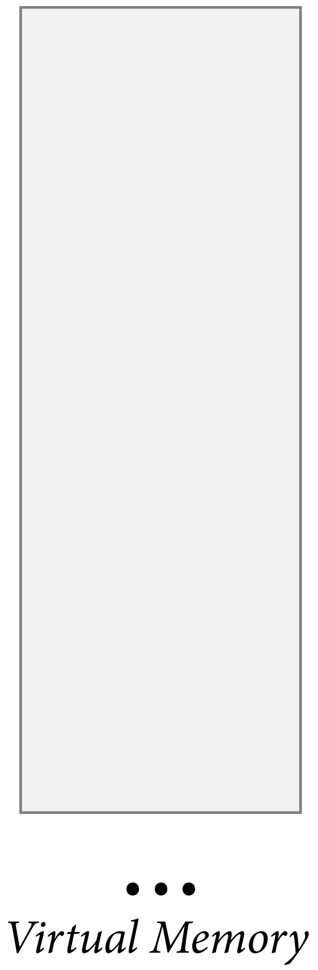
Binning Memory Allocators

```c
int main() {
    char* ptr = malloc(12);
    …
}
```



Binning Memory Allocators

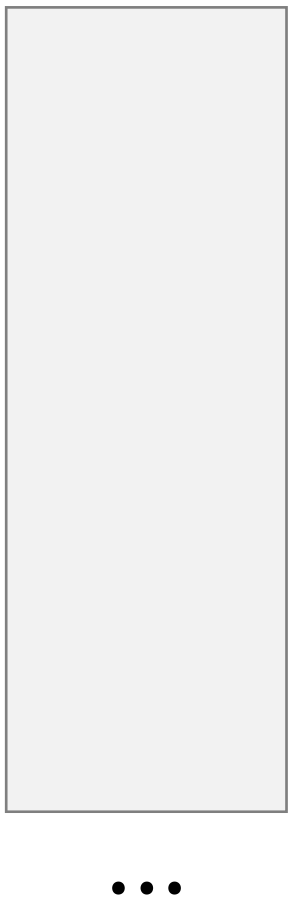
```
int main() {
    char* ptr = malloc(12);
    …
}
```



Binning Memory Allocators

```
int main() {
    char* ptr = malloc(12);
    …
}
```

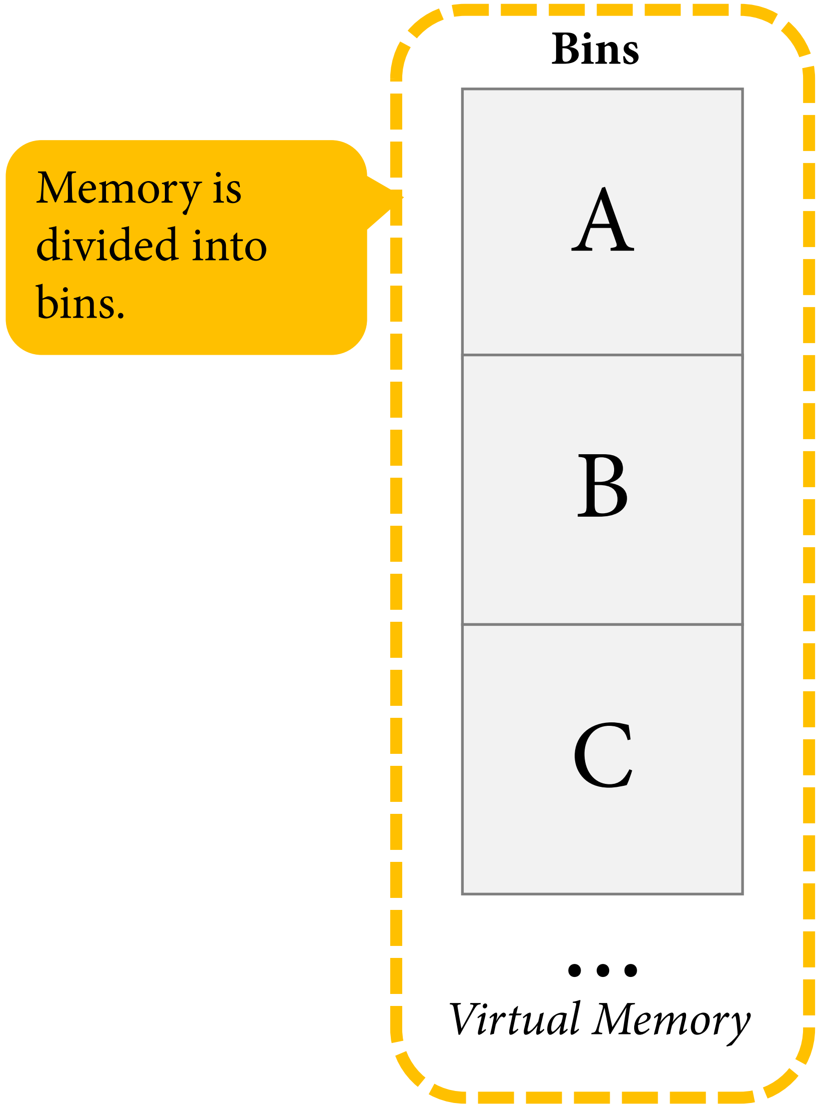
Memory is requested by the allocator.



Virtual Memory

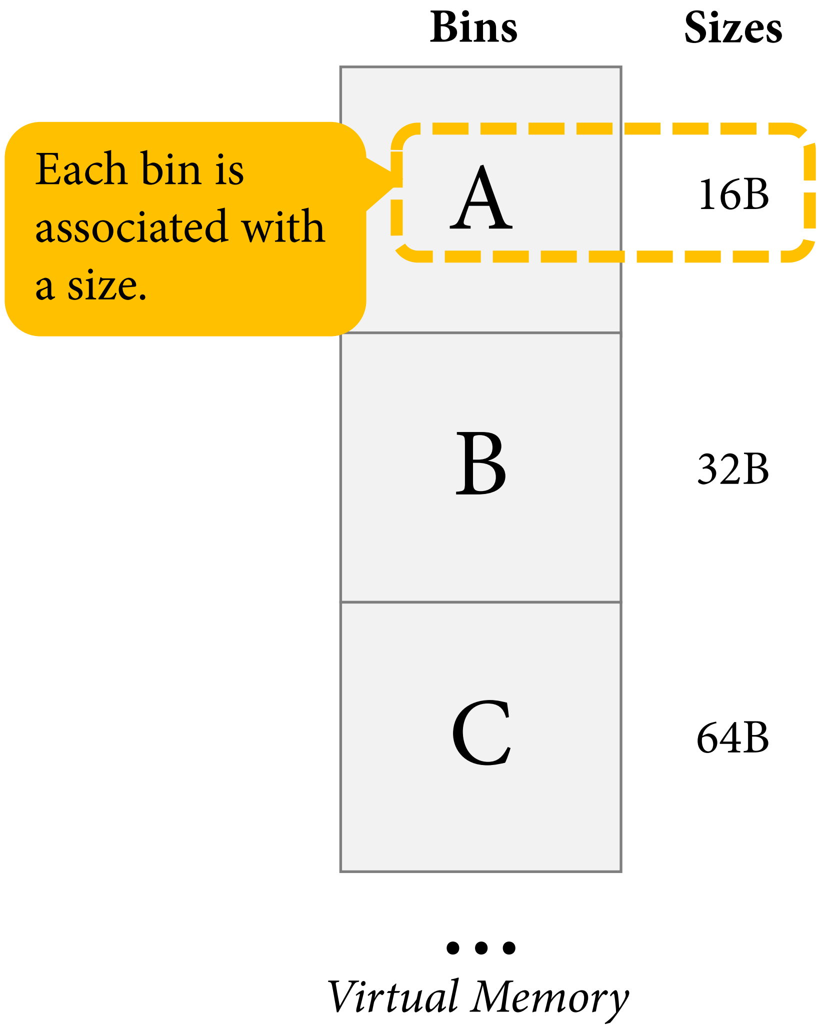
Binning Memory Allocators

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



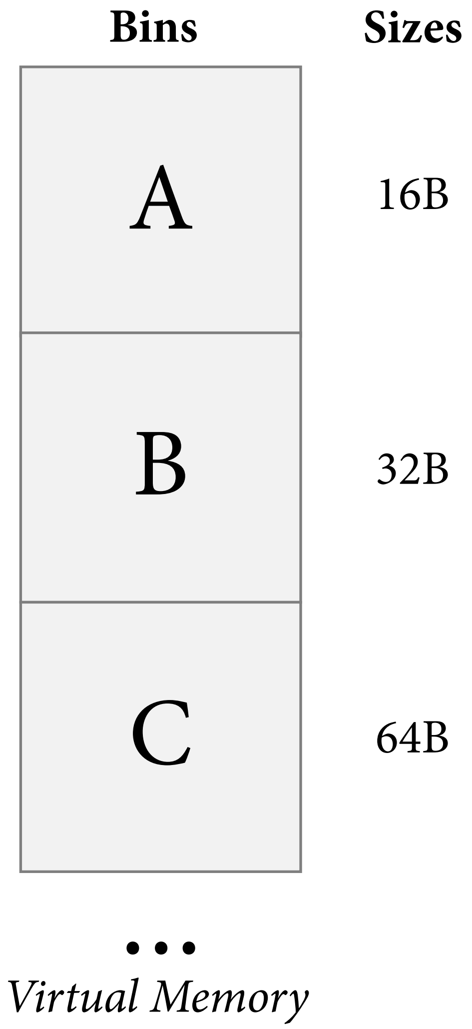
Binning Memory Allocators

```
int main() {
  char* ptr = malloc(12);
  …
}
```



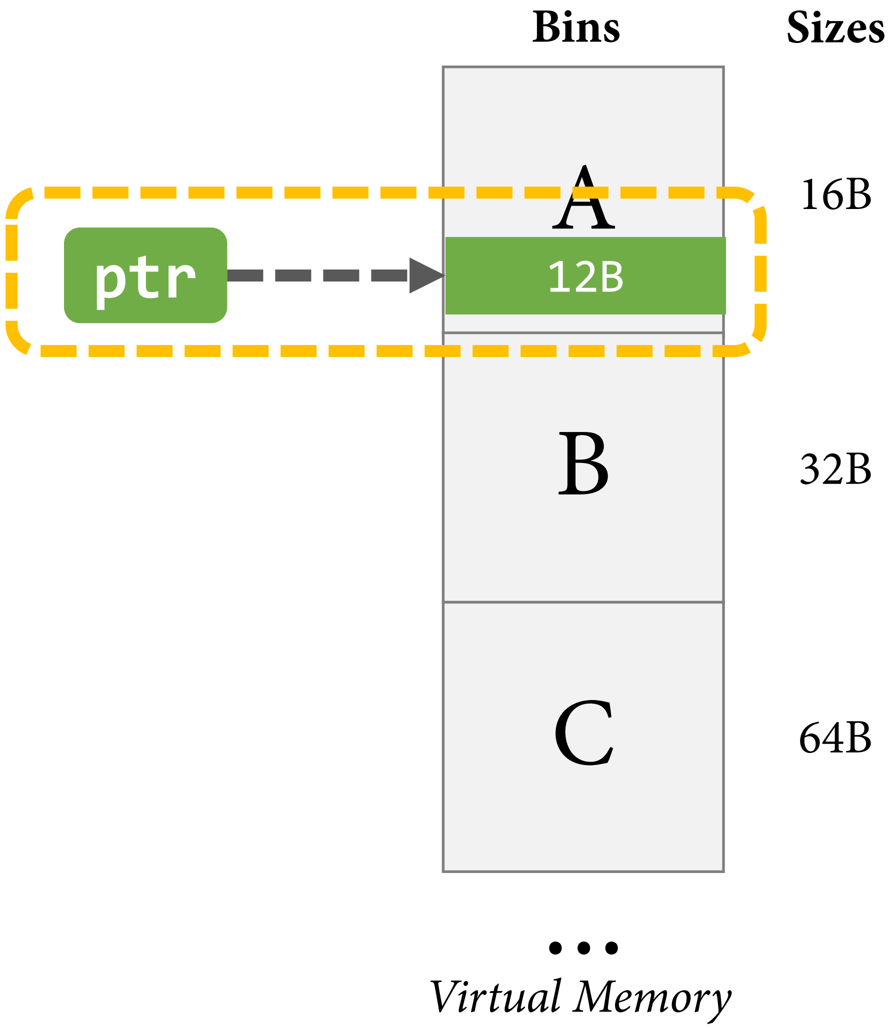
Binning Memory Allocators

```
int main() {
    char* ptr = 12B
    ...
}
```



