Hardware-Software Co-design For Practical Memory Safety

Mohamed Tarek

Ph.D. Defense - April 11th, 2022

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Hardware-Software Co-design For Practical **Memory Safety**

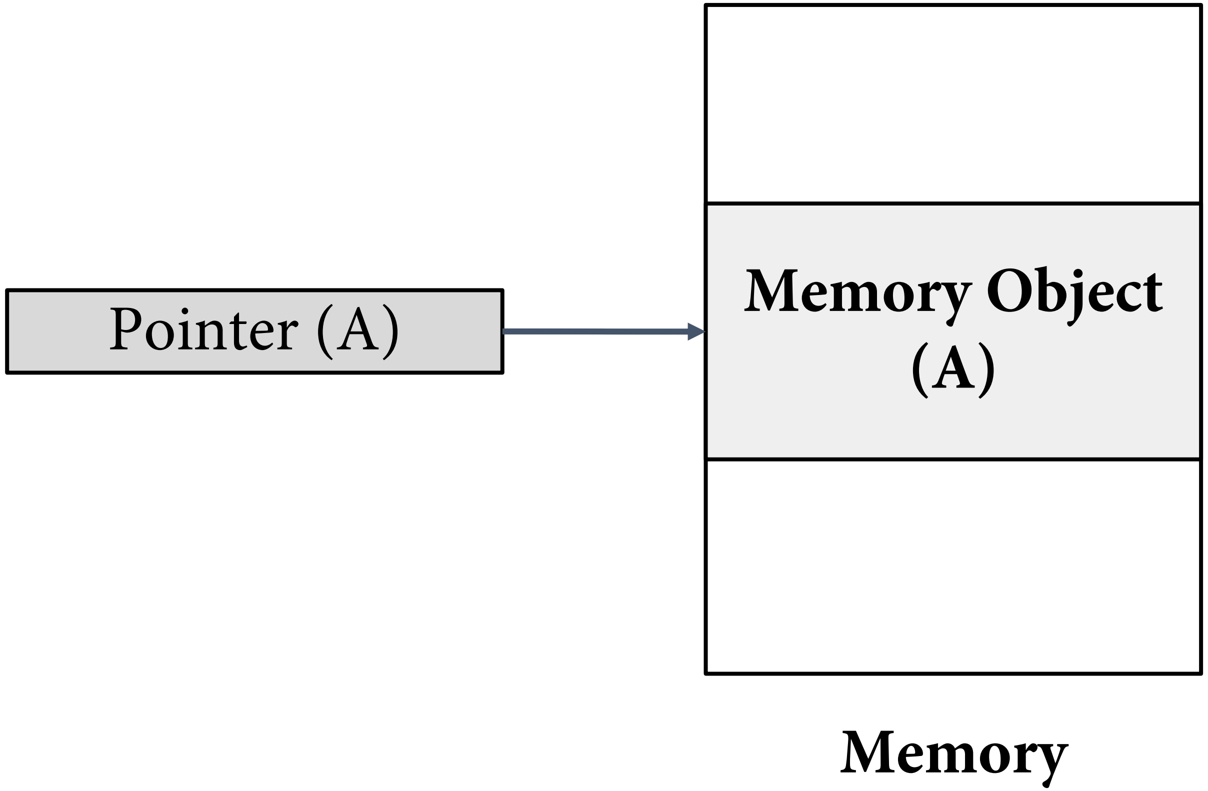
Mohamed Tarek

Ph.D. Defense - April 11th, 2022

Columbia | Engineering
The Fu Foundation School of Engineering and Applied Science

What is Memory Safety?

