

Practical Memory Safety

Mohamed Tarek Ibn Ziad

<https://research.nvidia.com/person/mohamed-tarek-ibn-ziad>

Guest Lecture at CU Boulder - October 31st, 2022

About Me

- Research Scientist @ NVIDIA
 - Member of the Architecture Research Group (ARG).

About Me

- Research Scientist @ NVIDIA
 - Member of the Architecture Research Group (ARG).
- PhD from CS Department @ Columbia University
 - Member of the Computer Architecture and Security Technologies Lab (CASTL)
 - Hardware-Software Co-design for Practical Memory Safety
 - Supervisor: Simha Sethumadhavan

About Me

- Research Scientist @ NVIDIA
 - Member of the Architecture Research Group (ARG).
- PhD from CS Department @ Columbia University
 - Member of the Computer Architecture and Security Technologies Lab (CASTL)
 - Hardware-Software Co-design for Practical Memory Safety
 - Supervisor: Simha Sethumadhavan
- Before joining Columbia
 - B.Sc and M.Sc in Computer Engineering from Ain Shams University, Egypt.
 - Software development engineer at Mentor Graphics (now acquired by Siemens)

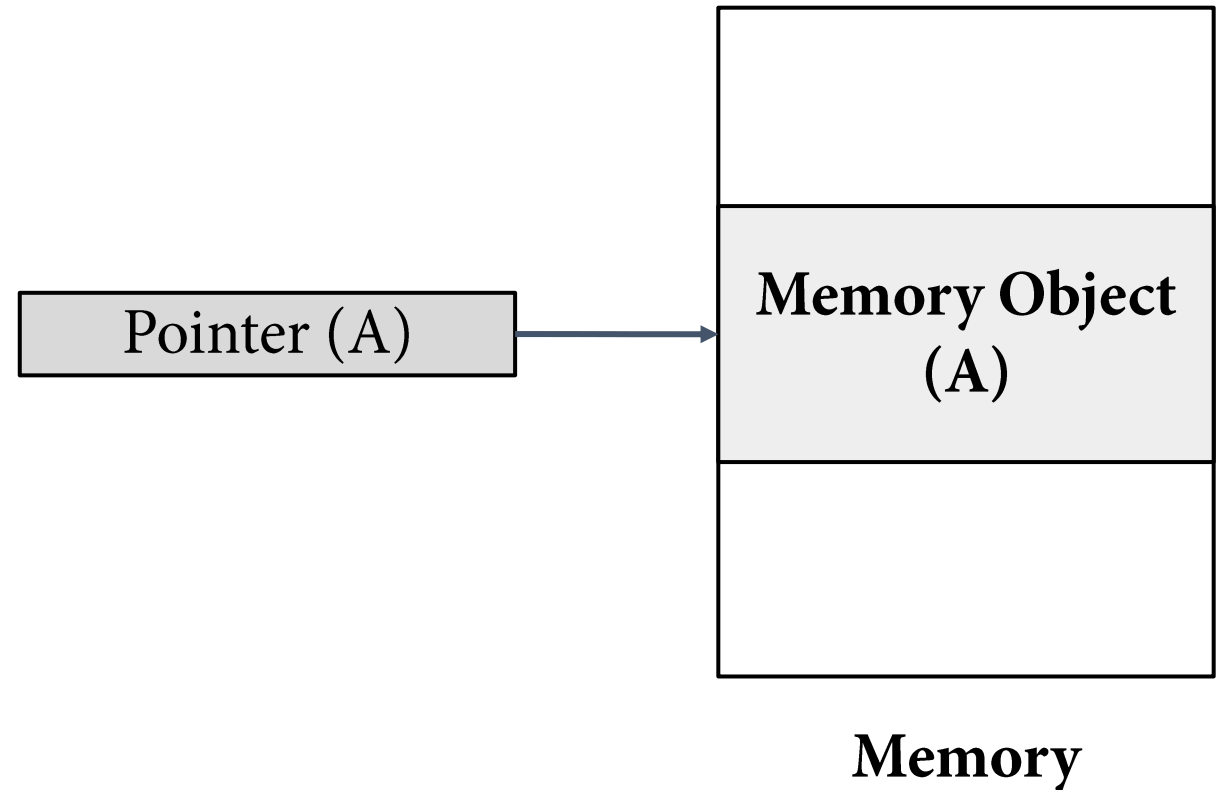
About Me

- Research Scientist @ NVIDIA
 - Member of the Architecture Research Group (ARG).
- PhD from CS Department @ Columbia University
 - Member of the Computer Architecture and Security Technologies Lab (CASTL)
 - Hardware-Software Co-design for Practical Memory Safety
 - Supervisor: Simha Sethumadhavan
- Before joining Columbia
 - B.Sc and M.Sc in Computer Engineering from Ain Shams University, Egypt.
 - Software development engineer at Mentor Graphics (now acquired by Siemens)

*Today's
Talk!*

What is Memory Safety?

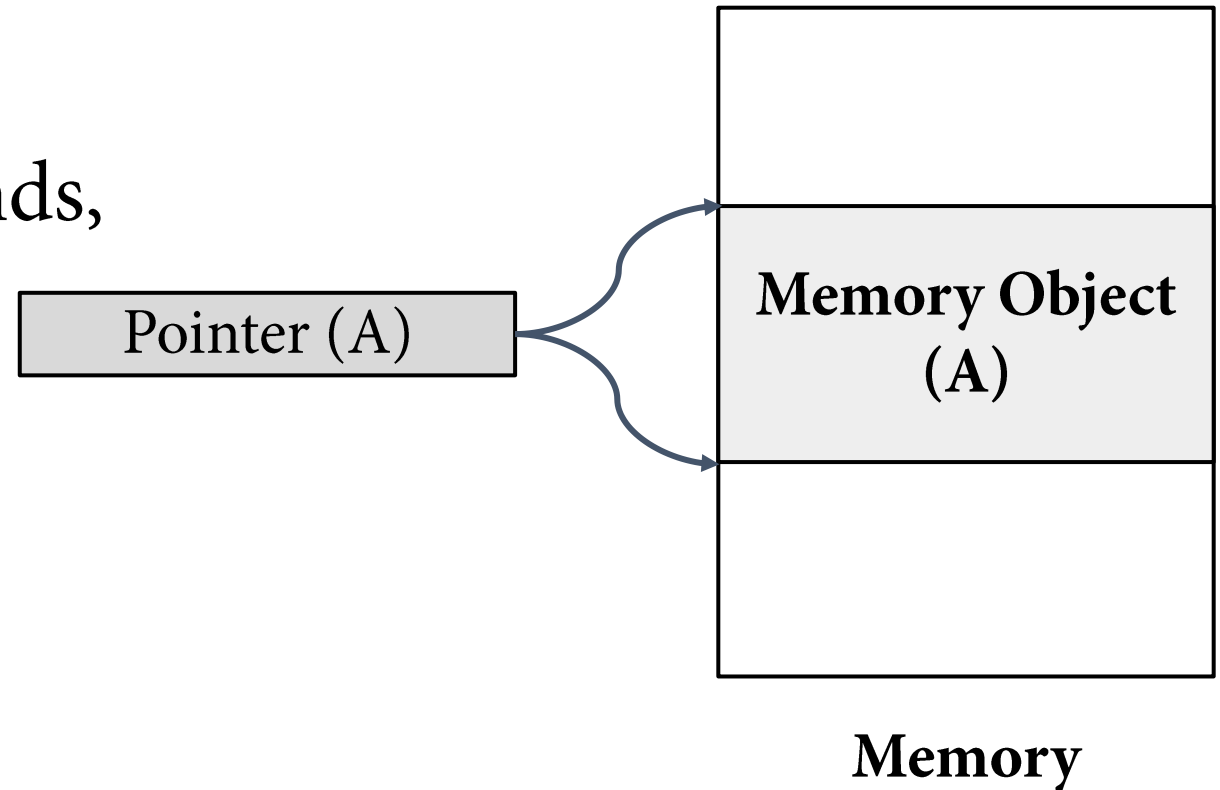
A program property that guarantees **memory objects** can only be accessed:



What is Memory Safety?

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

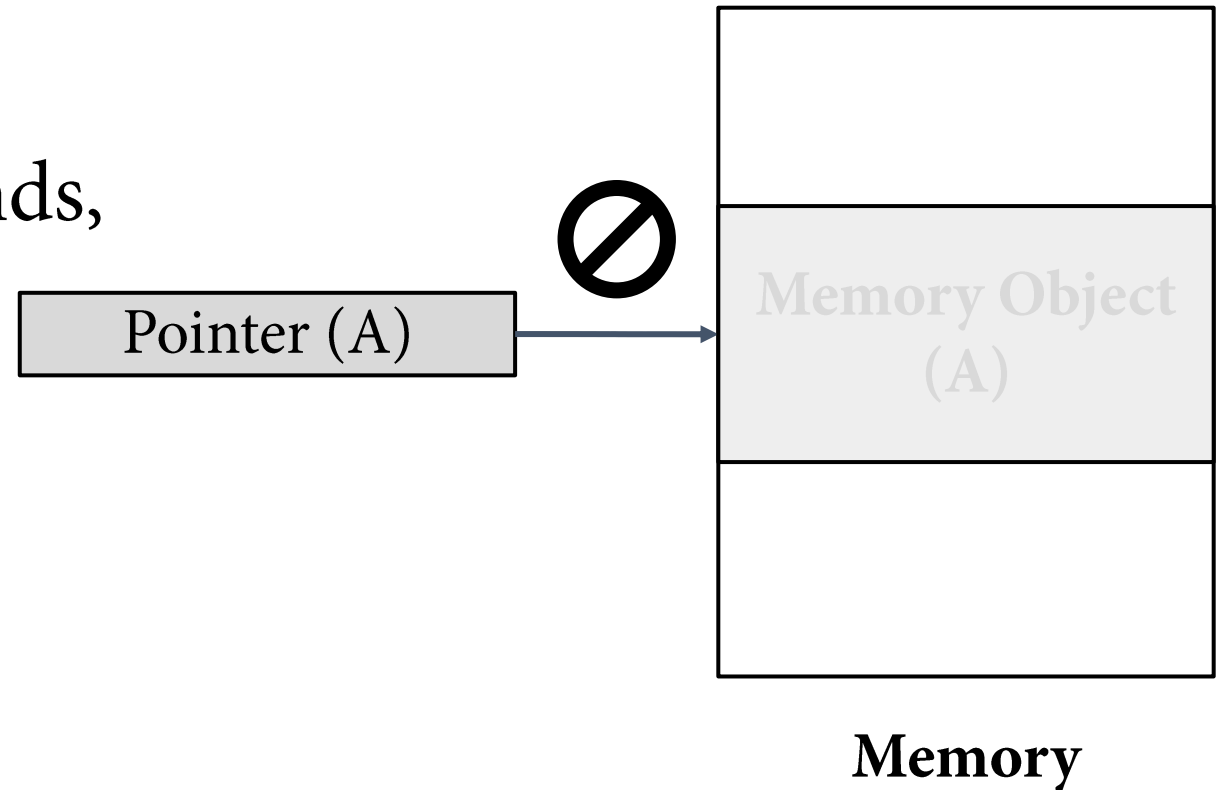
- Between their intended bounds,



What is Memory Safety?

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

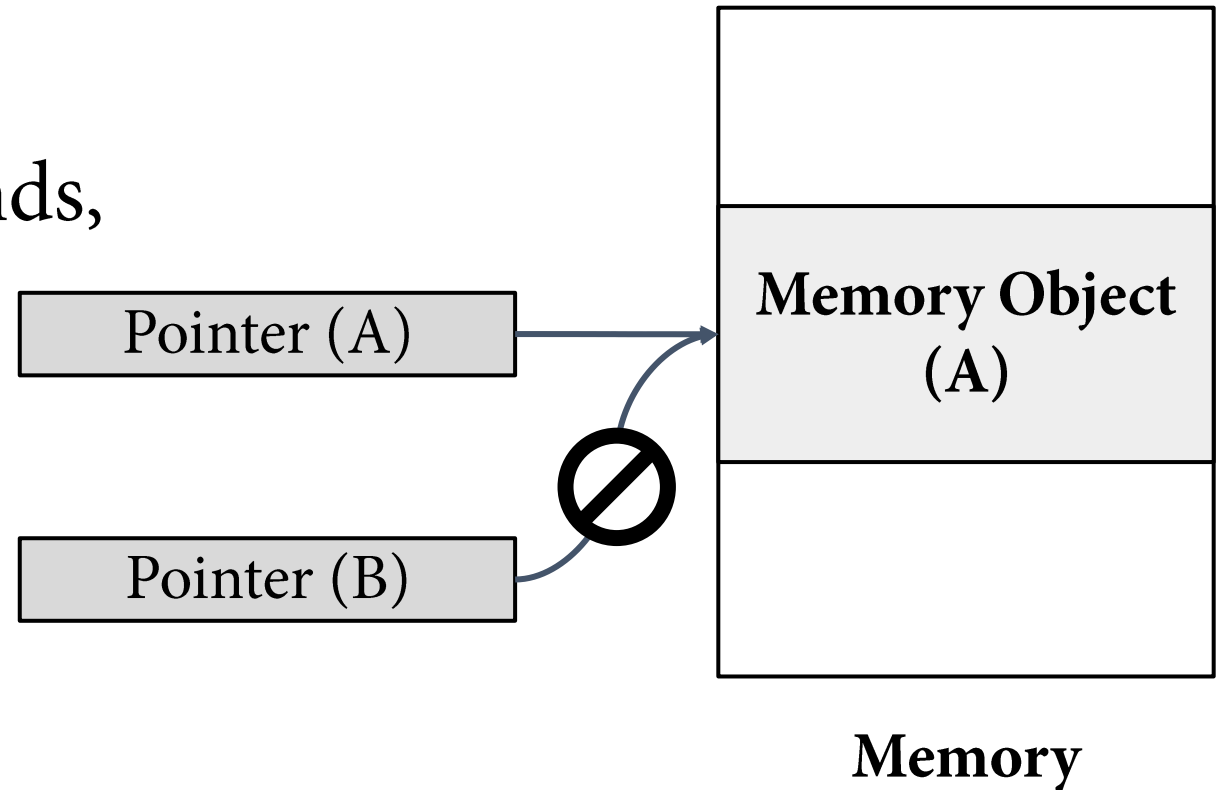
- Between their intended bounds,
- During their lifetime, and



What is Memory Safety?

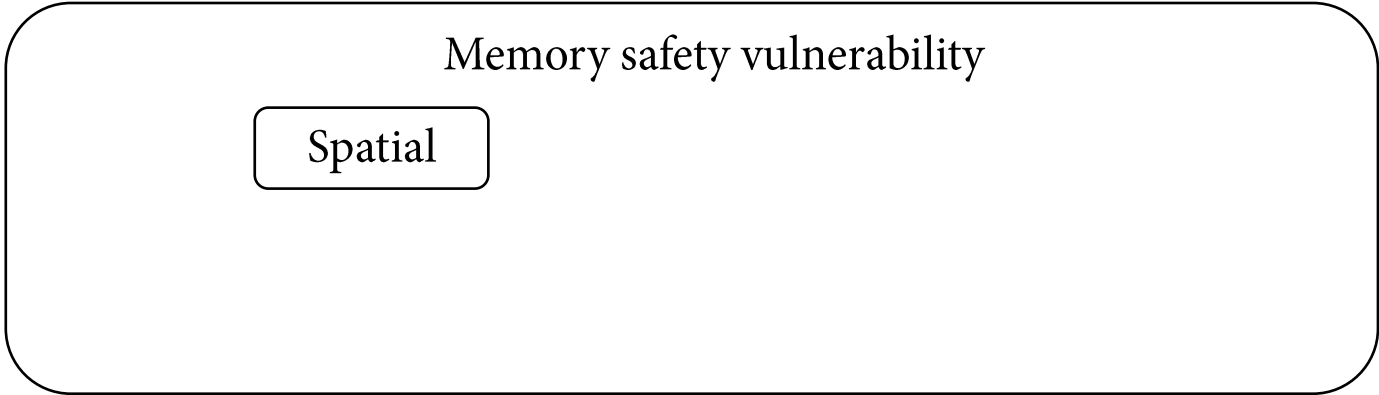
A program property that guarantees **memory objects** can only be accessed:

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



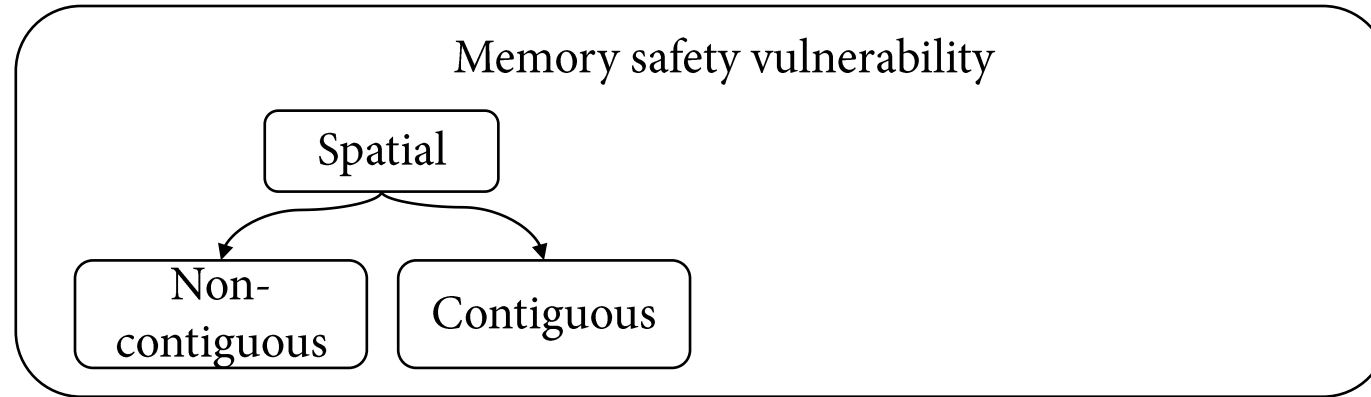
Memory Attacks Taxonomy

Root cause



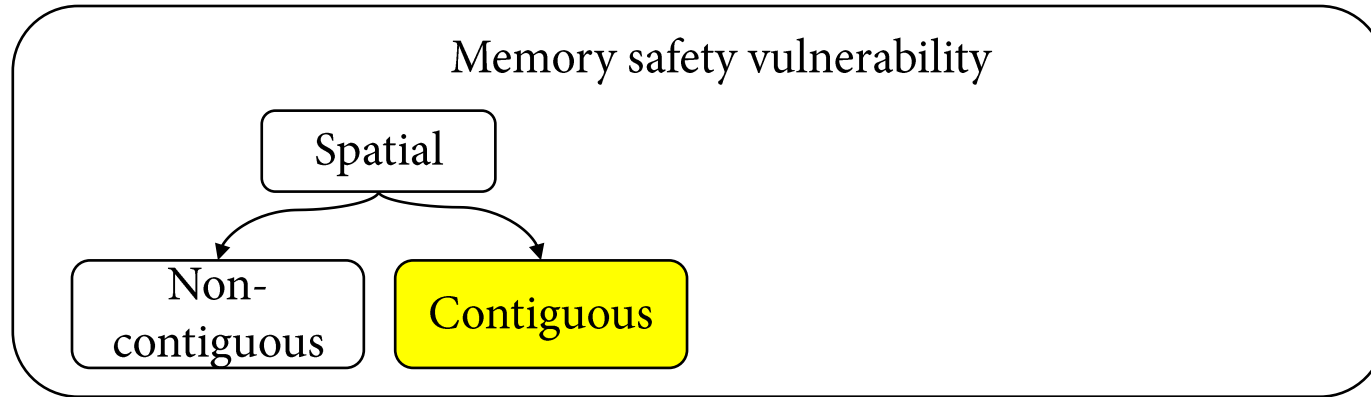
Memory Attacks Taxonomy

Root cause



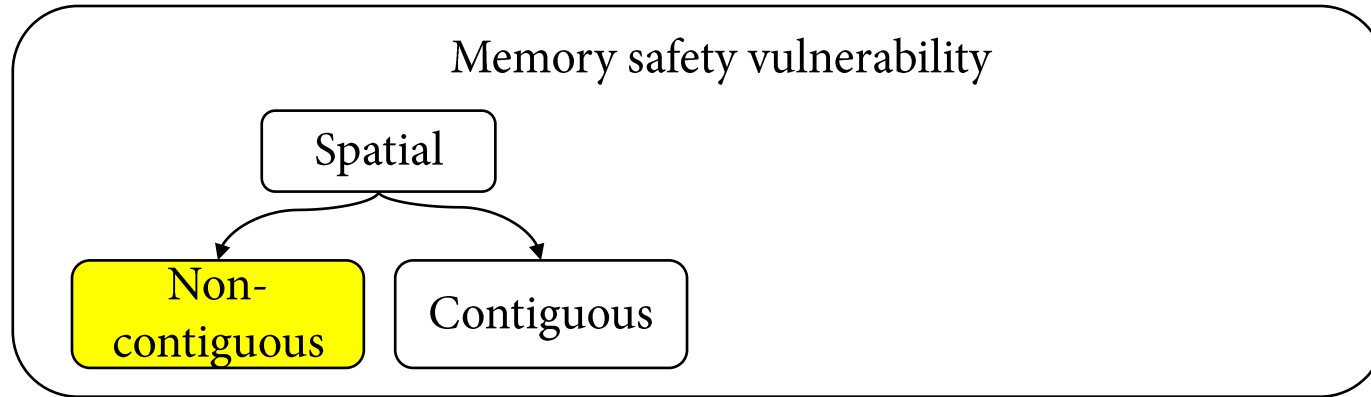
Memory Attacks Taxonomy

Root cause



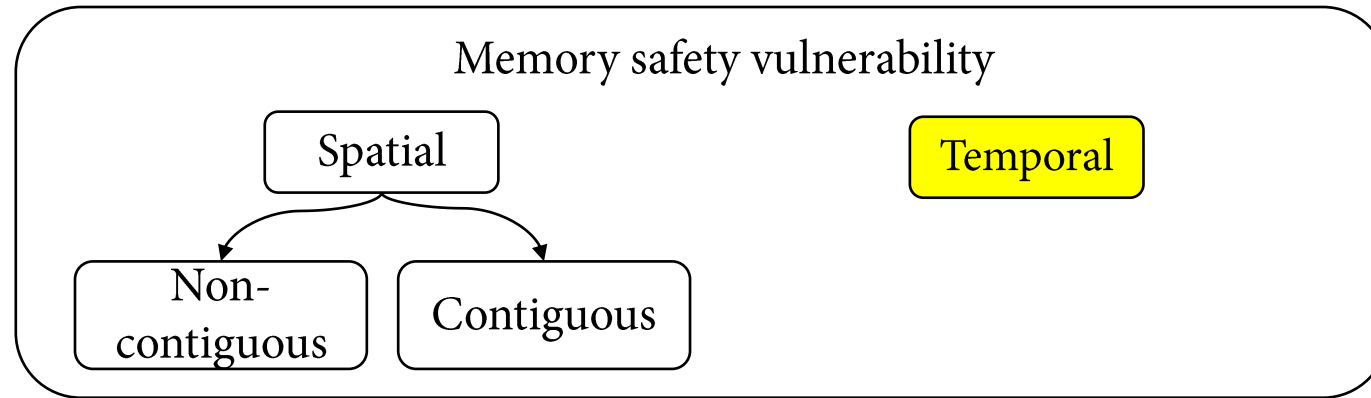
Memory Attacks Taxonomy

Root cause



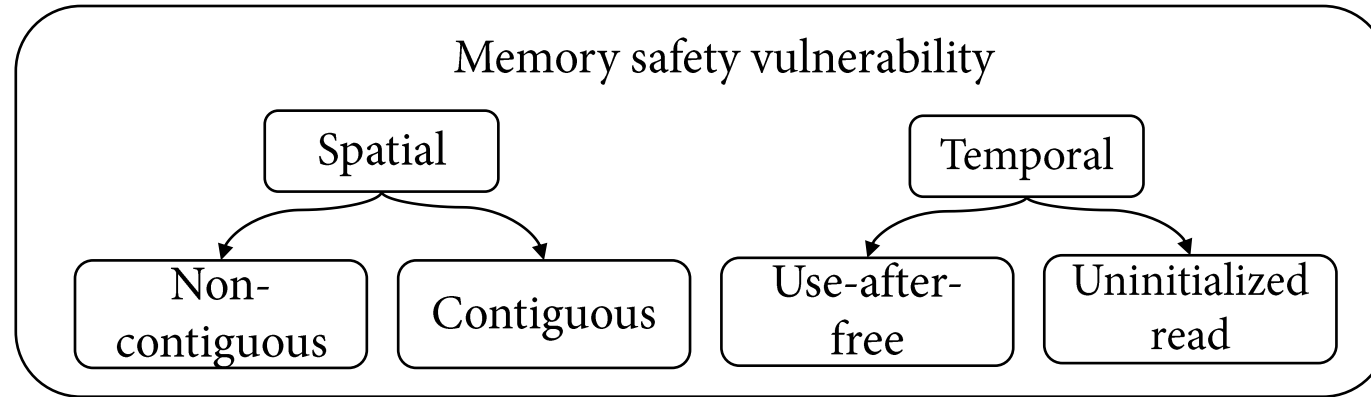
Memory Attacks Taxonomy

Root cause



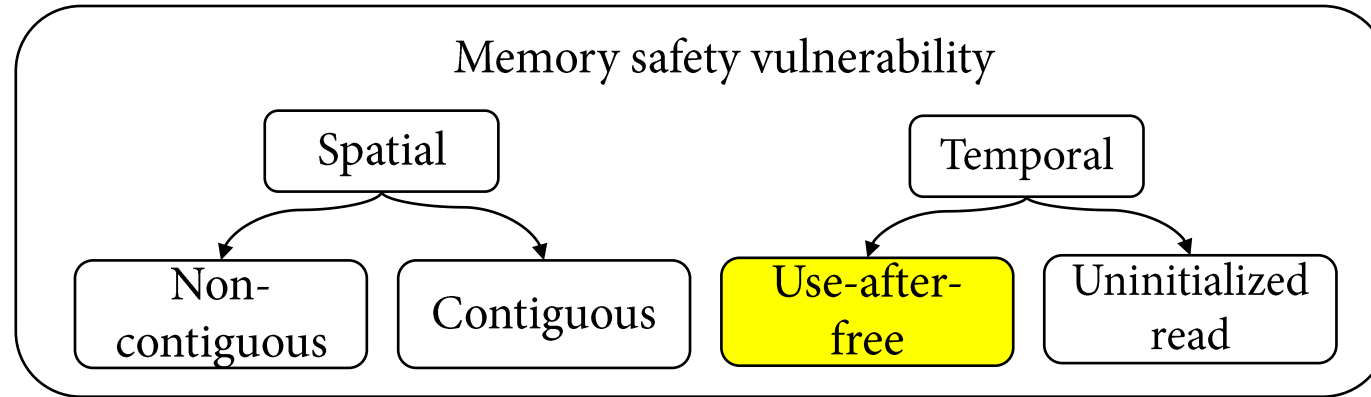
Memory Attacks Taxonomy

Root cause



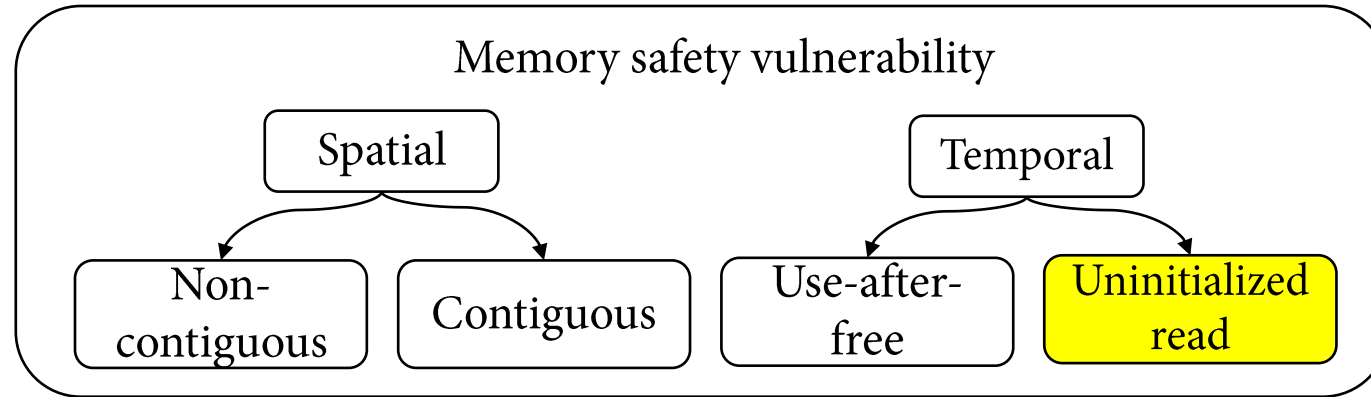
Memory Attacks Taxonomy

Root cause

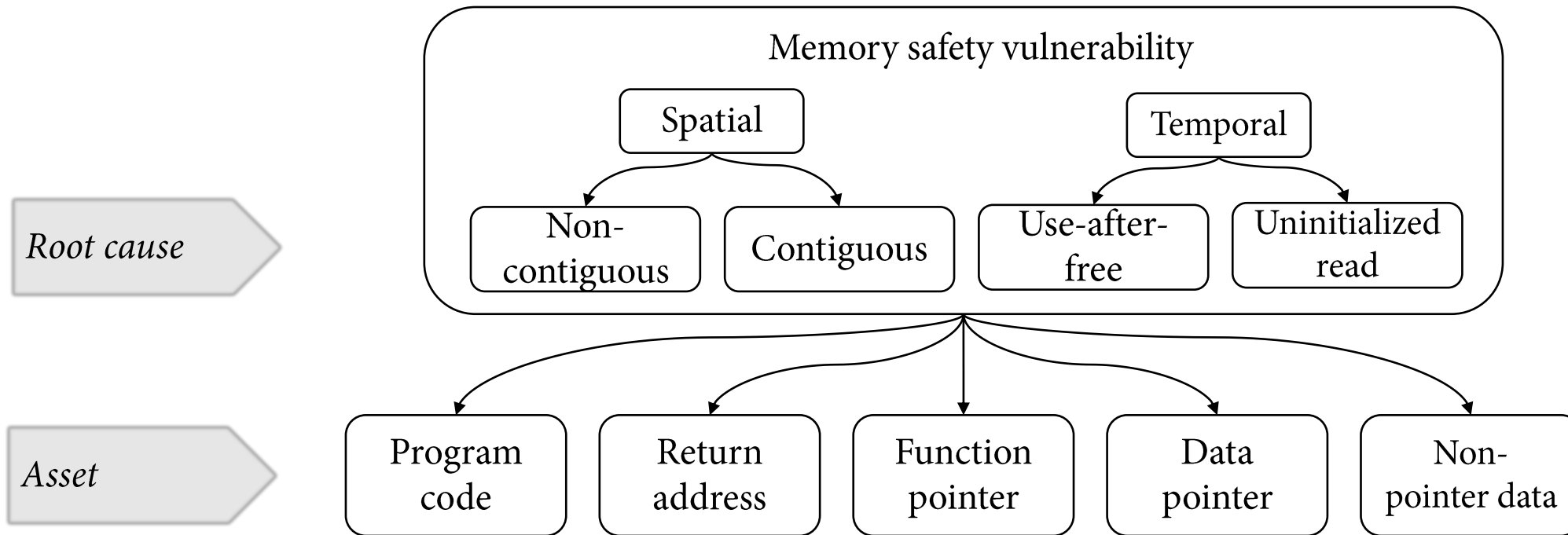


Memory Attacks Taxonomy

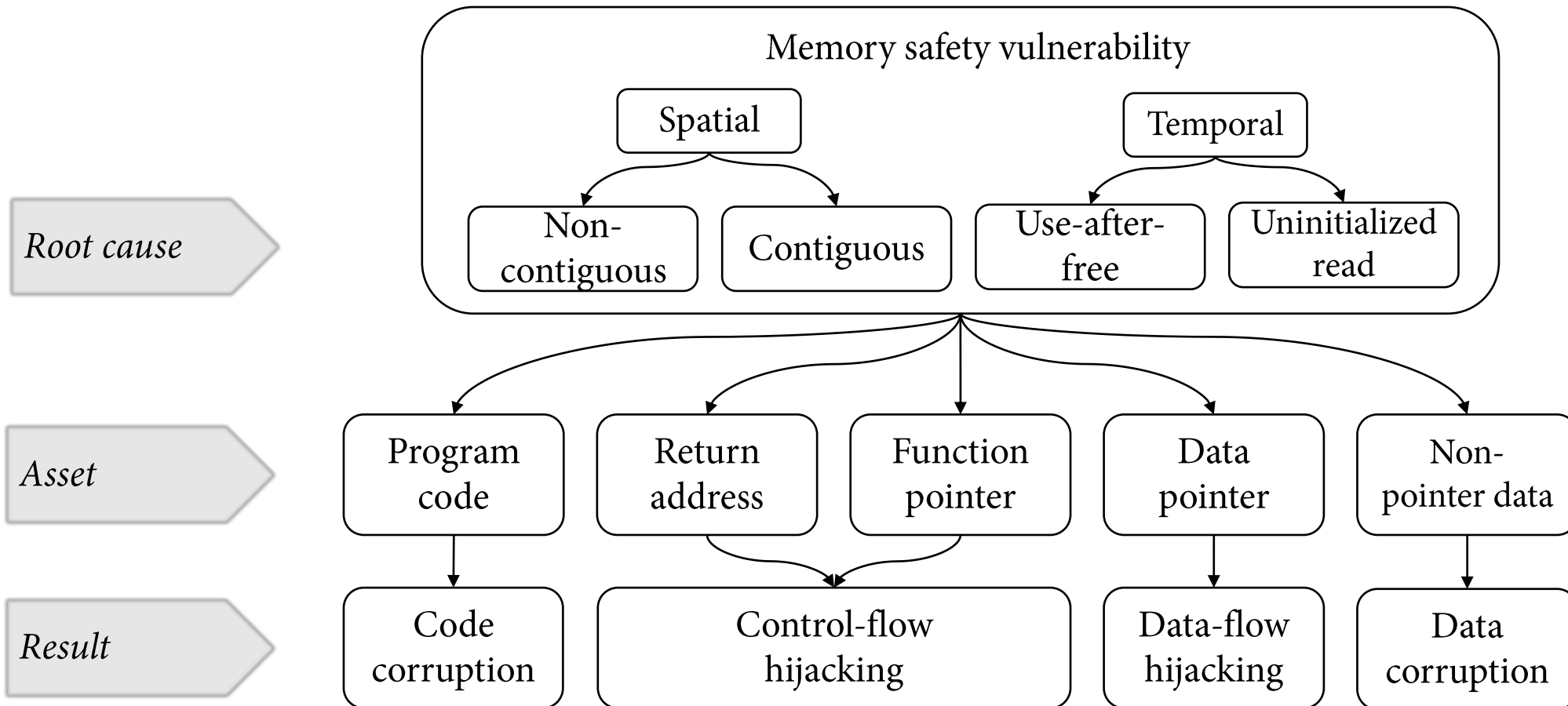
Root cause



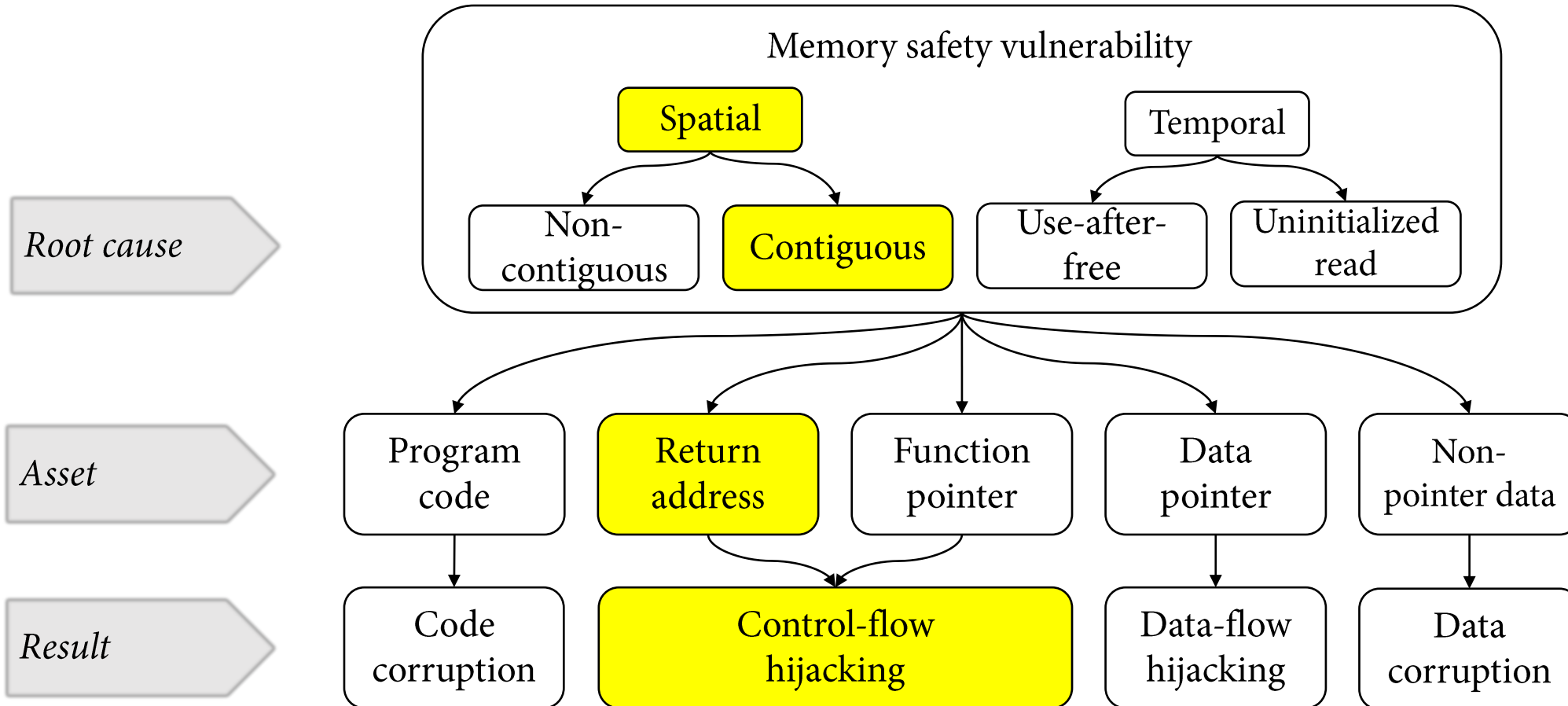
Memory Attacks Taxonomy



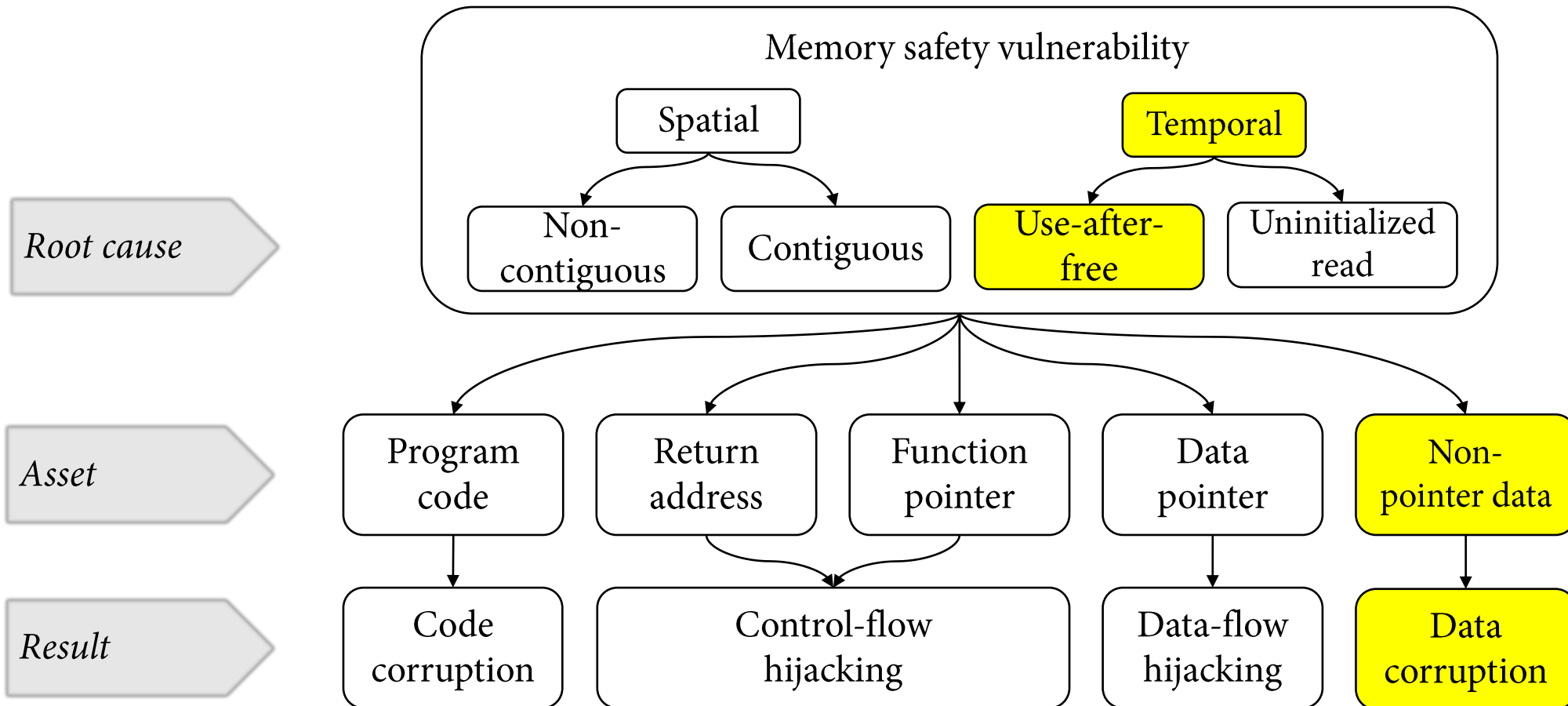
Memory Attacks Taxonomy



Memory Attacks Taxonomy



Memory Attacks Taxonomy



Why is memory safety a concern?