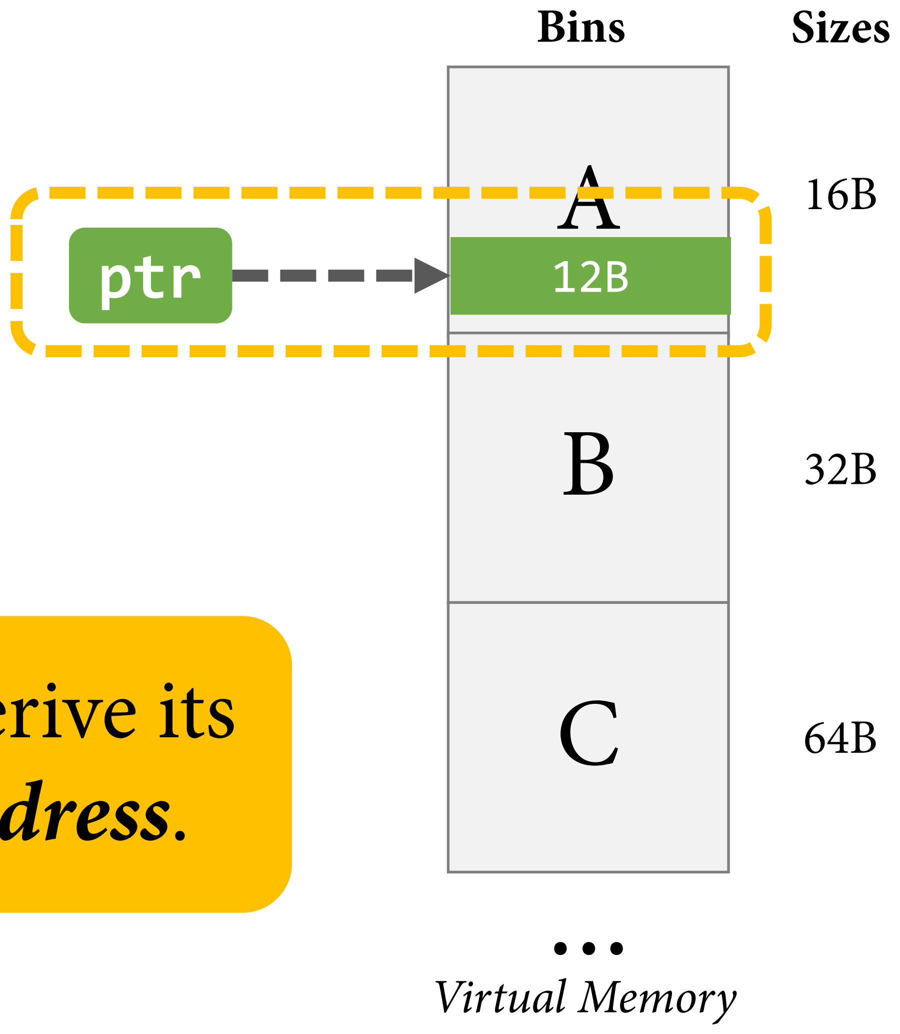
Binning Memory Allocators

```
int main() {
  char* ptr = malloc(12);
  ...
}
```

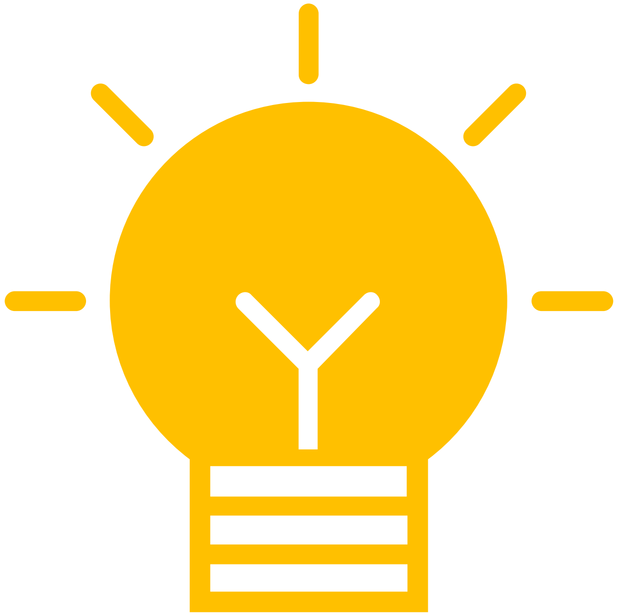


Binning Memory Allocators

```
int main() {
  char* ptr = malloc(12);
  …
}
```



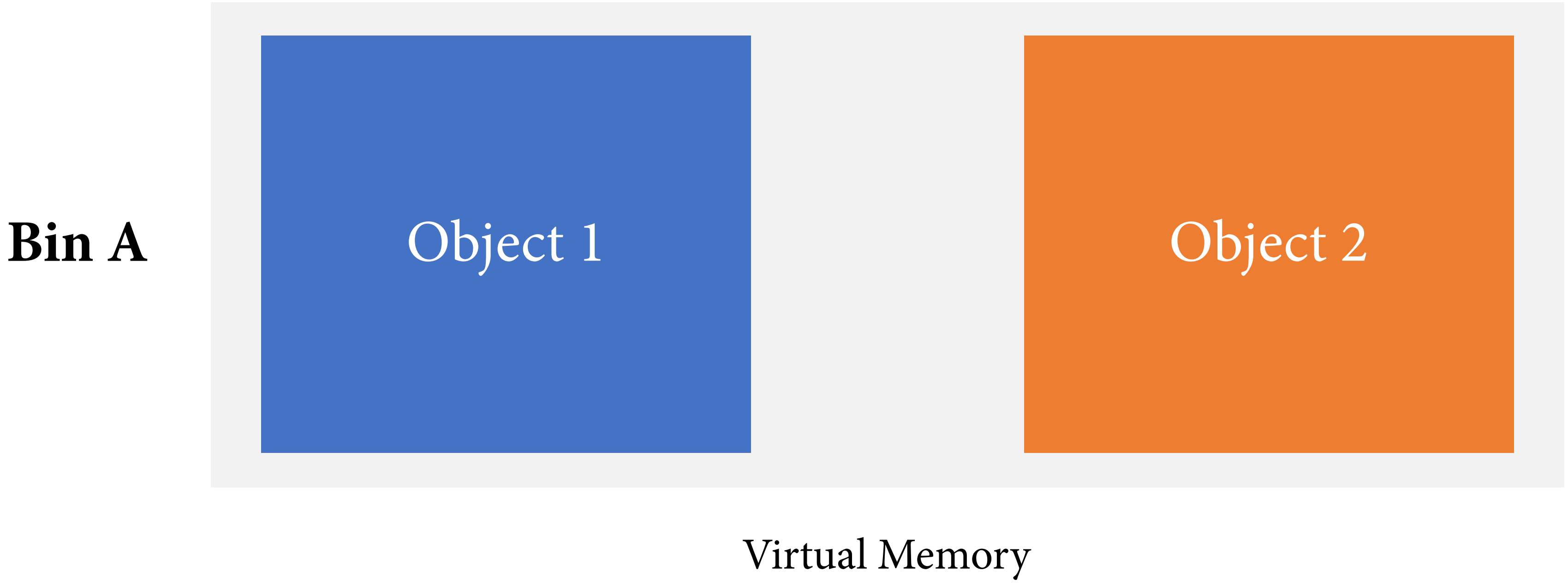
Given **any** pointer, we can derive its ***allocation size*** and ***base address***.



From Bins to Security

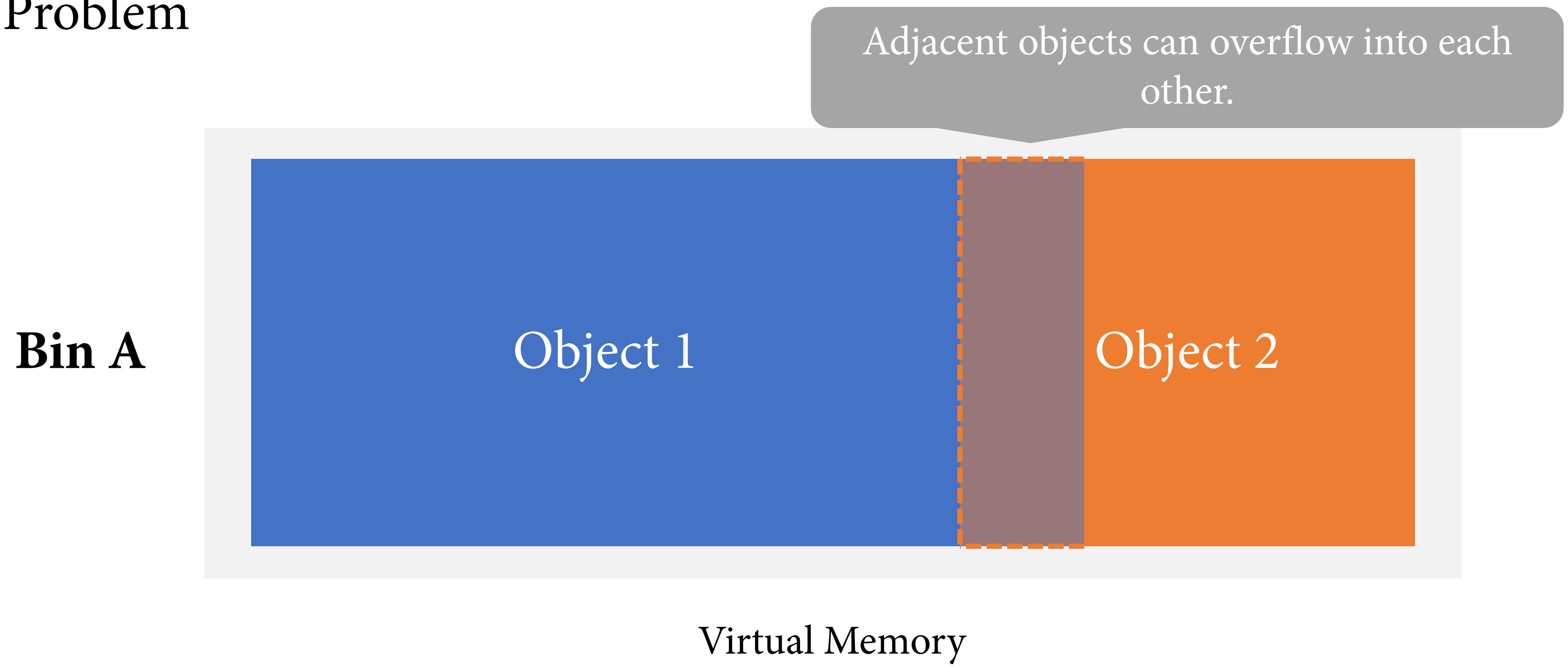
Spatial Memory Safety (Inter-Object)

The Problem



Spatial Memory Safety (Inter-Object)

The Problem



Spatial Memory Safety (Inter-Object)

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

Spatial Memory Safety (Inter-Object)

```
int main() {
  char* ptr = malloc(12);
  ptr[1] = ‘A’;        ➡  store ptr[1],‘A’
  …
}
```

Spatial Memory Safety (Inter-Object)

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

```
s_store ptr[1],‘A’,ptr_trusted_base
```

We add **one extra** operand for loads/stores.

Spatial Memory Safety (Inter-Object)

```
int main() {
  char* ptr = malloc(12);
  ptr[1] = 'A';
  …
}
```

→ $ptr_{trusted\_base}$

s_store ptr[1],'A',$ptr_{trusted\_base}$

Spatial Memory Safety (Inter-Object)

```
int main() {
  char* ptr = malloc(12);
  ptr[1] = 'A';
  …
}
```

s_store ptr[1], 'A' $ptr_{trusted\_base}$

The compiler **propagates** the allocation base address.

Spatial Memory Safety (Inter-Object)

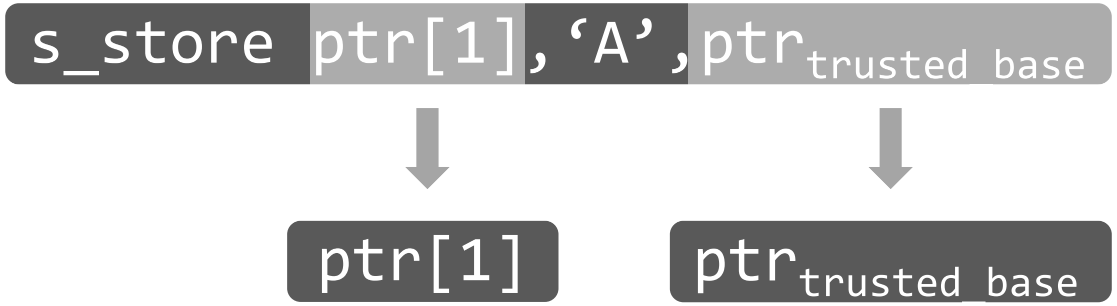
```
int main() {
  char* ptr = malloc(12);
  ptr[1] = ‘A’;
  …
}
```

s_store ptr[1],‘A’,$ptr_{trusted\_base}$

Spatial Memory Safety (Inter-Object)

```
s_store ptr[1],‘A’,ptr_trusted_base
```

Spatial Memory Safety (Inter-Object)



Spatial Memory Safety (Inter-Object)

```
s_store ptr[1],‘A’,ptr_trusted_base
```

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

Spatial Memory Safety (Inter-Object)

`s_store ptr[1],'A',ptr`$_{trusted\_base}$

offset = ptr[1] − ptr$_{trusted_base}$

size = getSize(ptr$_{trusted_base}$)

Spatial Memory Safety (Inter-Object)

`s_store ptr[1],'A',ptr`$_{trusted\_base}$

offset = ptr[1] − ptr$_{trusted_base}$

size = getSize(ptr$_{trusted_base}$)

Bounds Check

offset < size ?