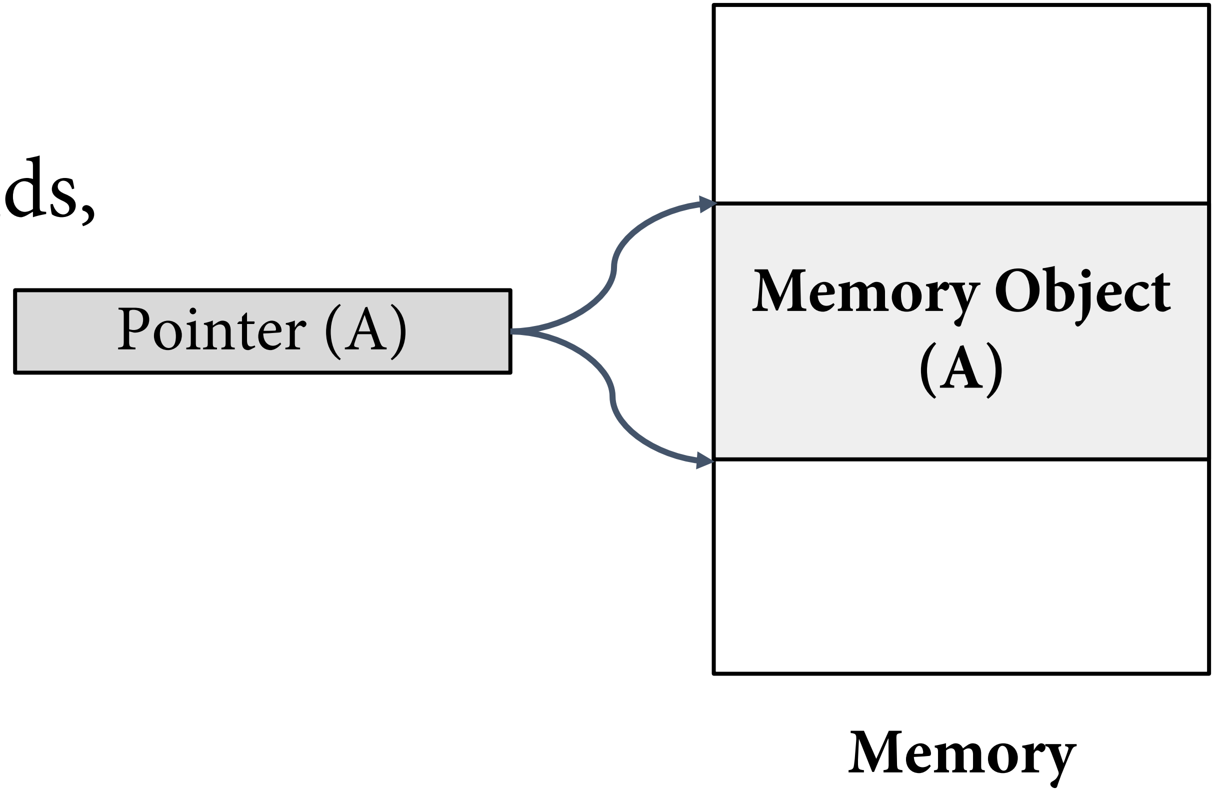
A program property that guarantees **memory objects** can only be accessed:



What is Memory Safety?