Memory Safety is a serious problem!

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.

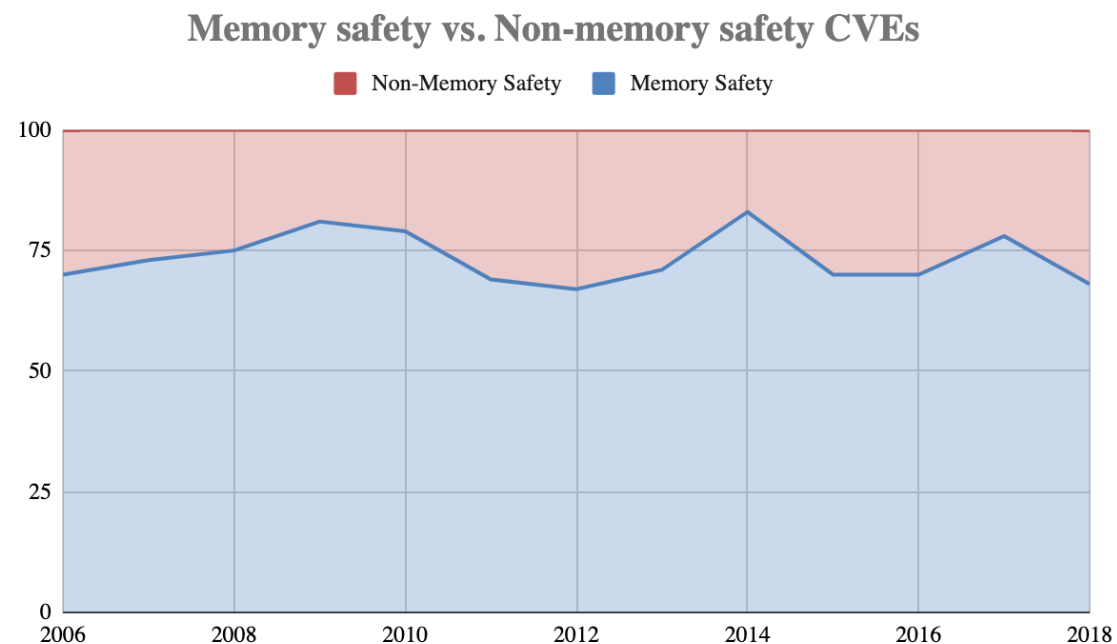
The New York Times

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

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

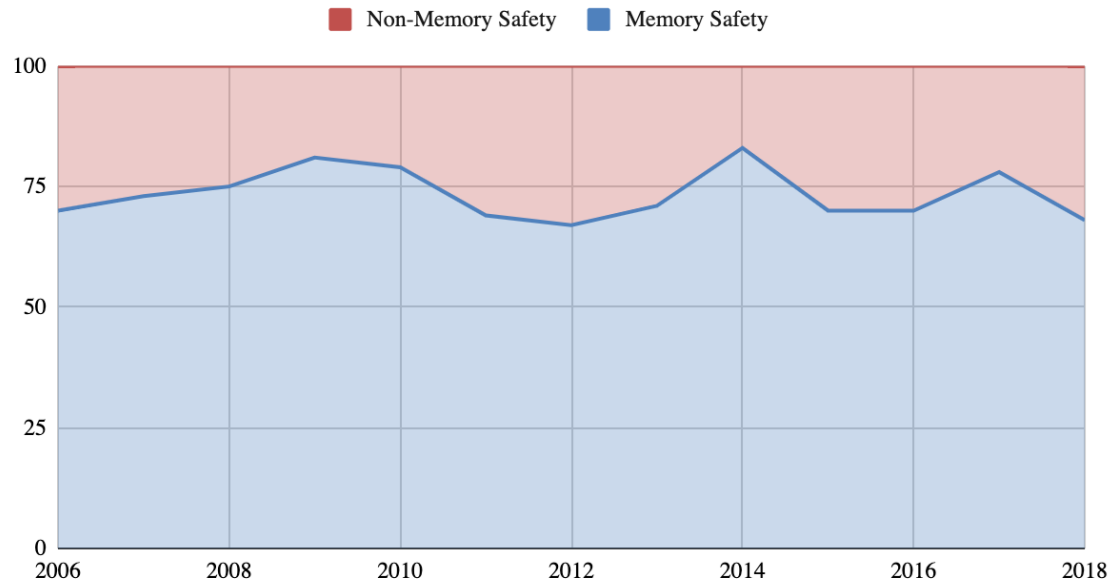
Prevalence of Memory Safety Vulns



Microsoft Product CVEs
between 2006-2018

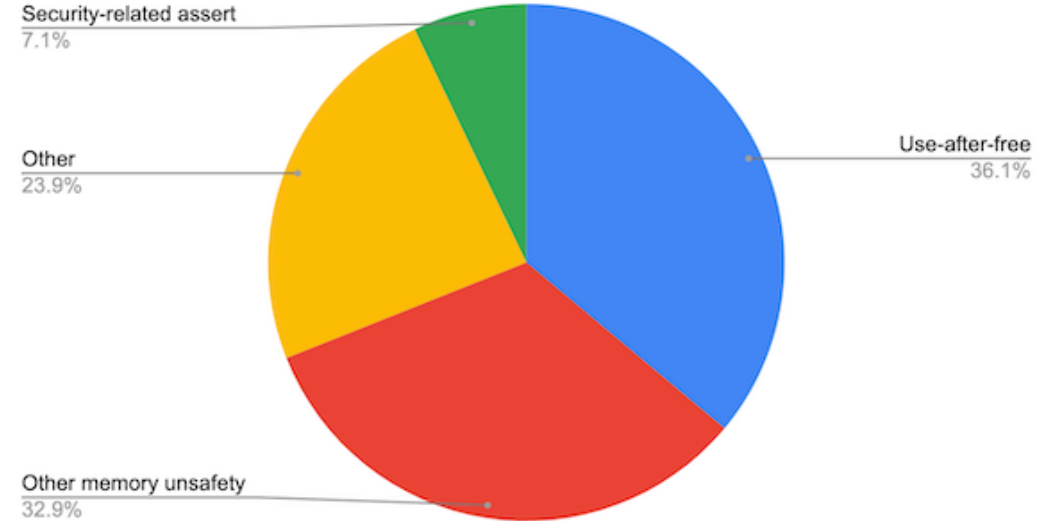
Prevalence of Memory Safety Vulns

Memory safety vs. Non-memory safety CVEs



Microsoft Product CVEs
between 2006-2018

High+, impacting stable



Chromium high severity security bugs
between 2015-2020

TEEN COMIX



BRANSON '15

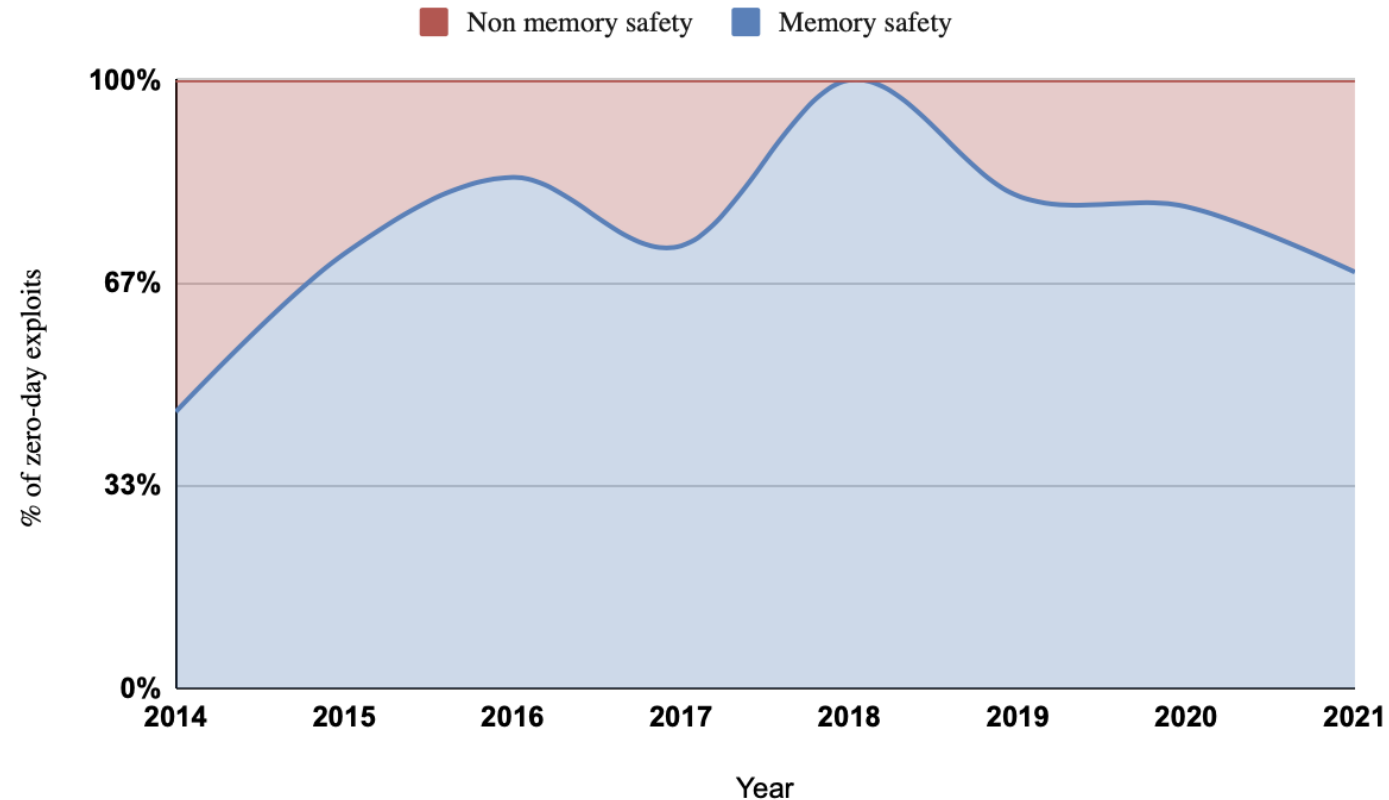


ATTACKERS

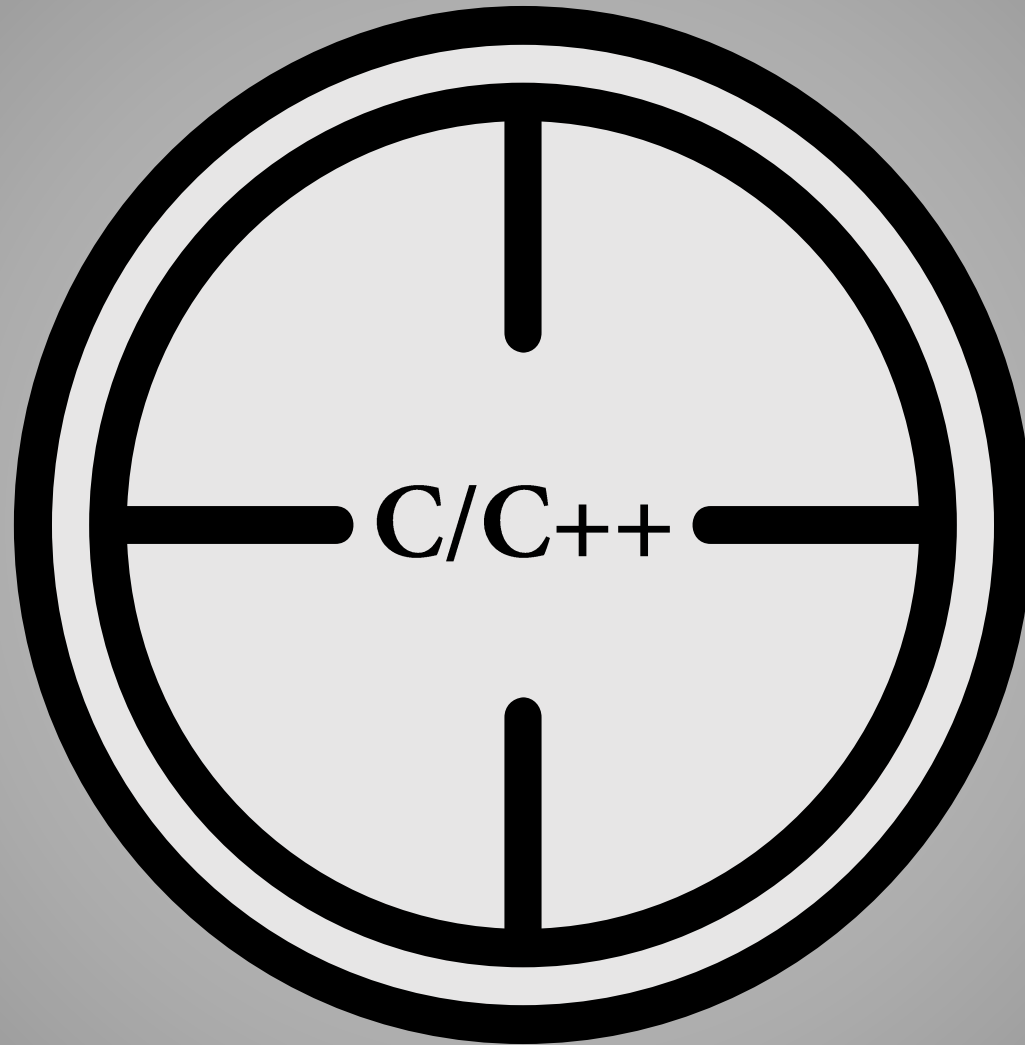


MEMORY SAFETY

Attackers prefer Memory Safety Vulns



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



C/C++ is here to stay!



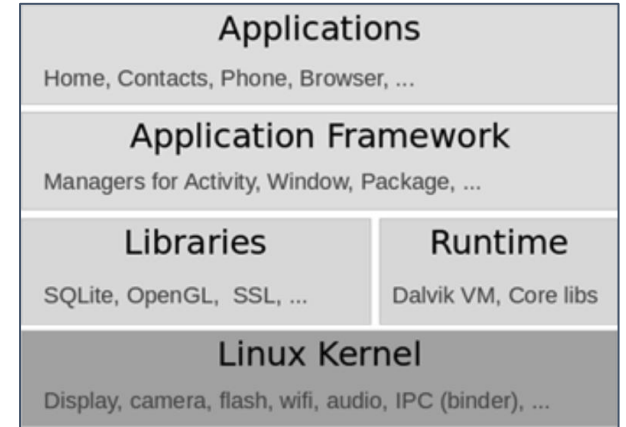
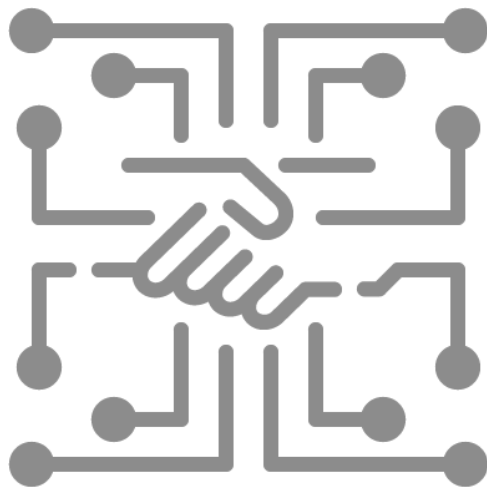
C/C++ is here to stay!



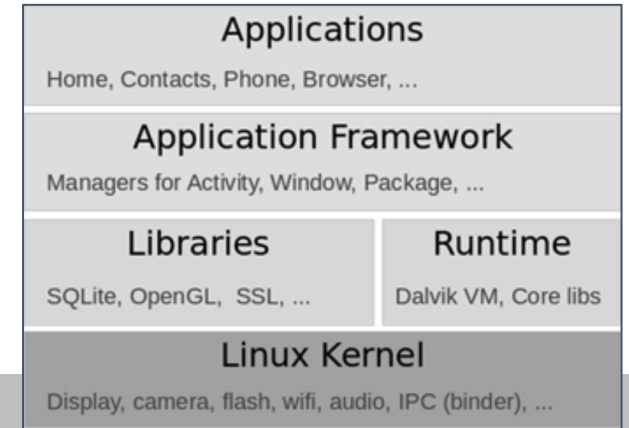
C/C++ is here to stay!



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**

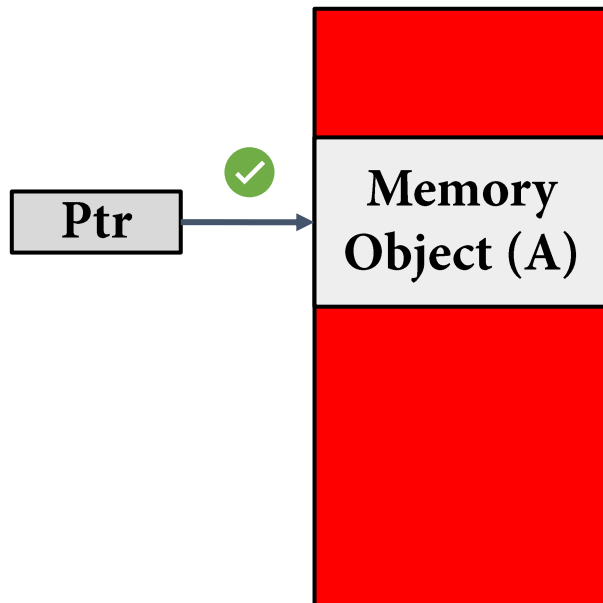
**Exploit
Mitigation**

How to fix C/C++ memory (un)safety?

**Memory
Blocklisting**

**Memory
Permitlisting**

**Exploit
Mitigation**

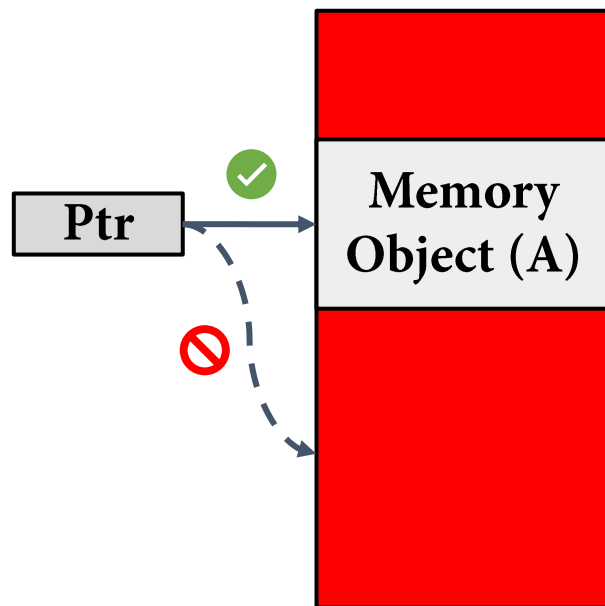


How to fix C/C++ memory (un)safety?

**Memory
Blocklisting**

**Memory
Permitlisting**

**Exploit
Mitigation**

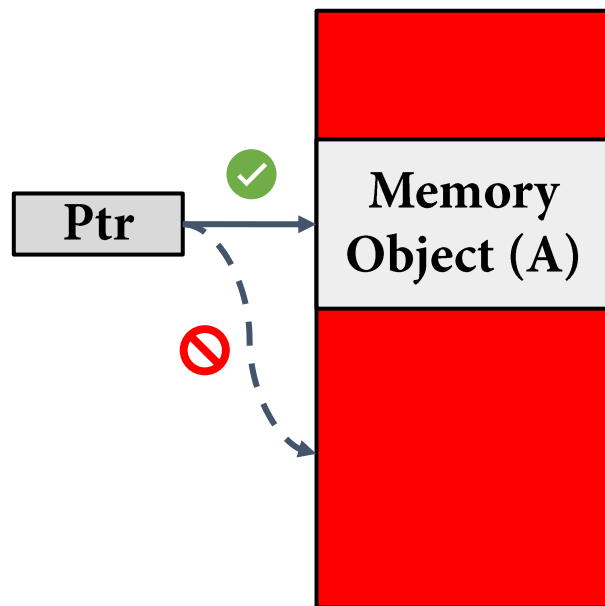


How to fix C/C++ memory (un)safety?

**Memory
Blocklisting**

**Memory
Permitlisting**

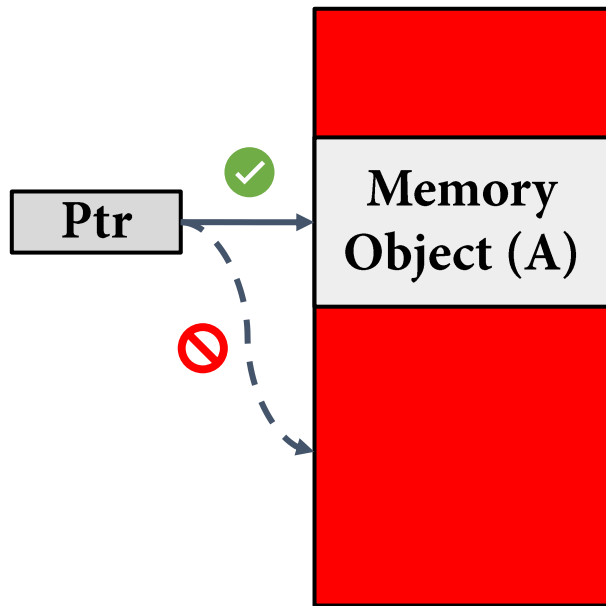
**Exploit
Mitigation**



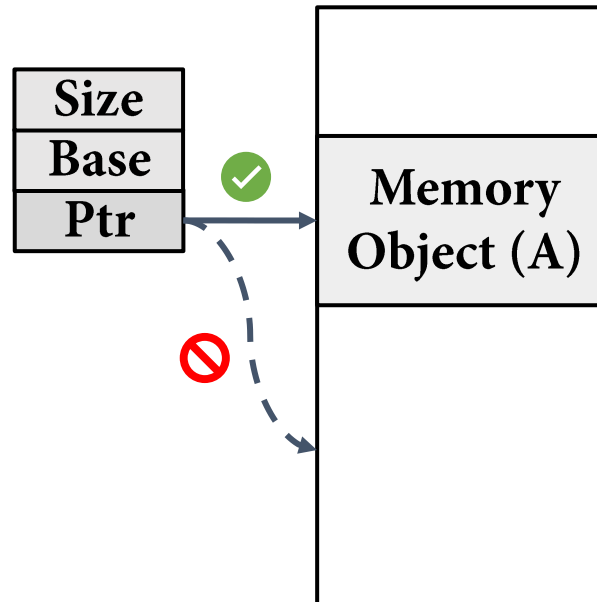
e.g., Google's Address Sanitizer

How to fix C/C++ memory (un)safety?

Memory Blocklisting



Memory Permitlisting

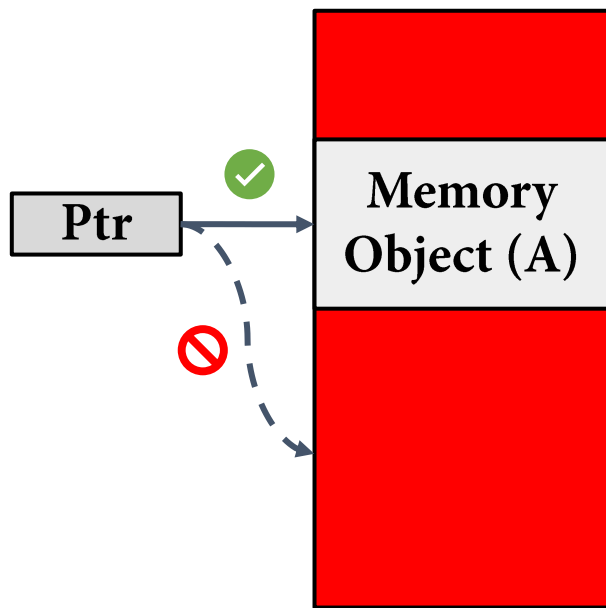


Exploit Mitigation

e.g., Google's Address Sanitizer

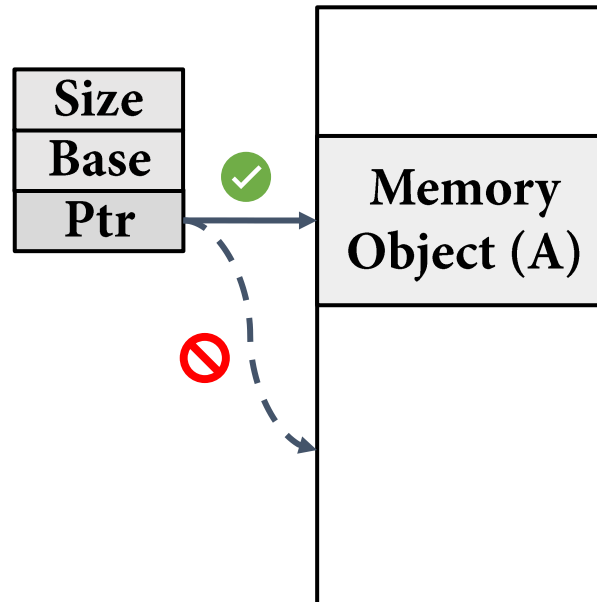
How to fix C/C++ memory (un)safety?

Memory Blocklisting



e.g., Google's Address Sanitizer

Memory Permitlisting



e.g., Intel's MPX and CHERI

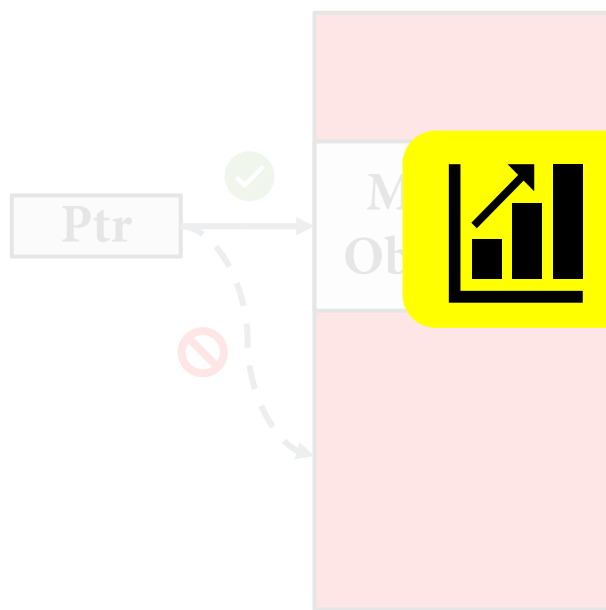
Exploit Mitigation

How to fix C/C++ memory (un)safety?

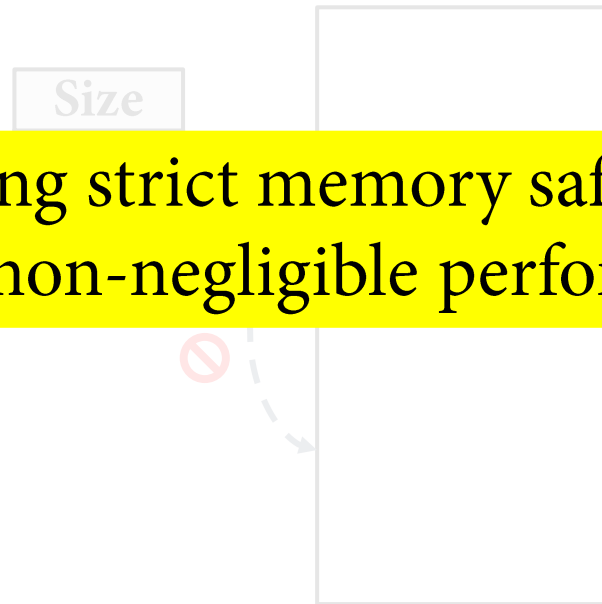
**Memory
Blocklisting**

**Memory
Permitlisting**

**Exploit
Mitigation**



e.g., Google's Address Sanitizer



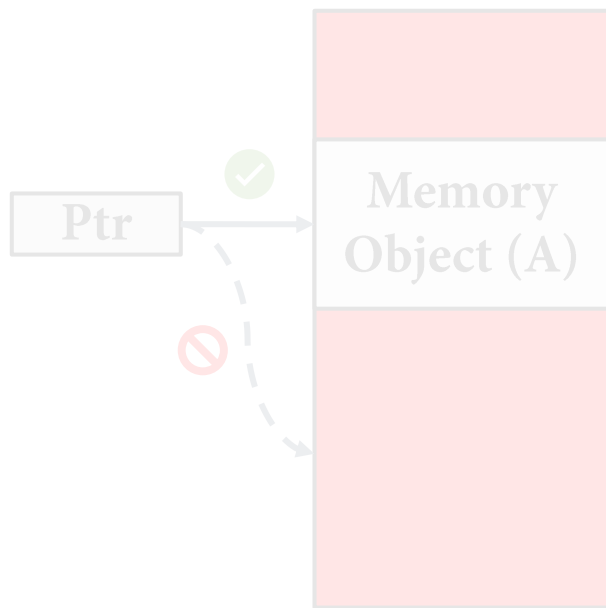
e.g., Intel's MPX and CHERI



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

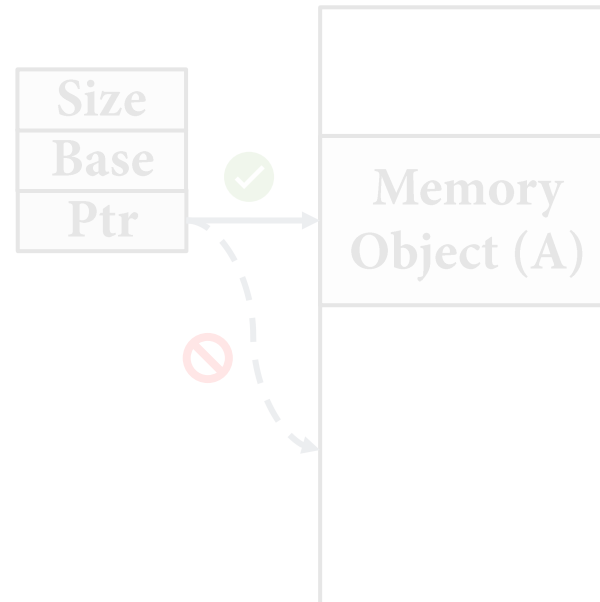
How to fix C/C++ memory (un)safety?