Spatial Memory Safety (Inter-Object)

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

size = getSize($ptr_{trusted\_base}$)

Bounds Check

offset < size ?

Spatial Memory Safety (Inter-Object)

```
int main() {
  char* ptr = malloc(12);
  ptr[1] = 'A';
  …
}
```

$ptr_{trusted\_base}$

s_store ptr[1],'A',$ptr_{trusted\_base}$

Spatial Memory Safety (Inter-Object)

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

$ptr_{trusted\_base}$

s_store ptr[1],‘A’,$ptr_{trusted\_base}$

Let’s pass the pointer to another context (e.g., `foo`).

Spatial Memory Safety (Inter-Object)

```
int main() {
  char* ptr = malloc(12);
  ptr[1] = ‘A’;
   …
   foo(ptr);
}
void Foo (char* xptr){
    …
    xptr[7] = ‘B’;
    …
}
```

$ptr_{trusted\_base}$

s_store ptr[1],‘A’,$ptr_{trusted\_base}$

Spatial Memory Safety (Inter-Object)

```
int main() {
  char* ptr = malloc(12);
  ptr[1] = ‘A’;
   …
   foo(ptr);
}
void Foo (char* xptr){
    …
    xptr[7] = ‘B’;
    …
}
```

$ptr_{trusted\_base}$

s_store ptr[1],‘A’,$ptr_{trusted\_base}$

→ s_store xptr[7],‘A’,$xptr_{trusted\_base}$

Spatial Memory Safety (Inter-Object)

```
int main() {
  char* ptr = malloc(12);
  ptr[1] = ‘A’;
   …
   foo(ptr);
}
void Foo (char* xptr){
    …
    xptr[7] = ‘B’;
    …
}
```

$ptr_{trusted\_base}$

s_store ptr[1],‘A’,$ptr_{trusted\_base}$

s_store xptr[7],‘A’,$xptr_{trusted\_base}$

How do we get this?

Spatial Memory Safety (Inter-Object)

```
int main() {
   char* ptr = malloc(12);
   ptr[1] = ‘A’;
   …
   foo(ptr);
}
void Foo (char* xptr){
    …
    xptr[7] = ‘B’;
    …
}
```

$ptr_{trusted\_base}$

s_store ptr[1],‘A’,$ptr_{trusted\_base}$

$xptr_{trusted\_base}$ ← compBase(xptr[7])

s_store xptr[7],‘A’,$xptr_{trusted\_base}$

Spatial Memory Safety (Inter-Object)

```
xptr_trusted base ← compBase(xptr[7])
```

Spatial Memory Safety (Inter-Object)

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

`Bin` = `xptr` >> $\log_2(S)$ where S is the size of the bins.

Spatial Memory Safety (Inter-Object)

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

`Bin` = `xptr` >> $log_2(S)$ where S is the size of the bins.

`size` = `getSize( Bin )`

Spatial Memory Safety (Inter-Object)

$xptr_{trusted\_base} \leftarrow \texttt{compBase(xptr[7])}$

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

$size = getSize(Bin)$

$xptr_{trusted\_base} = \lfloor xptr \times (1/size) \rfloor \times size$

Spatial Memory Safety (Inter-Object)

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

Bin = xptr >> $\log_2(S)$ where S is the size of the bins.

size = getSize(Bin)

$xptr_{trusted\_base} = \lfloor xptr \times (1/ size) \rfloor \times size$

Base pointer is **implicitly** derived!

Spatial Memory Safety (Inter-Object)

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

$ptr_{trusted\_base}$

s_store ptr[1],'A',$ptr_{trusted\_base}$

Spatial Memory Safety (Inter-Object)

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

$ptr_{trusted\_base}$

s_store ptr[1],'A',$ptr_{trusted\_base}$

Pointer arithmetic can push the pointer out-of-bounds before calling `foo`!

Spatial Memory Safety (Inter-Object)

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

ptr_trusted_base

s_store ptr[1],'A',ptr_trusted_base

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)

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

→

```
typedef struct {
  char a;
  double b;
  A_t_c *c_ptr;
  void (*fp)();
} A_t;
```

+

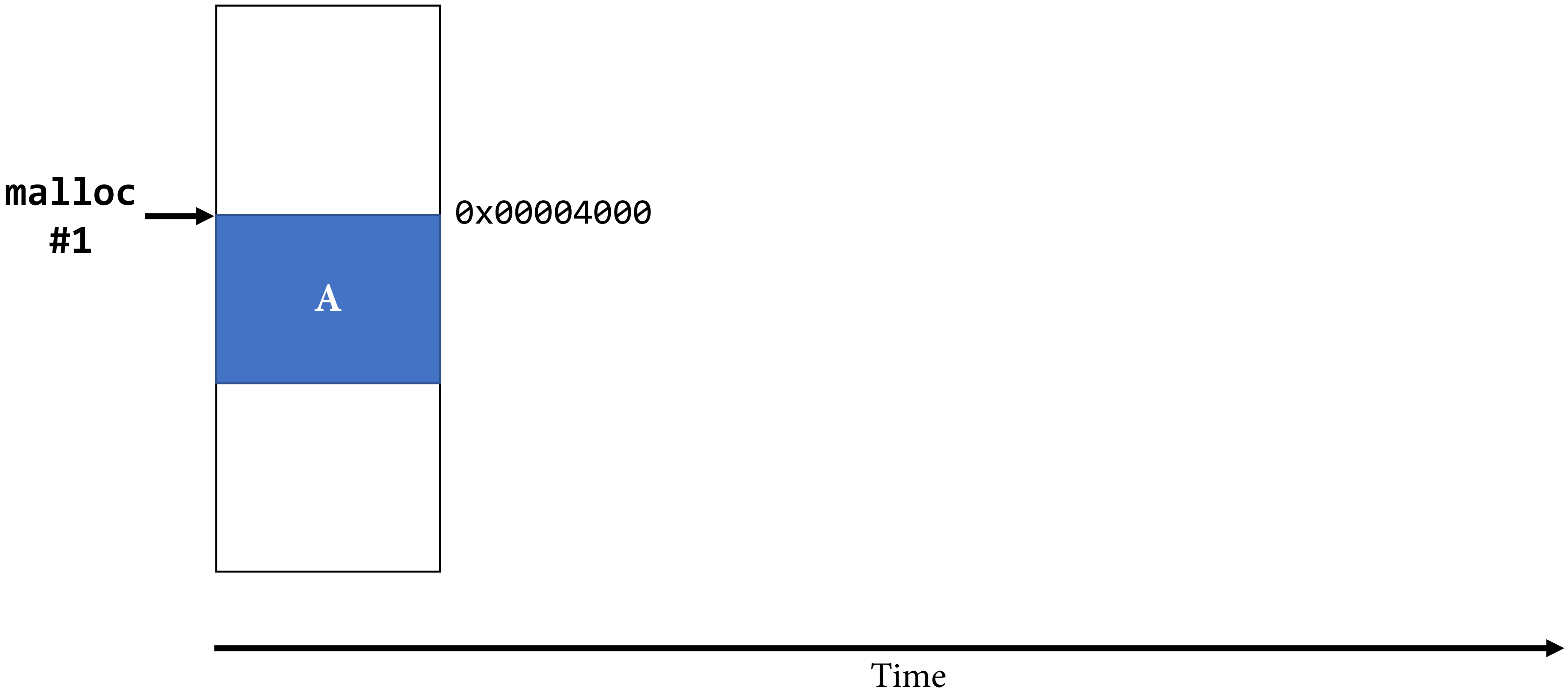
```
typedef struct {
  char c[3];
} A_t_c;
```