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

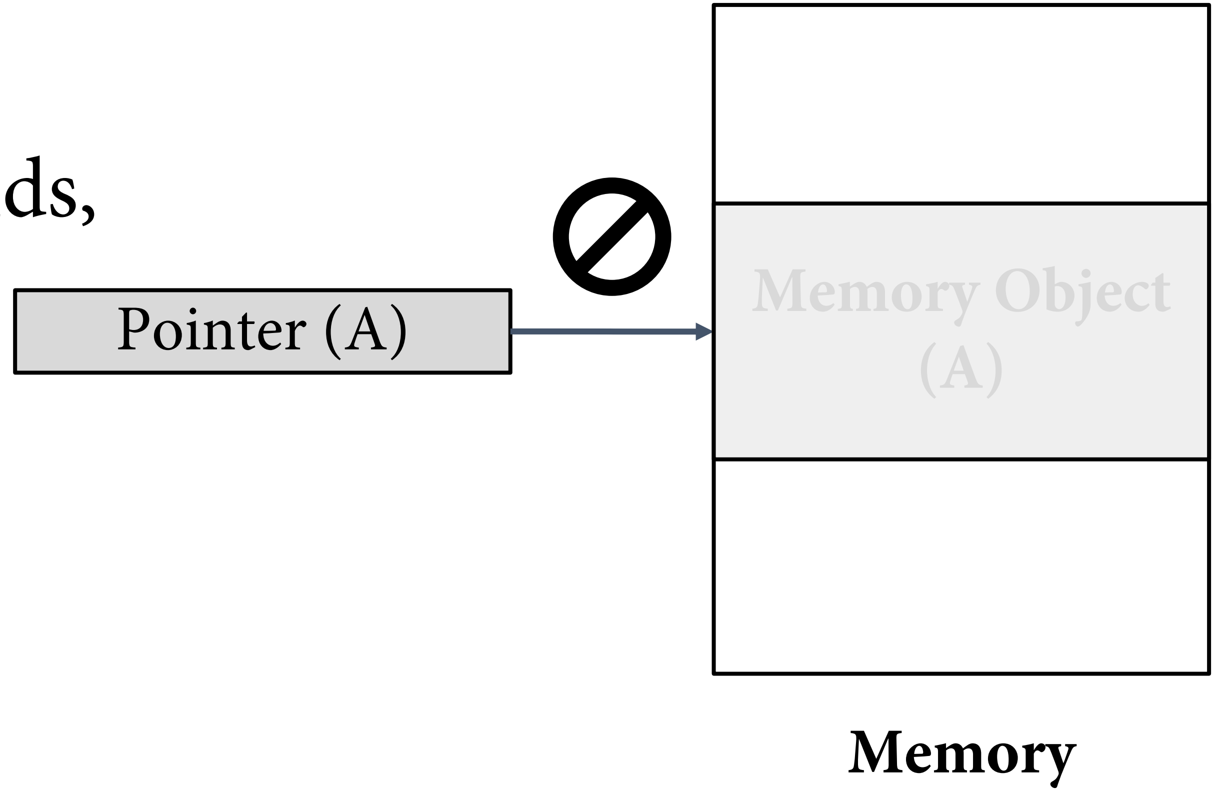
- Between their intended bounds,



What is Memory Safety?

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

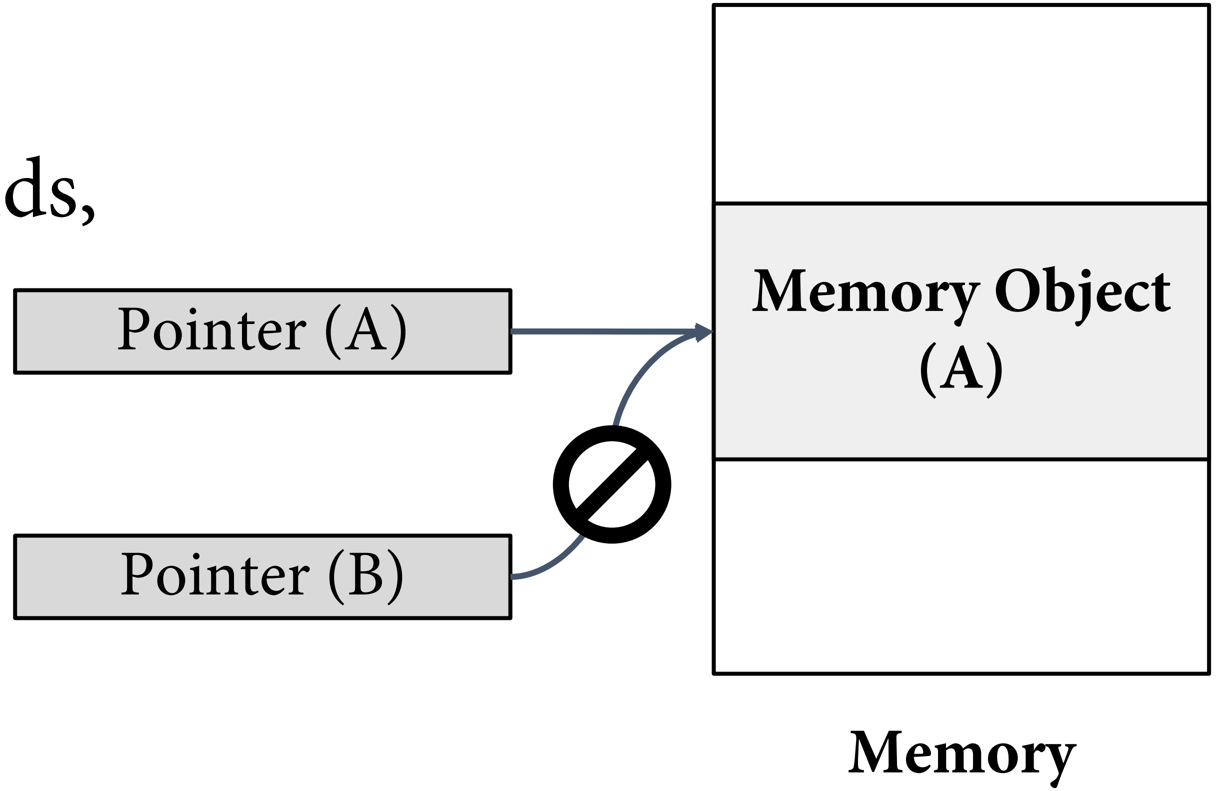
- Between their intended bounds,
- During their lifetime, and