Memory Blocklisting



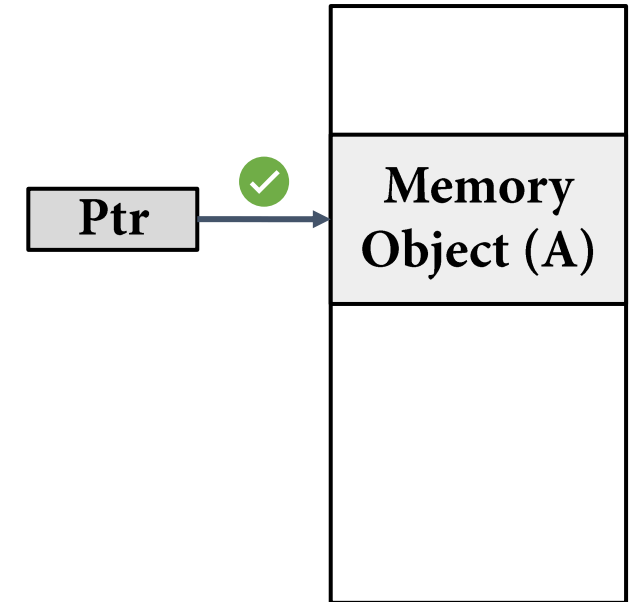
e.g., Google's Address Sanitizer

Memory Permitlisting



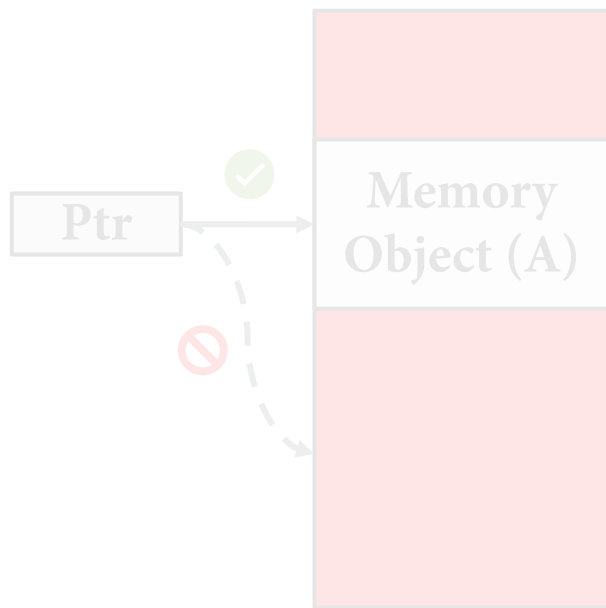
e.g., Intel's MPX and CHERI

Exploit Mitigation



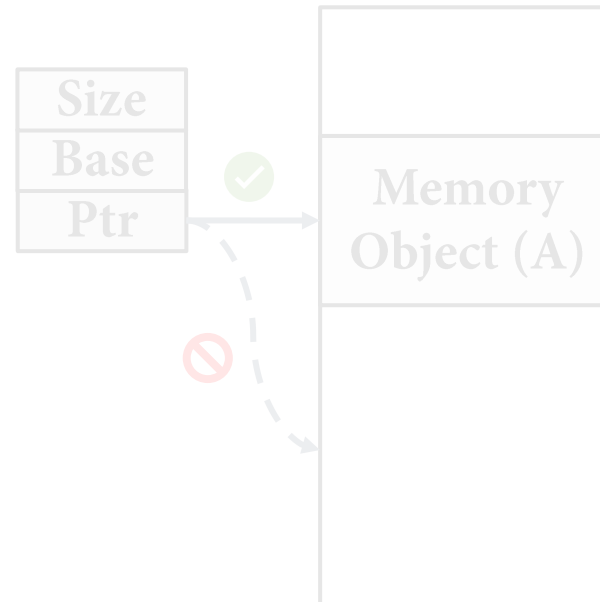
How to fix C/C++ memory (un)safety?

Memory Blocklisting



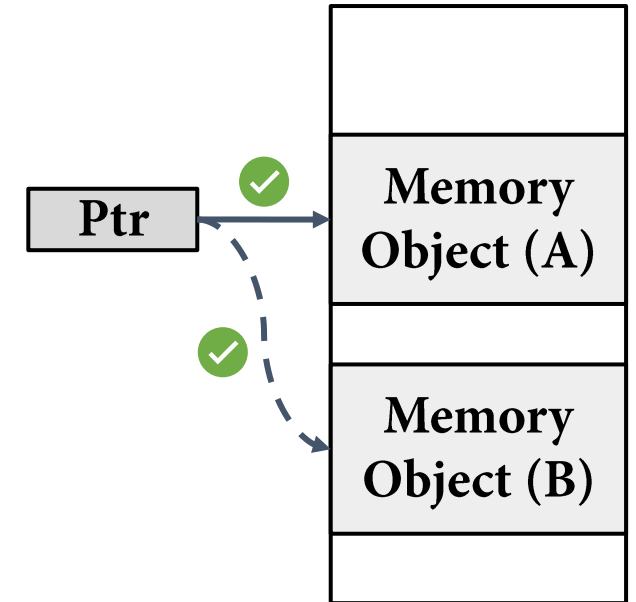
e.g., Google's Address Sanitizer

Memory Permitlisting



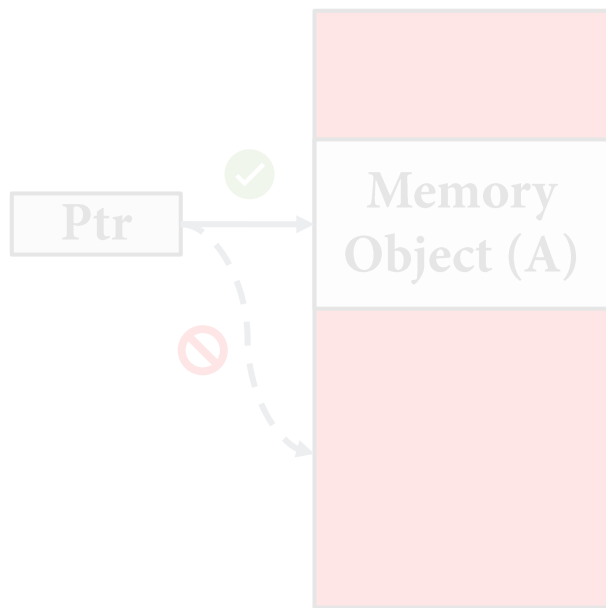
e.g., Intel's MPX and CHERI

Exploit Mitigation



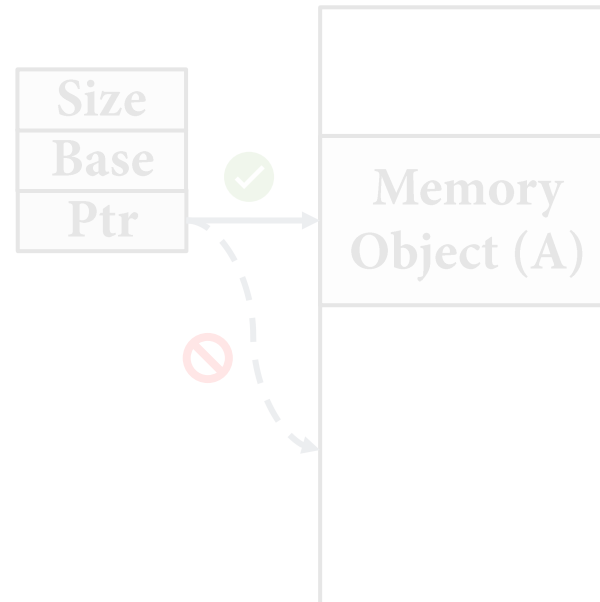
How to fix C/C++ memory (un)safety?

Memory Blocklisting



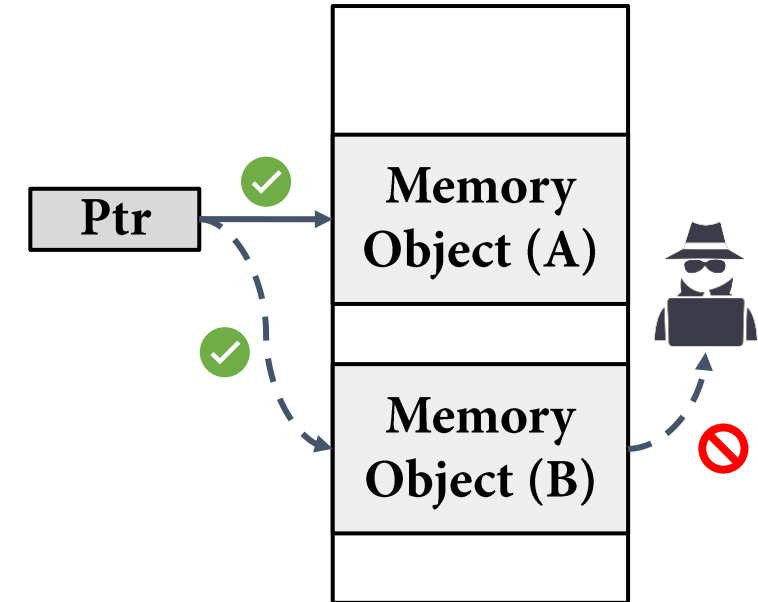
e.g., Google's Address Sanitizer

Memory Permitlisting



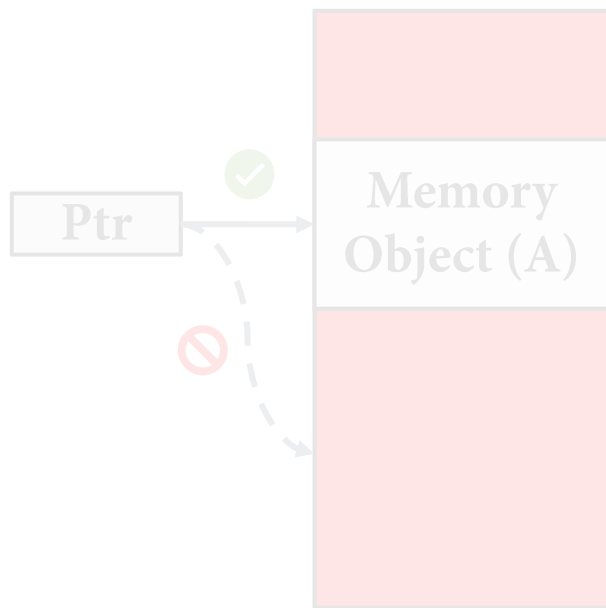
e.g., Intel's MPX and CHERI

Exploit Mitigation



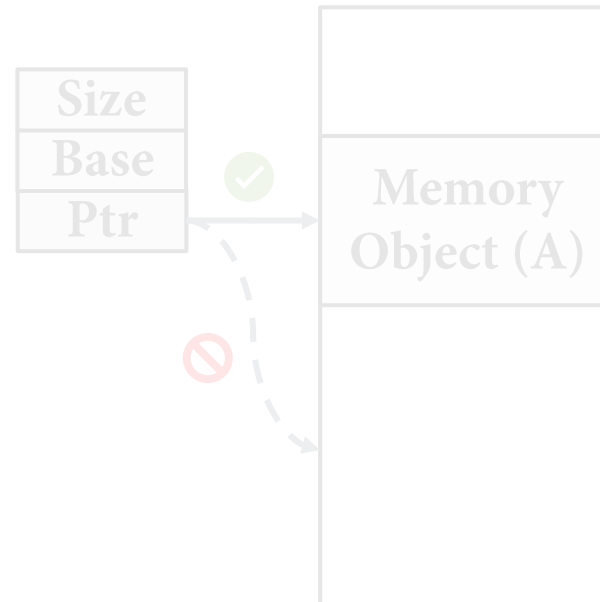
How to fix C/C++ memory (un)safety?

Memory Blocklisting



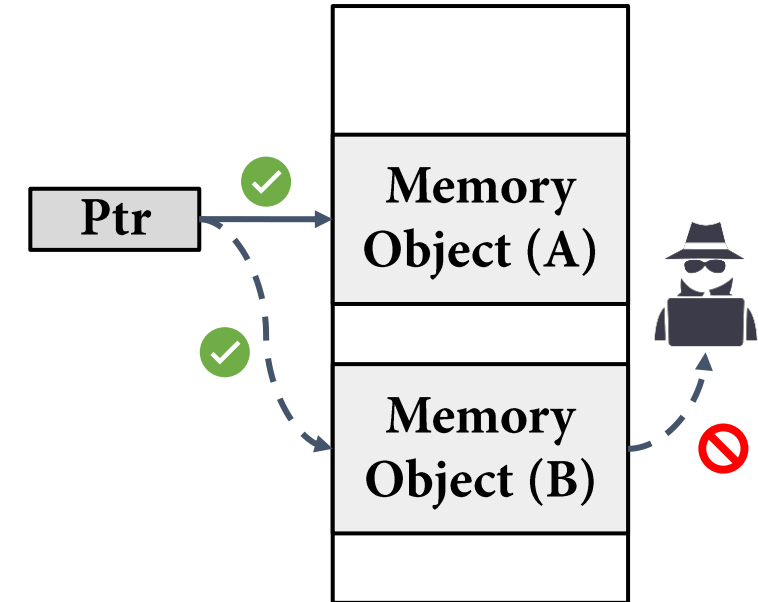
e.g., Google's Address Sanitizer

Memory Permitlisting



e.g., Intel's MPX and CHERI

Exploit Mitigation



e.g., ARM's PAC

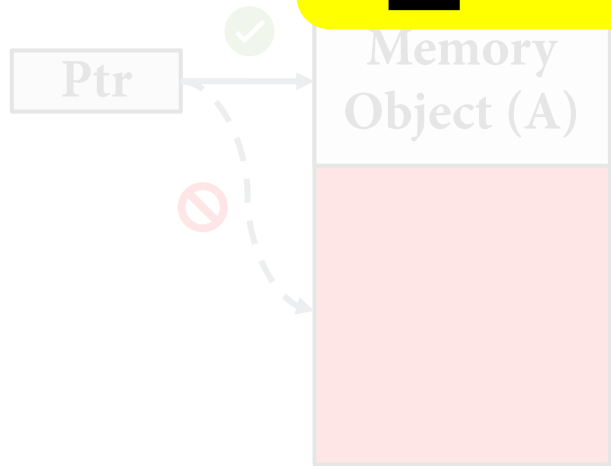
How to fix C/C++ memory (un)safety?

**Memory
Blocklisting**

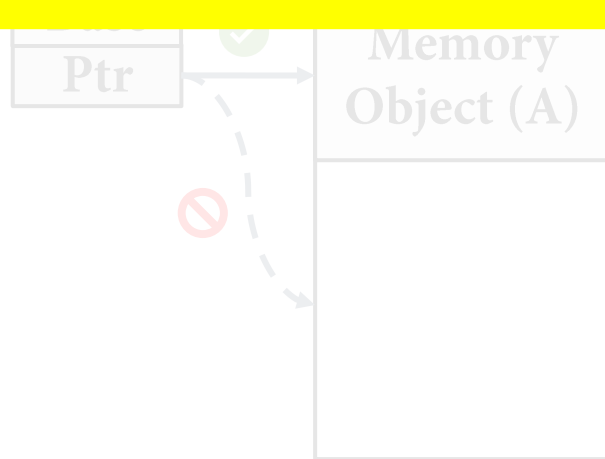
**Memory
Permitlisting**

**Exploit
Mitigation**

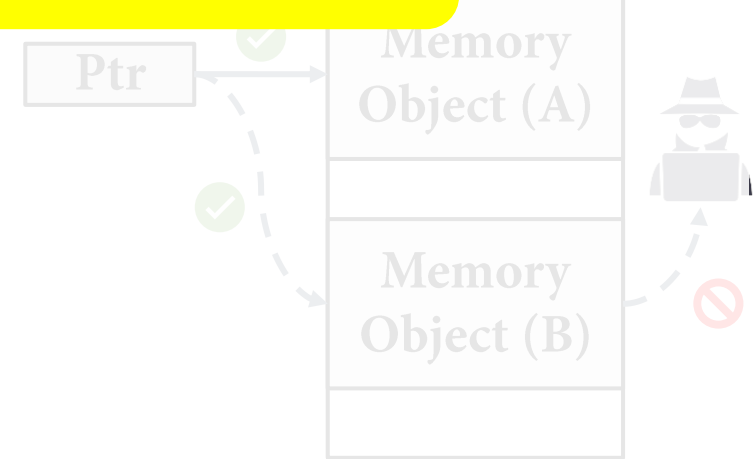
 All prior approaches share a common theme:



e.g., Google's Address Sanitizer



e.g., Intel's MPX and CHERI




e.g., ARM's PAC

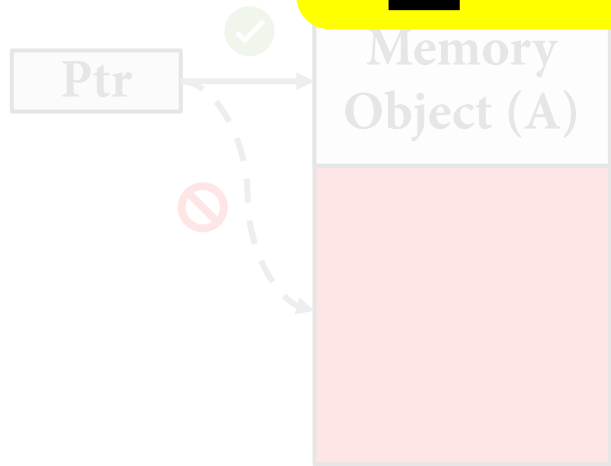
How to fix C/C++ memory (un)safety?

**Memory
Blocklisting**

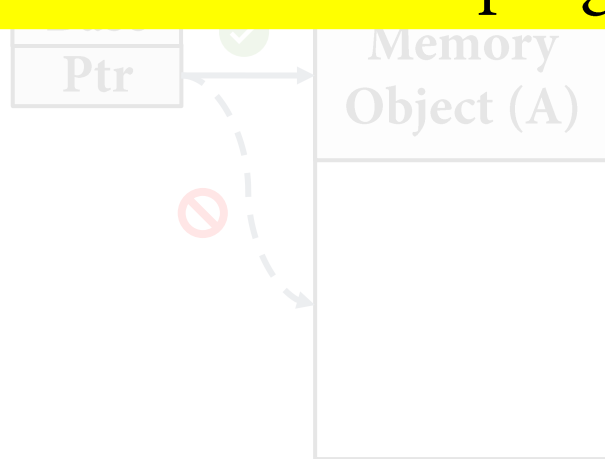
**Memory
Permitlisting**

**Exploit
Mitigation**

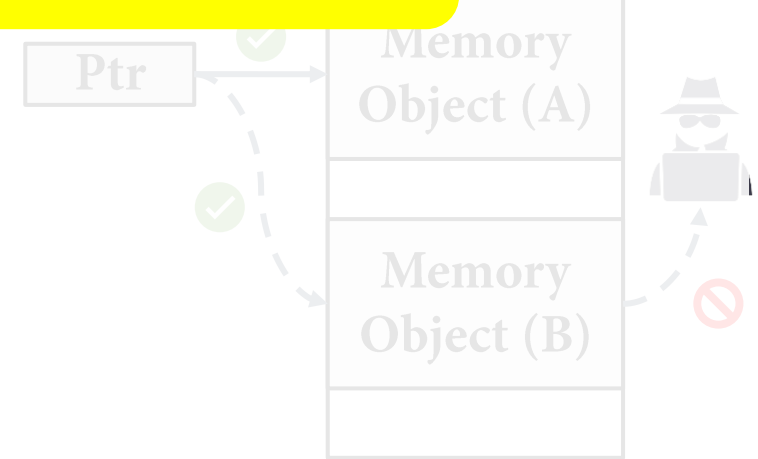
 All prior approaches share a common theme:
Adding more features to a program to make it secure



e.g., Google's Address Sanitizer



e.g., Intel's MPX and CHERI



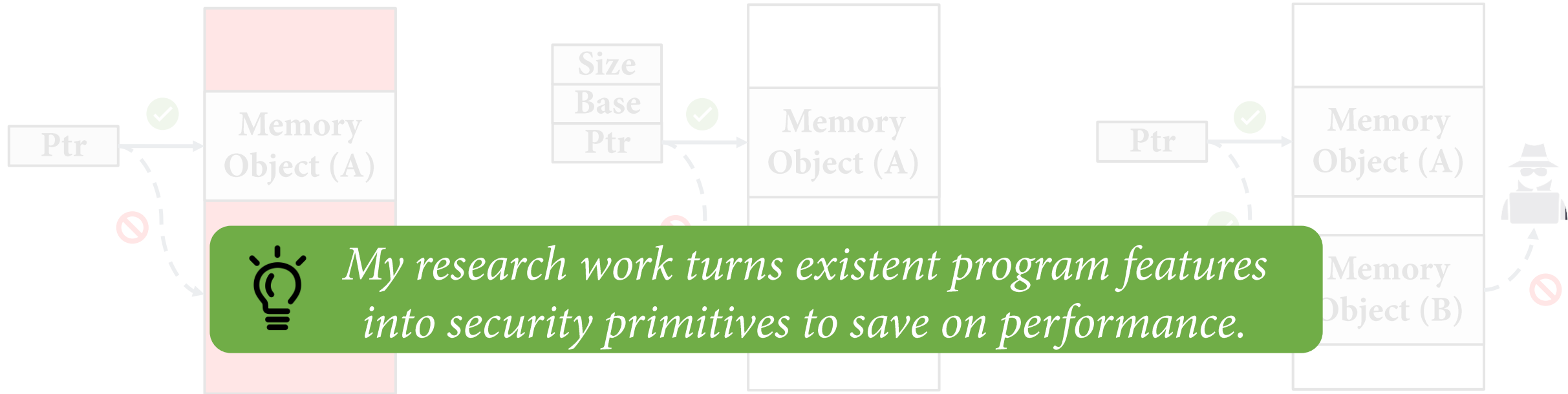
e.g., ARM's PAC

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

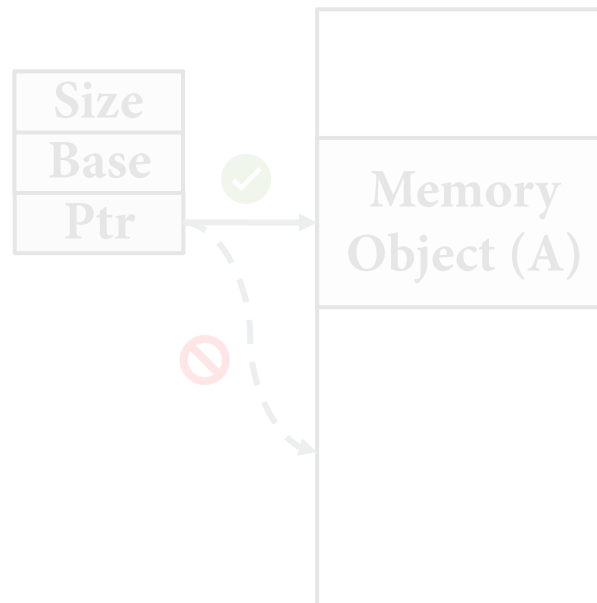
My solutions for C/C++ memory (un)safety

Memory Blacklisting

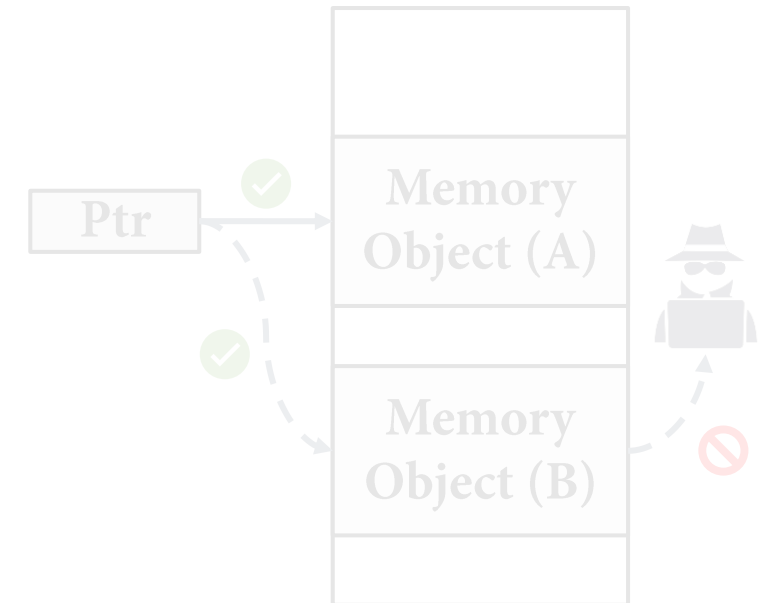


*Uses dead bytes in
program memory*

Memory Permitlisting



Exploit Mitigation

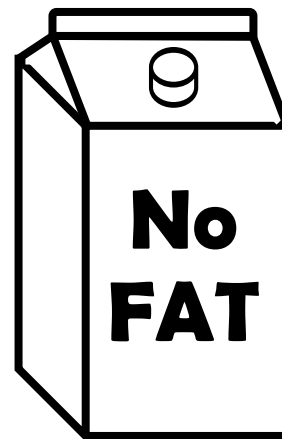


My solutions for C/C++ memory (un)safety

Memory Blocklisting

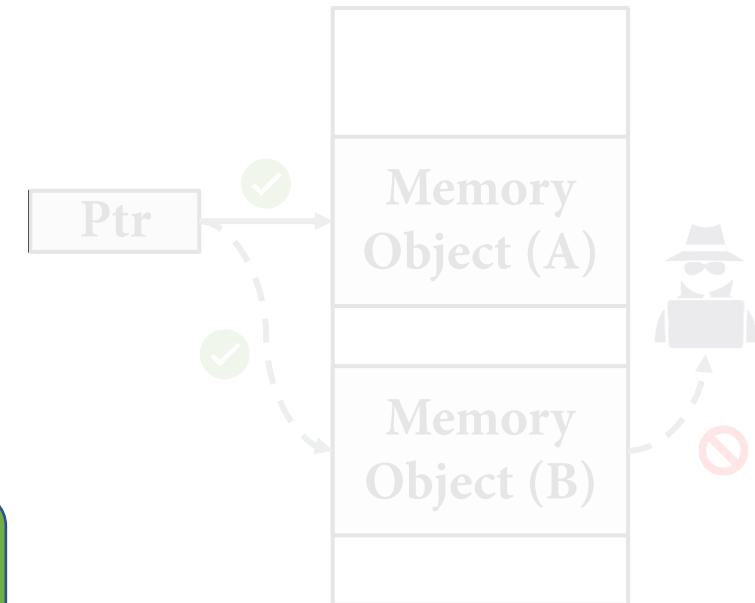


Memory Permitlisting



Leverages modern software trends

Exploit Mitigation

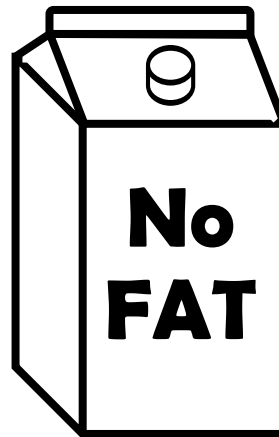


My solutions for C/C++ memory (un)safety

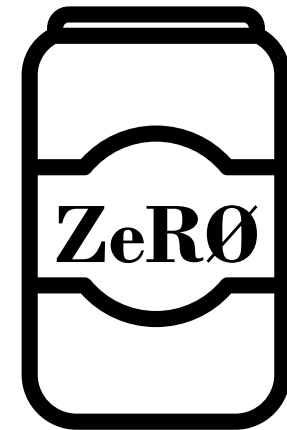
**Memory
Blocklisting**



**Memory
Permitlisting**



**Exploit
Mitigation**



*Mitigates all known exploits
with zero runtime overheads.*

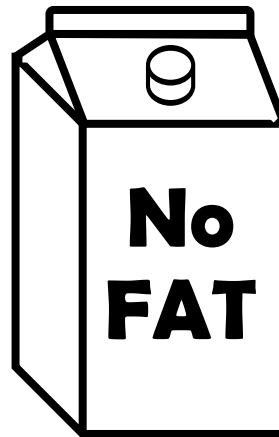
My solutions for C/C++ memory (un)safety

Memory Blocklisting



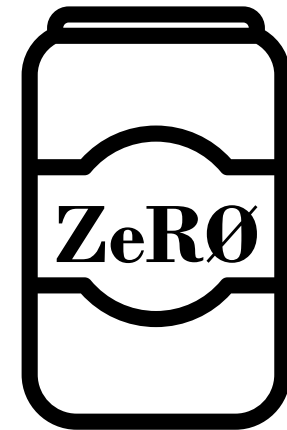
[[MICRO 2019](#)]

Memory Permitlisting



[[ISCA 2021](#)]

Exploit Mitigation



[[ISCA 2021](#)]



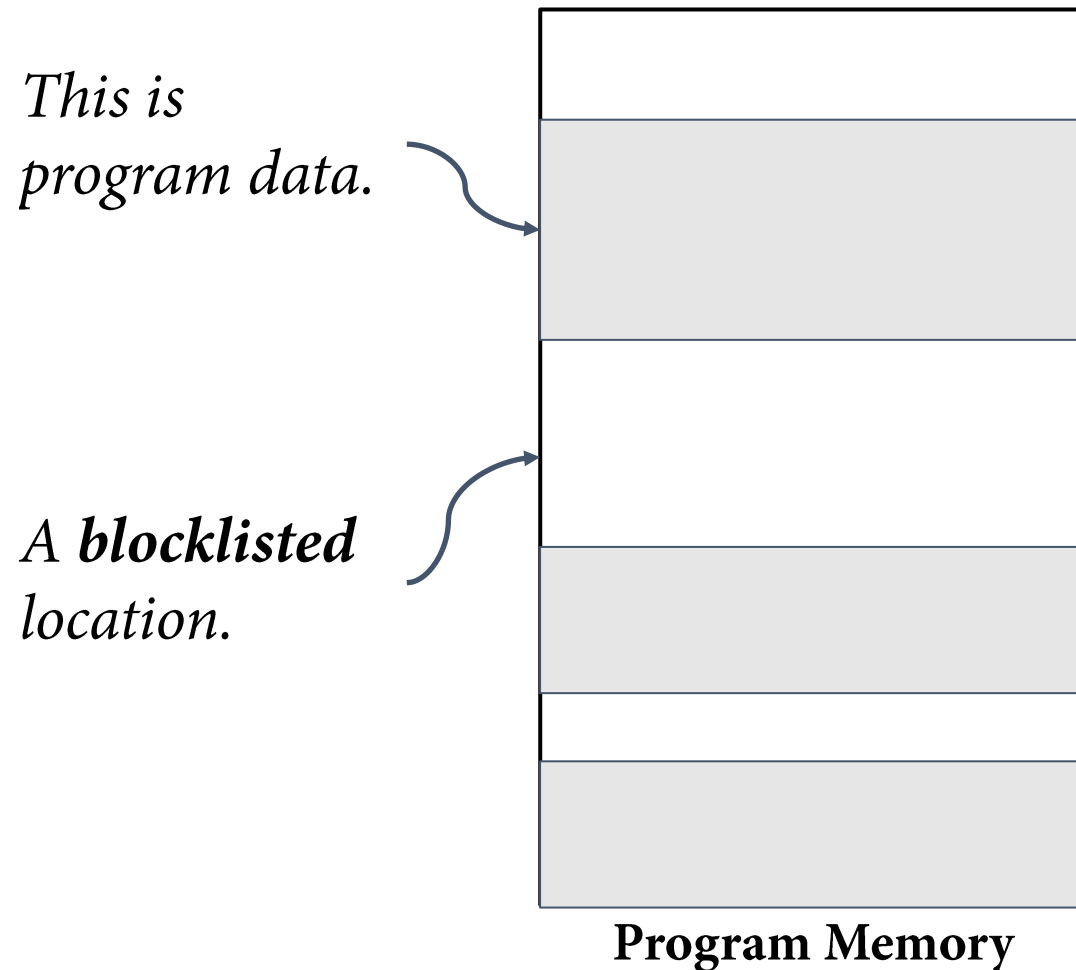
Cache Line Formats

Hiroshi Sasaki, Miguel A. Arroyo, **Mohamed Tarek Ibn Ziad**, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan, Practical byte-granular memory blacklisting using Califorms.

[[MICRO 2019](#)] [IEEE Micro Top Picks Honorable Mention]

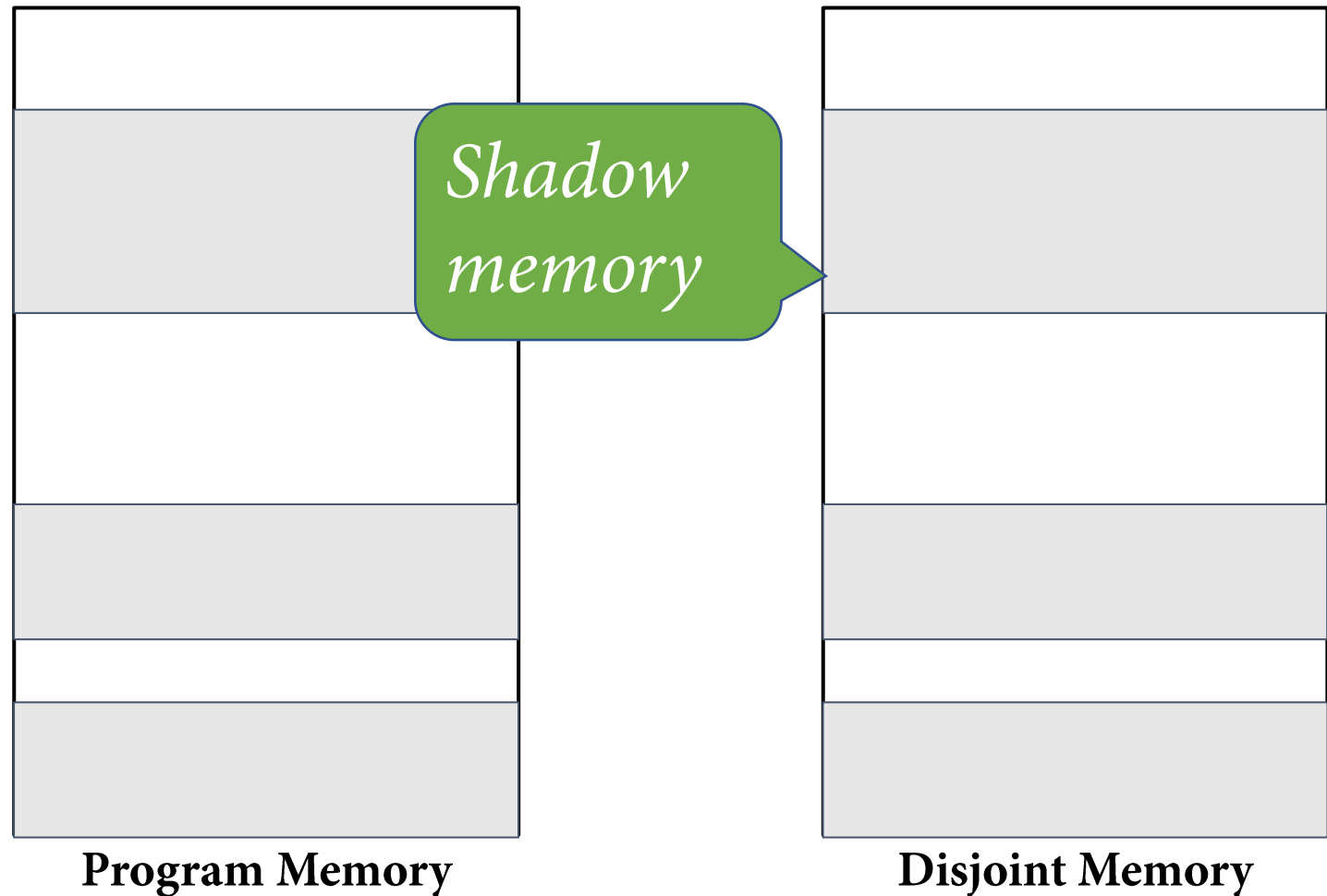


CaLiForms Memory Blocklisting

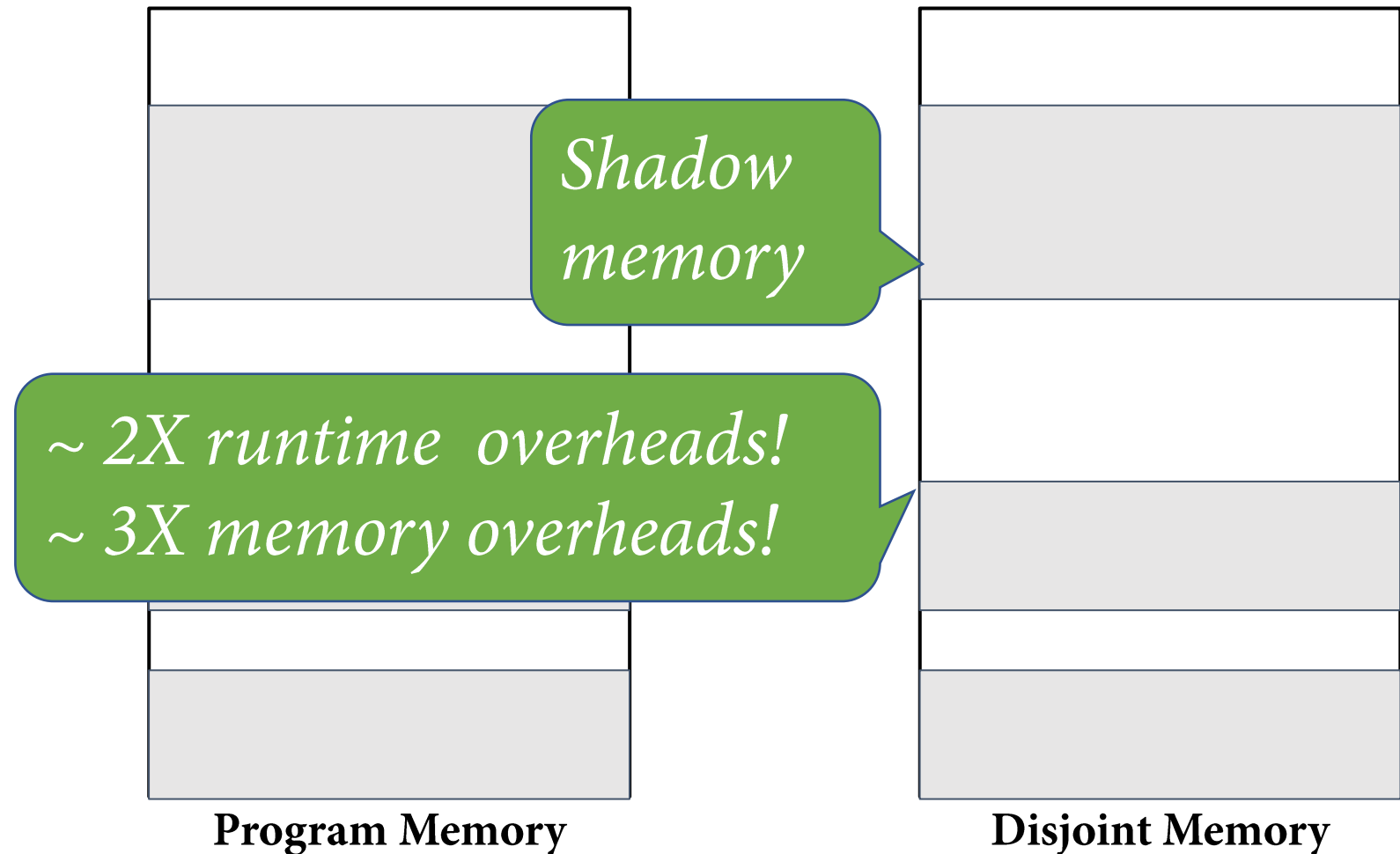


Challenge
How to efficiently track the state of memory locations?