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

Temporal Memory Safety

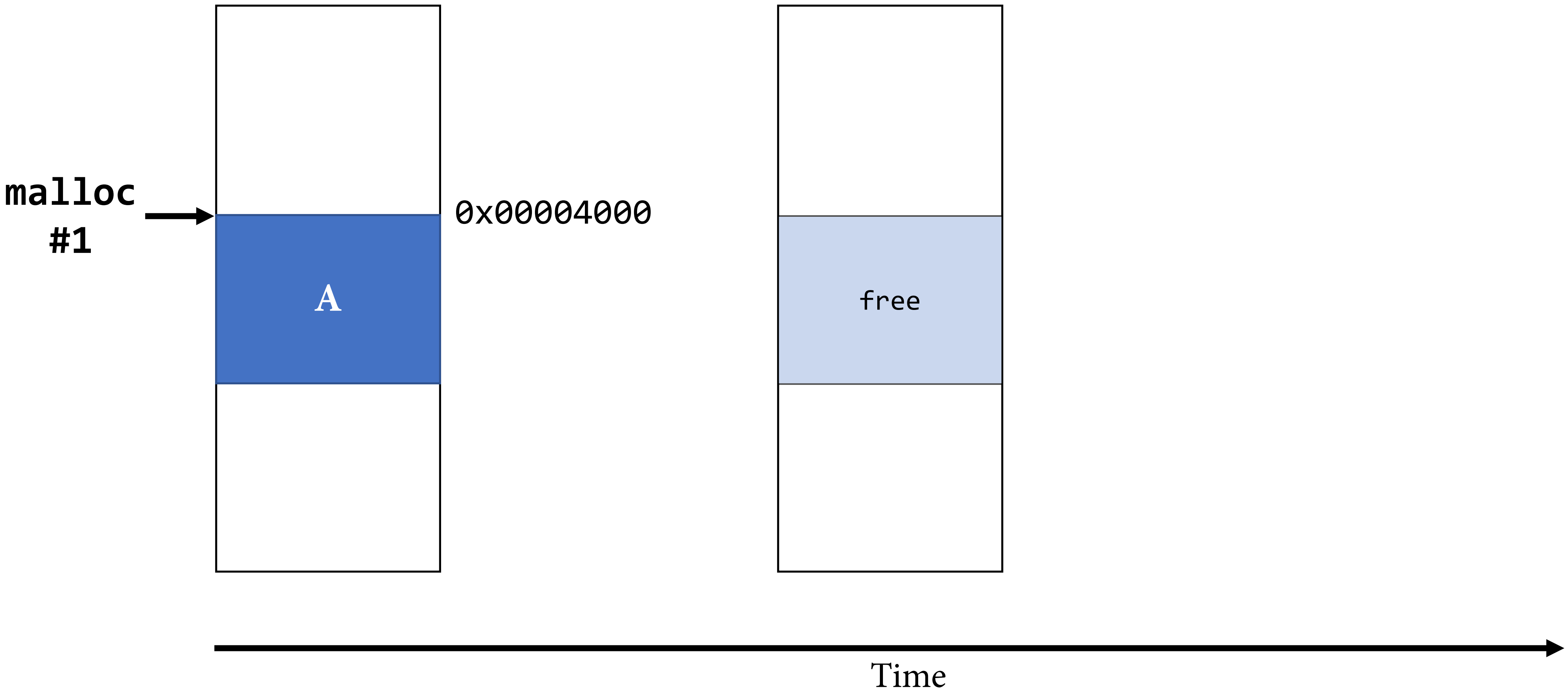
Temporal Memory Safety

The Problem



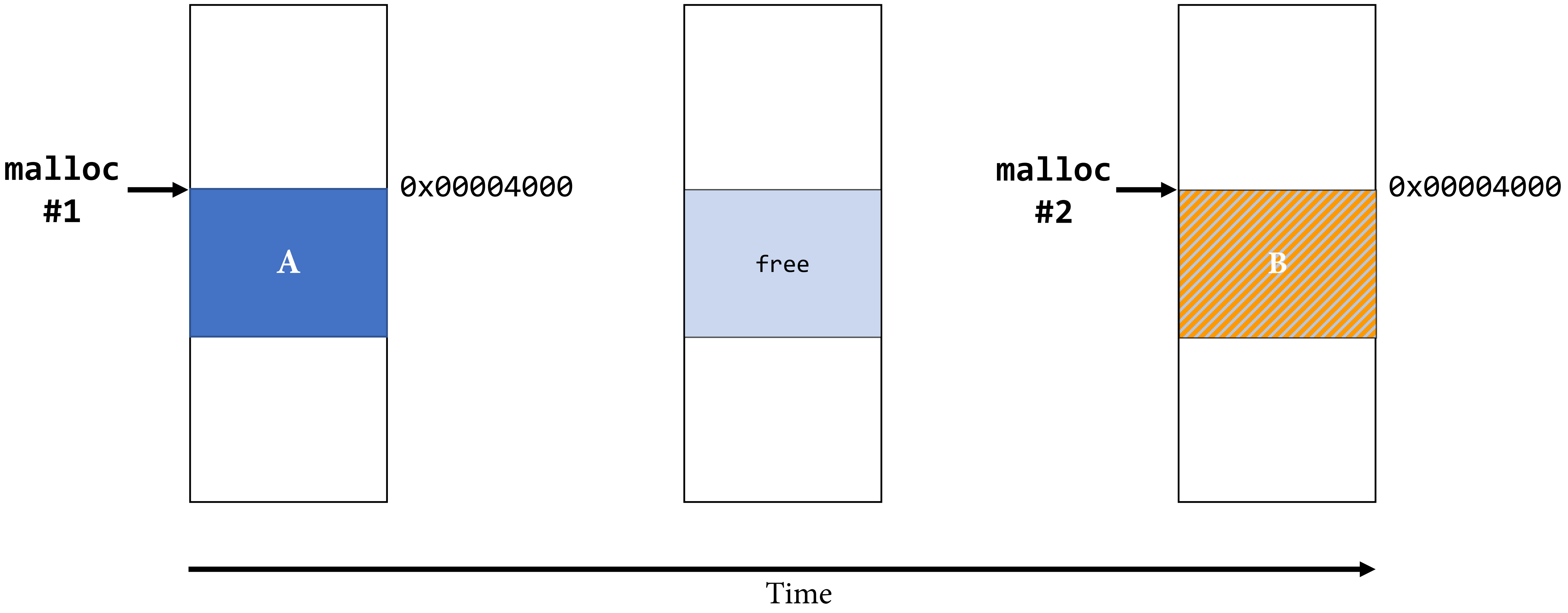
Temporal Memory Safety

The Problem



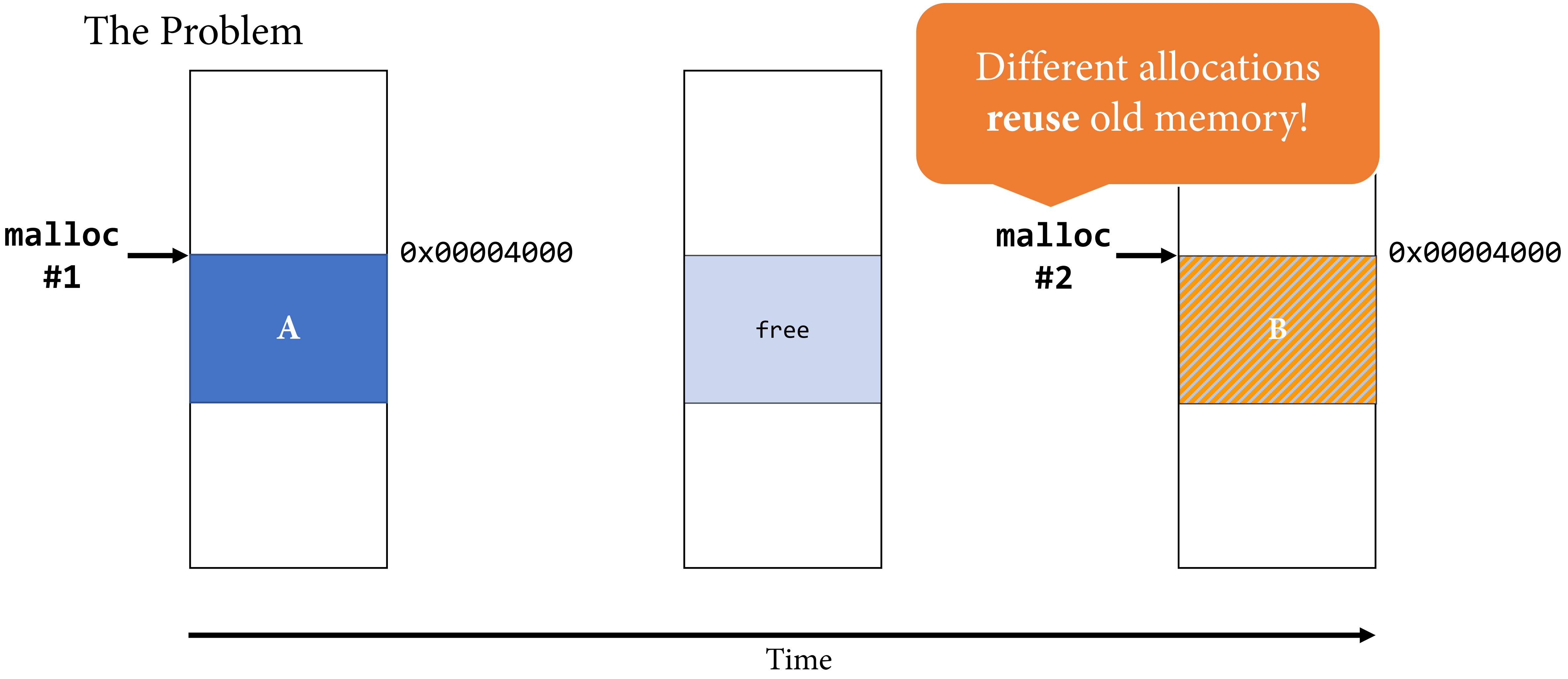
Temporal Memory Safety

The Problem

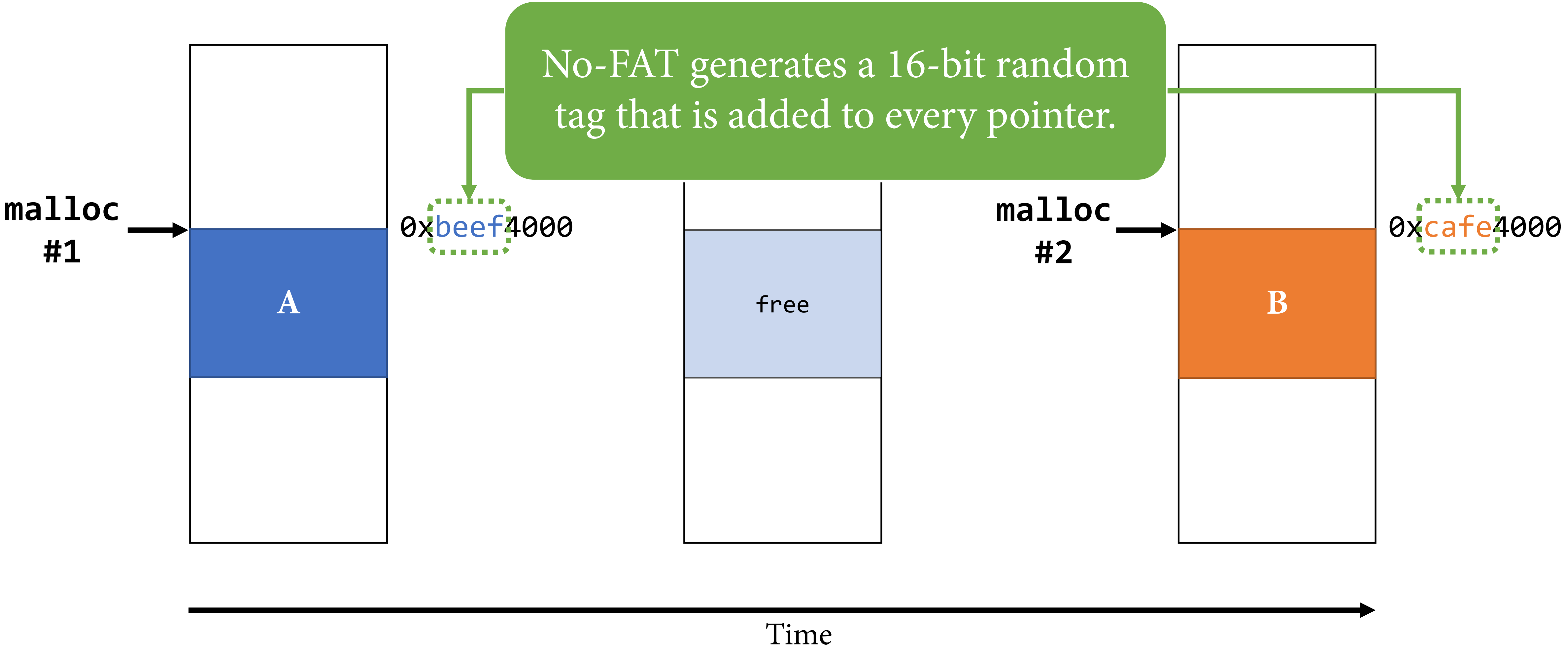


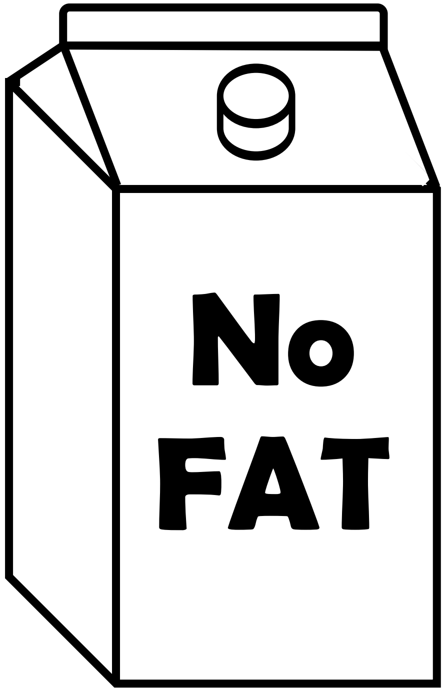
Temporal Memory Safety

The Problem



Temporal Memory Safety





ISA Extensions

ISA Extensions

1. **s_store** *Addr, Dest, BaseAddr*
2. **s_load** *Addr, Src, BaseAddr*

ISA Extensions

1. `s_store Addr, Dest, BaseAddr`
2. `s_load  Addr, Src,  BaseAddr`
3. `verifyBounds Addr, BaseAddr`

ISA Extensions

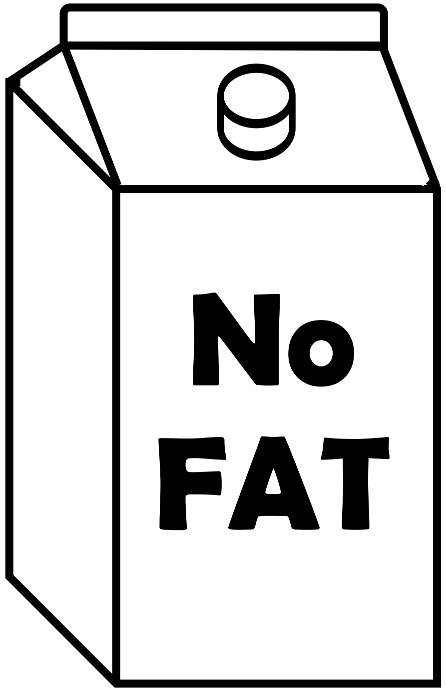
1. `s_store` *Addr, Dest, BaseAddr*
2. `s_load` *Addr, Src, BaseAddr*
3. `verifyBounds` *Addr, BaseAddr*

Exceptions are thrown in the case that the target memory address does not match `BaseAddr`.