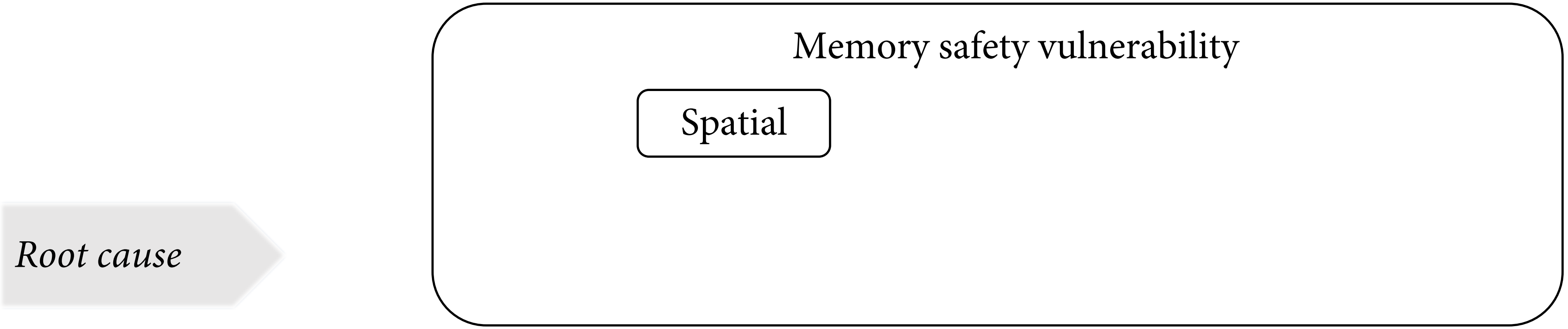
What is Memory Safety?