CaLiForms Memory Blocklisting



CaLiForms Memory Blocklisting

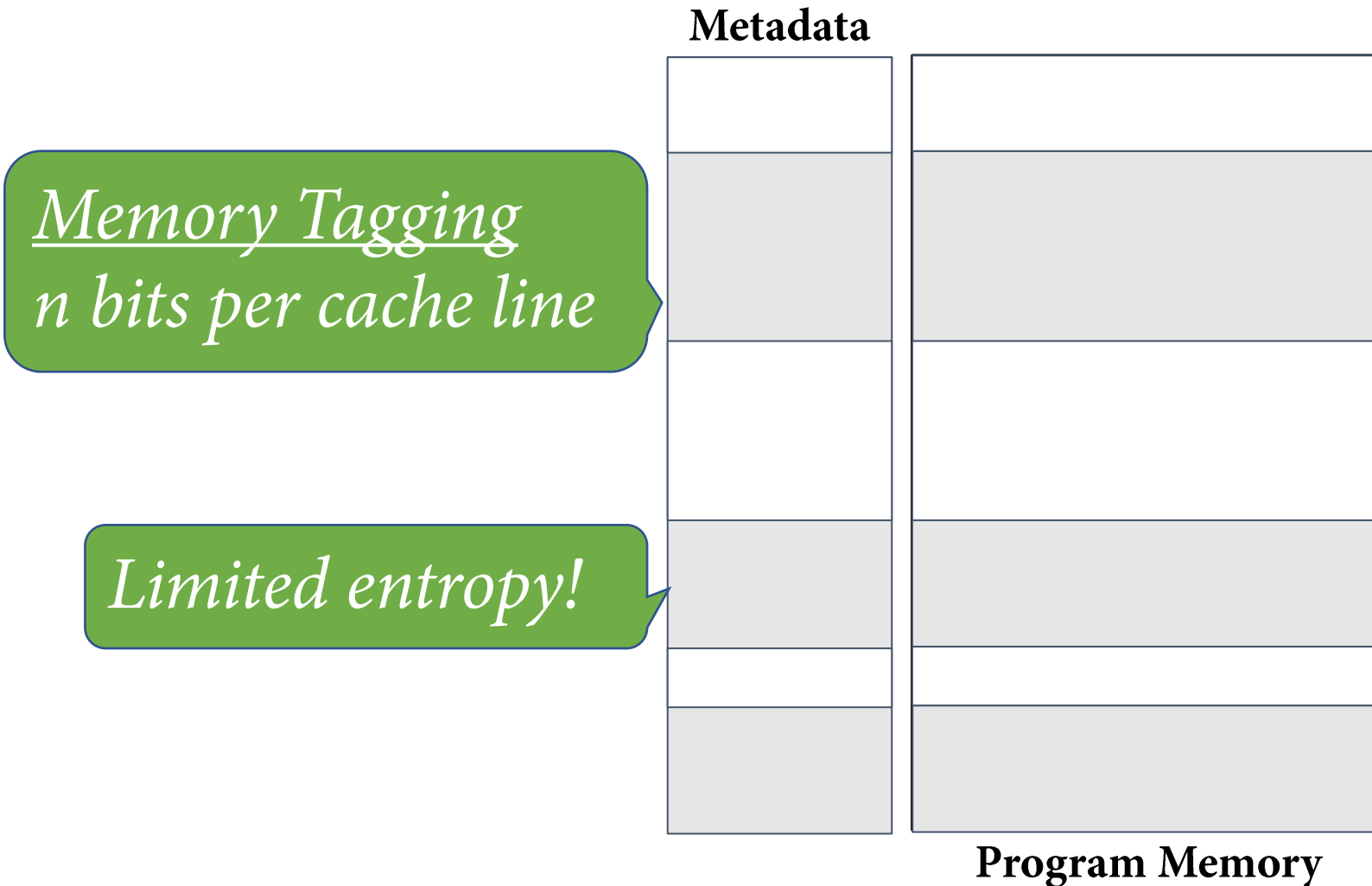


CaLiForms Memory Blocklisting

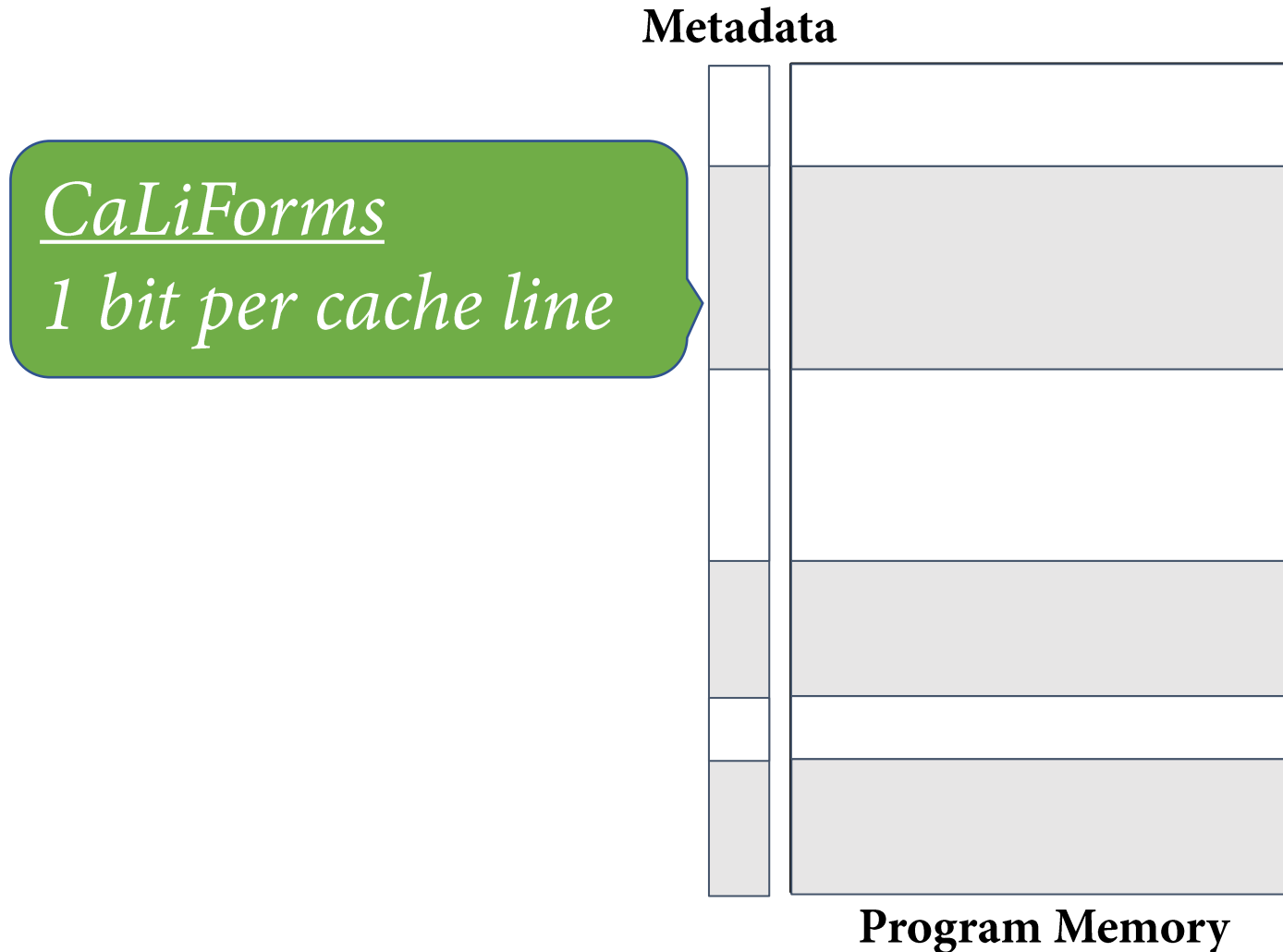
*Memory Tagging
n bits per cache line*



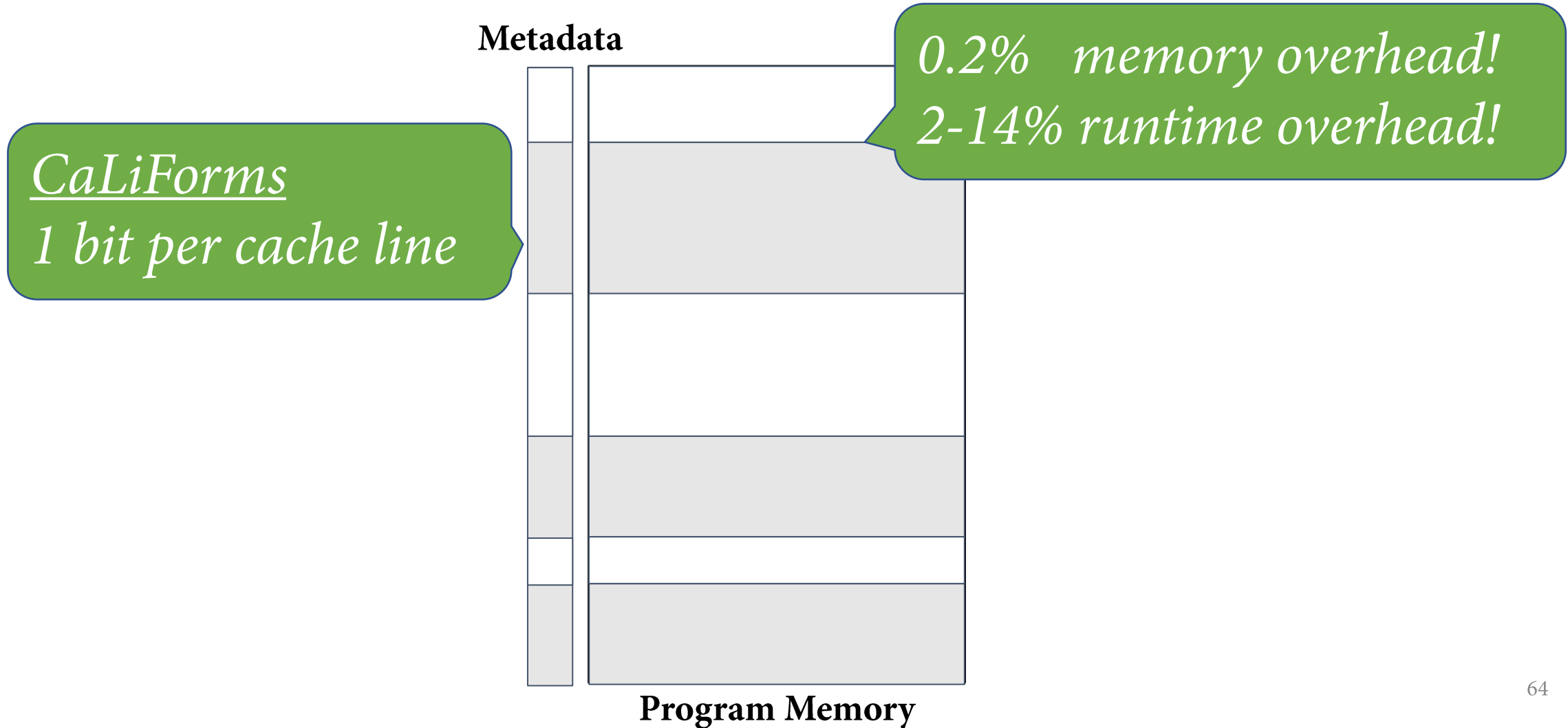
CaLiForms Memory Blocklisting



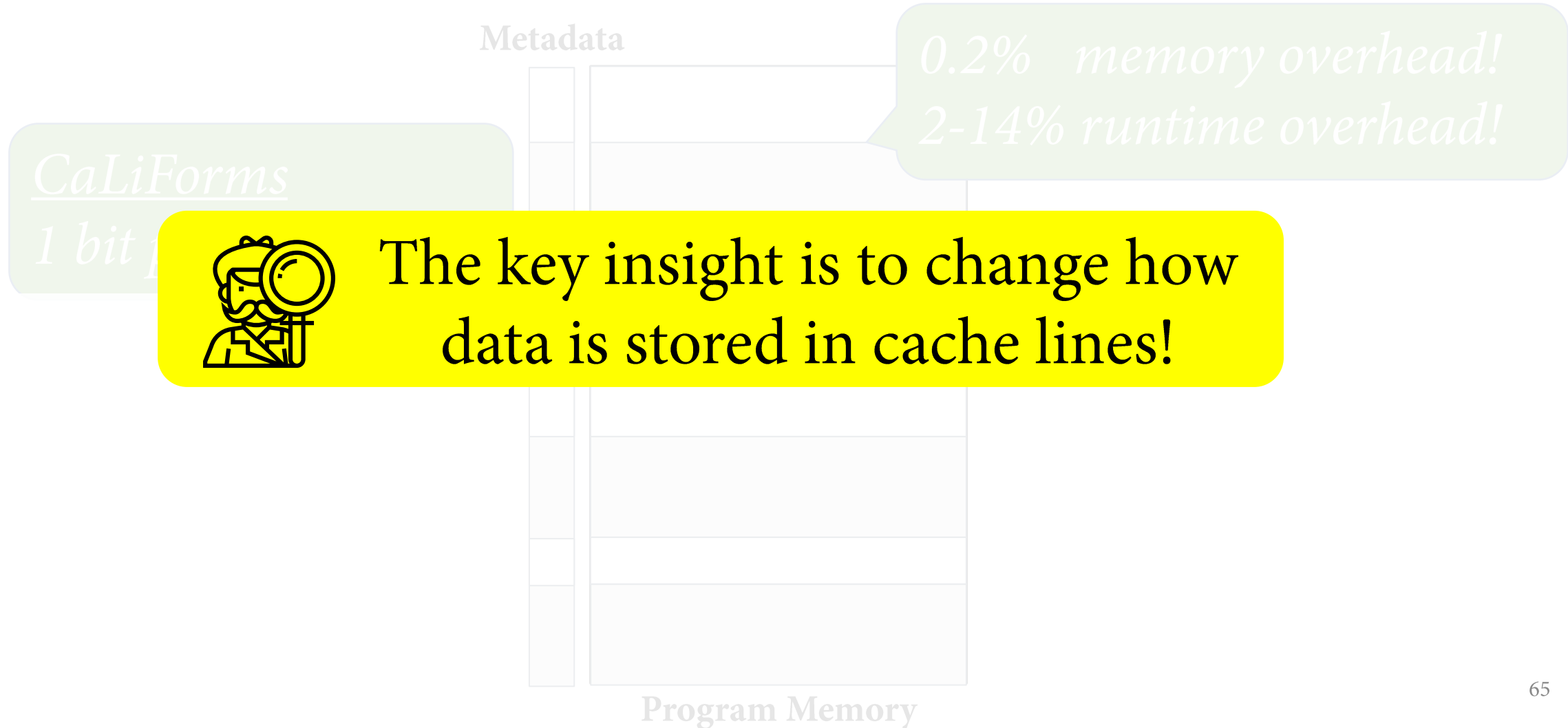
CaLiForms Memory Blocklisting



CaLiForms Memory Blocklisting



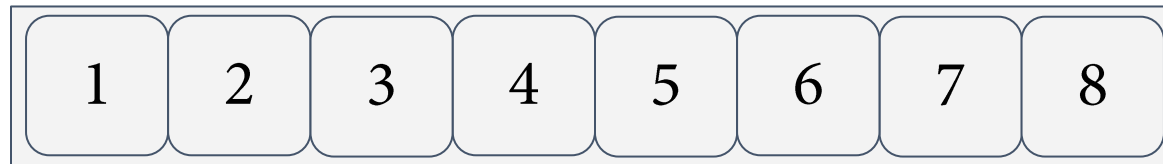
CaLiForms Memory Blocklisting



CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

Normal

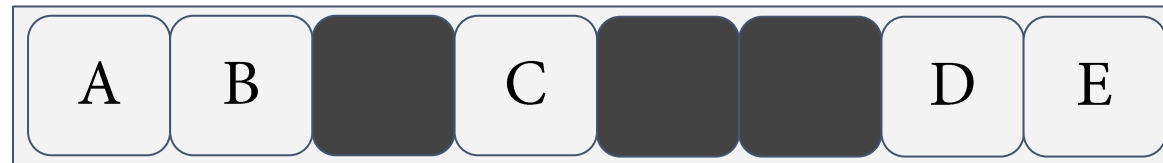


CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

 Blocklisted
Location

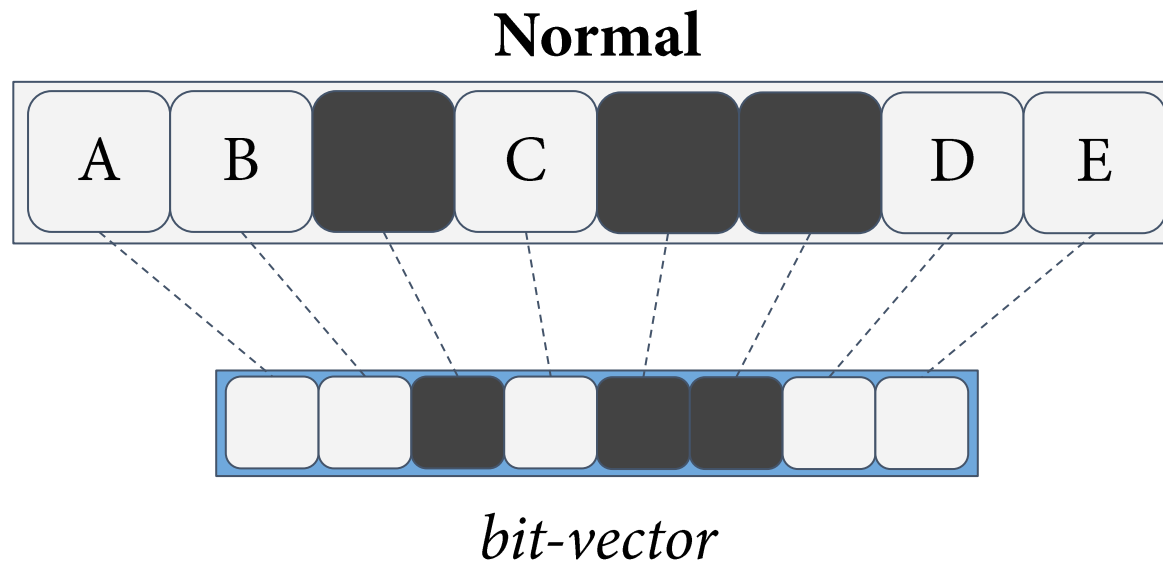
Normal



CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

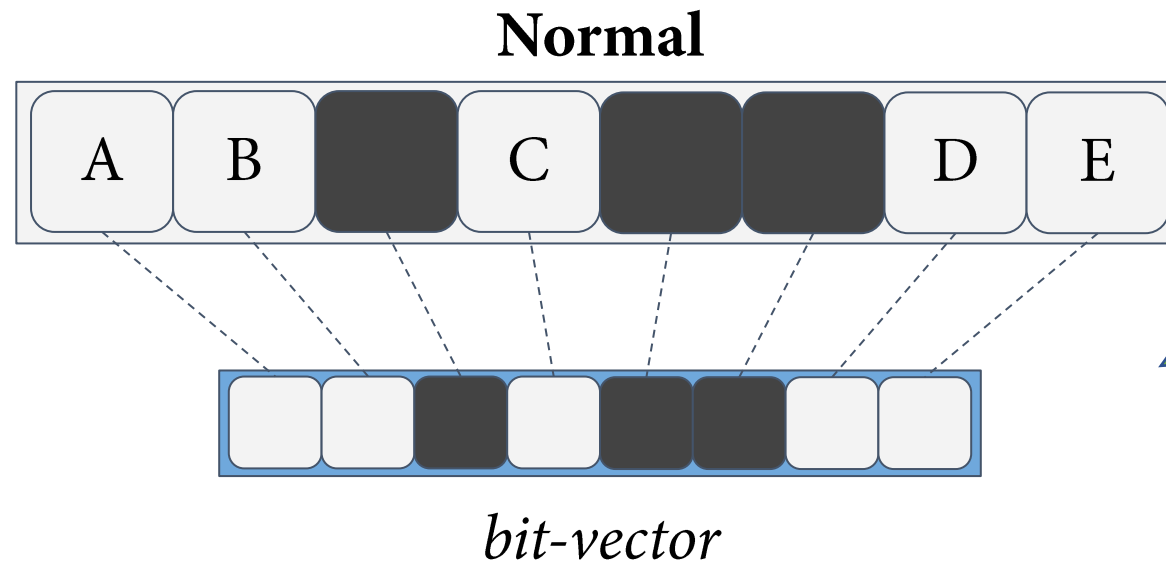
 Blocklisted
Location



CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

 Blocklisted
Location



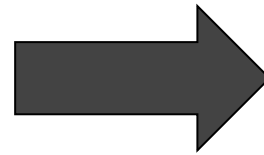
12.5% memory overhead

CaLiForms Cache Line Formats

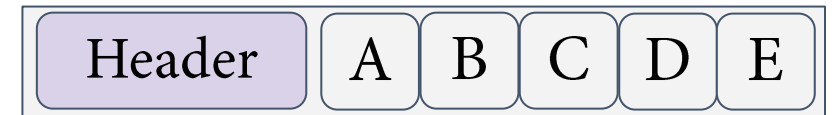
Our Metadata: Encoded within unused data.

 Blocklisted
Location

Normal



Califorms

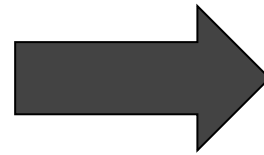


CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

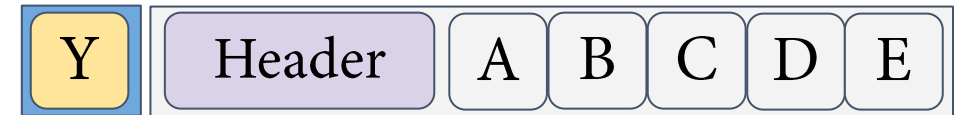
 Blocklisted
Location

Normal



Is
Califormed?

Califorms

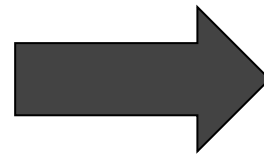


CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

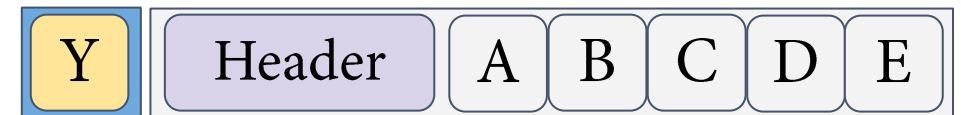
 Blocklisted
Location

Normal

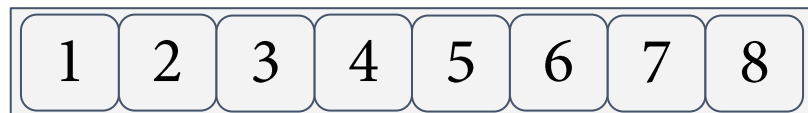


Is
Califormed?

Califorms



Normal

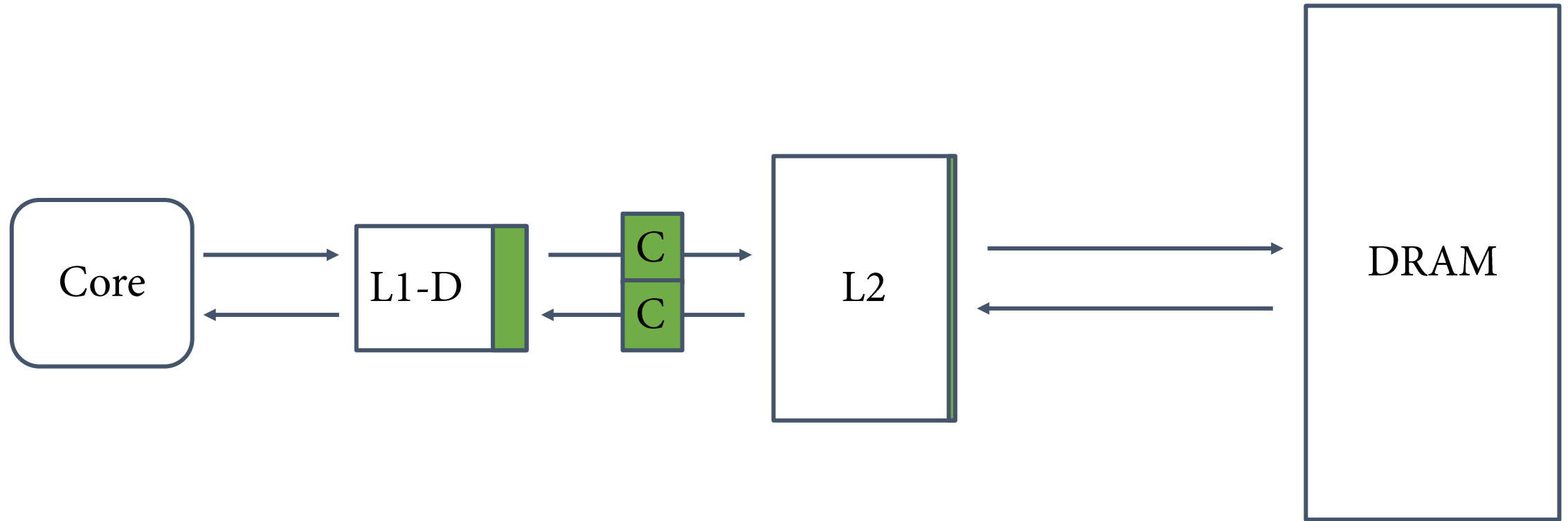


Is
Califormed?

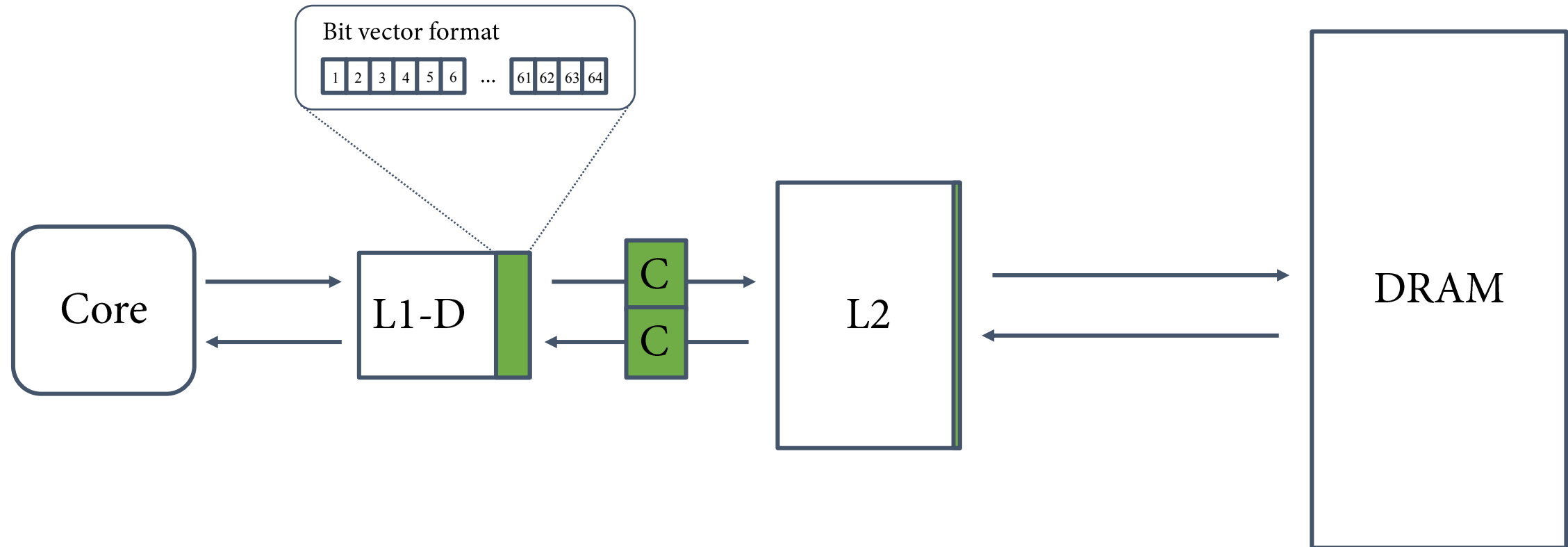
Normal



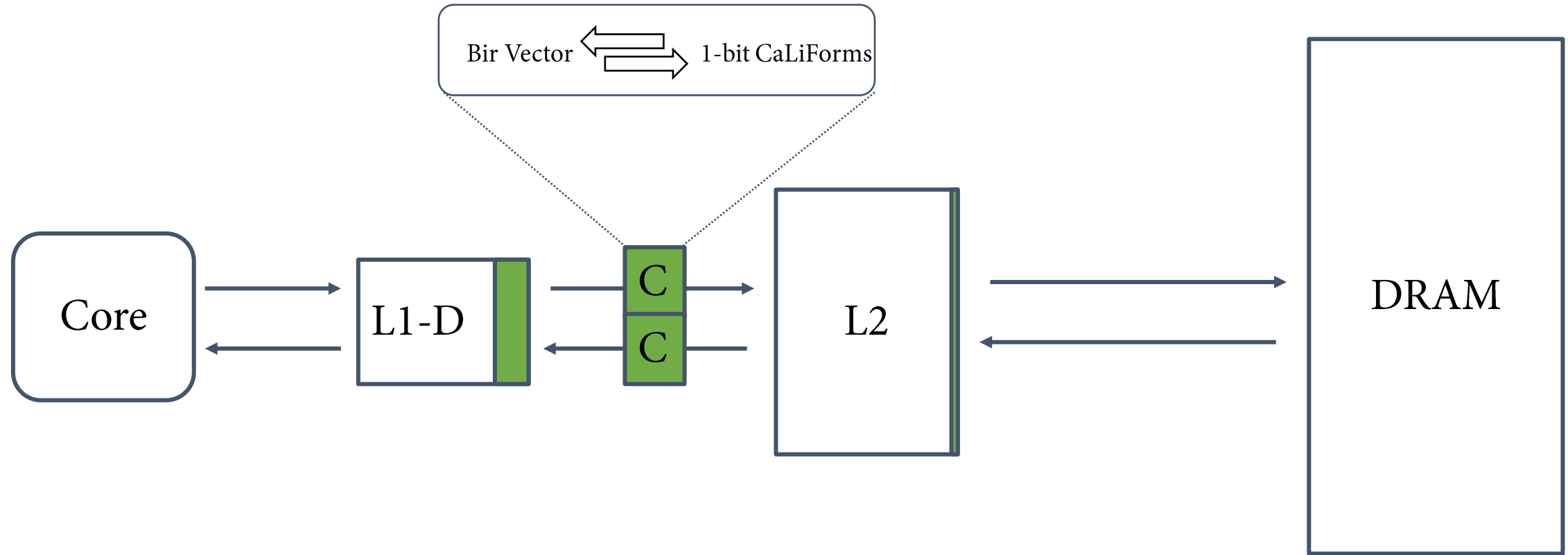
CaLiForms Microarchitectural Overview



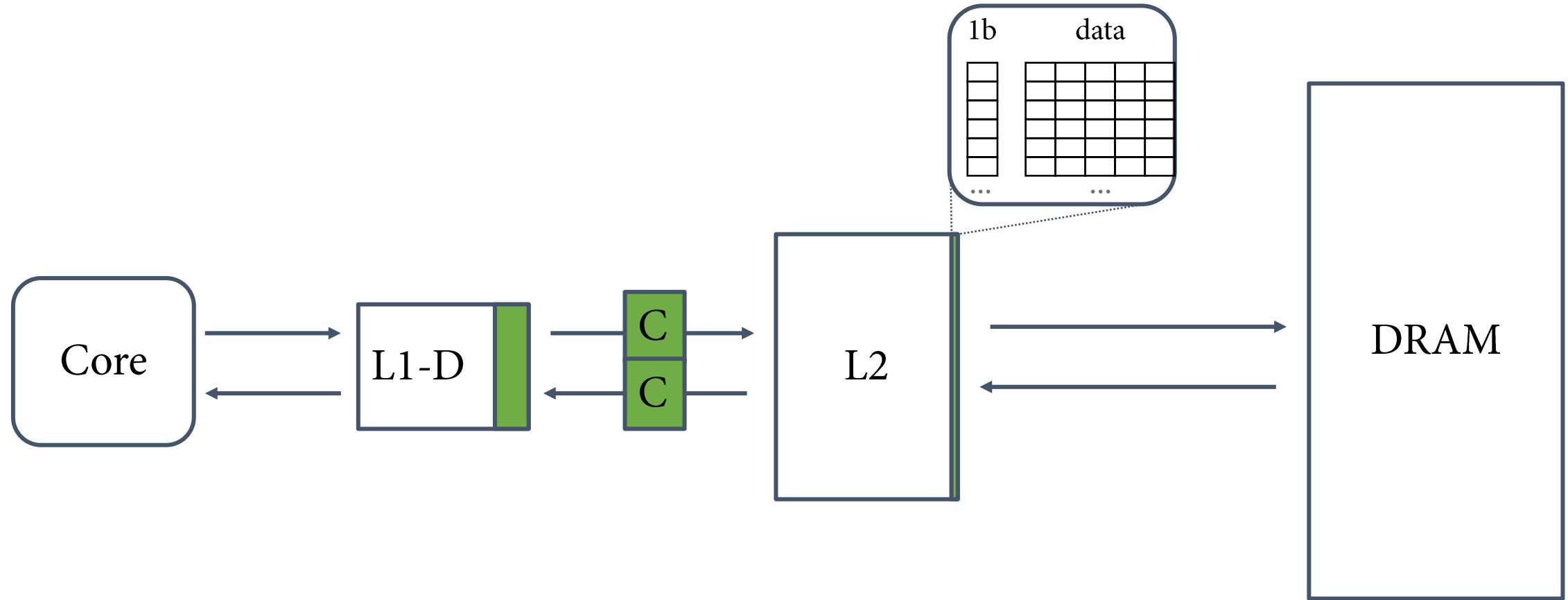
CaLiForms Microarchitectural Overview



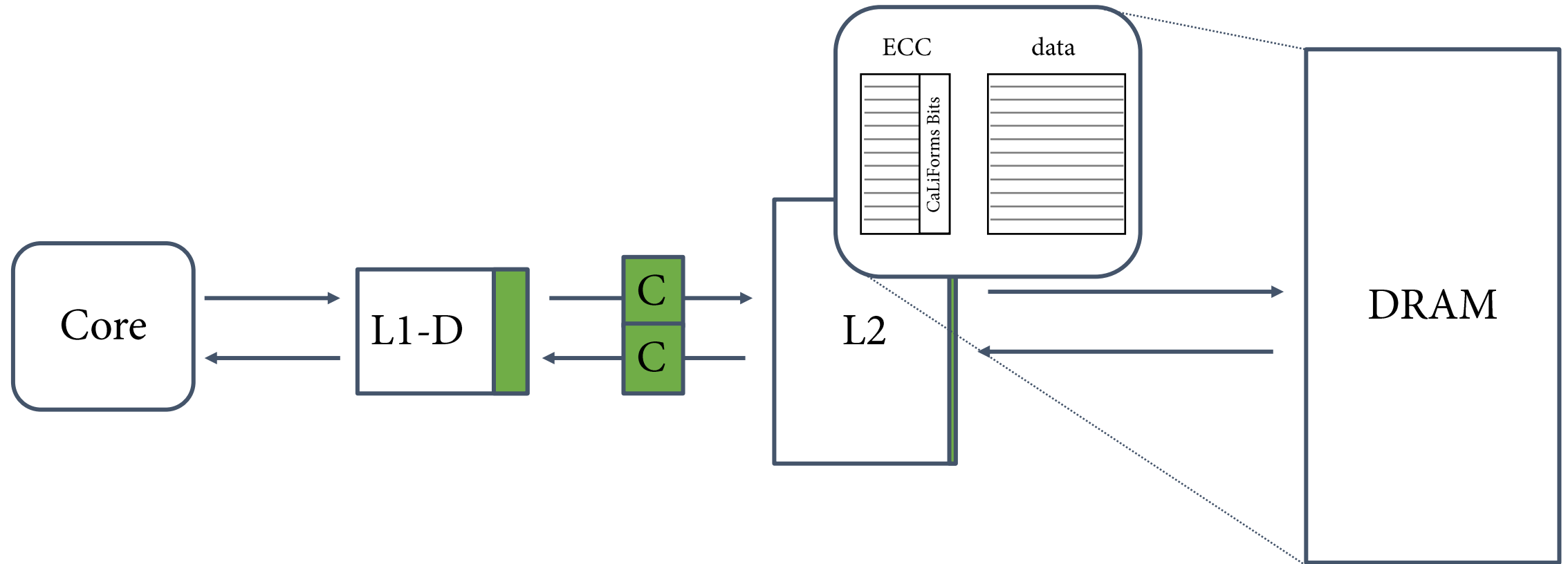
CaLiForms Microarchitectural Overview



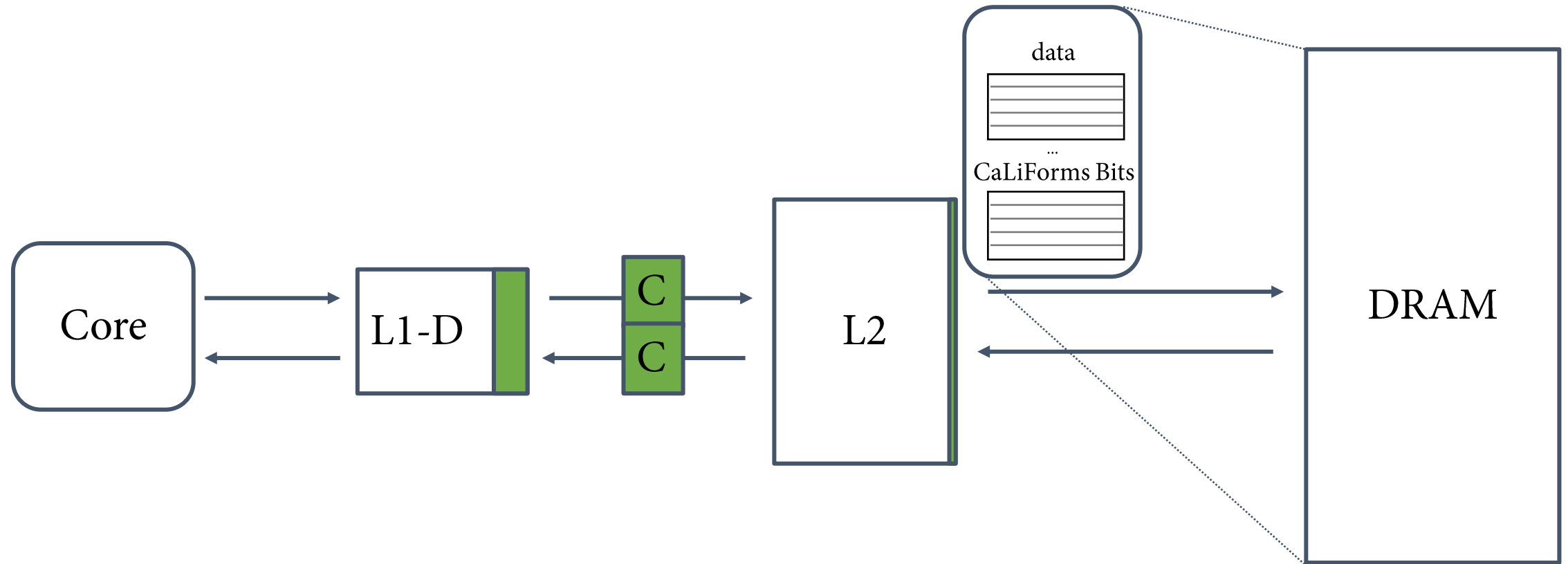
CaLiForms Microarchitectural Overview



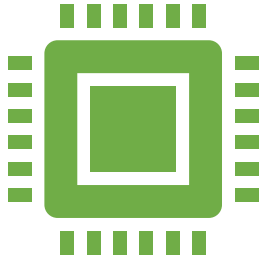
CaLiForms Microarchitectural Overview



CaLiForms Microarchitectural Overview



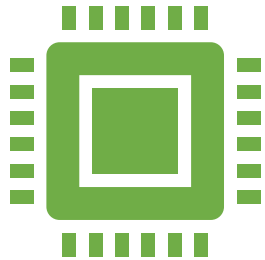
CaLiForms Performance Overheads



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.