ISA Extensions

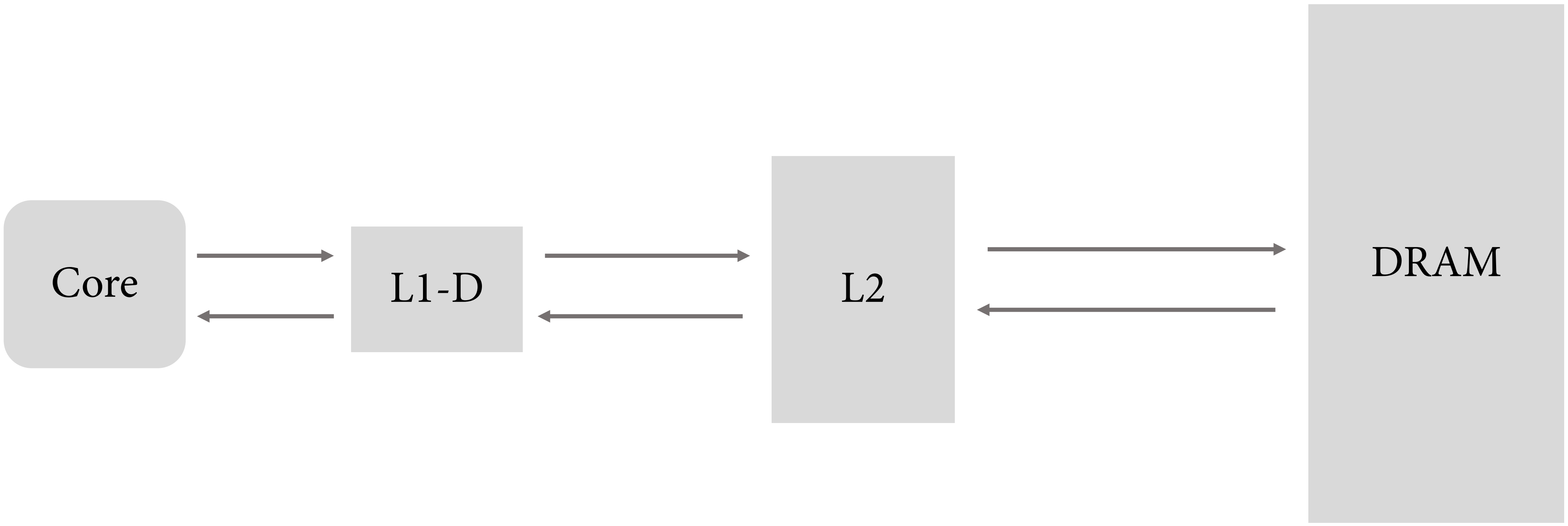
1. `s_store` *Addr, Dest, BaseAddr*
2. `s_load` *Addr, Src, BaseAddr*
3. `verifyBounds` *Addr, BaseAddr*
4. `compBase` *Addr, Dest*

Exceptions are thrown in the case that the target memory address does not match `BaseAddr`.

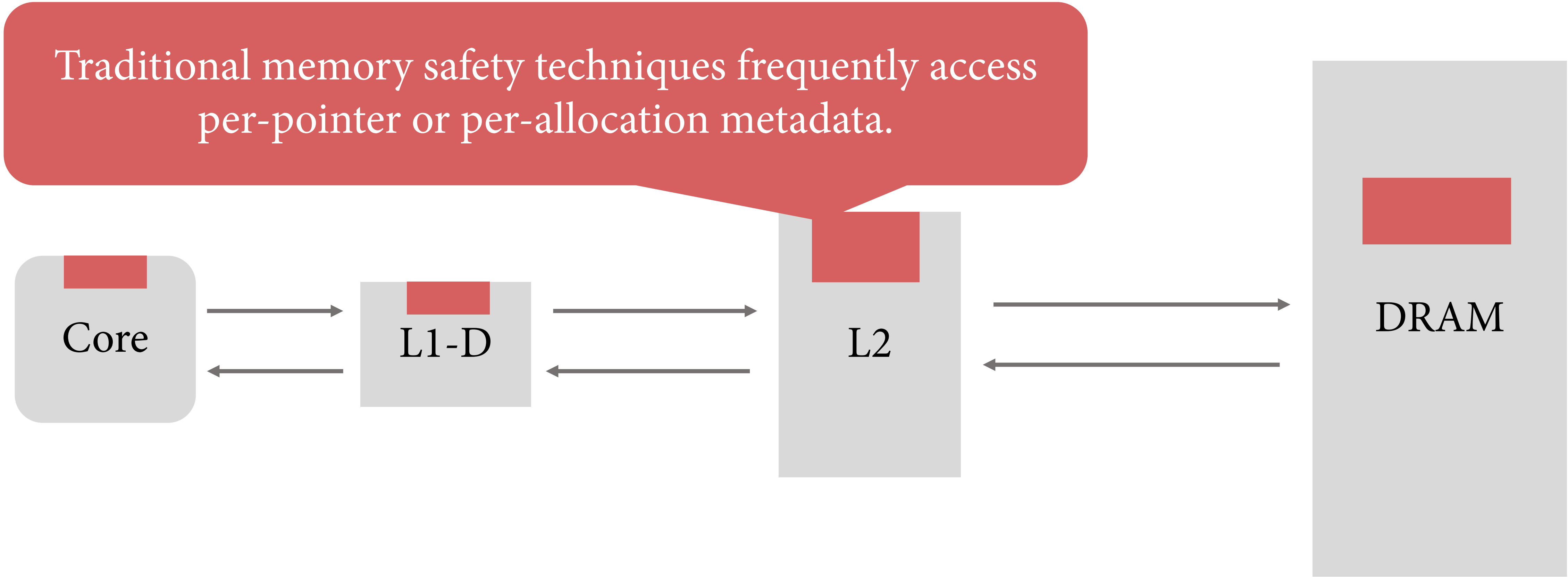


Microarchitectural Overview

Microarchitectural Overview

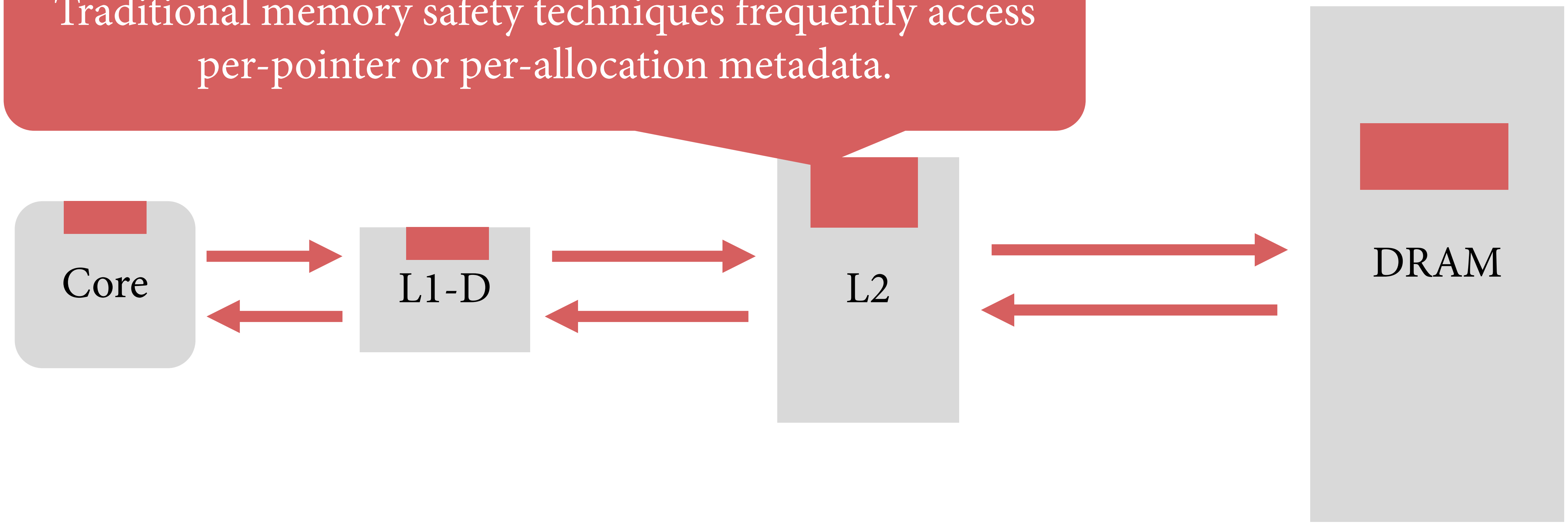


Microarchitectural Overview

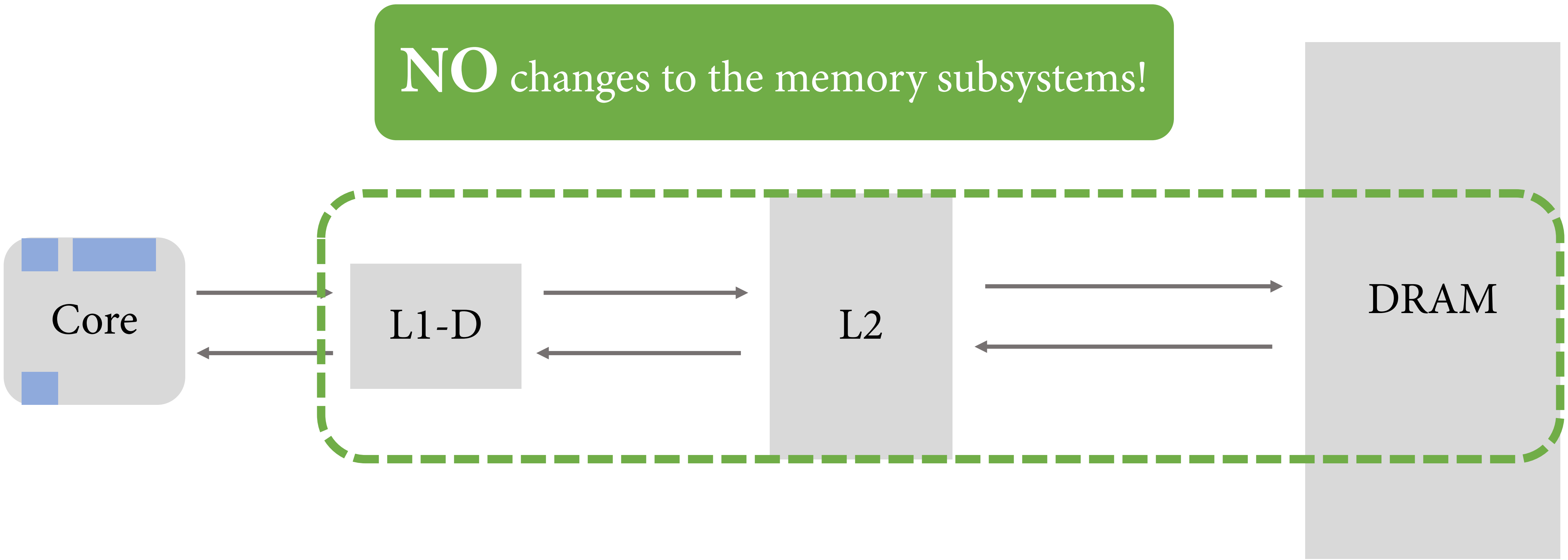


Microarchitectural Overview

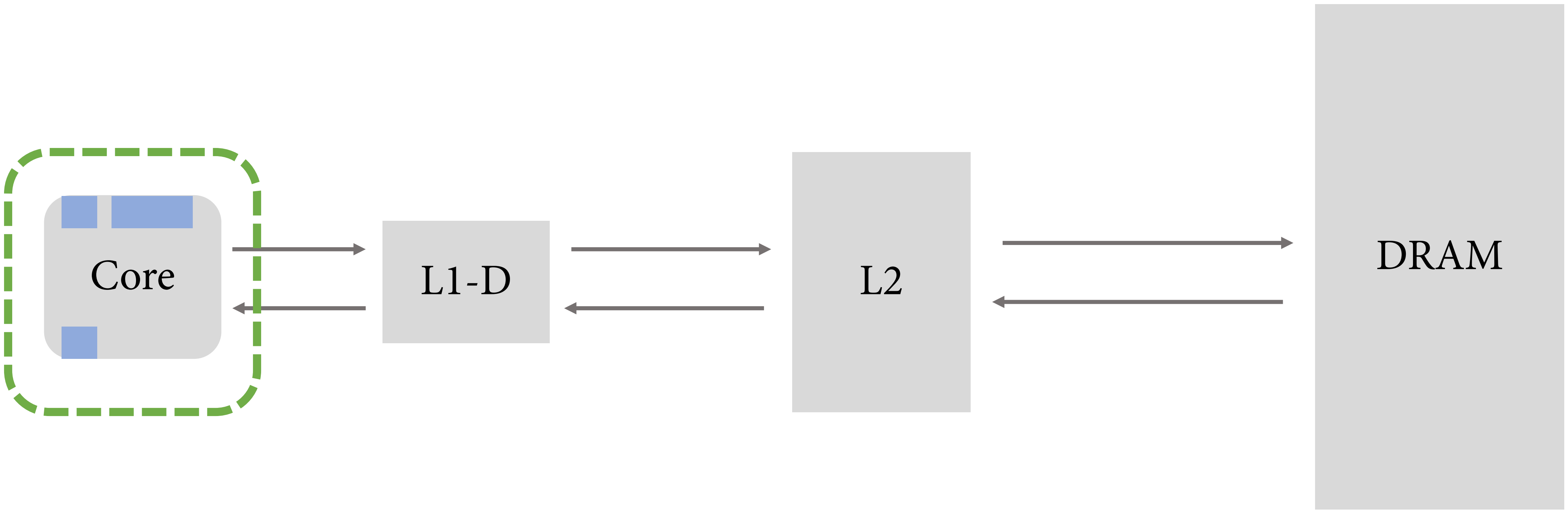
Traditional memory safety techniques frequently access per-pointer or per-allocation metadata.