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

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



Memory Attacks Taxonomy

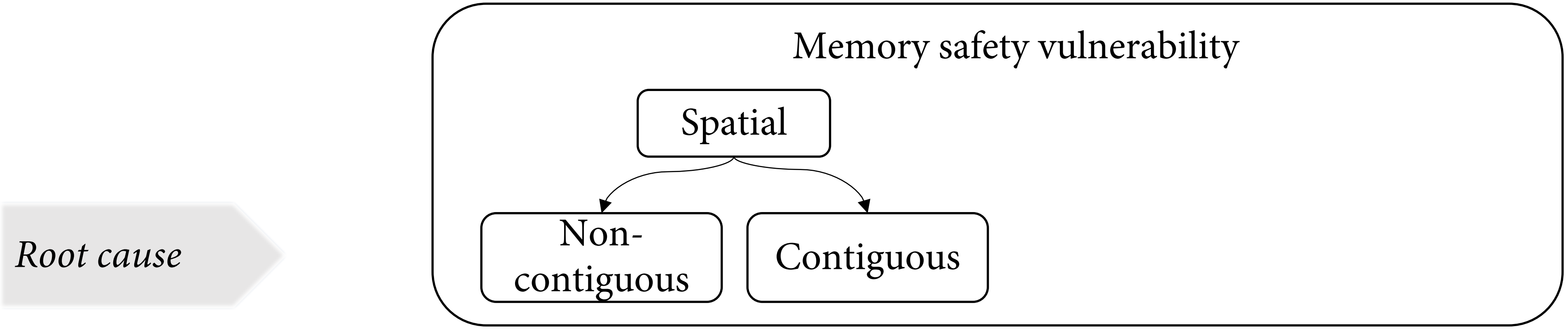


Root cause

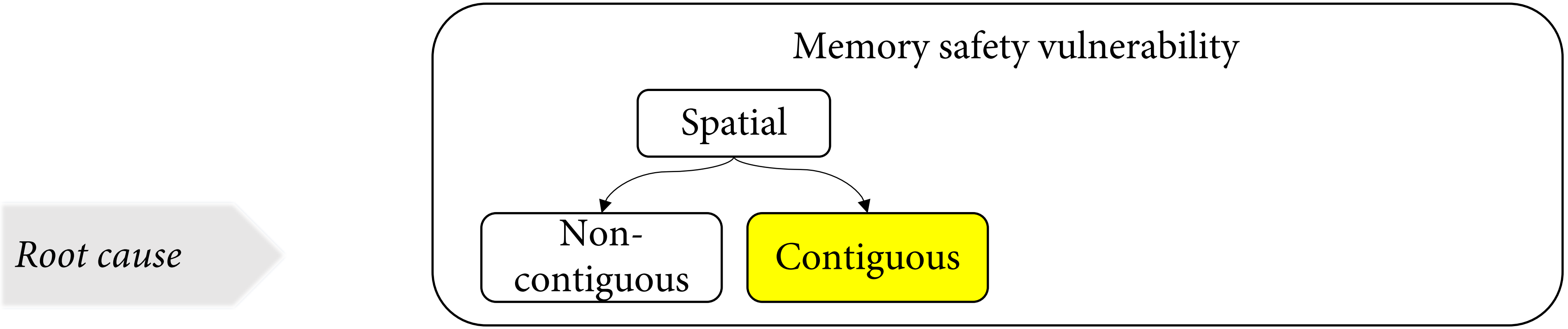
Memory safety vulnerability