00010010
101001101
00010010
111001001
00010010

Software Modifications

- We evaluate three different insertion policies using Clang/LLVM.

CaLiForms Insertion Polices

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

(1) Opportunistic

CaLiForms Insertion Polices

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

(1) Opportunistic

```
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

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

(1) Opportunistic

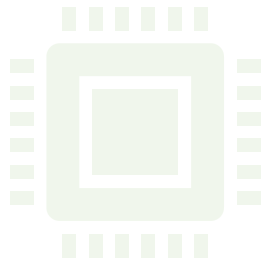
```
struct A_full {
    char tripwire[2];
    char c;
    char tripwire[1];
    int i;
    char tripwire[3];
    char buf[64];
    char tripwire[2];
    void (*fp)();
    char tripwire[1];
}
```

(2) Full

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

(3) Intelligent

CaLiForms Performance Overheads



Hardware Modifications

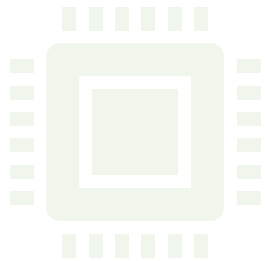
Our measurements show no impact on the cache access latency.

00010010
101001101
00010010
111001001
00010010

Software Modifications

- We evaluate three different insertion policies using Clang/LLVM.

CaLiForms Performance Overheads



Hardware Modifications

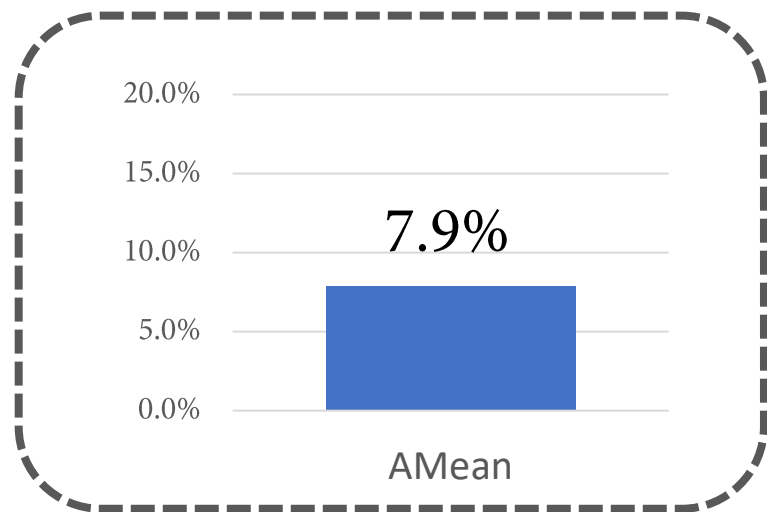
Our measurements show no impact on the cache access latency.

00010010
101001101
00010010
111001001
00010010

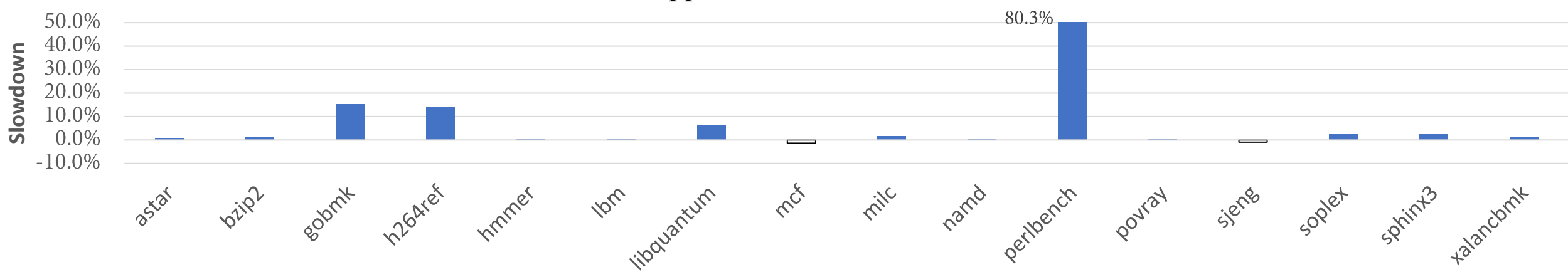
Software Modifications

- 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.

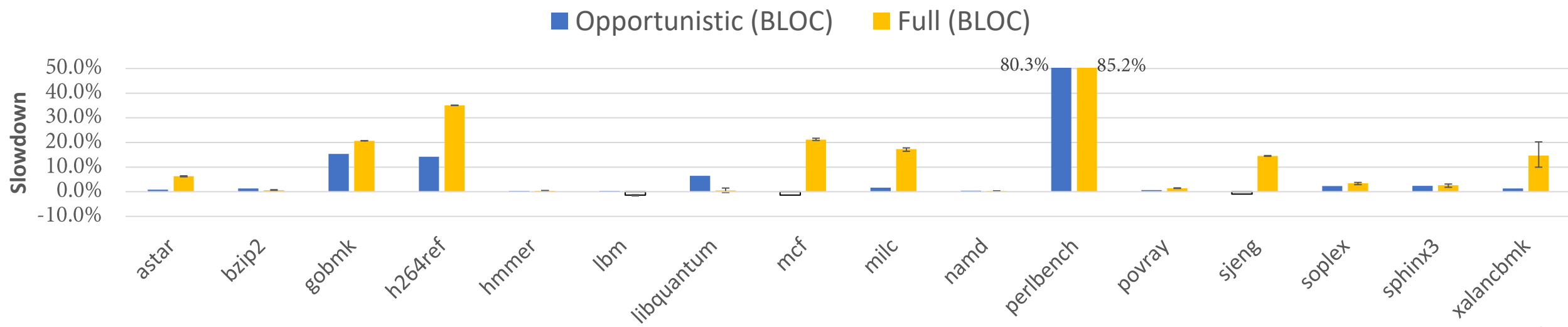
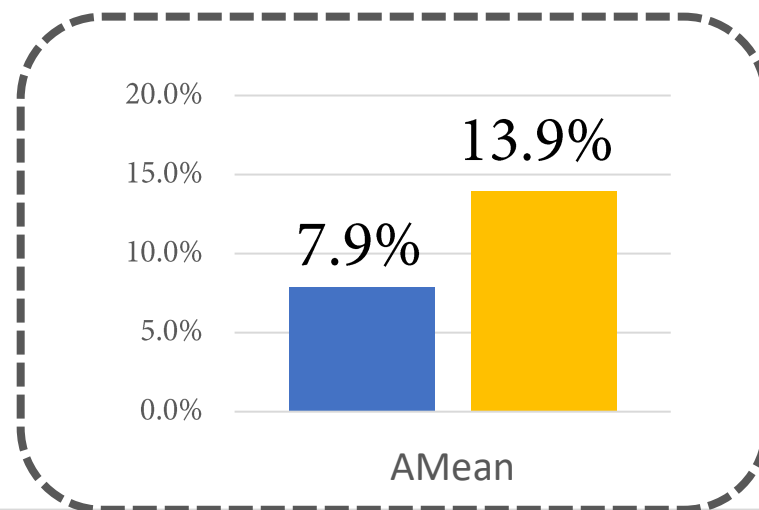
CaLiForms Performance Results (x86_64)



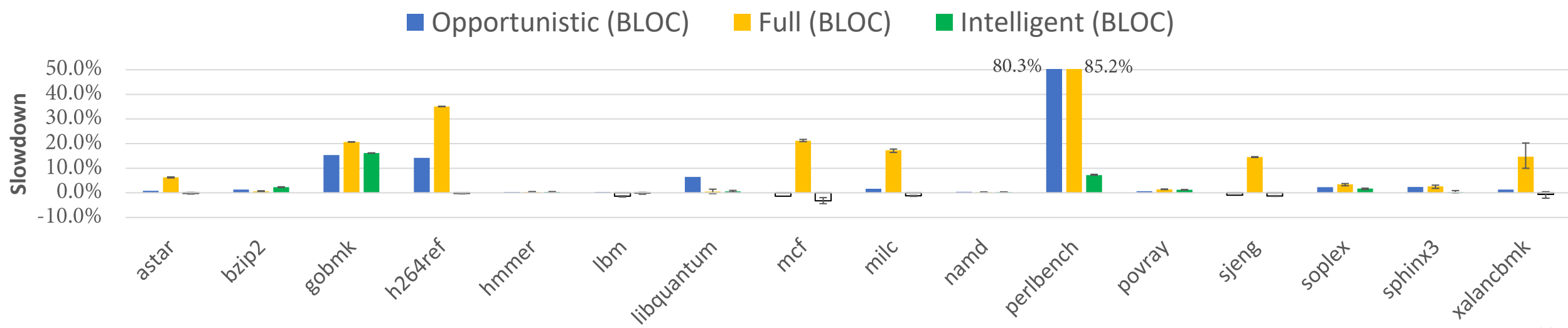
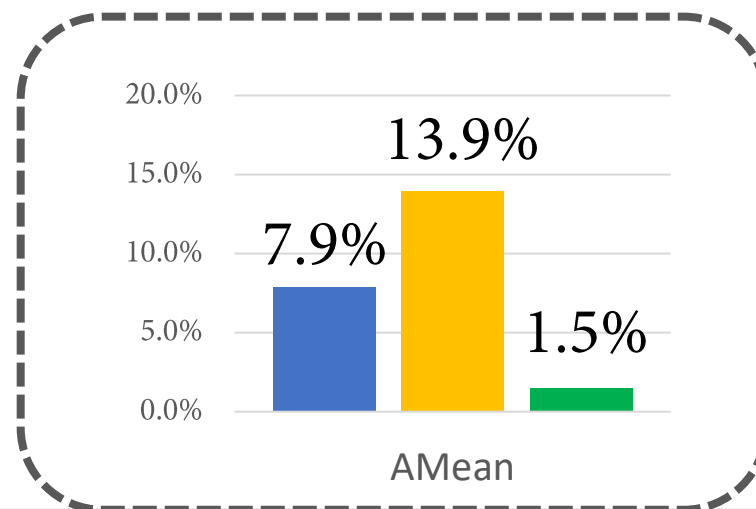
Opportunistic (BLOC)



CaLiForms Performance Results (x86_64)



CaLiForms Performance Results (x86_64)



CaLiForms Performance Overheads

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

(1) Opportunistic

```
struct A_full {  
    char tripwire[2];  
    char c;
```

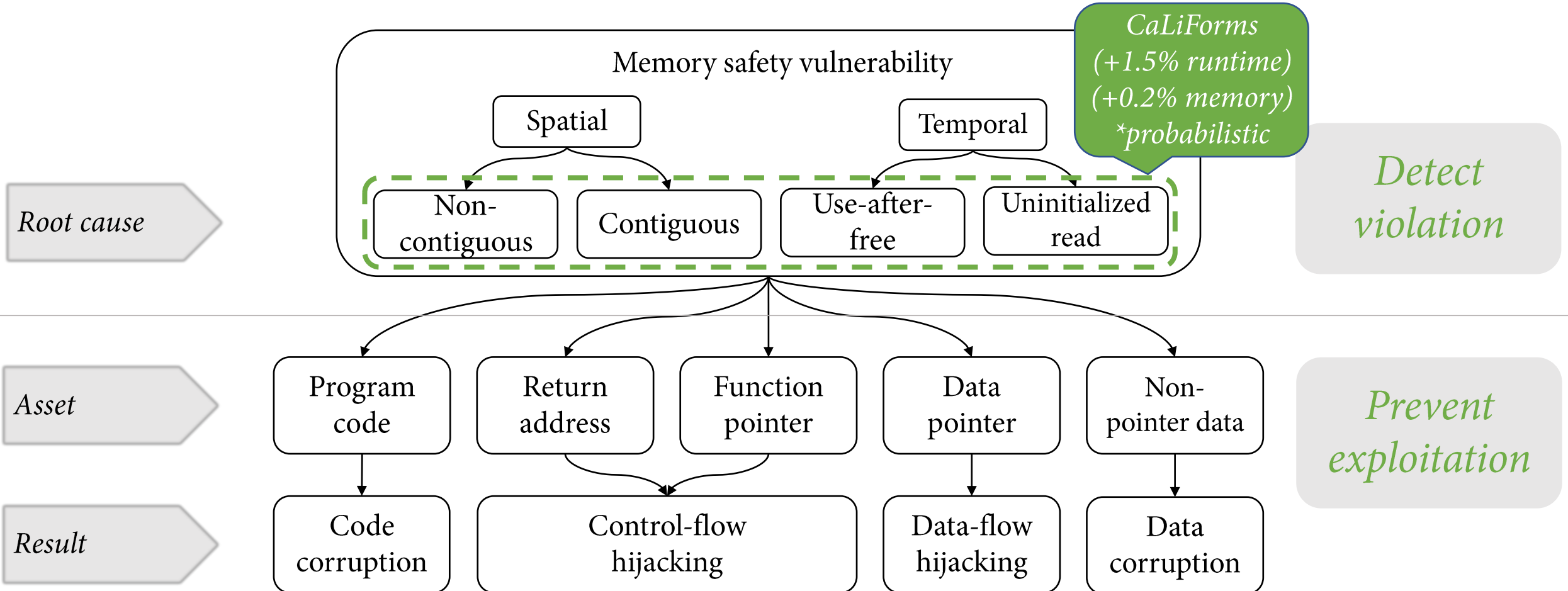
The *intelligent* policy provides the best performance-security tradeoff.

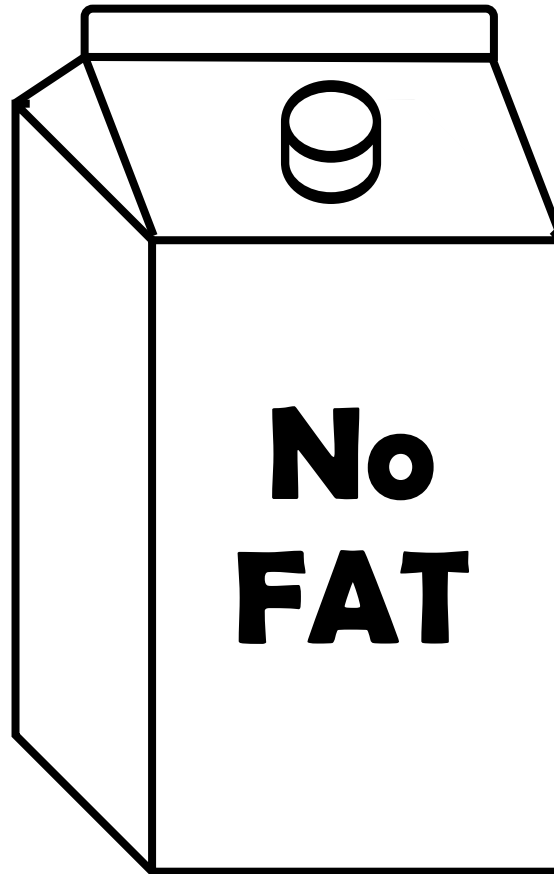
(2) Full

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

(3) Intelligent

Memory Attacks Taxonomy





Mohamed Tarek Ibn Ziad, Miguel A. Arroyo Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan, Architectural Support for Low Overhead Memory Safety Checks. [[ISCA 2021](#)]
[IEEE Top Picks in Hardware and Embedded Security 2022 Candidate]



No-FAT: Key Observation

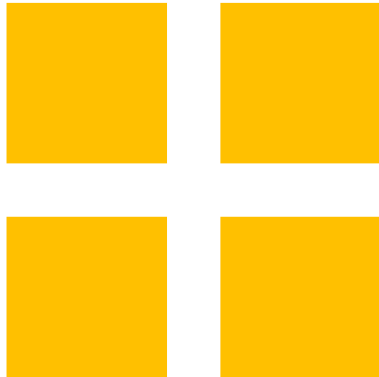


Current software trends can be used to enhance systems security

No-FAT: Key Observation



Current software trends can be used to enhance systems security

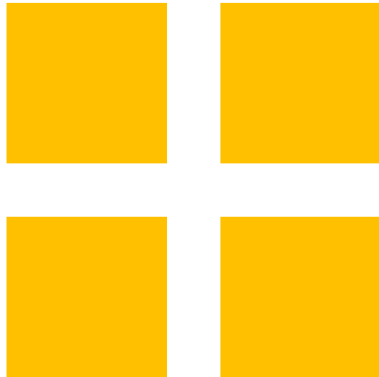


Increasing adoption of binning allocators

No-FAT: Key Observation



Current software trends can be used to enhance systems security



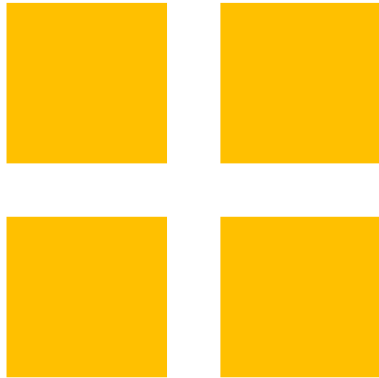
Increasing adoption of binning allocators

- 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.



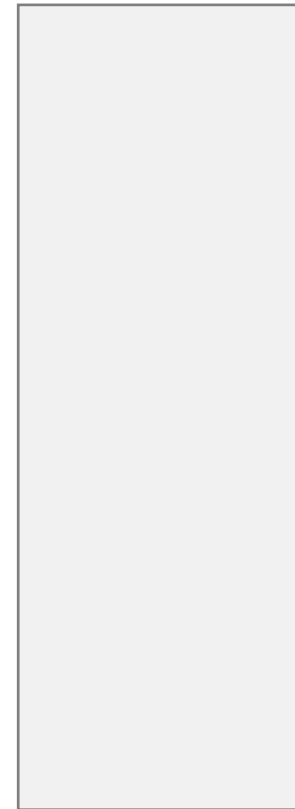
mi-malloc



tcMalloc

Binning Memory Allocators

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

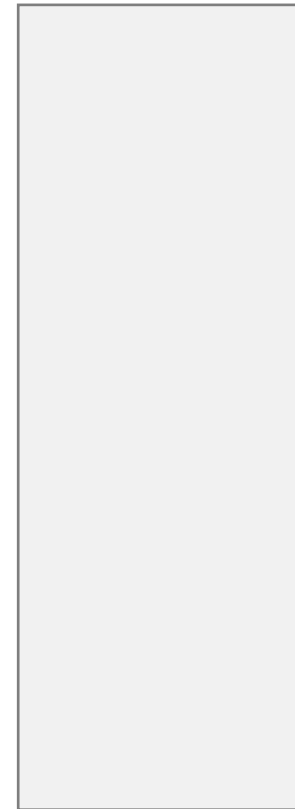


...

Virtual Memory

Binning Memory Allocators

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

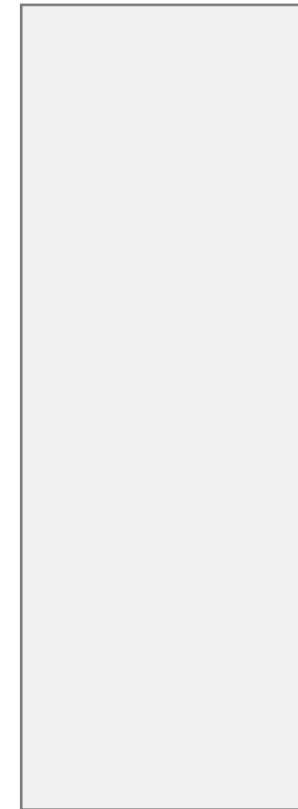


...
Virtual Memory

Binning Memory Allocators

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

Memory is requested by the allocator.

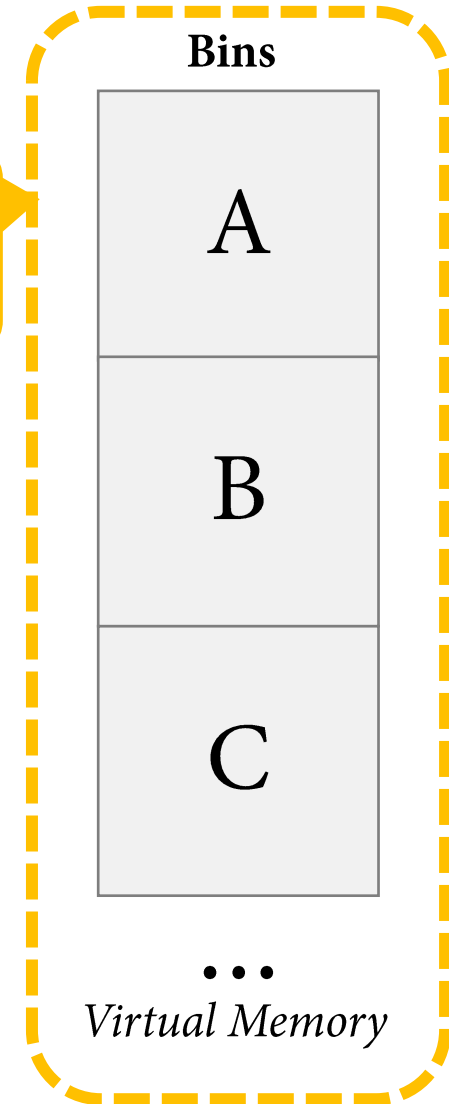


...
Virtual Memory

Binning Memory Allocators

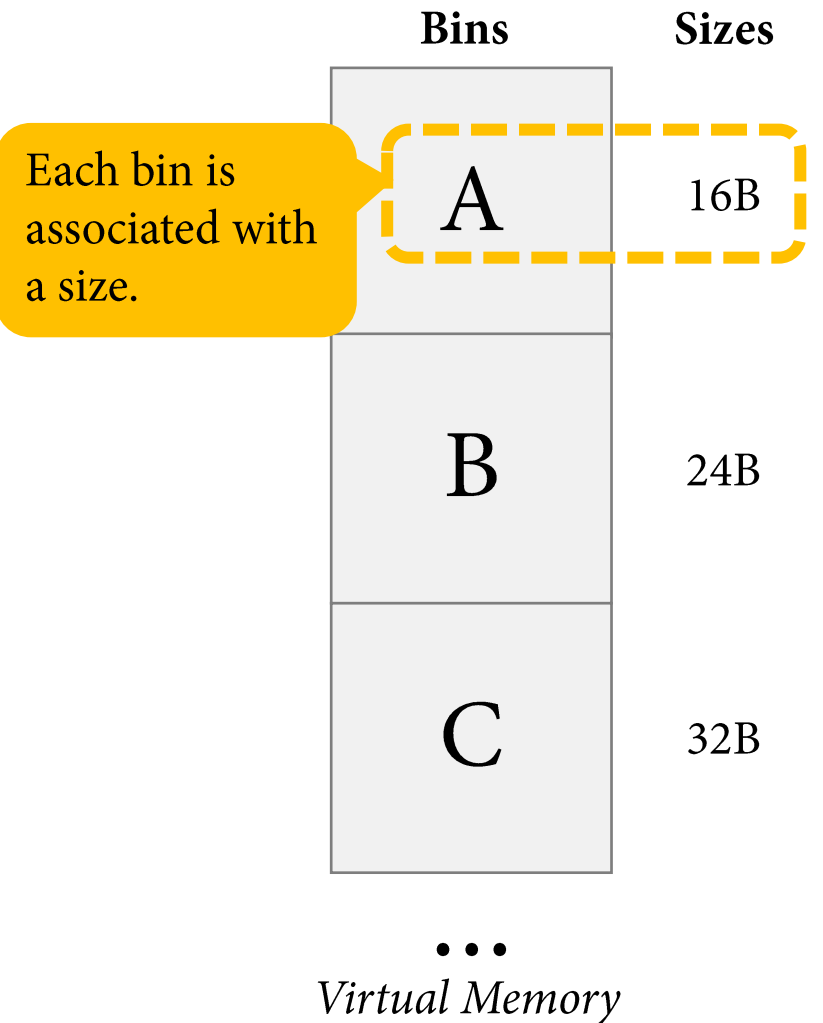
```
40. int main() {  
41.     char* ptr = malloc(12);  
42.     ...  
50. }
```

Memory is
divided into
bins.



Binning Memory Allocators

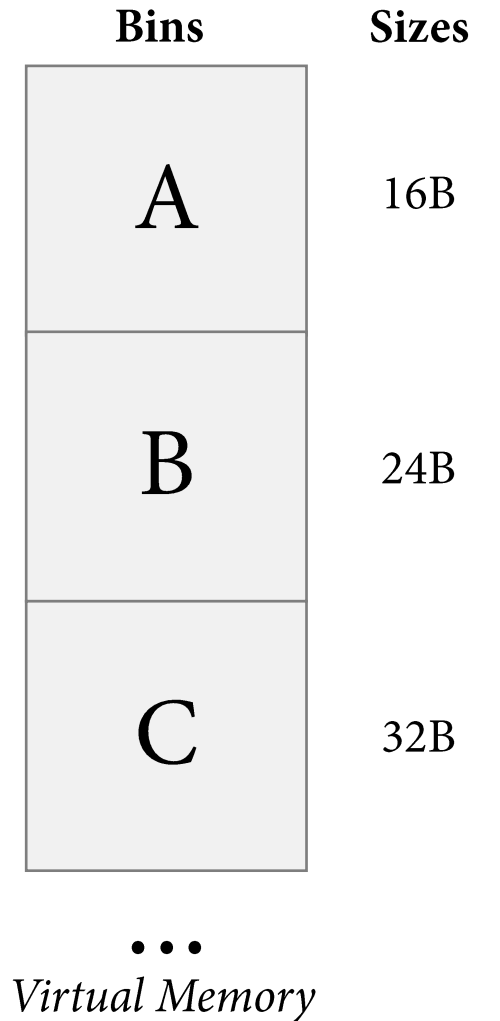
```
40. int main() {  
41.     char* ptr = malloc(12);  
42.     ...  
50. }
```



Binning Memory Allocators

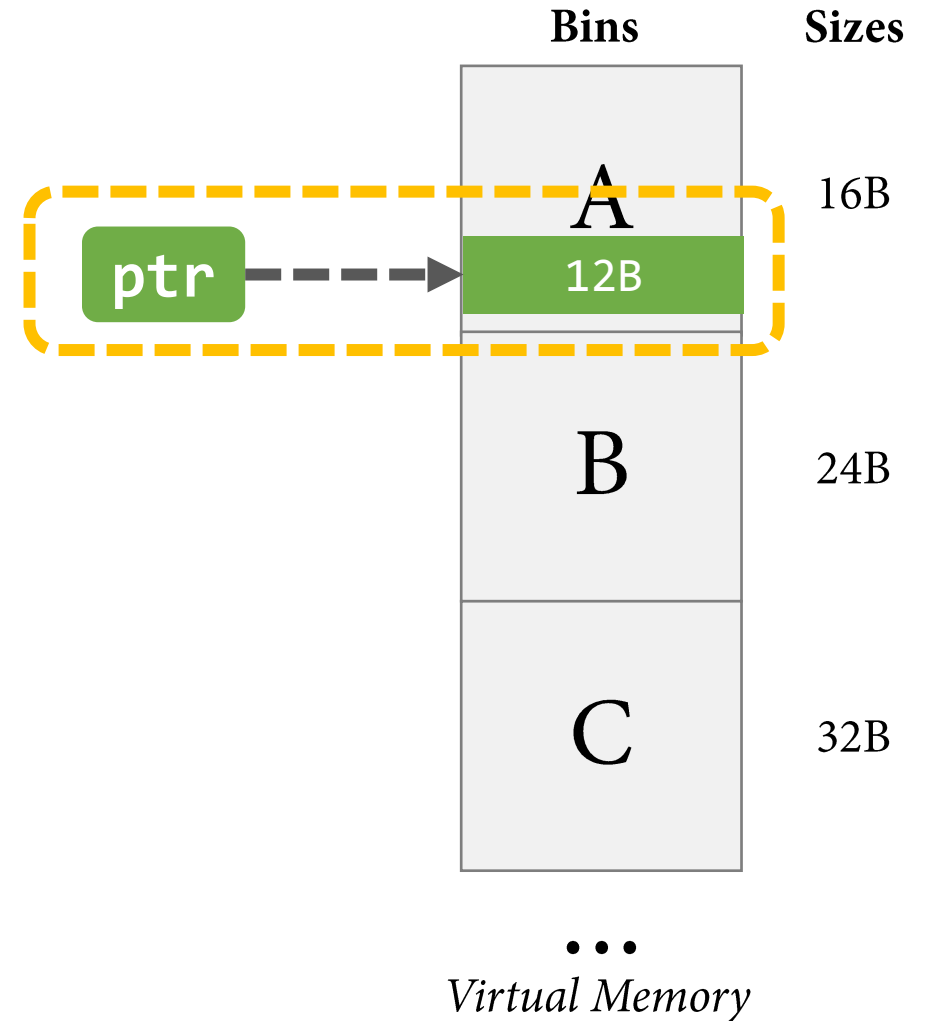


```
40. int main() {  
41.     char* ptr = 12B  
42.     ...  
50. }
```



Binning Memory Allocators

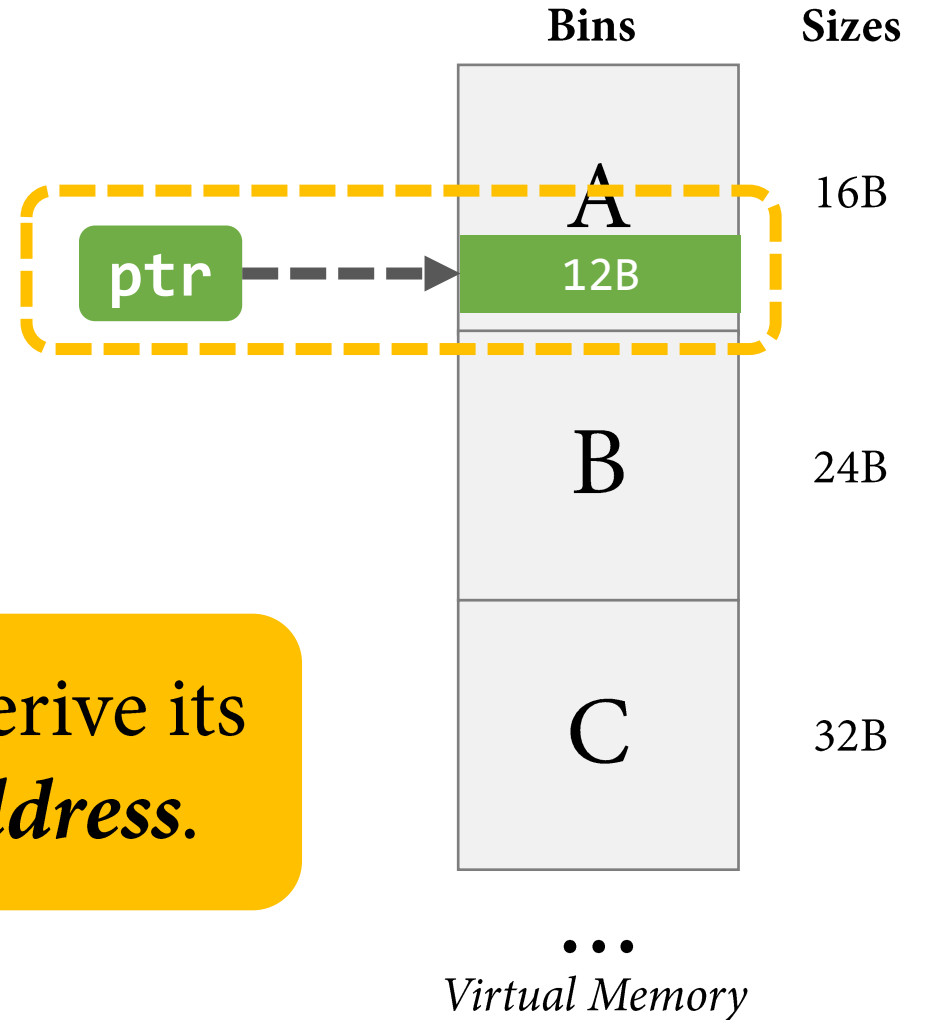
```
40. int main() {  
41.     char* ptr = malloc(12);  
42.     ...  
50. }
```



Binning Memory Allocators



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



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

How No-FAT Provides Memory Safety

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


How No-FAT Provides Memory Safety

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

How No-FAT Provides Memory Safety

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

We add one extra operand for loads/stores.

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



The compiler propagates the allocation base address.

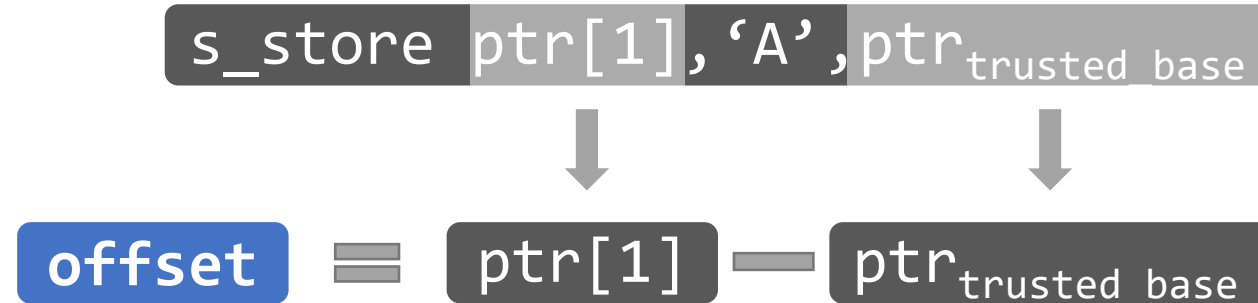
How No-FAT Provides Memory Safety

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

How No-FAT Provides Memory Safety

```
s_store ptr[1], 'A', ptr_trusted base
```

How No-FAT Provides Memory Safety



How No-FAT Provides Memory Safety

```
s_store ptr[1], 'A', ptr_trusted base
```

```
offset = ptr[1] - ptr_trusted base
```

```
size = getSize( ptr_trusted base )
```

How No-FAT Provides Memory Safety

```
s_store ptr[1], 'A', ptr_trusted base
```

offset = `ptr[1] - ptr_trusted base`

size = `getSize(ptr_trusted base)`

**Bounds
Check**

offset < **size** ?

How No-FAT Provides Memory Safety

```
s_store ptr[1], 'A', ptr_trusted_base
```

offset = `ptr[1] - ptr_trusted_base`

size = `getSize(ptr_trusted_base)`

**Bounds
Check**

offset < **size** ?

**Temporal
Check**

`ptr[1]` [63:48] = `ptr_trusted_base` [63:48] ?

How No-FAT Provides Memory Safety

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

size = getSize(ptr_{trusted base})

Bounds
Check

offset < size ?

Temporal
Check

ptr[1] [63:48] = ptr_{trusted_base} [63:48] ?

How No-FAT Provides Memory Safety

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

`ptrtrusted_base`

`s_store ptr[1], 'A', ptrtrusted_base`

How No-FAT Provides Memory Safety

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

`ptrtrusted_base`
`s_store ptr[1], 'A', ptrtrusted_base`


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

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. }
```

`ptrtrusted_base`

`s_store ptr[1], 'A', ptrtrusted_base`

An orange arrow originates from the parameter `xptr` in the `Foo` function signature (line 51) and points to the `ptr` variable in the `main` function (line 41), illustrating that `xptr` is a pointer to the memory managed by `ptr`.