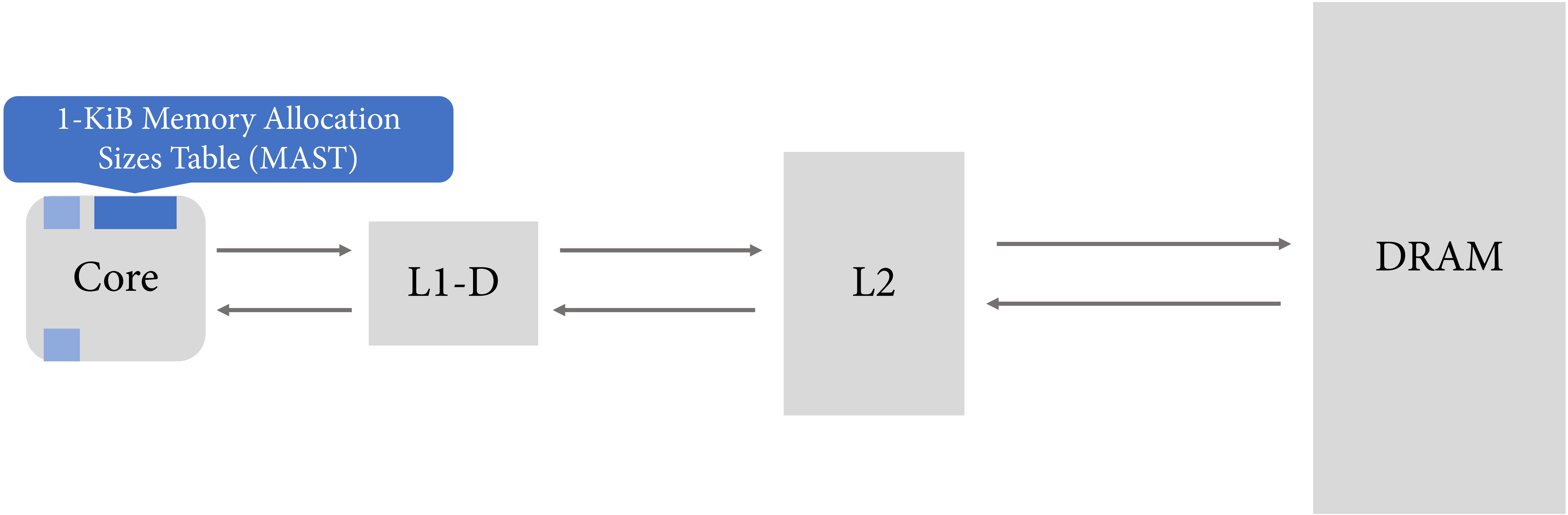
Microarchitectural Overview



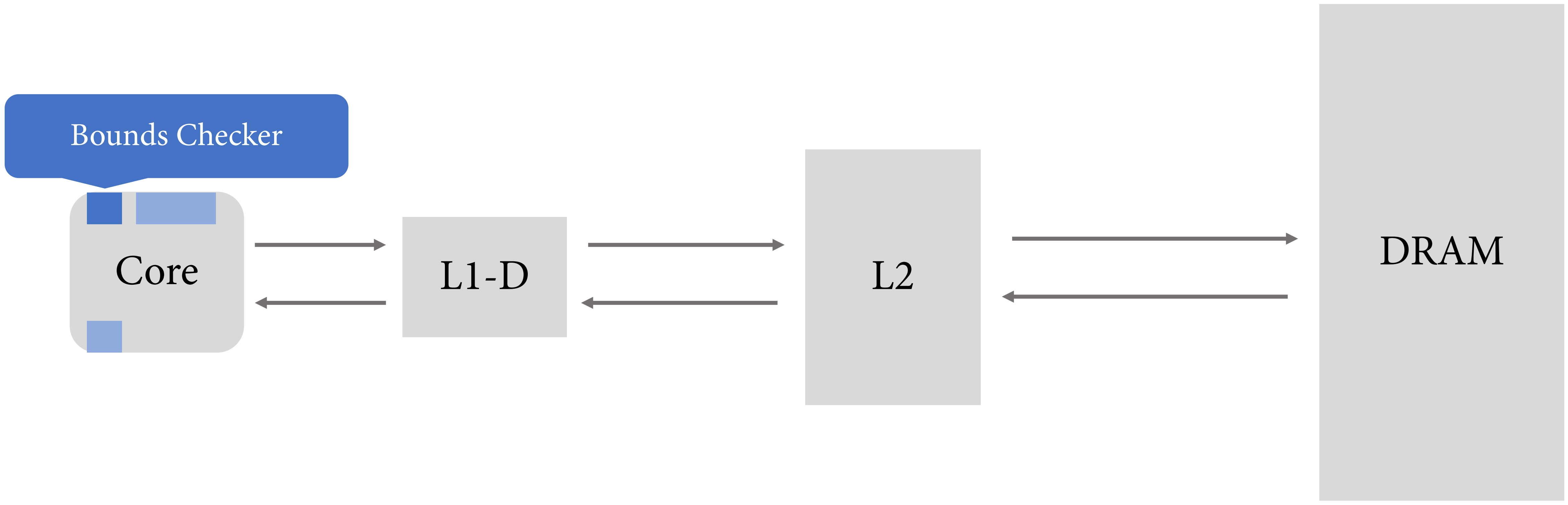
Microarchitectural Overview



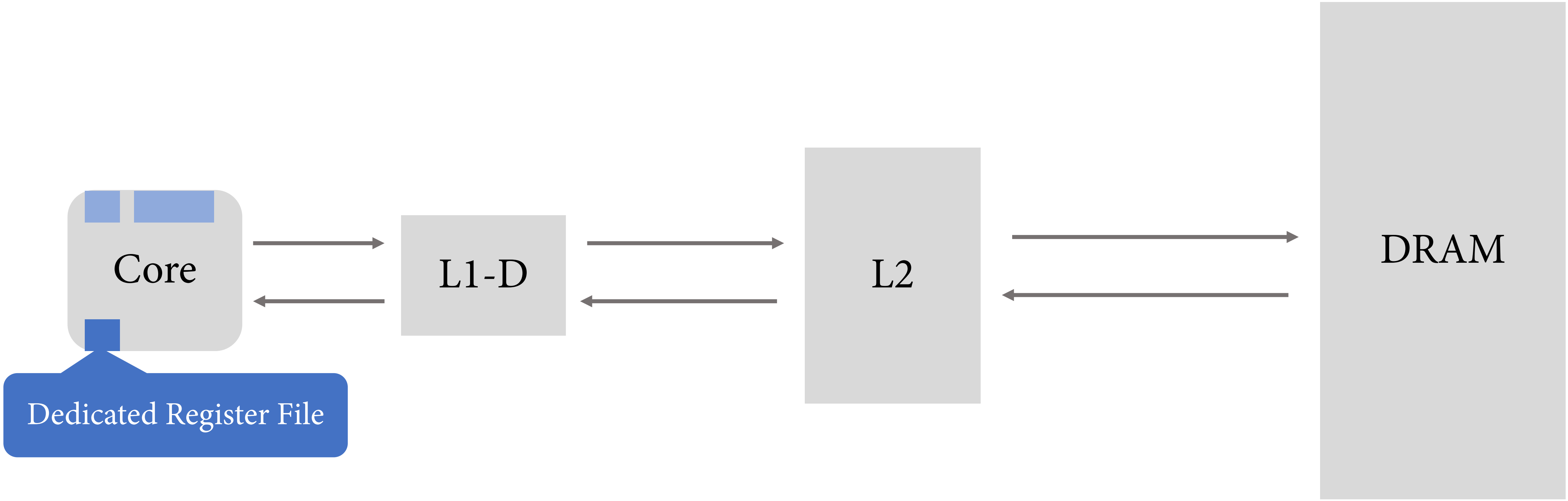
Microarchitectural Overview

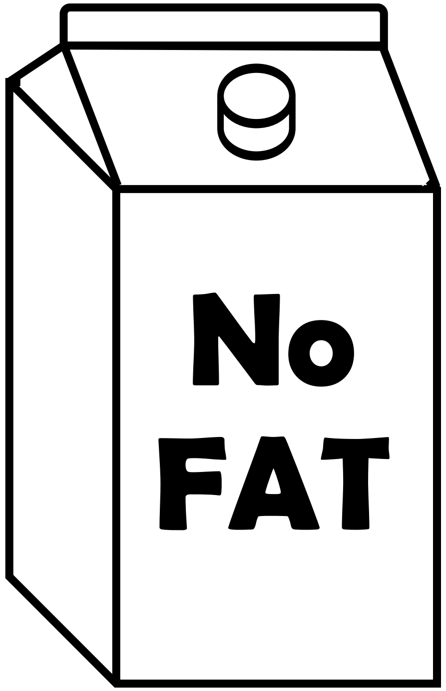


Microarchitectural Overview



Microarchitectural Overview





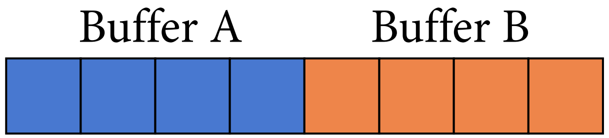
Resilience to Common Exploits

Resilience to Common Exploits

1

Buffer Over-/Under-flows

Cannot corrupt memory.

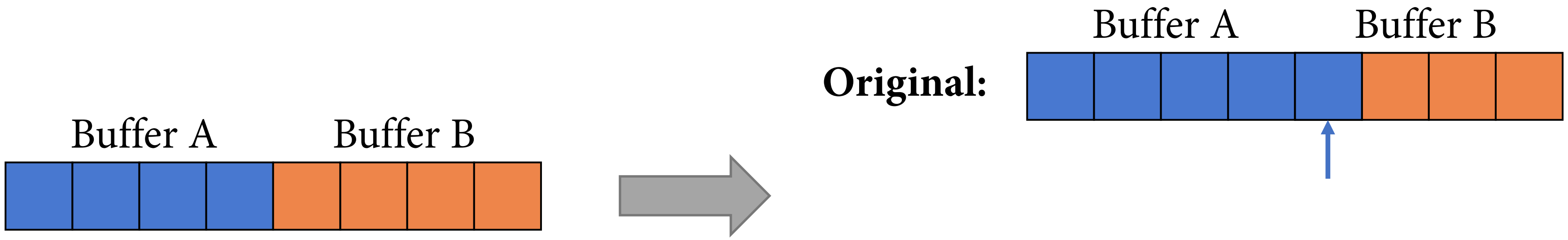


Resilience to Common Exploits

1

Buffer Over-/Under-flows

Cannot corrupt memory.

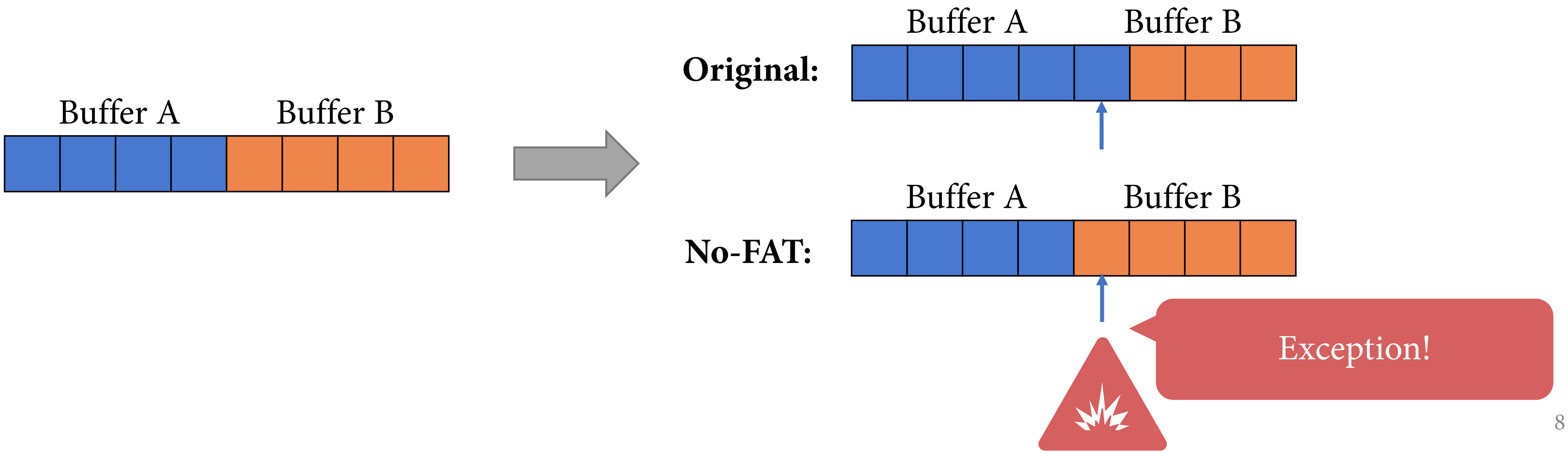


Resilience to Common Exploits

1

Buffer Over-/Under-flows

Cannot corrupt memory.