Spatial

Memory Attacks Taxonomy

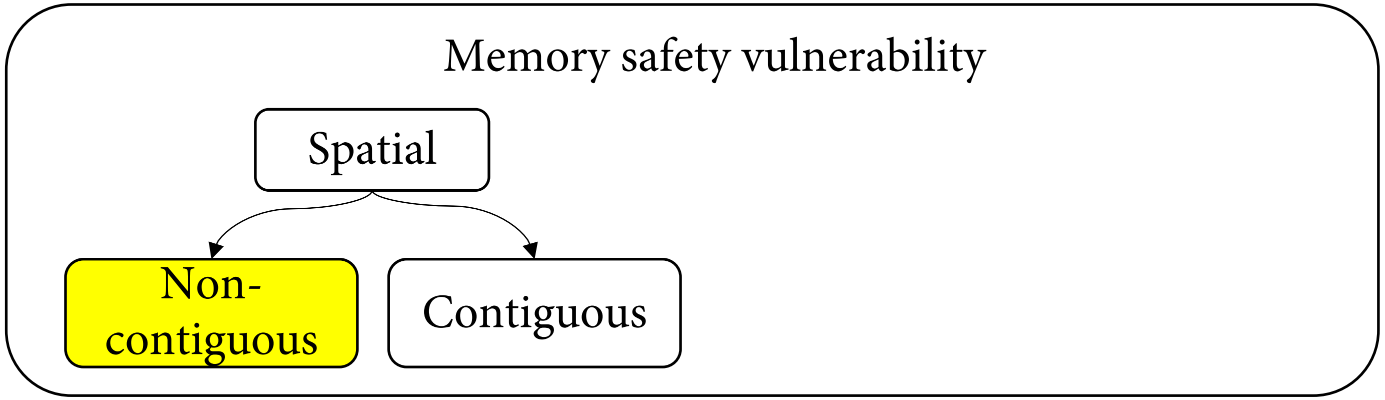


Memory Attacks Taxonomy



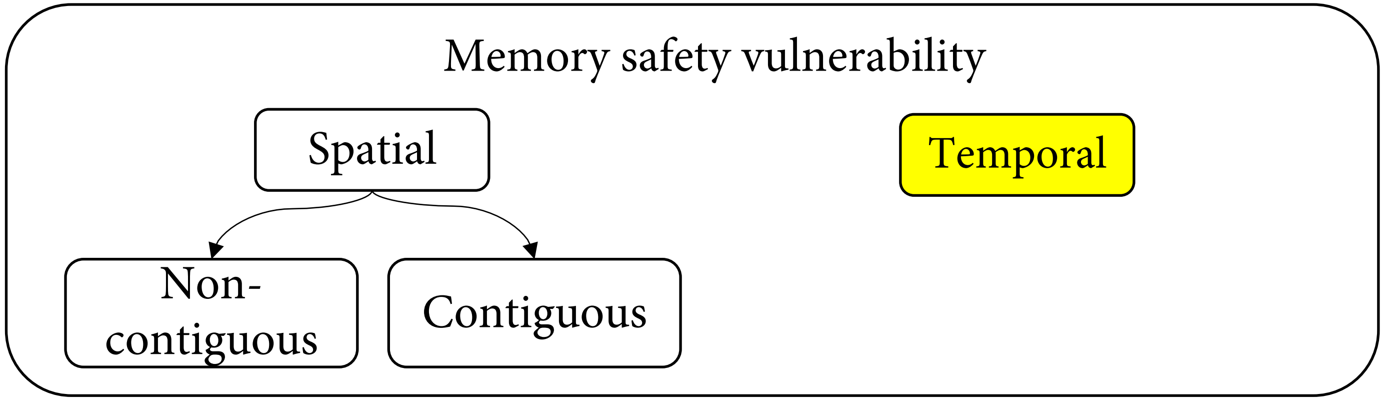
Memory Attacks Taxonomy

Root cause



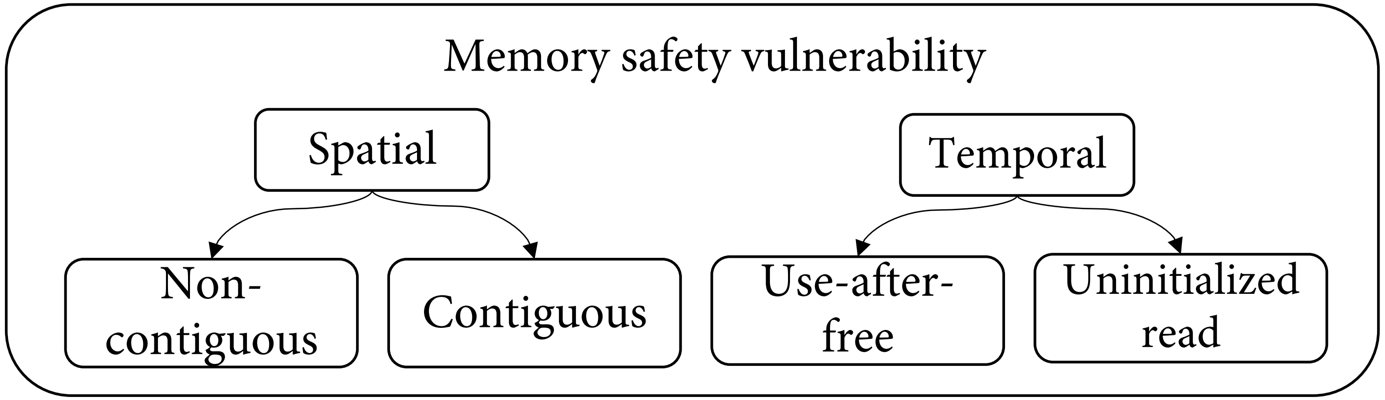
Memory Attacks Taxonomy

Root cause