How No-FAT Provides Memory Safety

```
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], 'B', xptr_trusted_base  
54.     ...  
60. }
```

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. }
```

`ptrtrusted_base`

`s_store ptr[1], 'A', ptrtrusted_base`

`s_store xptr[7], 'B', xptrtrusted_base`

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. }
```

`ptrtrusted_base`

`s_store ptr[1], 'A', ptrtrusted_base`

`xptrtrusted_base ← compBase(xptr[7])`

`s_store xptr[7], 'B', xptrtrusted_base`

How No-FAT Provides Memory Safety

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

How No-FAT Provides Memory Safety

`xptr`_{trusted base} ← `compBase(xptr[7])`

`Bin` = `xptr` >> `log2(S)` where S is the size of the bins.

How No-FAT Provides Memory Safety

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

Bin = `xptr` >> `log2(S)` where S is the size of the bins.

size = `getSize(Bin)`

How No-FAT Provides Memory Safety

```
xptr_trusted_base ← compBase(xptr[7])
```

Bin = `xptr` >> `log2(S)` where S is the size of the bins.

size = `getSize(Bin)`

`xptr_trusted_base` = `[xptr × (1/size)] × size`

How No-FAT Provides Memory Safety

```
xptrtrusted_base ← compBase(xptr[7])
```

Bin = `xptr` >> `log2(S)` where S is the size of the bins.

size = `getSize(Bin)`

`xptrtrusted_base` = `[xptr × (1 / size)] × size`

Base pointer is **implicitly** derived!

How No-FAT Provides Memory Safety

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

`ptrtrusted_base`

`s_store ptr[1], 'A', ptrtrusted_base`

How No-FAT Provides Memory Safety

```
40. int main() {  
41.     char* ptr = malloc(12);  
42.     ptr[1] = 'A';  
43.     ptr = ptr + 100;  
44.     ...  
49.     foo(ptr);  
50. }
```

`ptrtrusted_base`

`s_store ptr[1], 'A', ptrtrusted_base`

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

How No-FAT Provides Memory Safety

```
40. int main() {  
41.     char* ptr = malloc(12);  
42.     ptr[1] = 'A';  
43.     ptr = ptr + 100;  
44.     verifyBounds ptr, ptr_trusted_base  
45.     ...  
49.     foo(ptr);  
50. }
```

ptr_trusted_base

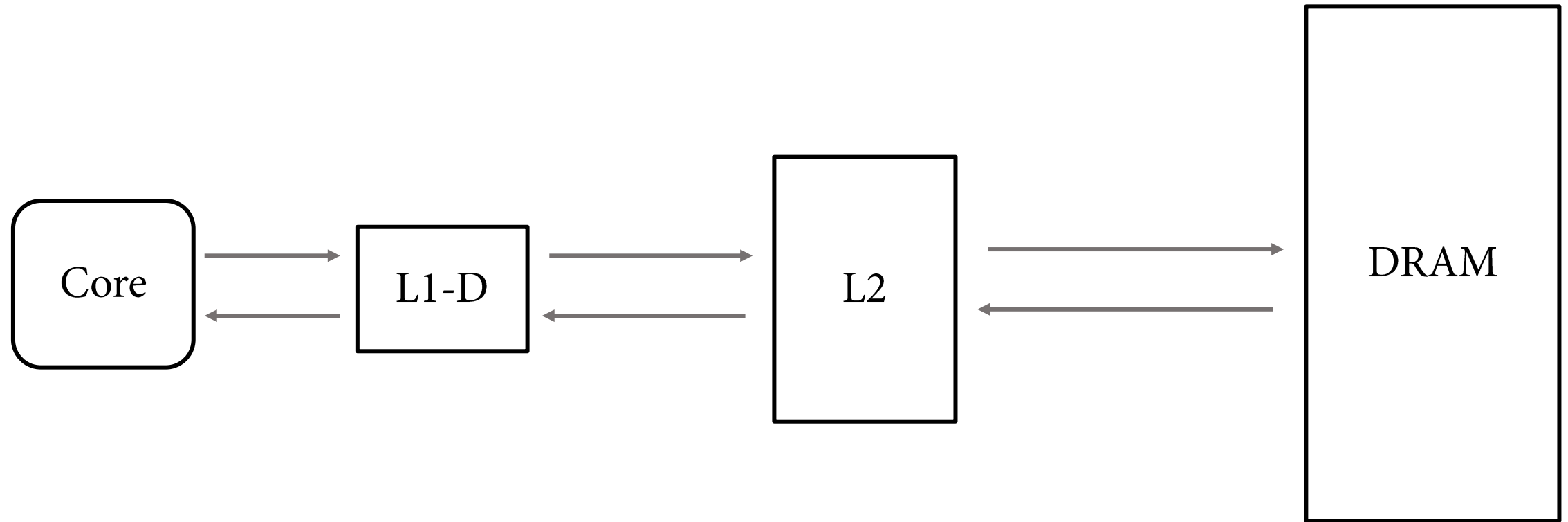
s_store ptr[1], 'A', ptr_trusted_base

verifyBounds ptr, ptr_trusted_base

...

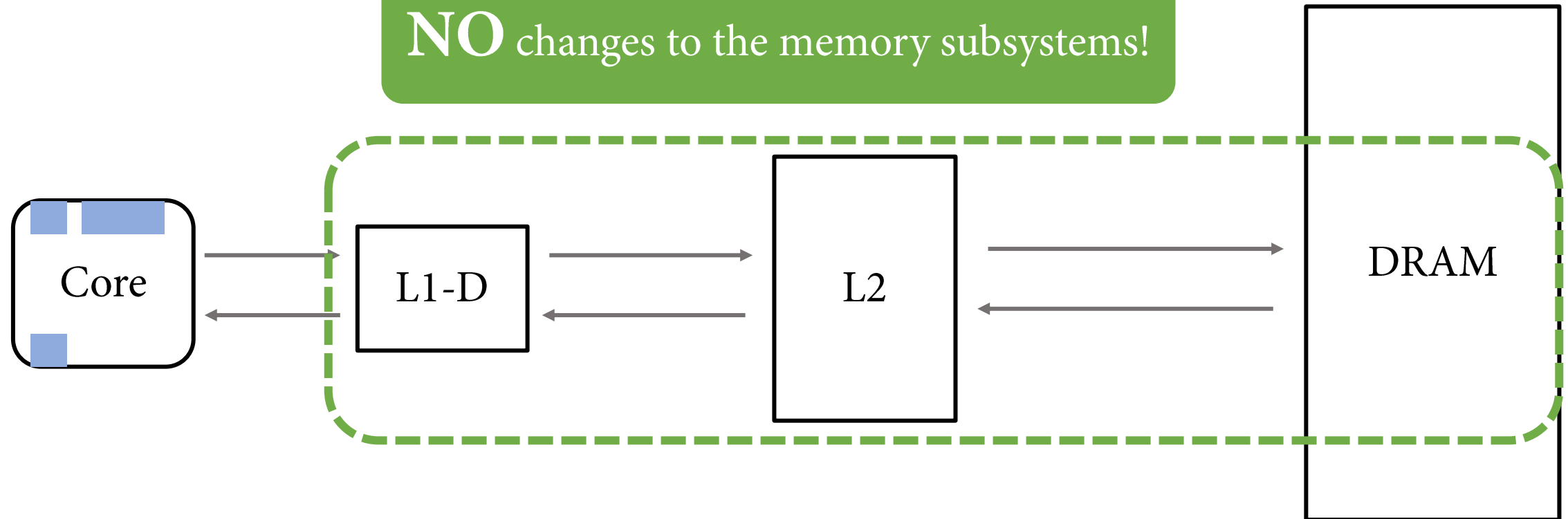
Verify the bounds of all pointers that escape to memory (or another function).

No-FAT Microarchitectural Overview

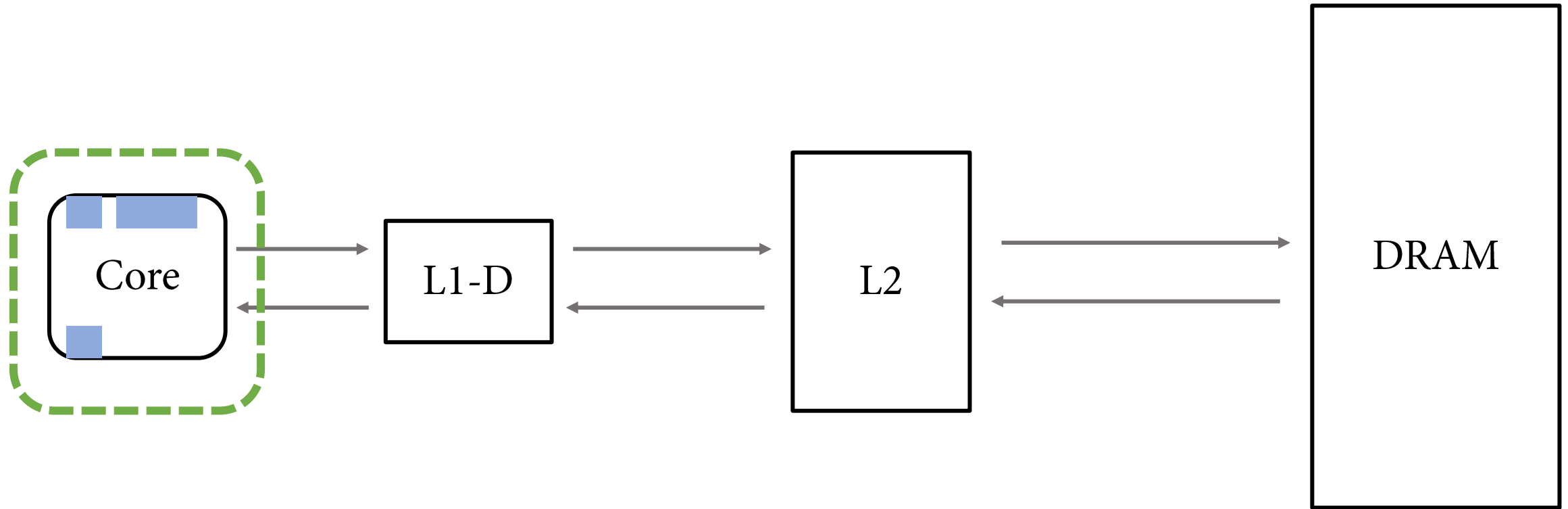


No-FAT Microarchitectural Overview

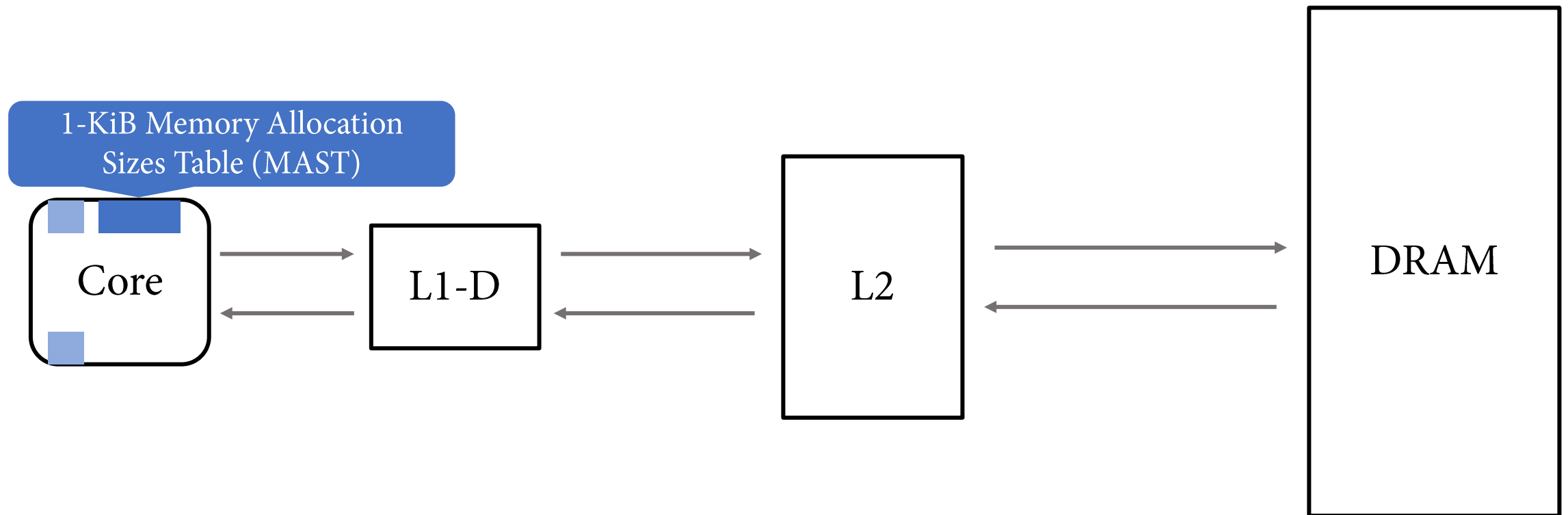
NO changes to the memory subsystems!



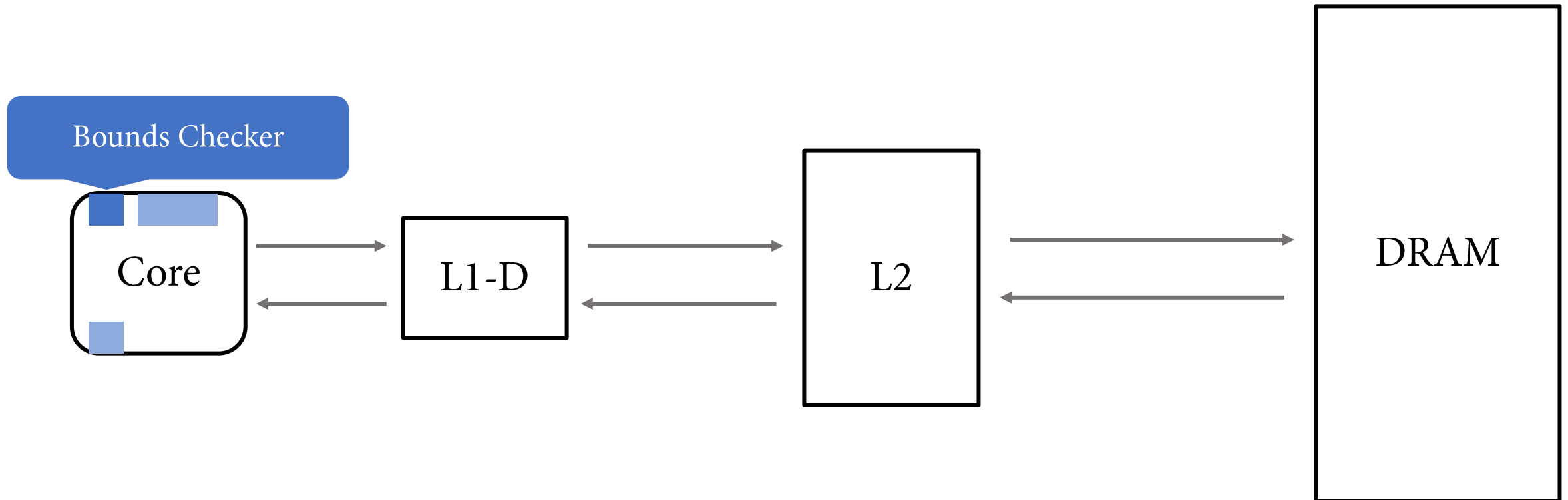
No-FAT Microarchitectural Overview



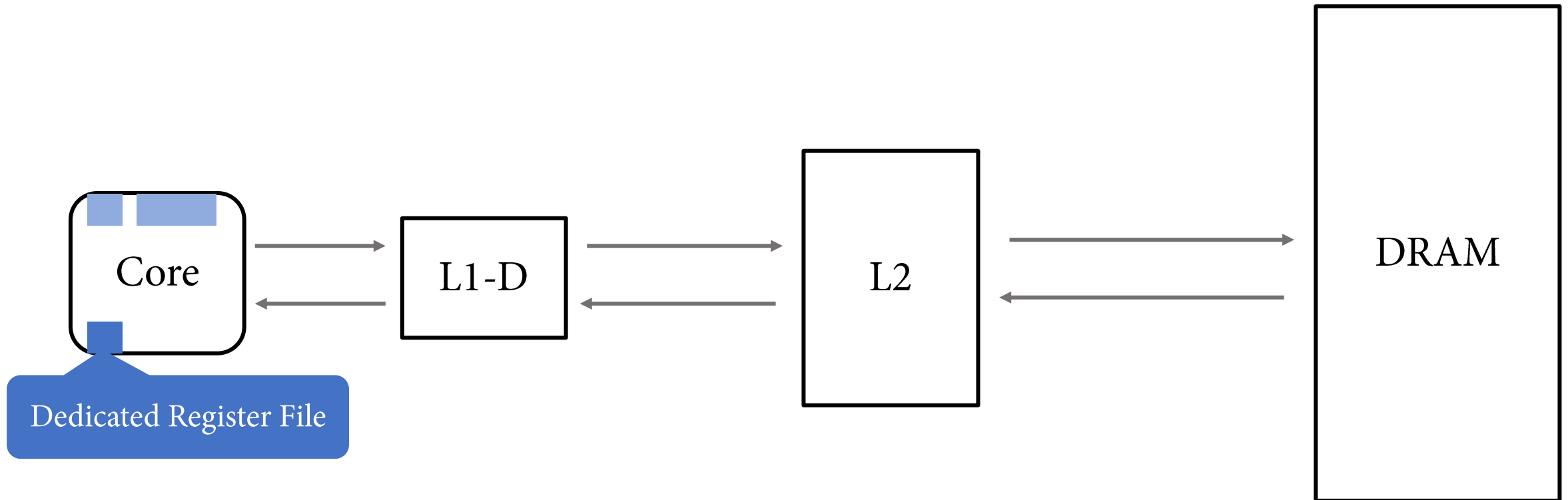
No-FAT Microarchitectural Overview



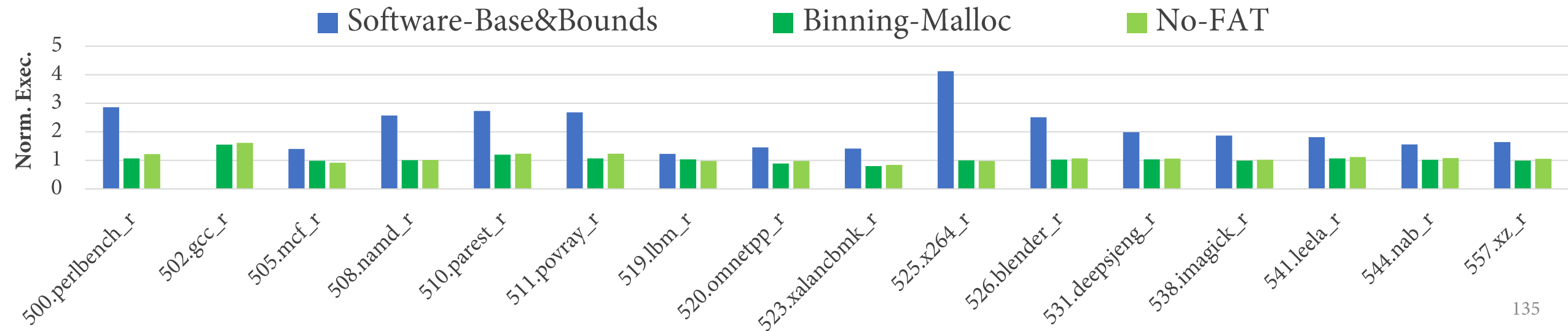
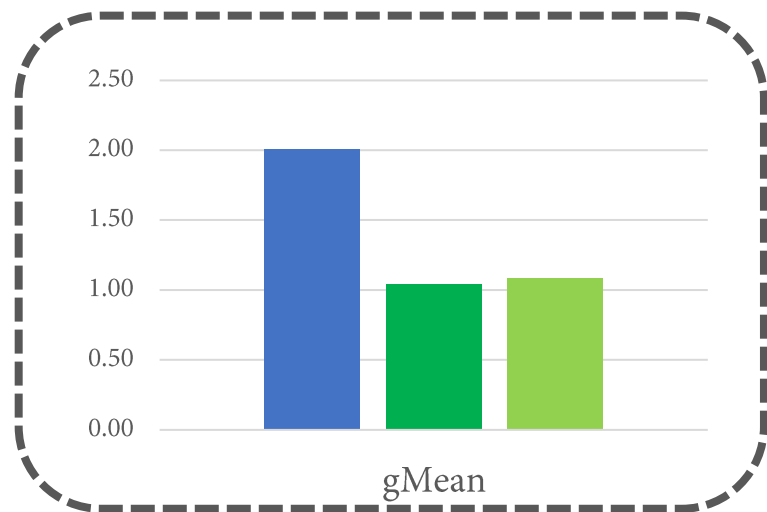
No-FAT Microarchitectural Overview



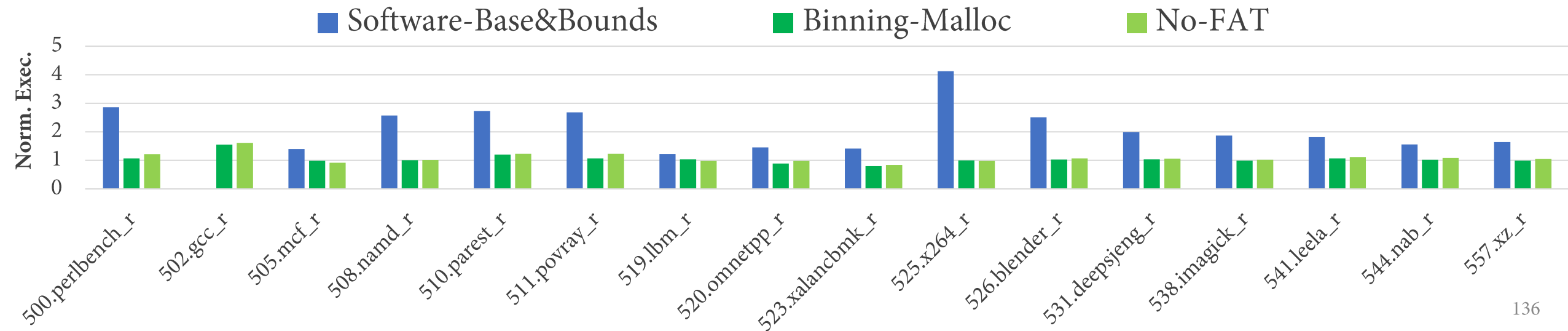
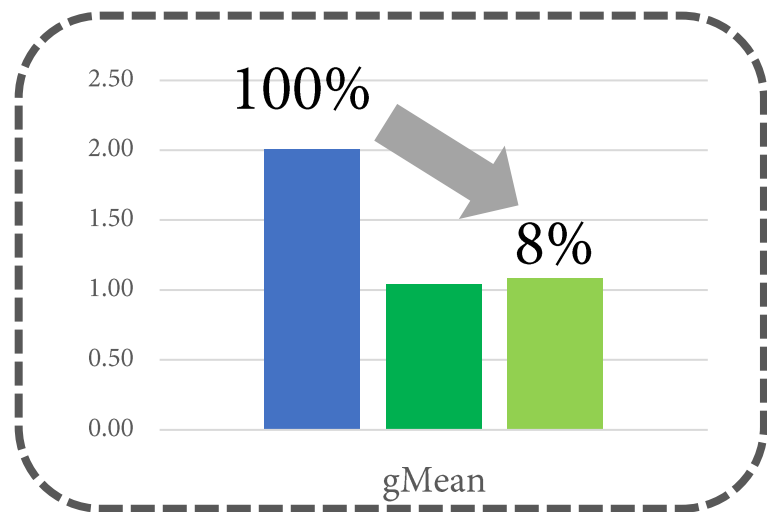
No-FAT Microarchitectural Overview



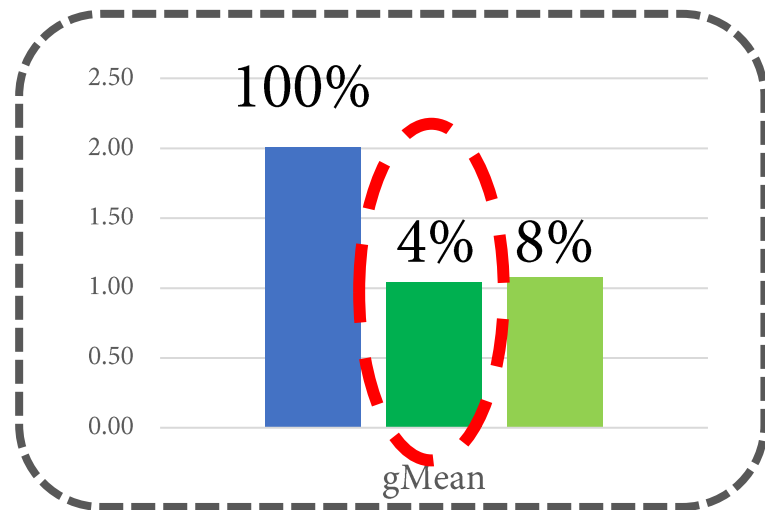
No-FAT Performance Results (x86_64)



No-FAT Performance Results (x86_64)



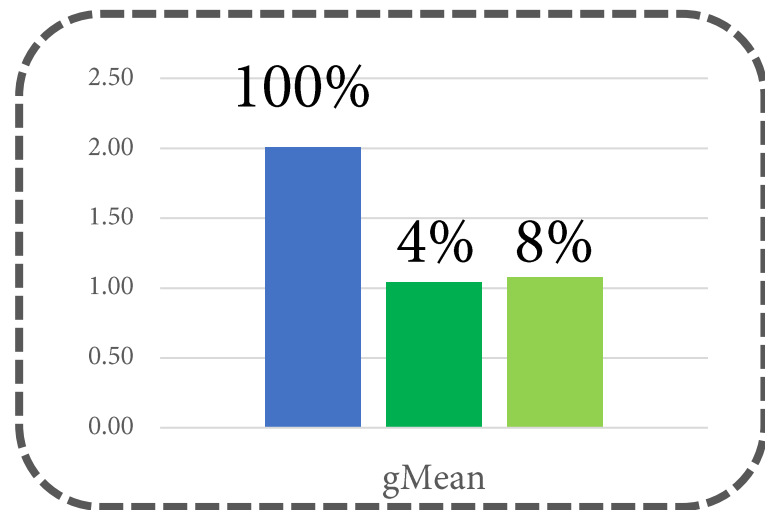
No-FAT Performance Results (x86_64)



Most of No-FAT's overheads are attributed to:

- The binning memory allocator, and

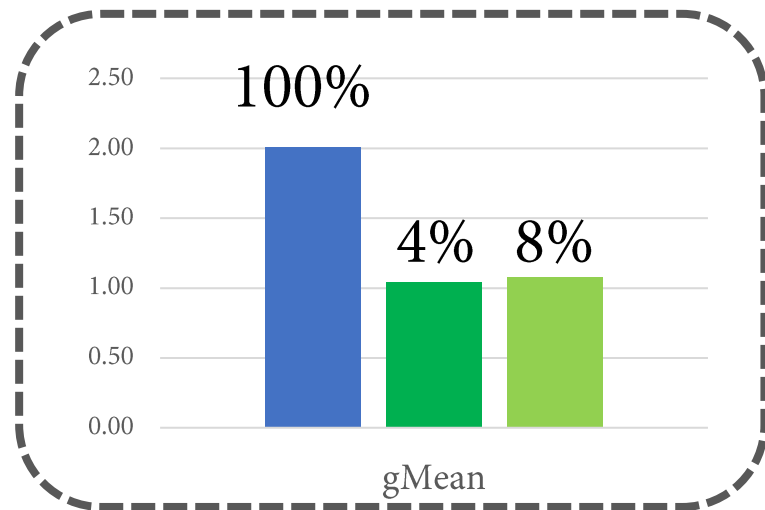
No-FAT Performance Results (x86_64)



Most of No-FAT's overheads are attributed to:

- The binning memory allocator, and
- The back-to-back MULs during base address computation

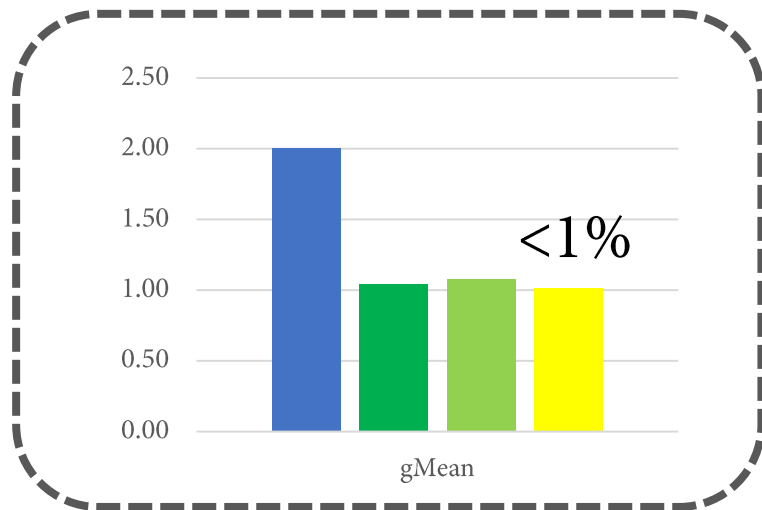
No-FAT Performance Results (x86_64)



Most of No-FAT's overheads are eliminated with:

- A performant binning memory allocator (e.g., MiMalloc), and

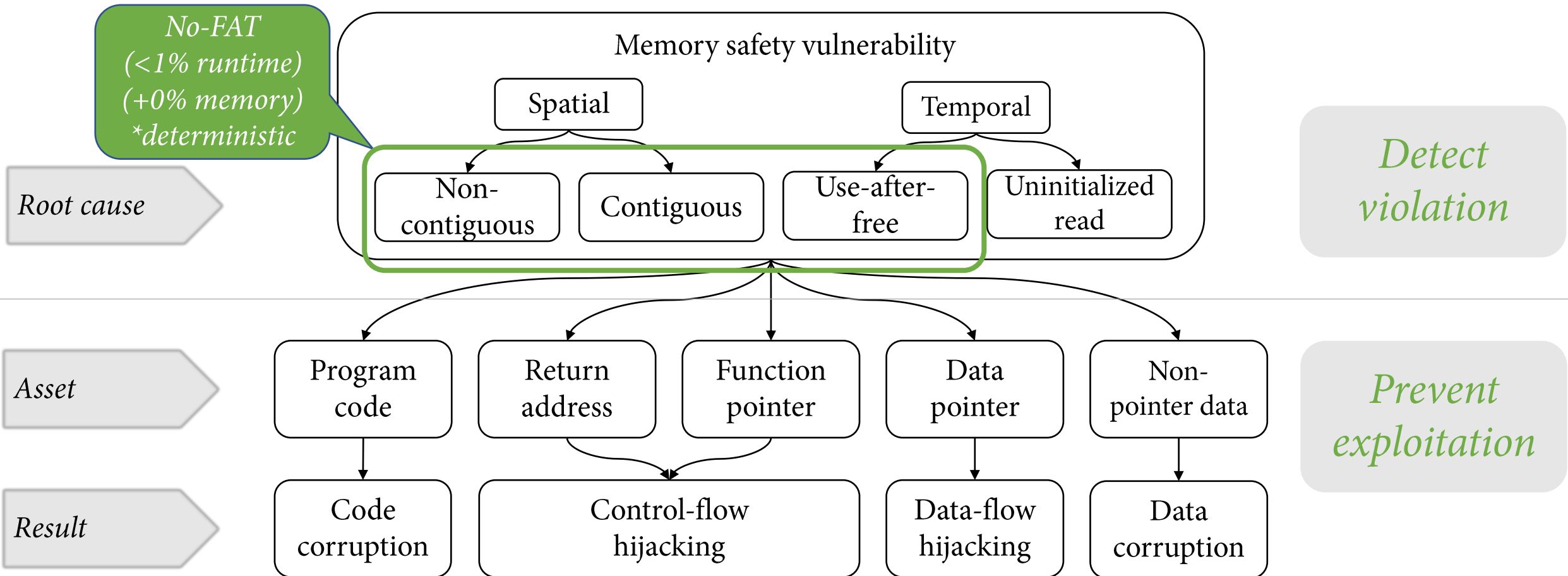
No-FAT Performance Results (x86_64)



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.

Memory Attacks Taxonomy



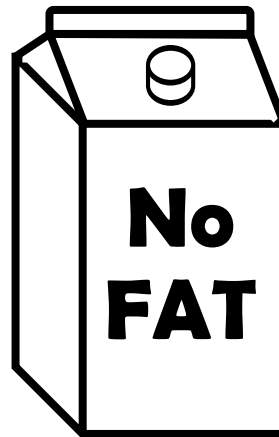
My solutions for C/C++ memory (un)safety

Memory Blocklisting



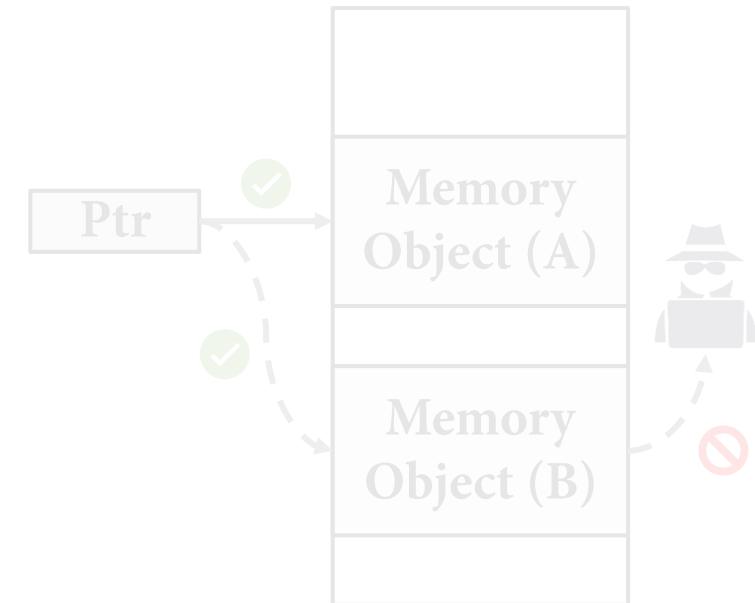
[MICRO 2019]

Memory Permitlisting



[ISCA 2021]

Exploit Mitigation



Comparison with prior work

Comparison with prior work

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

Comparison with prior 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

Comparison with prior 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
CaLiForms	1 bit per cache line	Provides probabilistic guarantees

Comparison with prior 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
CaLiForms	1 bit per cache line	Provides probabilistic guarantees
Explicit Base & Bounds	N bits per pointer or allocation	Breaks compatibility with the rest of the system (eg. unprotected libraries).