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.

```
// 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.

```
// 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.

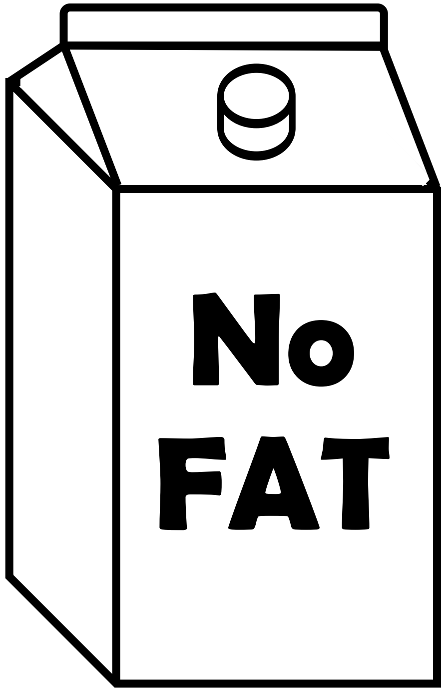
```
// mispredicted branch
if (i < sizeof(a)) {

  secret = a[i];

  // secret is leaked
  val = b[64 * secret];
}
```

Detected!

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



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 loads/stores do not change the binary size.

Performance Overheads

Hardware Modifications

Our measurements show minimal latency/area/power overheads.

Software Modifications

- Our special loads/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 loads/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.

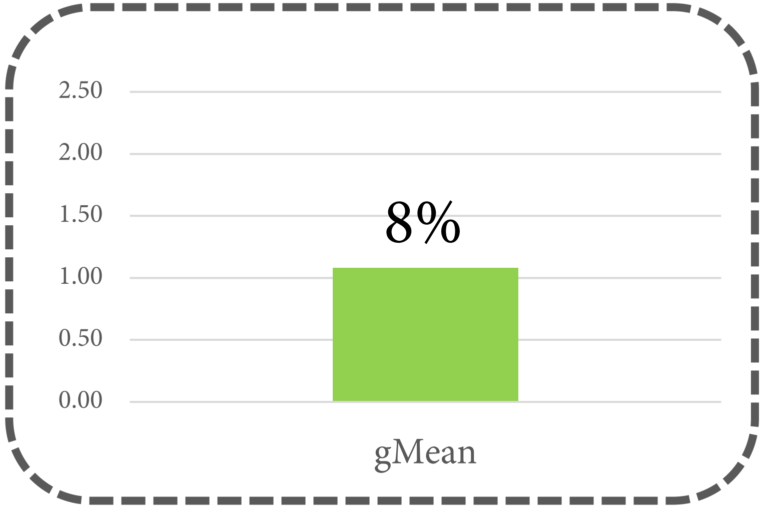
Performance Results (x86_64)

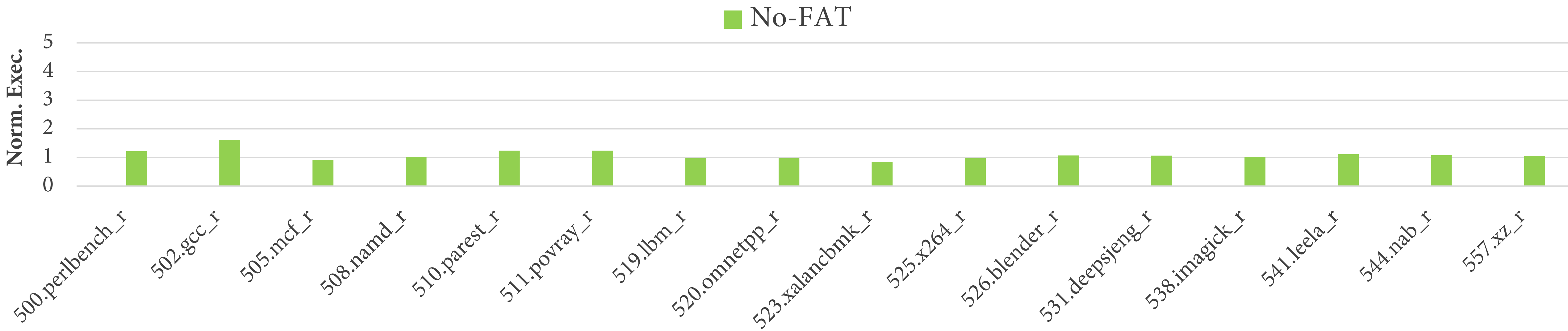
Experimental Setup

We emulate No-FAT on x86_64 by modifying LLVM to emit new instructions.

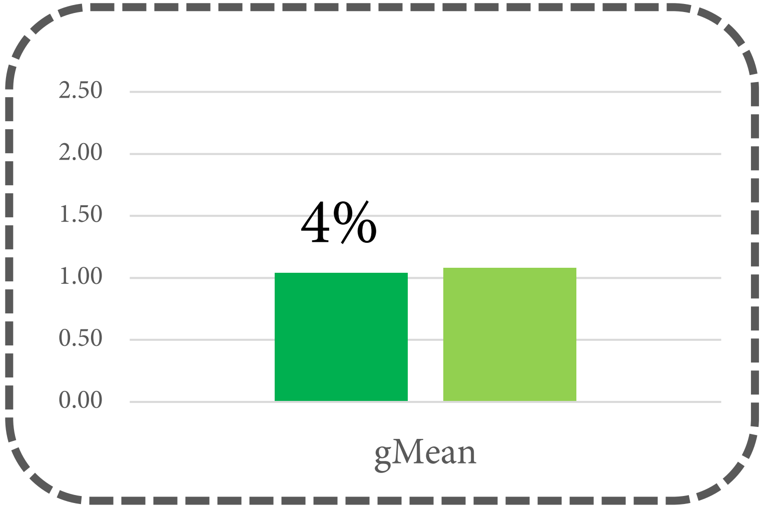
- `CompBase` is emulated using two multiplications followed by a `store`.
- `VerifyBounds` is emulated using dummy `stores`.

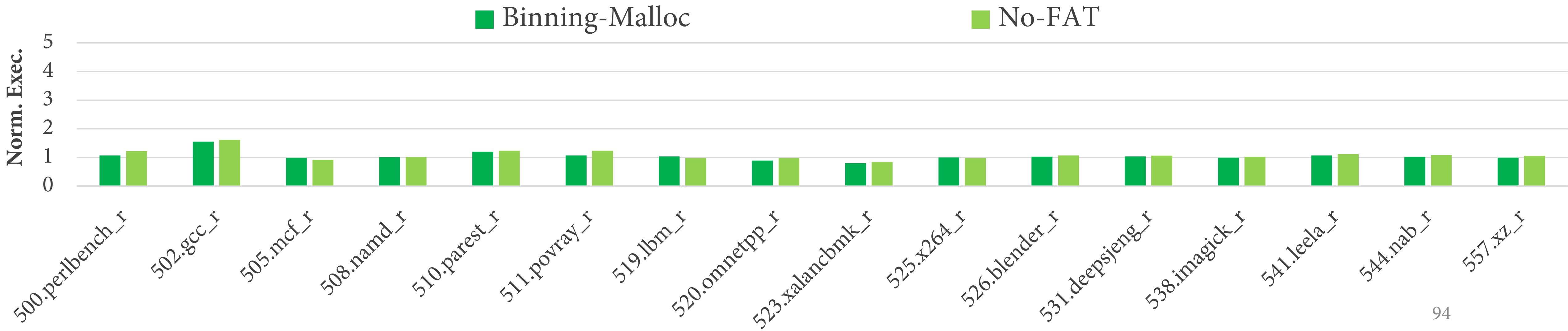
Performance Results (x86_64)



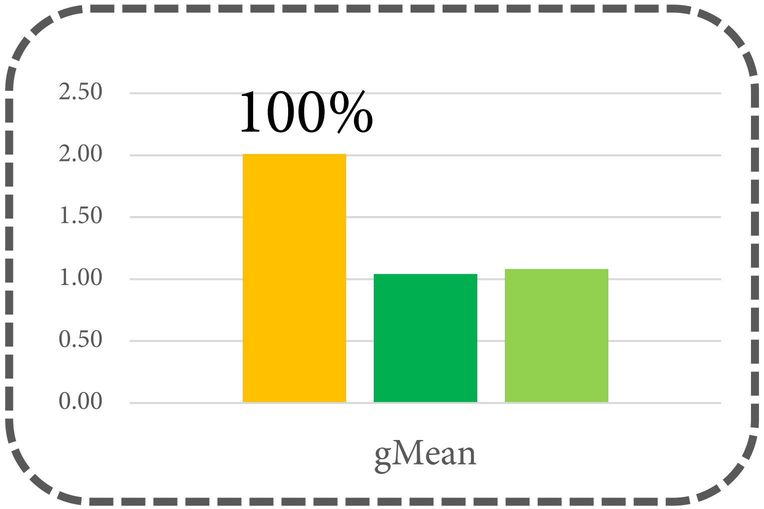


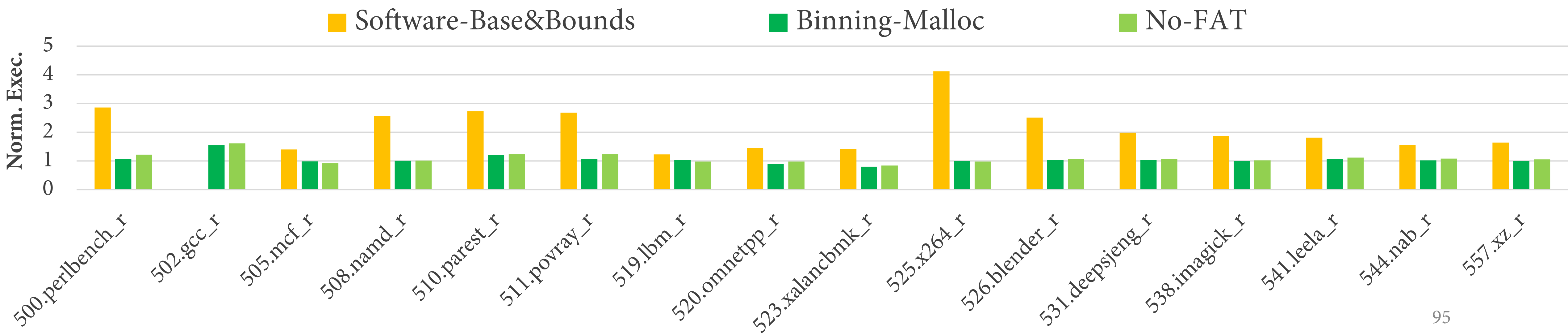
Performance Results (x86_64)



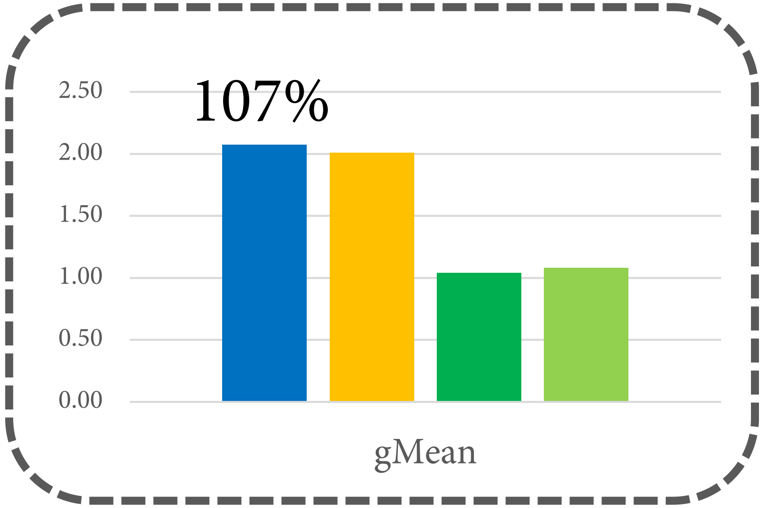


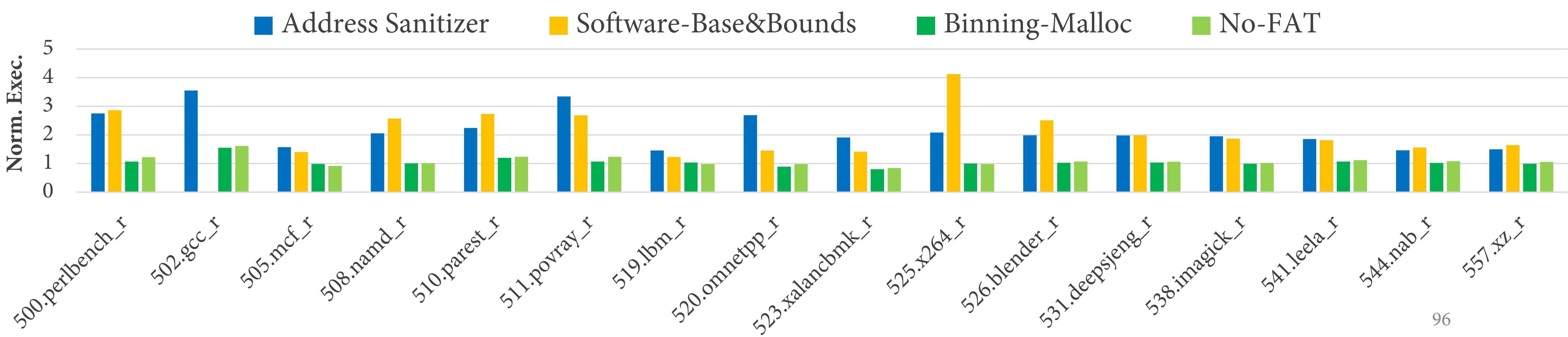
Performance Results (x86_64)