Comparison with prior 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
CaLiForms	1 bit per cache line	Provides probabilistic guarantees
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

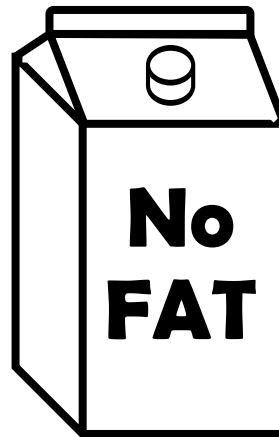
My solutions for C/C++ memory (un)safety

Memory Blocklisting



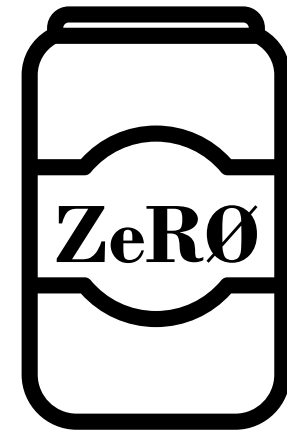
[[MICRO 2019](#)]

Memory Permitlisting

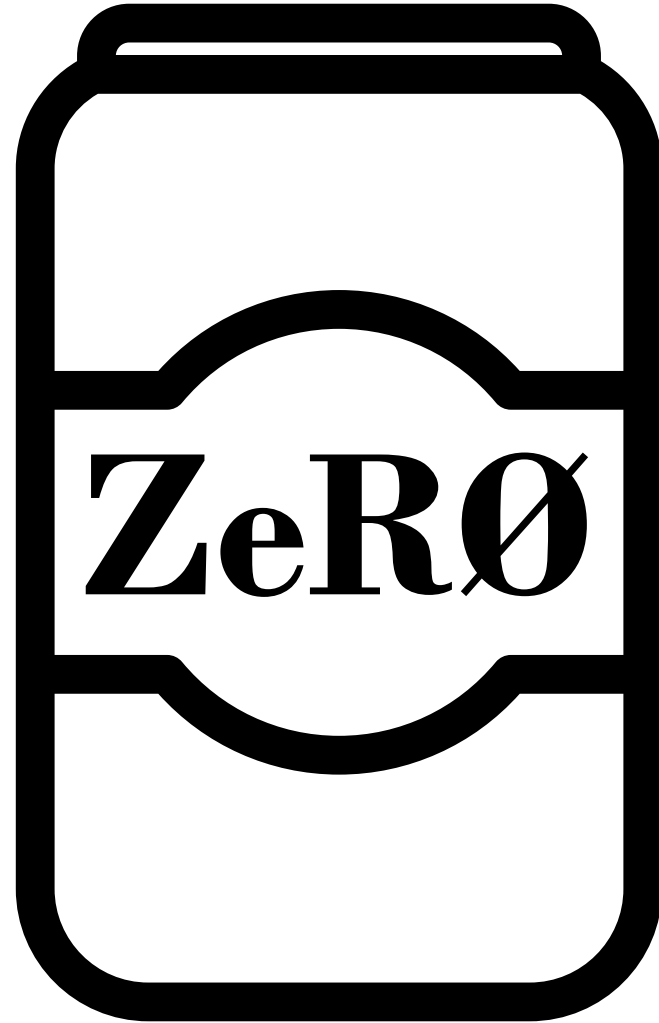


[[ISCA 2021](#)]

Exploit Mitigation



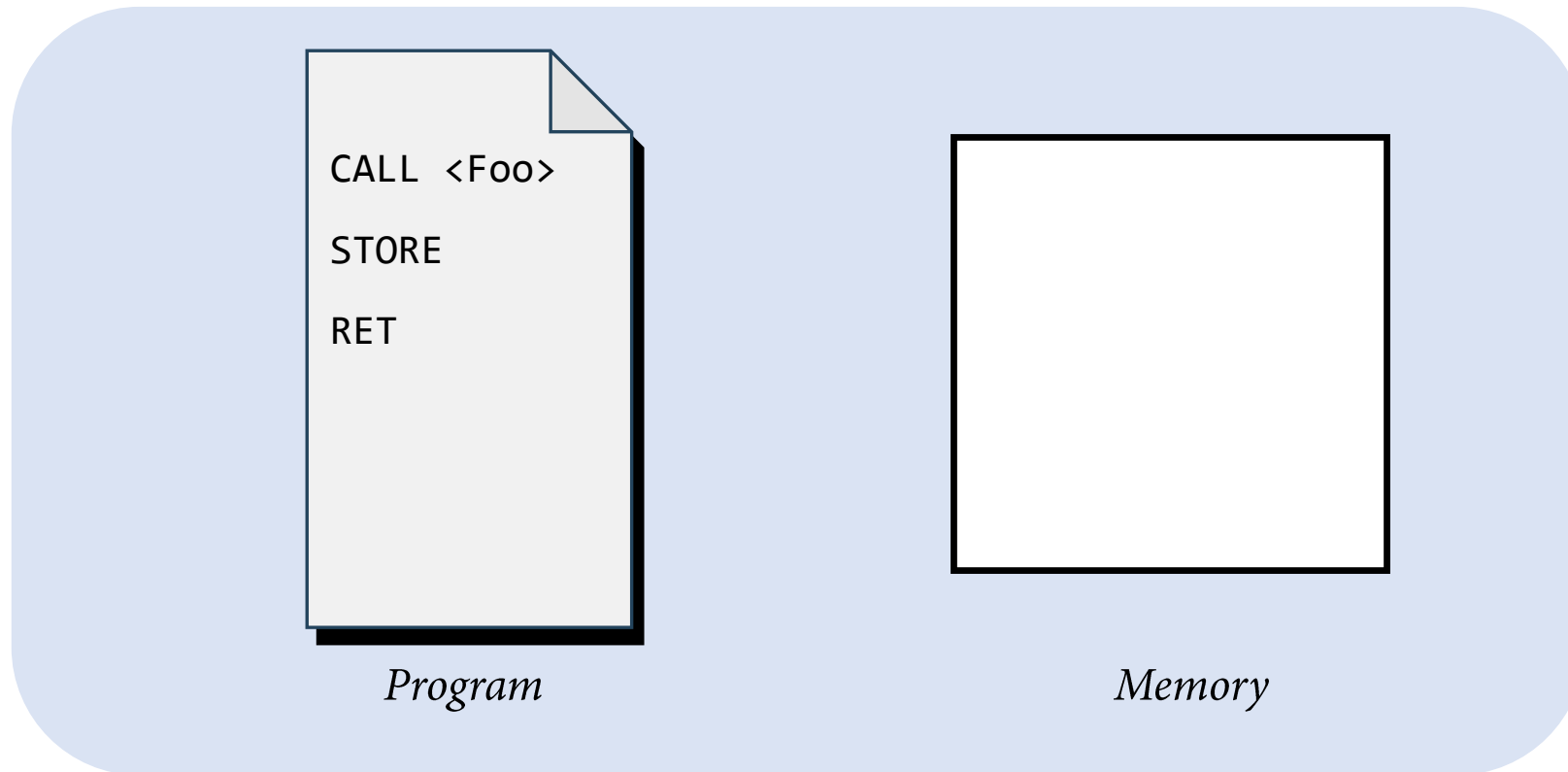
[[ISCA 2021](#)]



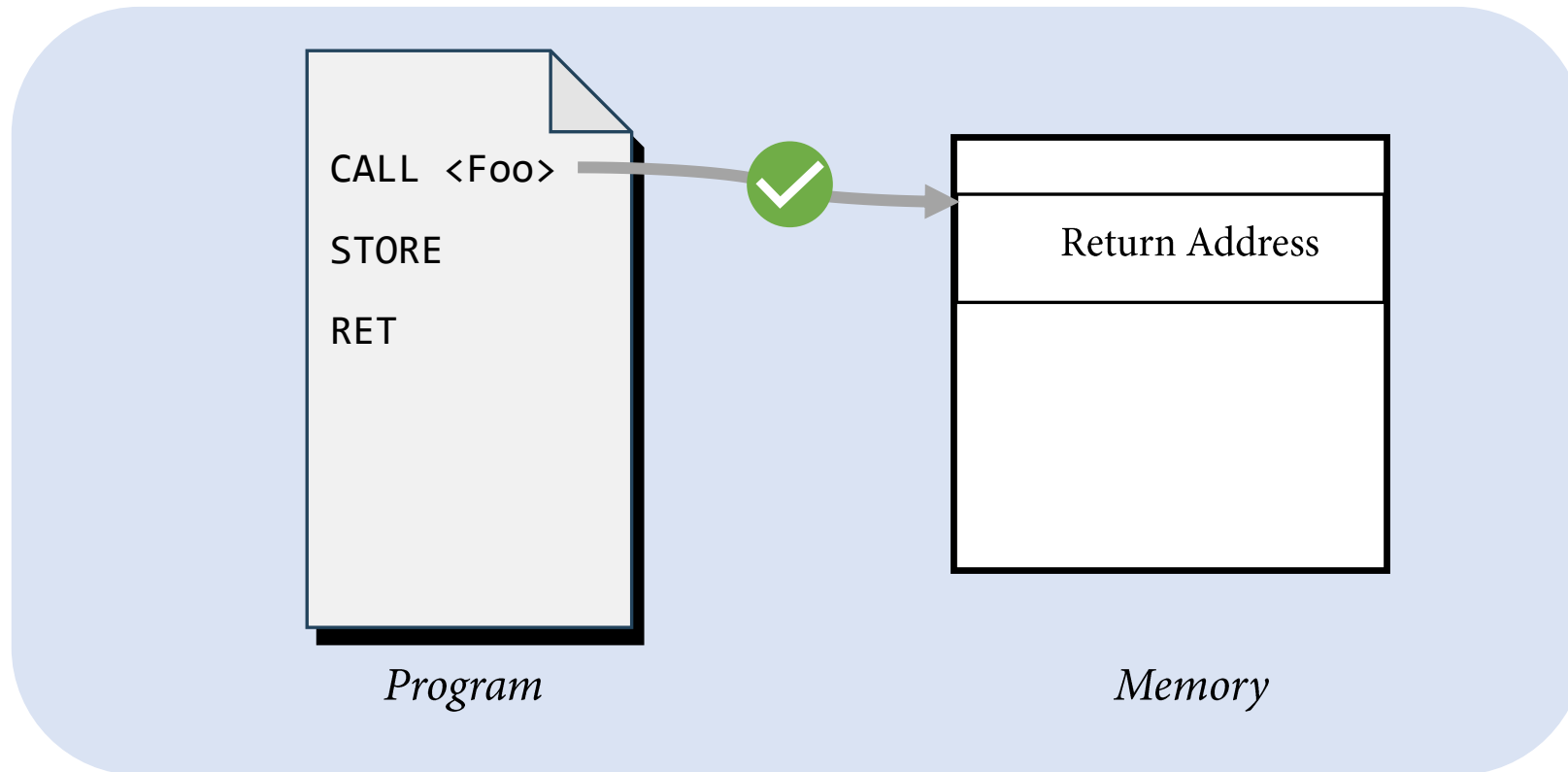
Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan,
ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks. [[ISCA 2021](#)]