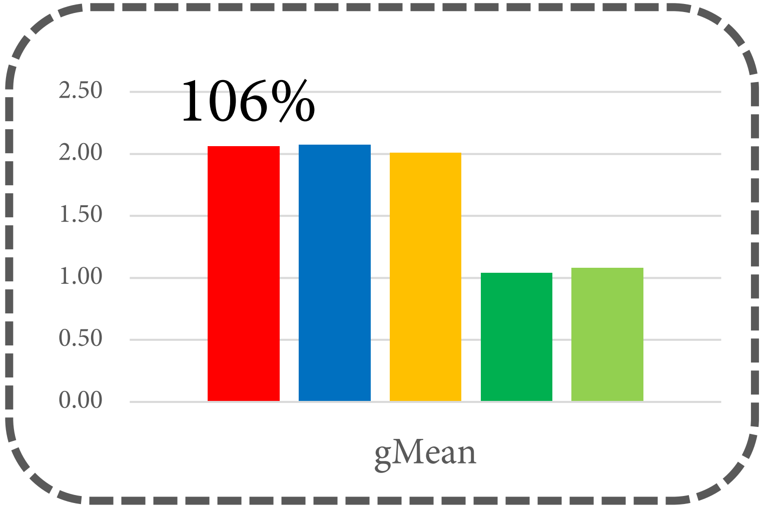


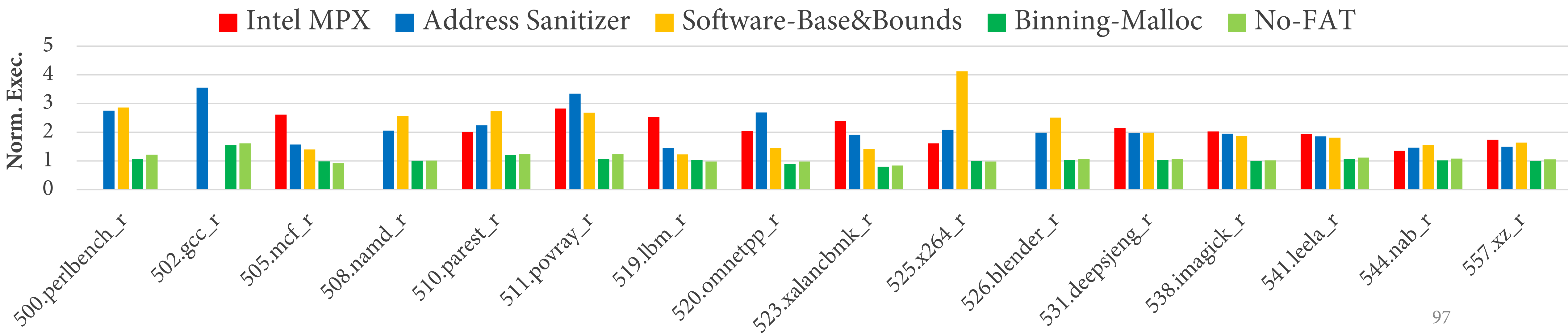
Performance Results (x86_64)





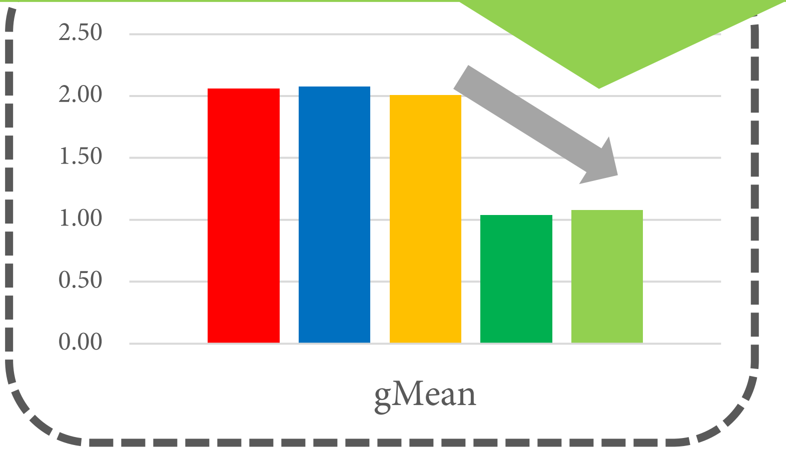
Performance Results (x86_64)

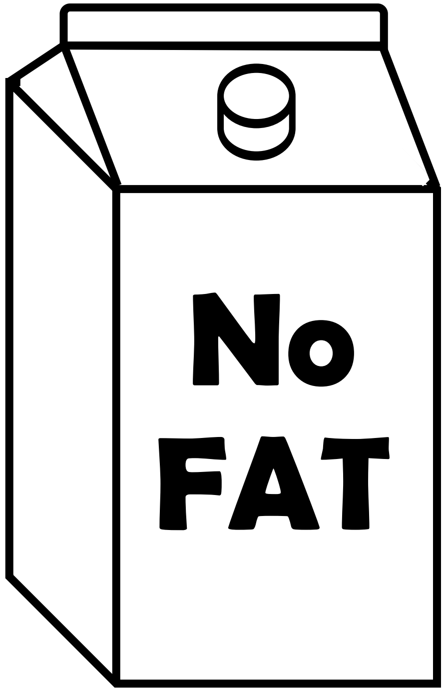




Performance Results (x86_64)

We reduce the average runtime overheads of full memory safety **from 100% to 8%!**





Related Work

Related Work

	Metadata	Concerns
Memory Tagging	N-bits per pointer & allocation	Spatial & temporal safety limited by tag width

Related Work

	Metadata	Concerns
Memory Tagging	N-bits per pointer & allocation	Spatial & temporal safety limited by tag width
Tripwires	N-bits per allocation	Susceptible to non-adjacent overflows

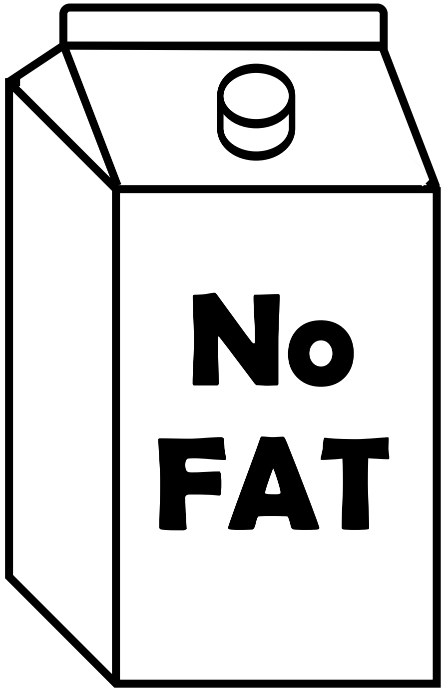
Related Work

	Metadata	Concerns
Memory Tagging	N-bits per pointer & allocation	Spatial & temporal safety limited by tag width
Tripwires	N-bits per allocation	Susceptible to non-adjacent overflows
Explicit Base & Bounds	N-bits per pointer or allocation	Breaks compatibility with the rest of the system (eg. unprotected libraries).

Related Work

	Metadata	Concerns
Memory Tagging	N-bits per pointer & allocation	Spatial & temporal safety limited by tag width
Tripwires	N-bits per allocation	Susceptible to non-adjacent overflows
Explicit Base & Bounds	N-bits per pointer or allocation	Breaks compatibility with the rest of the system (eg. unprotected libraries).
No-FAT	Fixed (1K) bits per process	Requires binning allocator

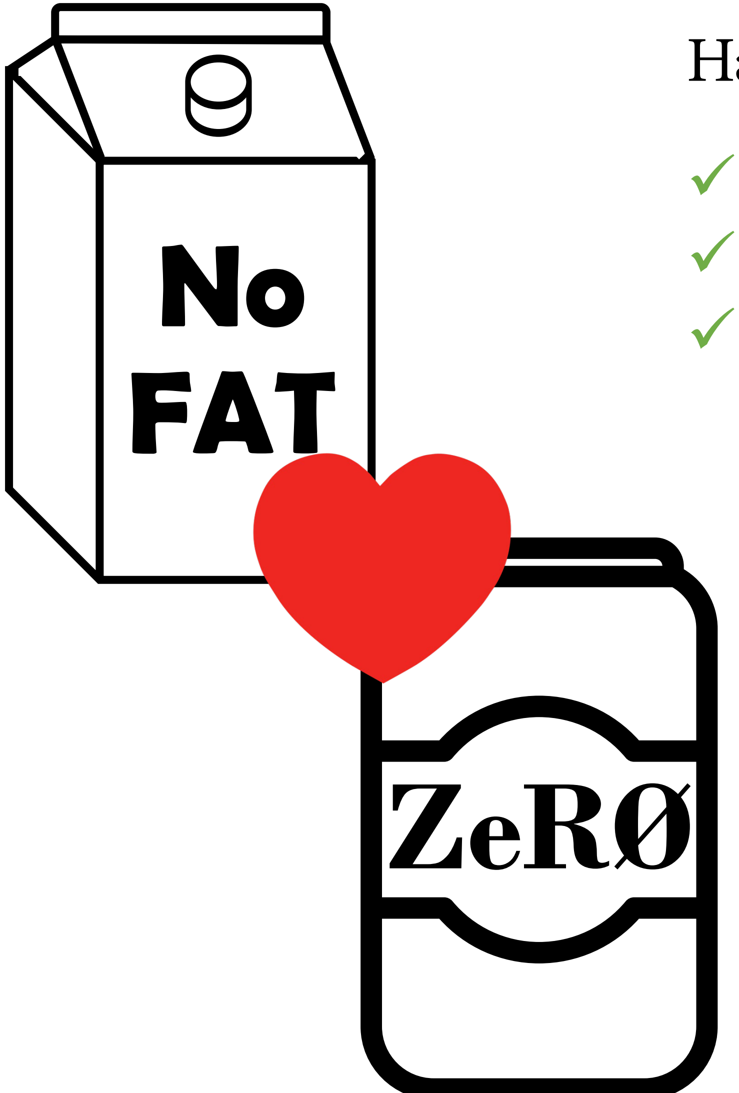
Takeaways



Having no metadata

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

Takeaways



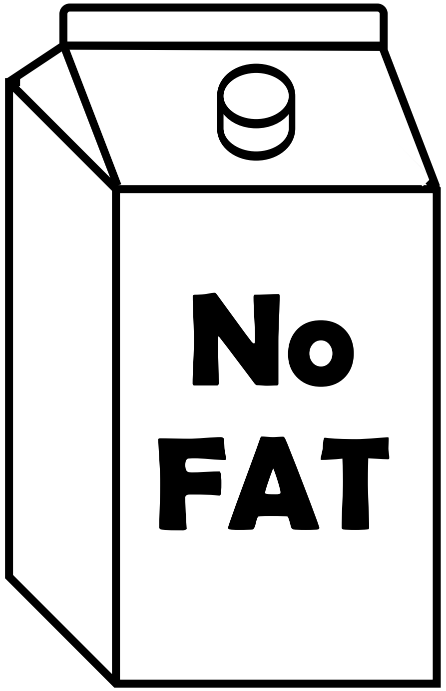
Having no metadata

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

For applications where an 8% overhead is too much, checkout ZeRØ.

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!