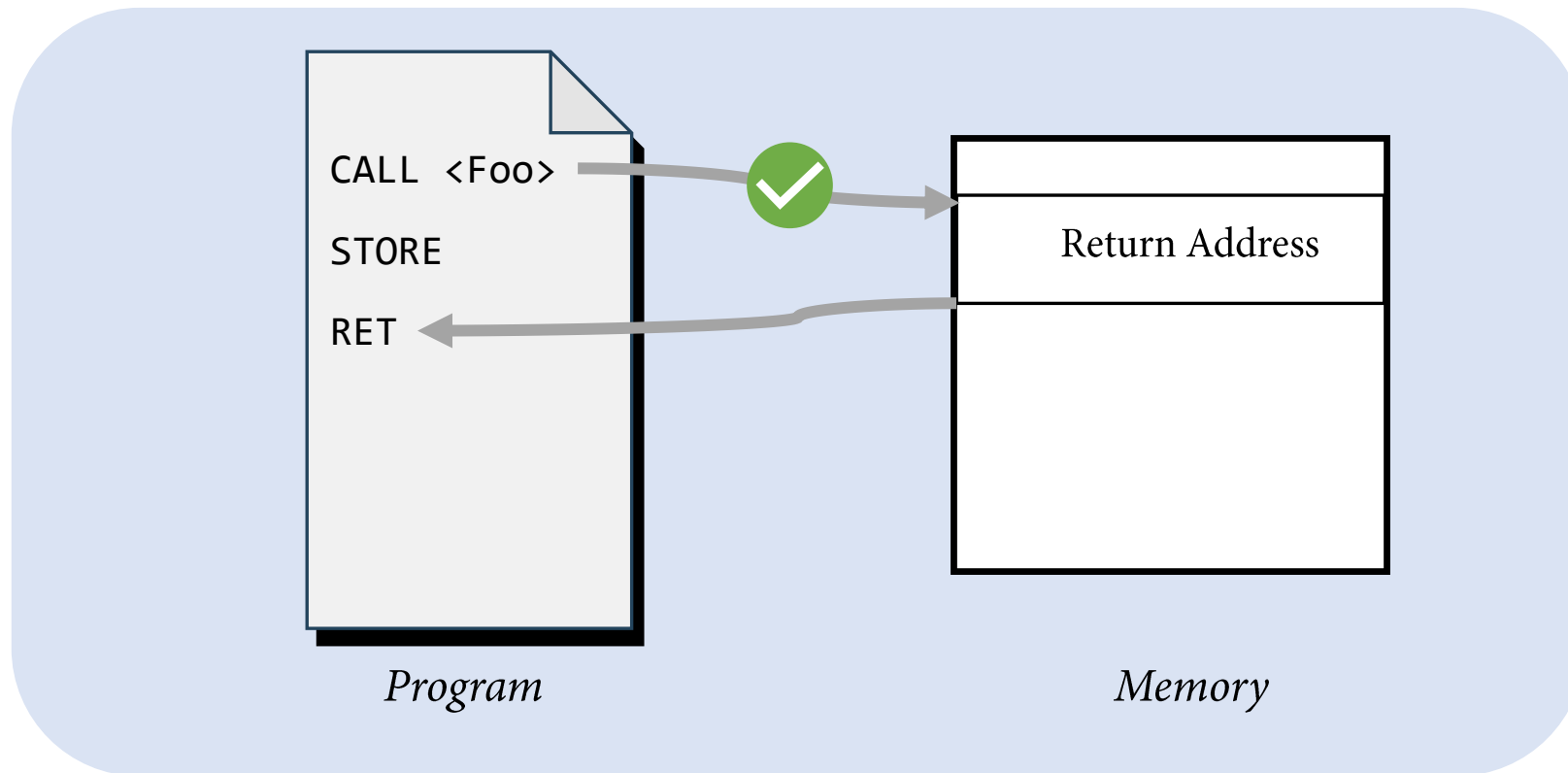
Return Address Protection with ZeRØ



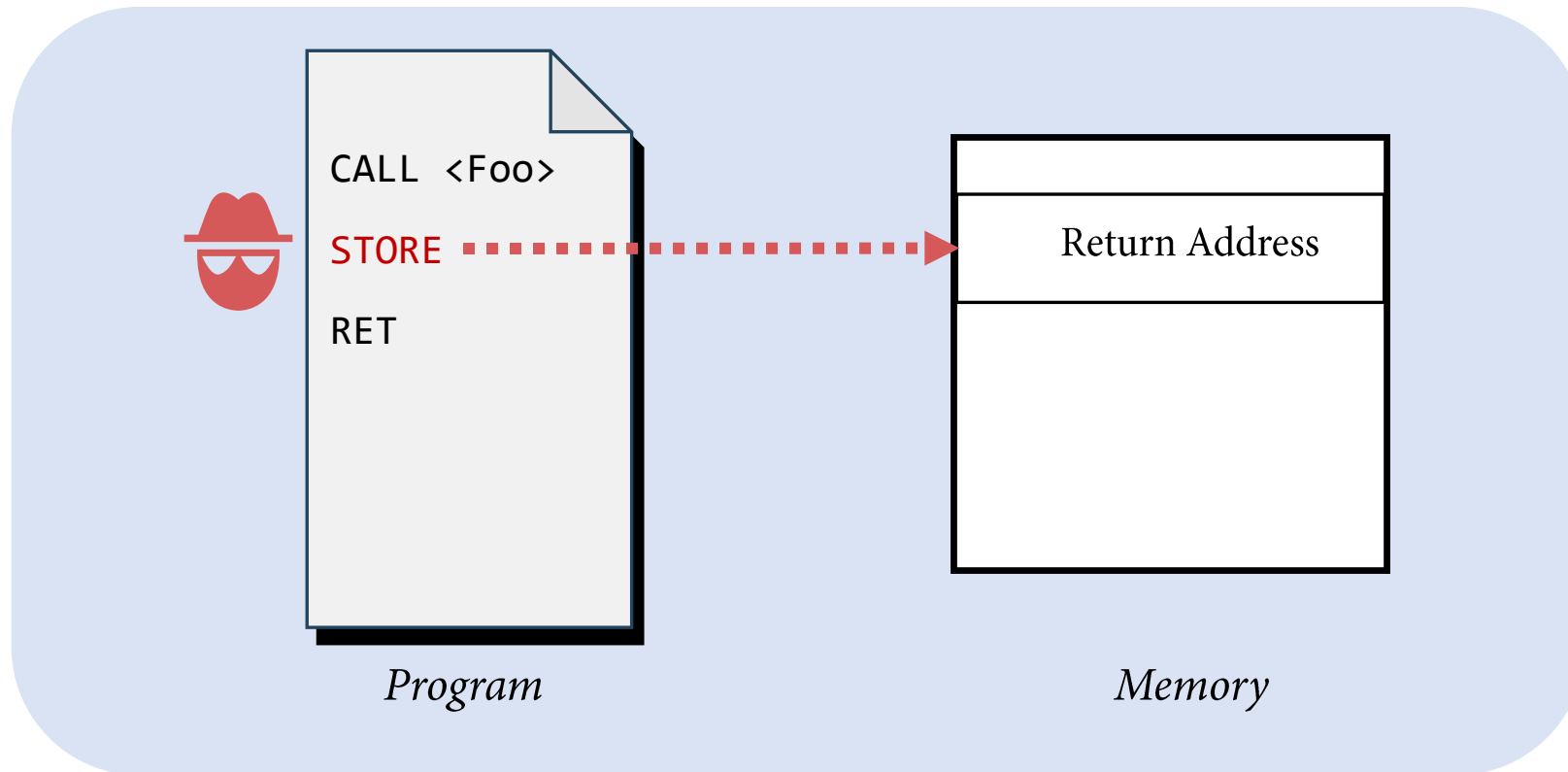
Return Address Protection with ZeRØ



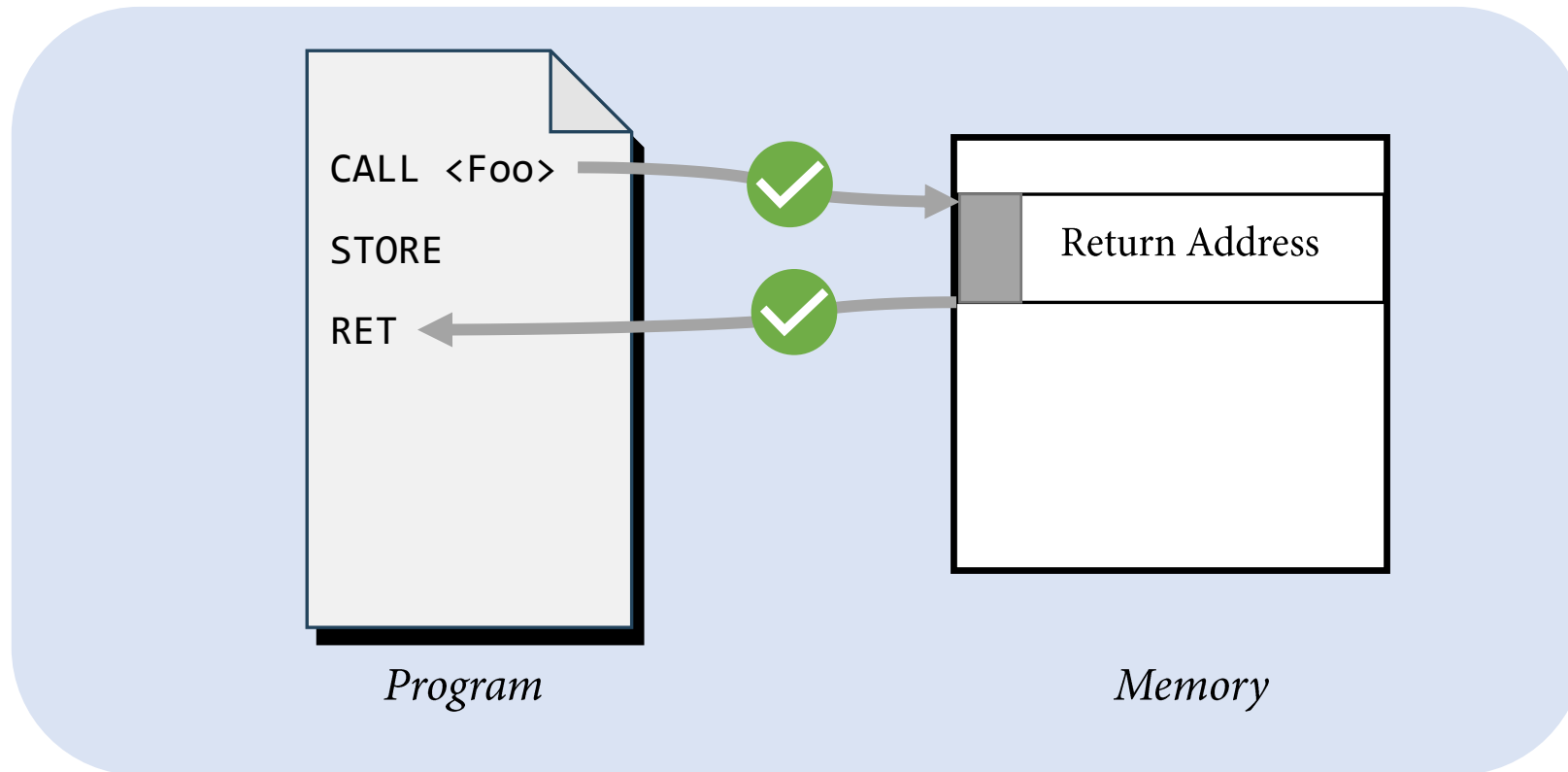
Return Address Protection with ZeRØ



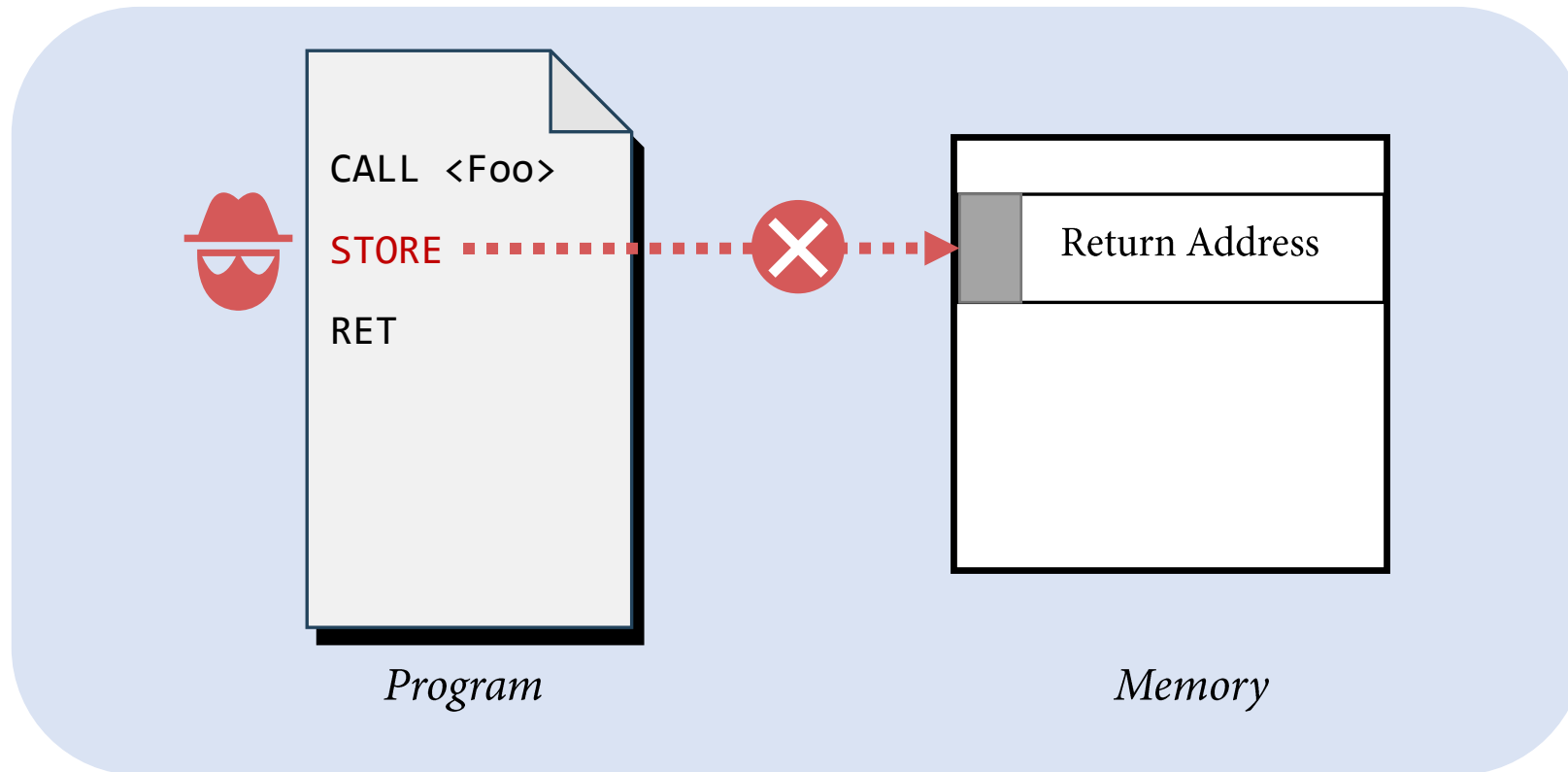
Return Address Protection with ZeRØ



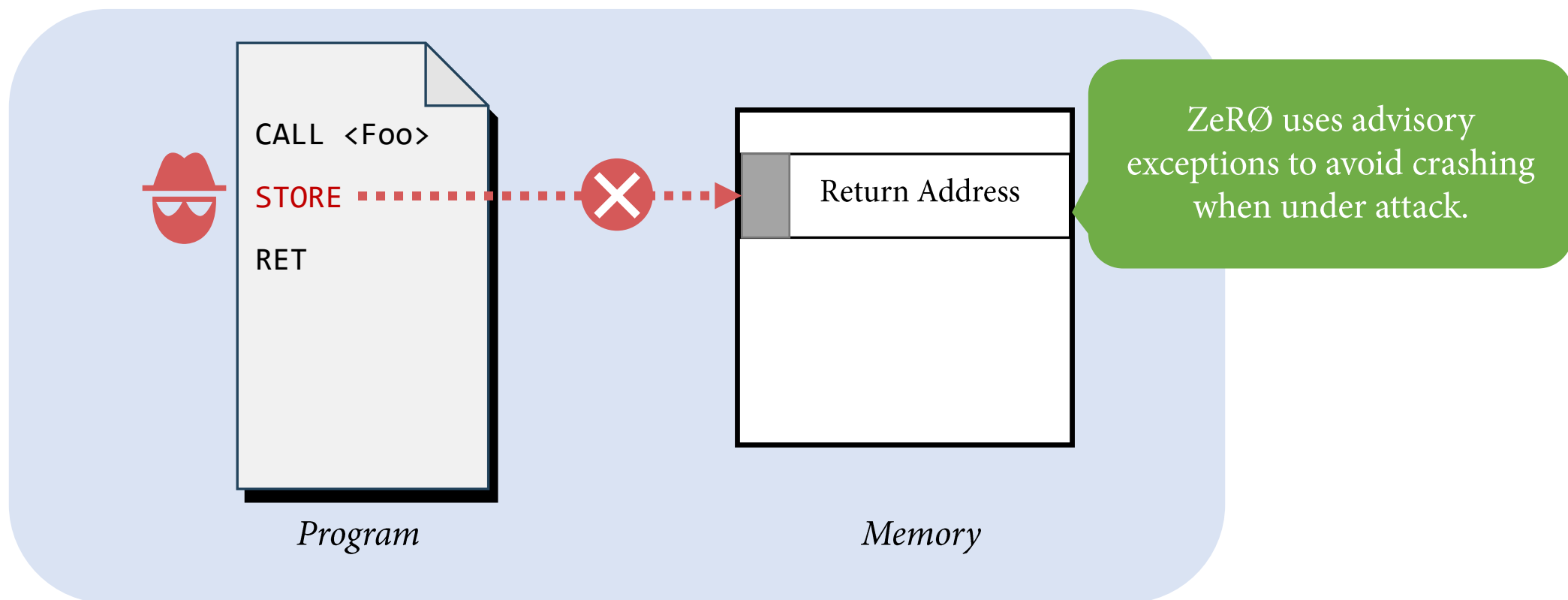
Return Address Protection with ZeRØ



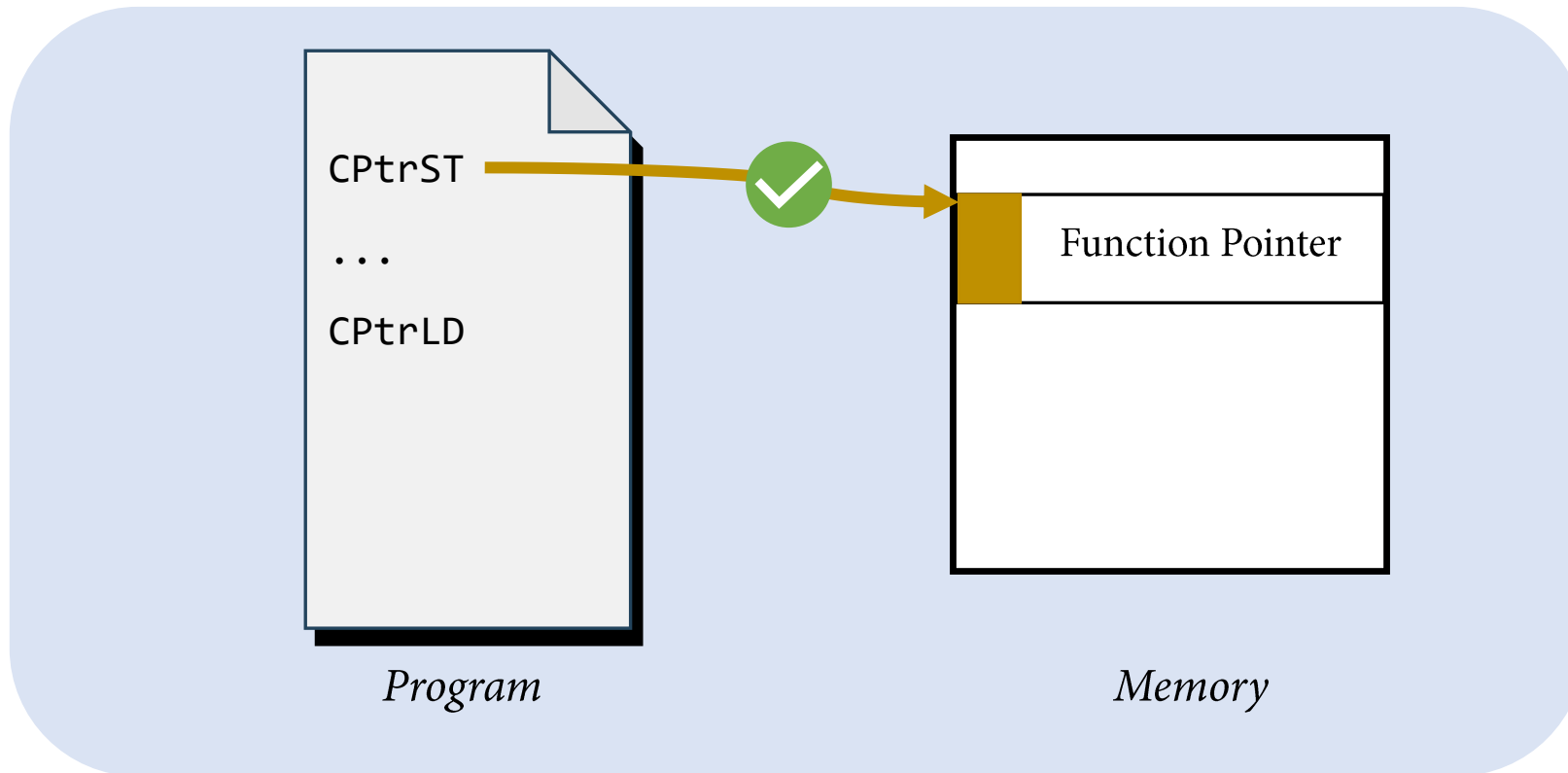
Return Address Protection with ZeRØ



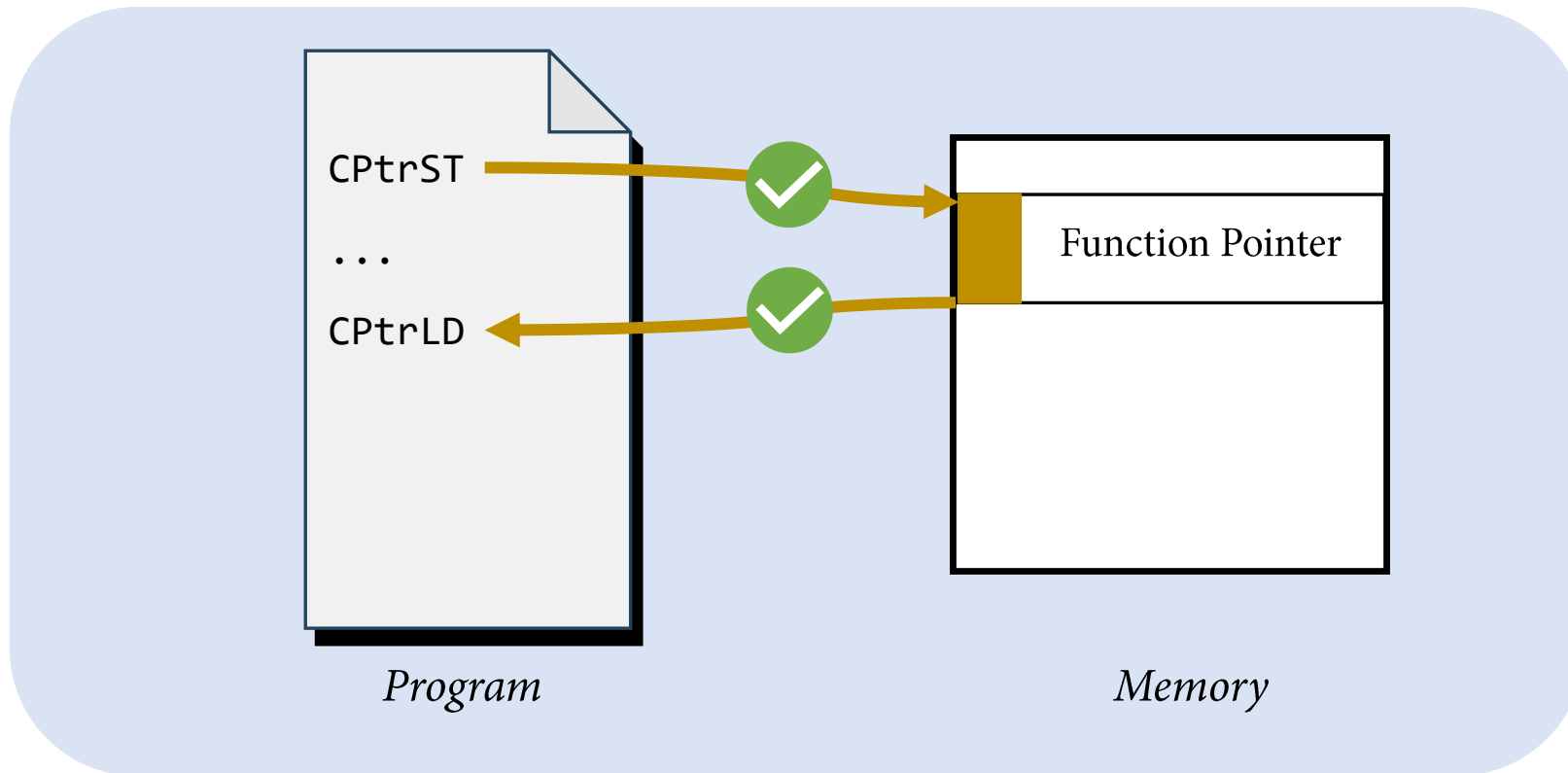
Return Address Protection with ZeRØ



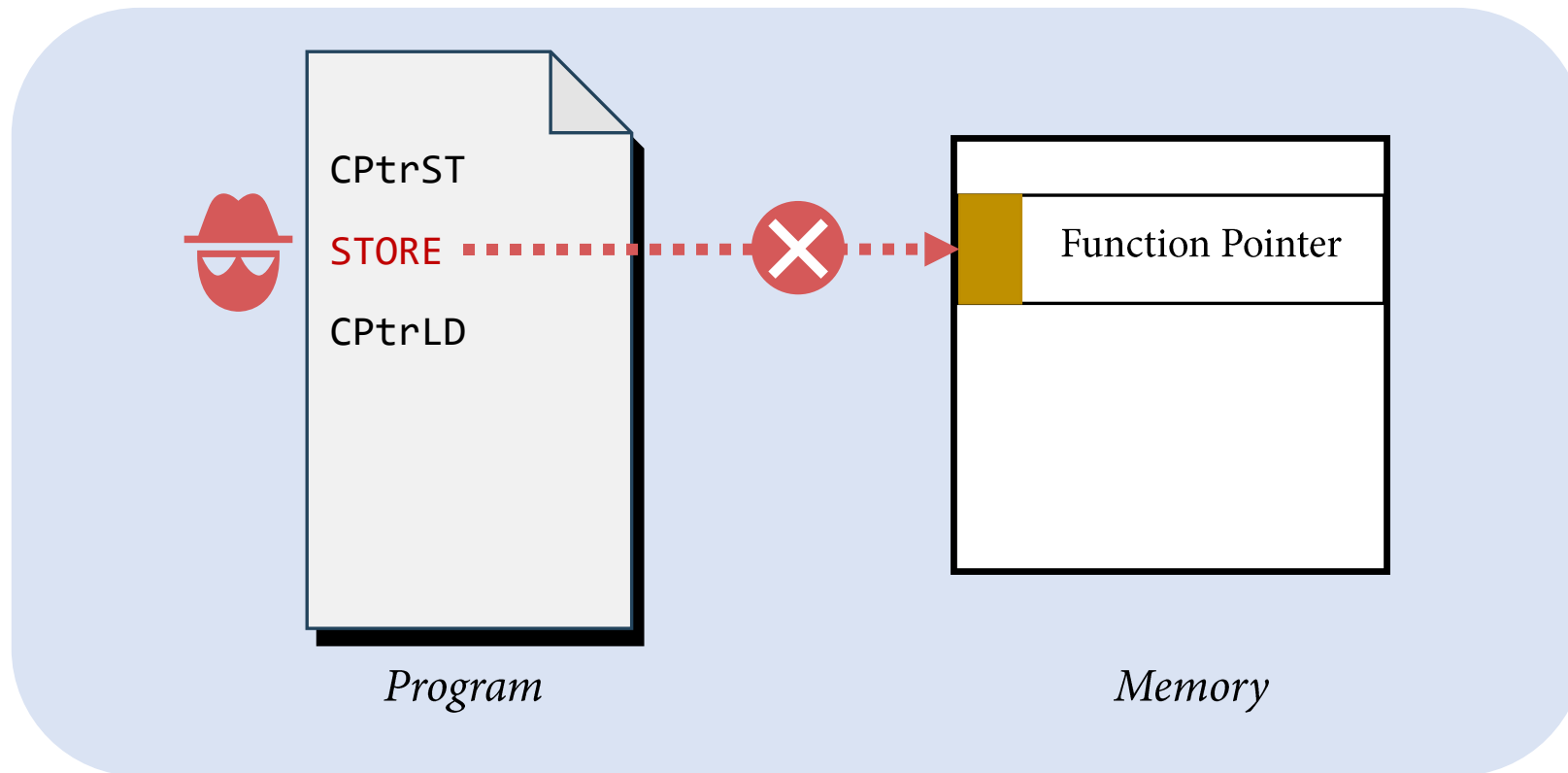
Code Pointer Integrity with ZeRØ



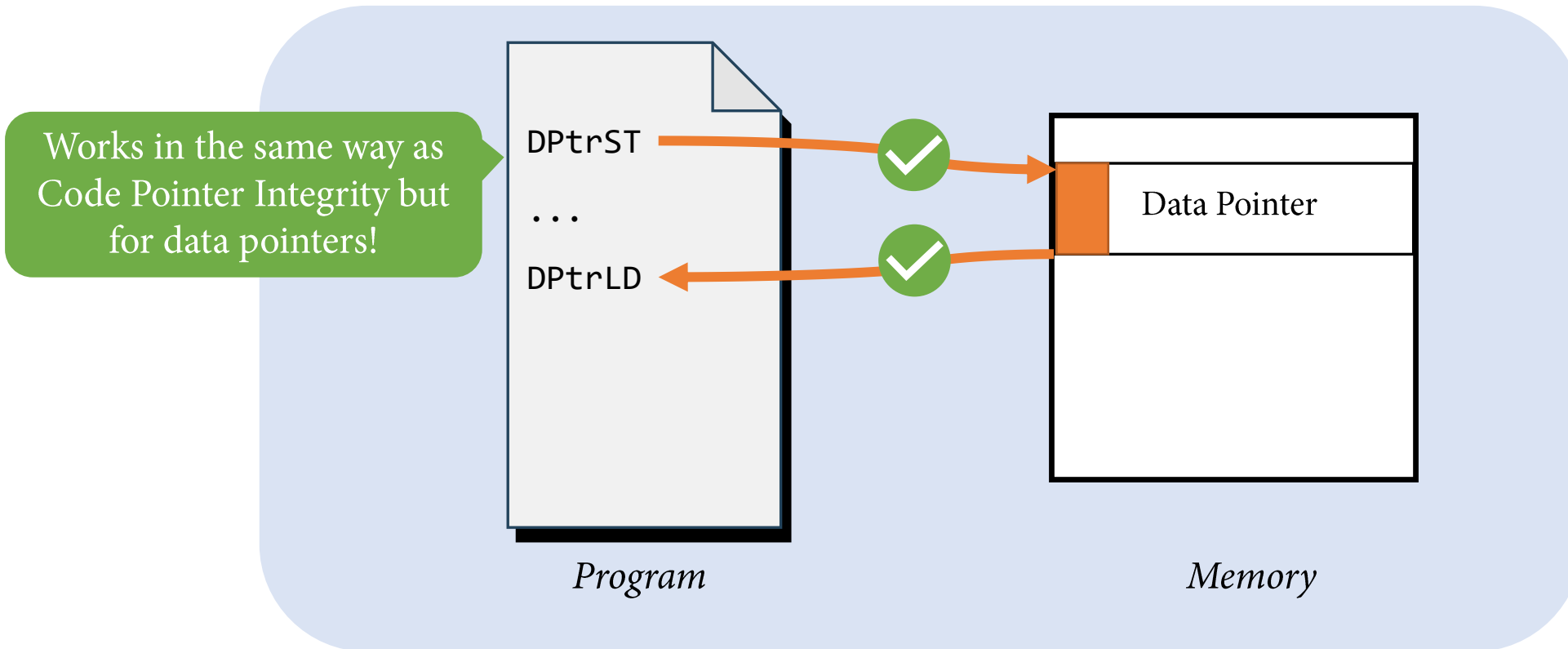
Code Pointer Integrity with ZeRØ



Code Pointer Integrity with ZeRØ



Data Pointer Integrity with ZeRØ



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



Has
Pointers?



Encoded



Normal



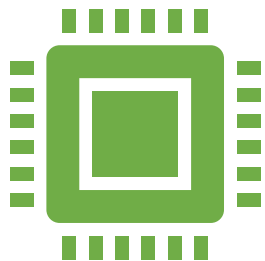
Has
Pointers?



Normal



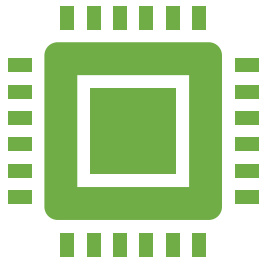
ZeRØ Performance Overheads



Hardware Modifications

Our measurements show no impact on the cache access latency.

ZeRØ Performance Overheads



Hardware Modifications

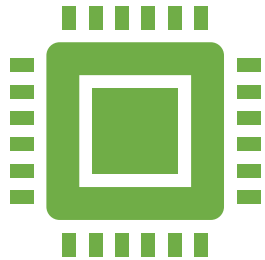
Our measurements show no impact on the cache access latency.

00010010
101001101
00010010
111001001
00010010

Software Modifications

- Our special load/stores do not change the binary size.

ZeRØ Performance Overheads



Hardware Modifications

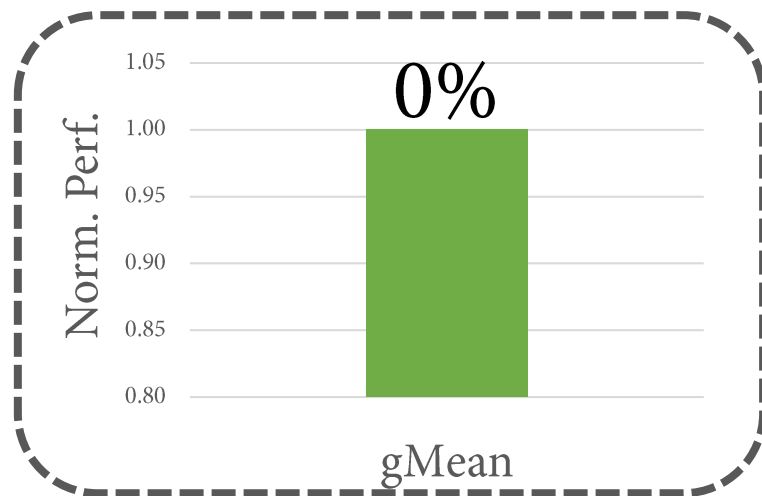
Our measurements show no impact on the cache access latency.

00010010
101001101
00010010
111001001
00010010

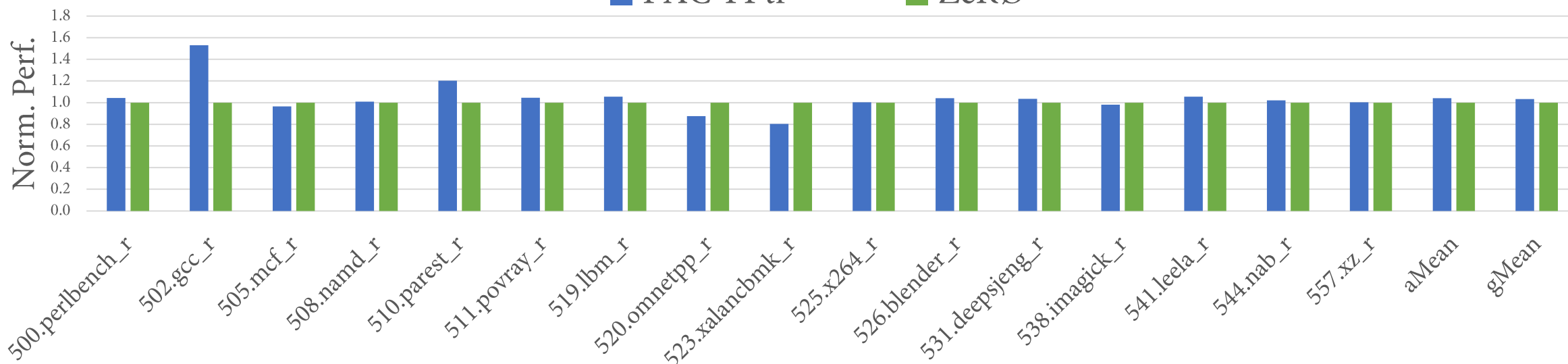
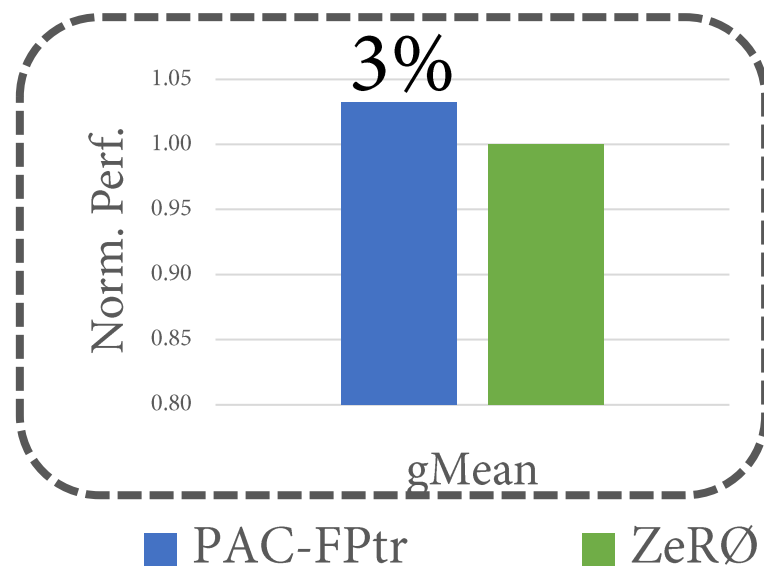
Software Modifications

- Our special load/stores do not change the binary size.
- The ClearMeta instructions are only called on memory deletion.

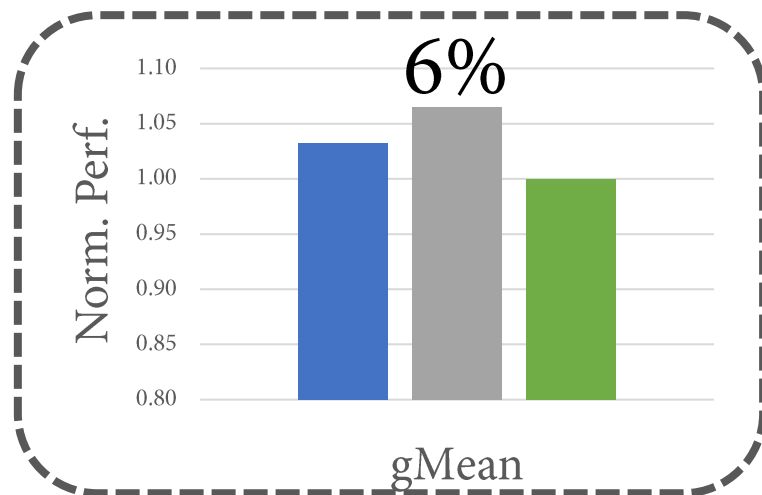
ZeRØ Performance Results (x86_64)



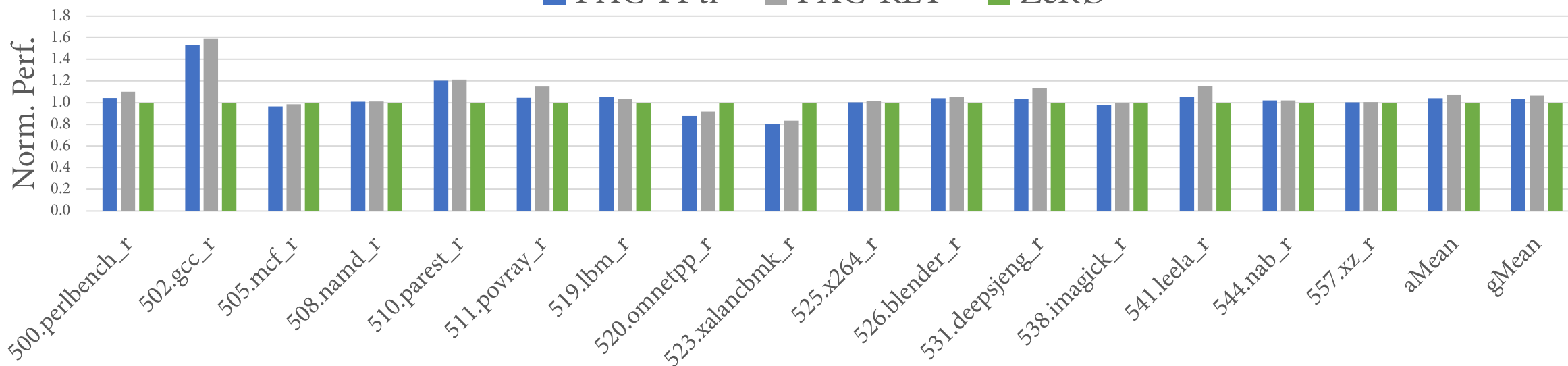
ZeRØ Performance Results (x86_64)



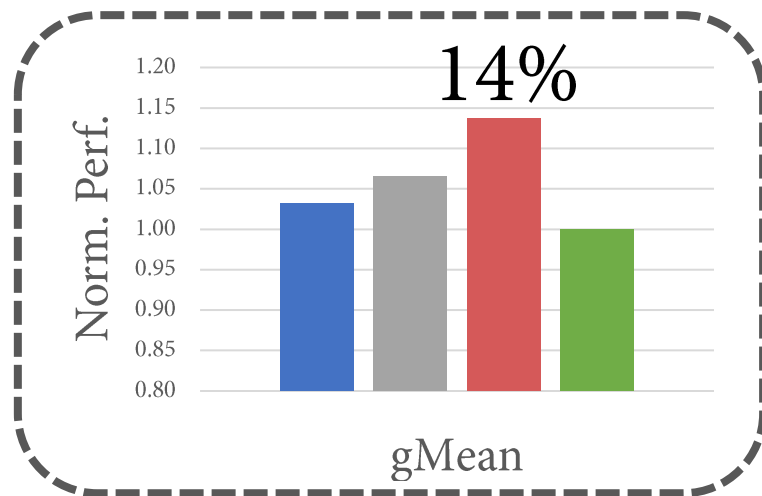
ZeRØ Performance Results (x86_64)



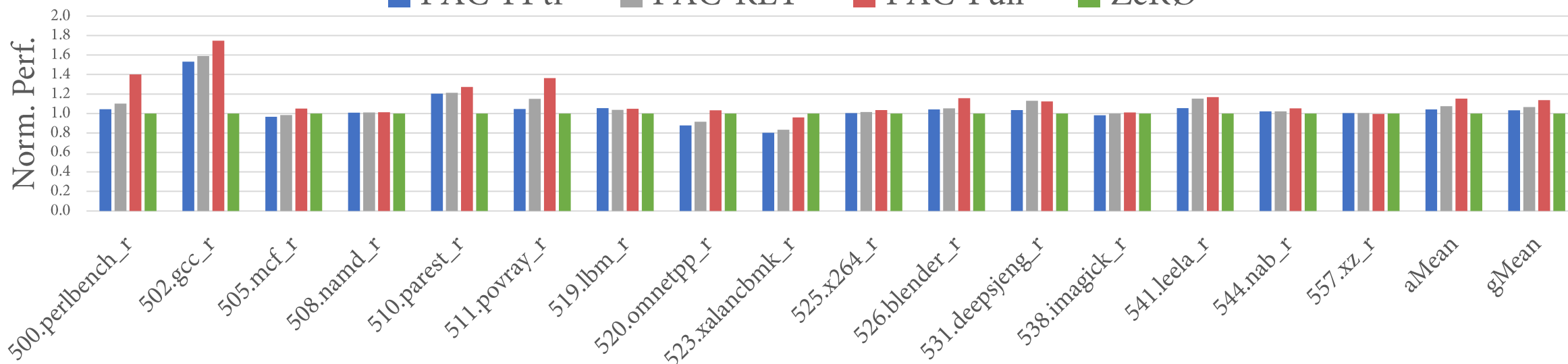
■ PAC-FPtr ■ PAC-RET ■ ZeRØ



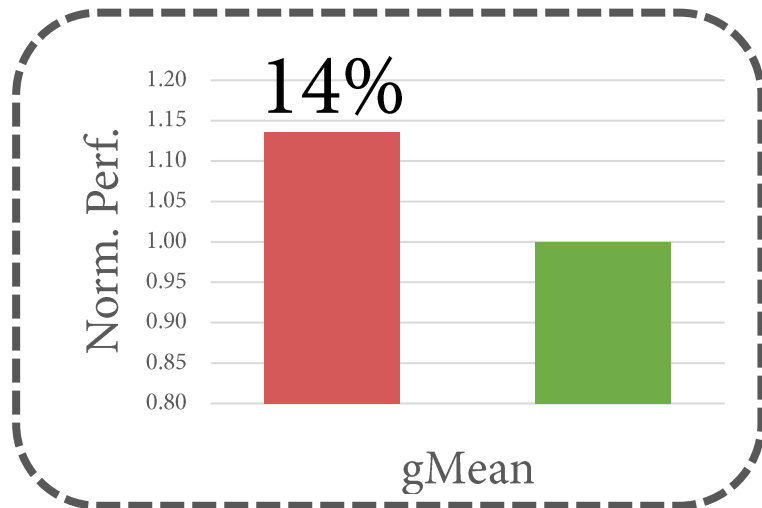
ZeRØ Performance Results (x86_64)



■ PAC-FPtr ■ PAC-RET ■ PAC-Full ■ ZeRØ



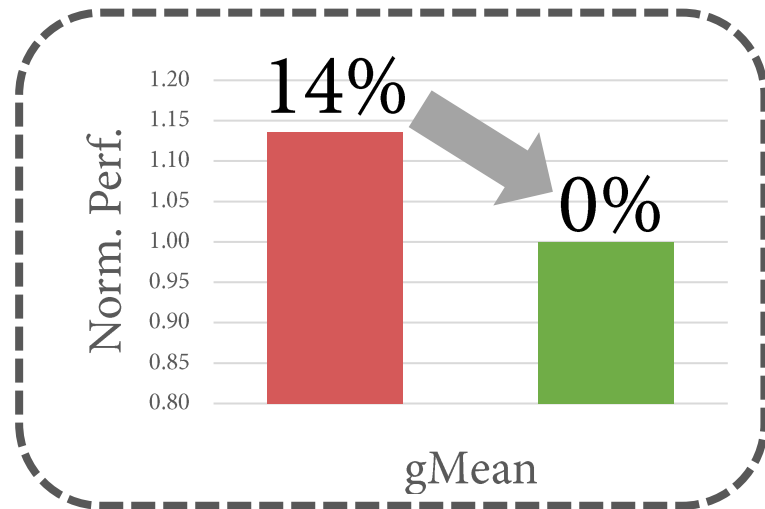
ZeRØ Performance Results (x86_64)



PAC's overheads are attributed to the extra QARMA encryption invocations upon pointer:

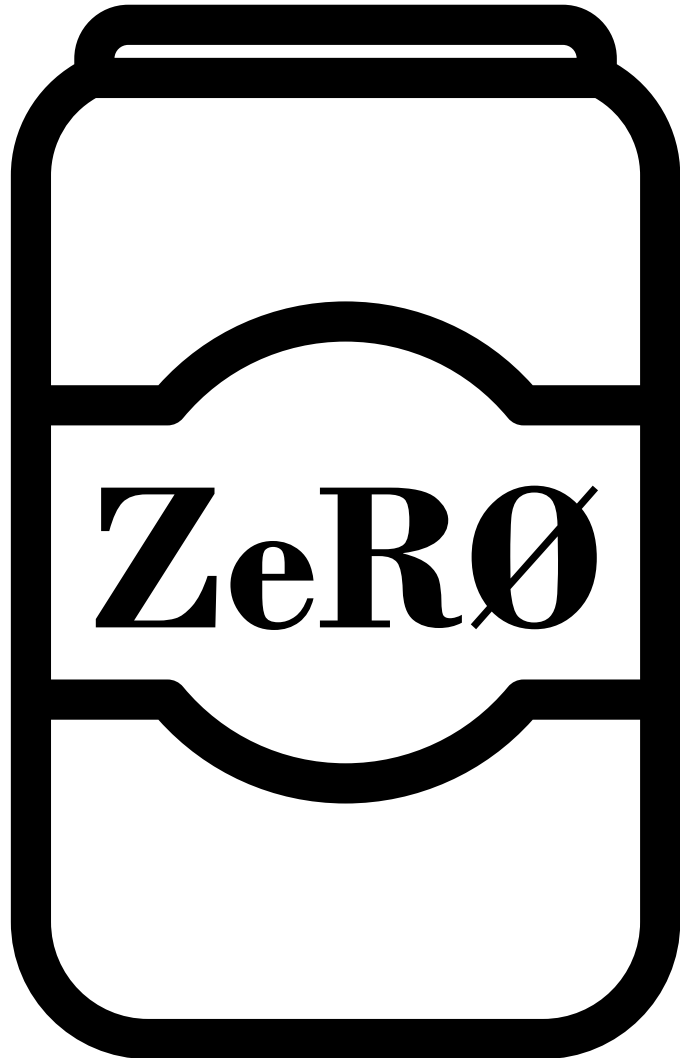
- loads/stores
- usages

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

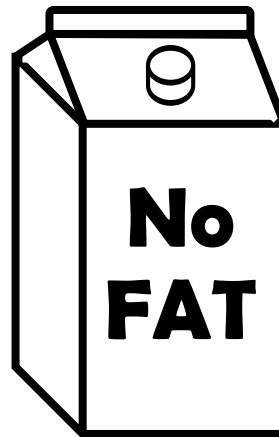
My solutions for C/C++ memory (un)safety

Memory Blocklisting



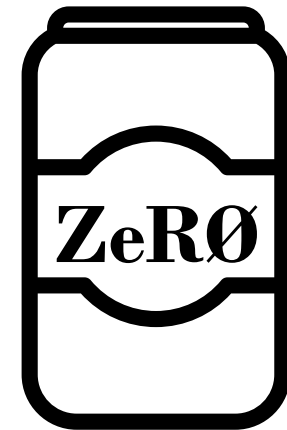
[[MICRO 2019](#)]

Memory Permitlisting



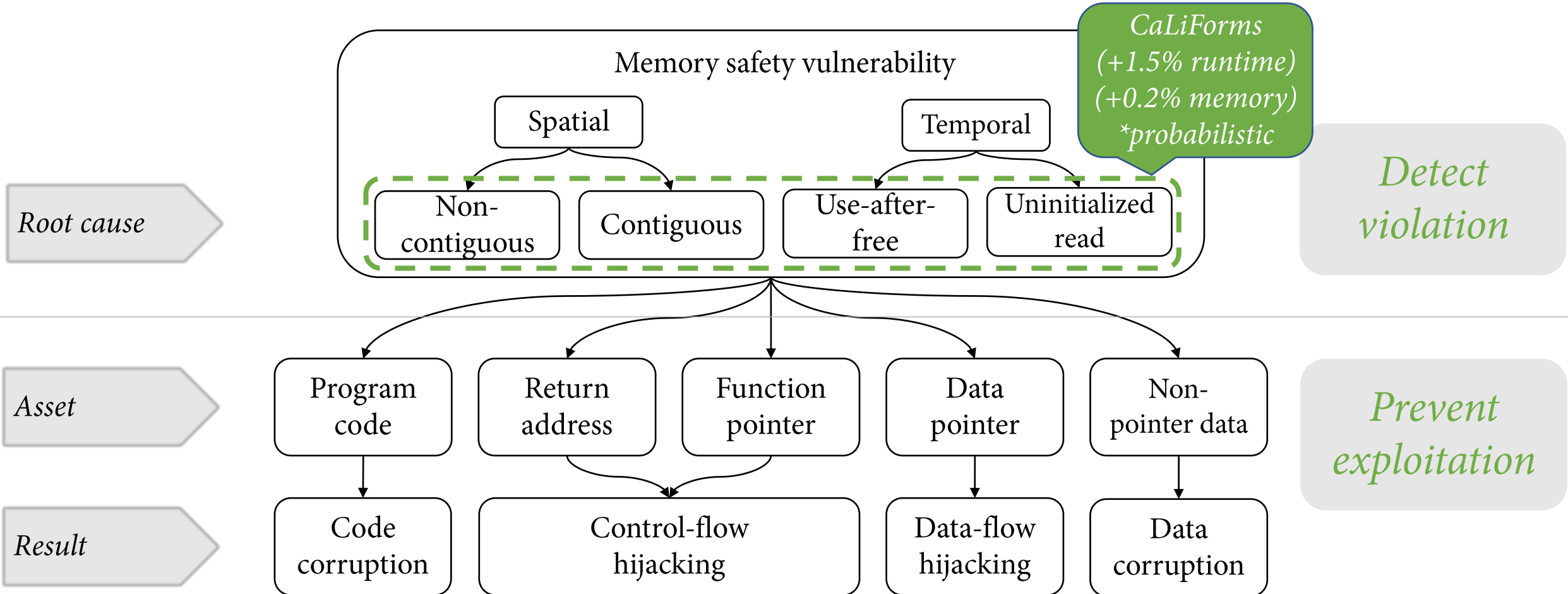
[[ISCA 2021](#)]

Exploit Mitigation

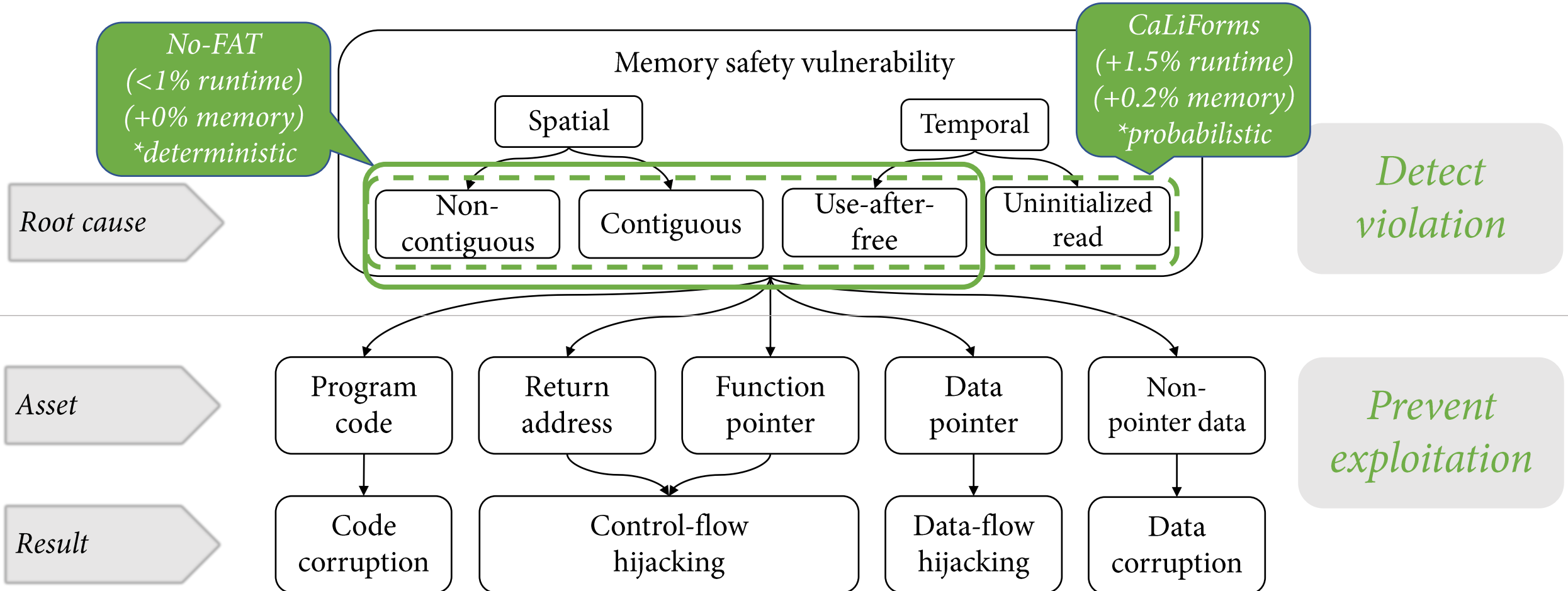


[[ISCA 2021](#)]

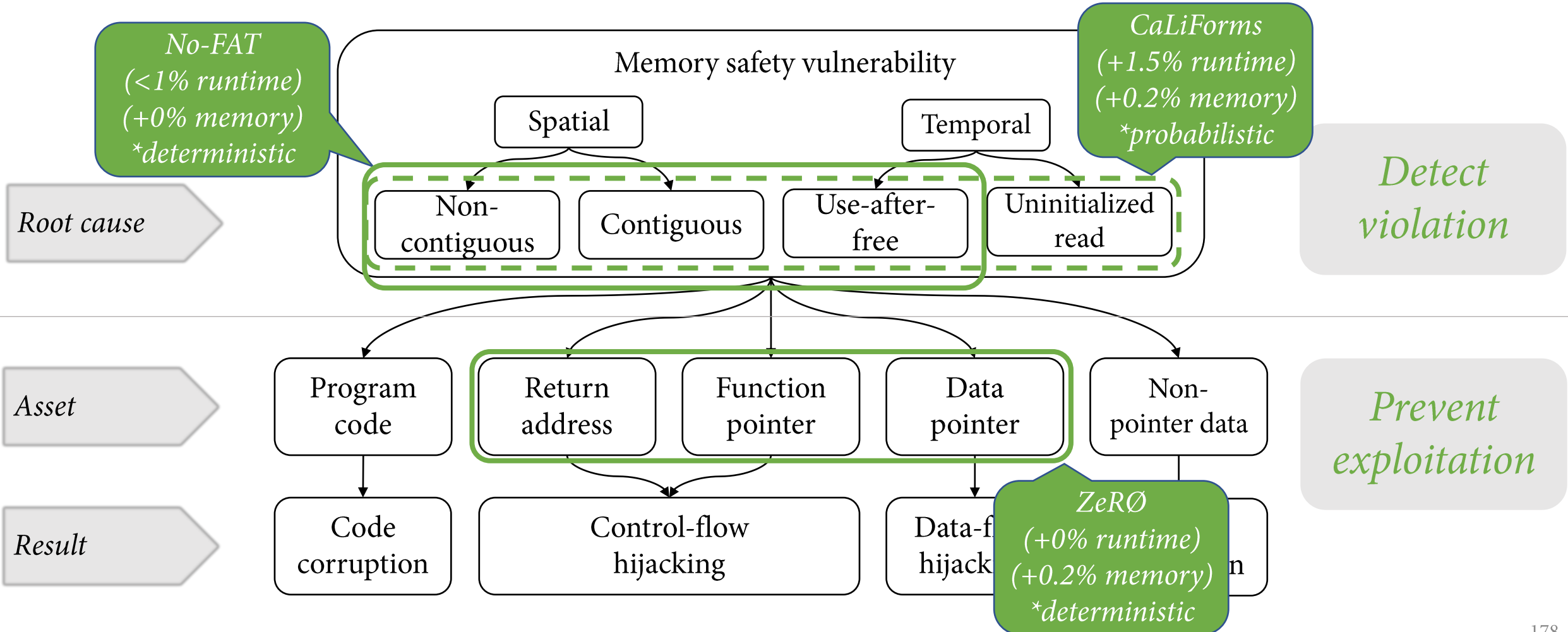
Memory Attacks Taxonomy



Memory Attacks Taxonomy



Memory Attacks Taxonomy



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

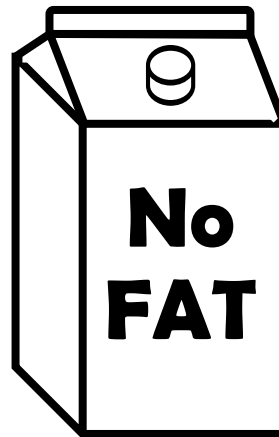
My solutions for C/C++ memory (un)safety

**Memory
Blocklisting**



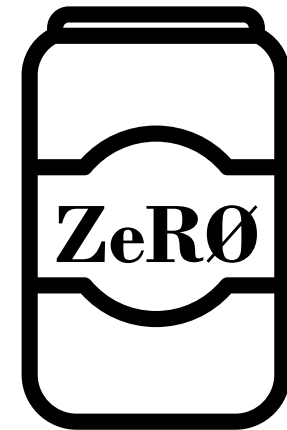
[[MICRO 2019](#)]

**Memory
Permitlisting**



[[ISCA 2021](#)]

**Exploit
Mitigation**



[[ISCA 2021](#)]

Thank You!