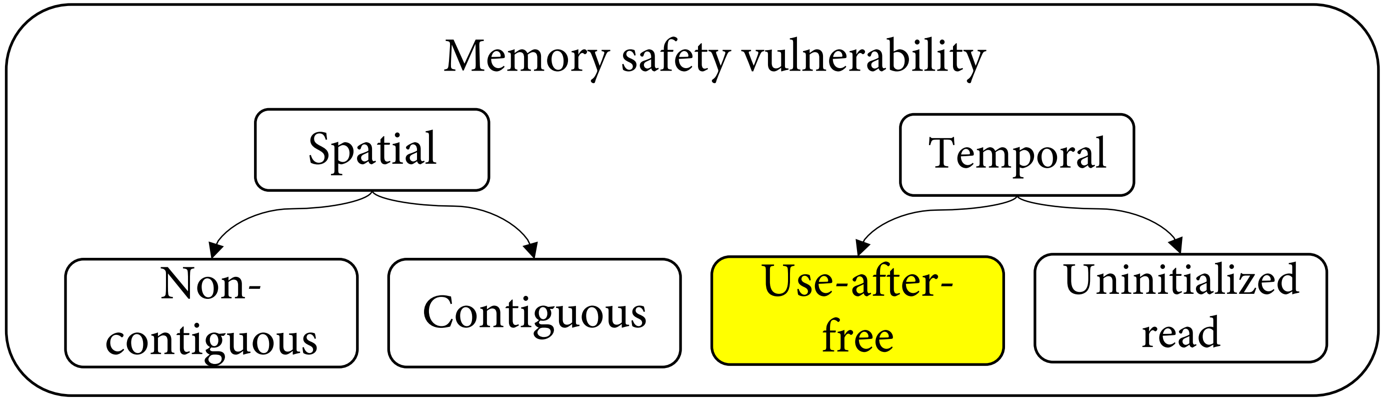
Memory Attacks Taxonomy

Root cause



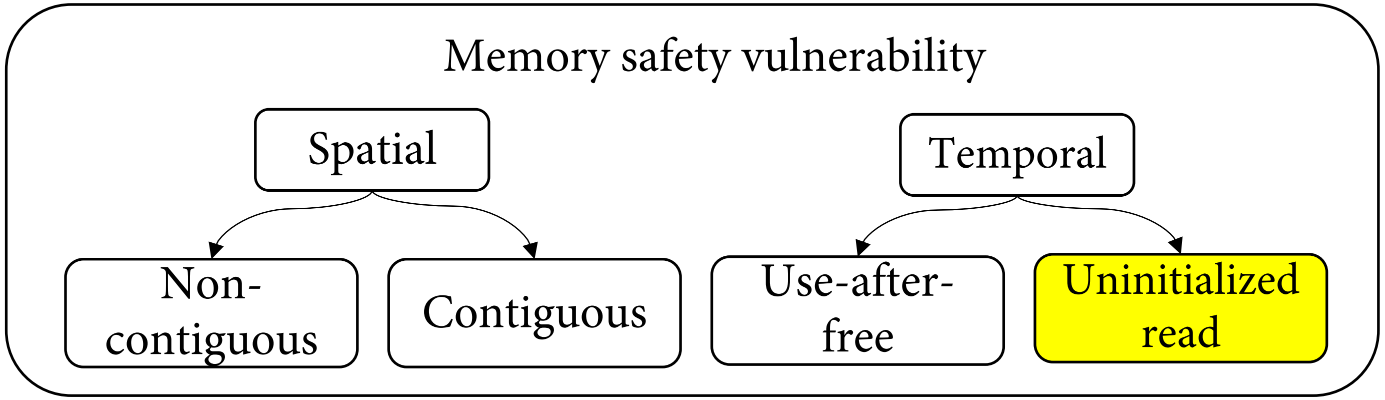
Memory Attacks Taxonomy

Root cause



Memory Attacks Taxonomy

Root cause



Memory safety vulnerability

Spatial

Non-contiguous

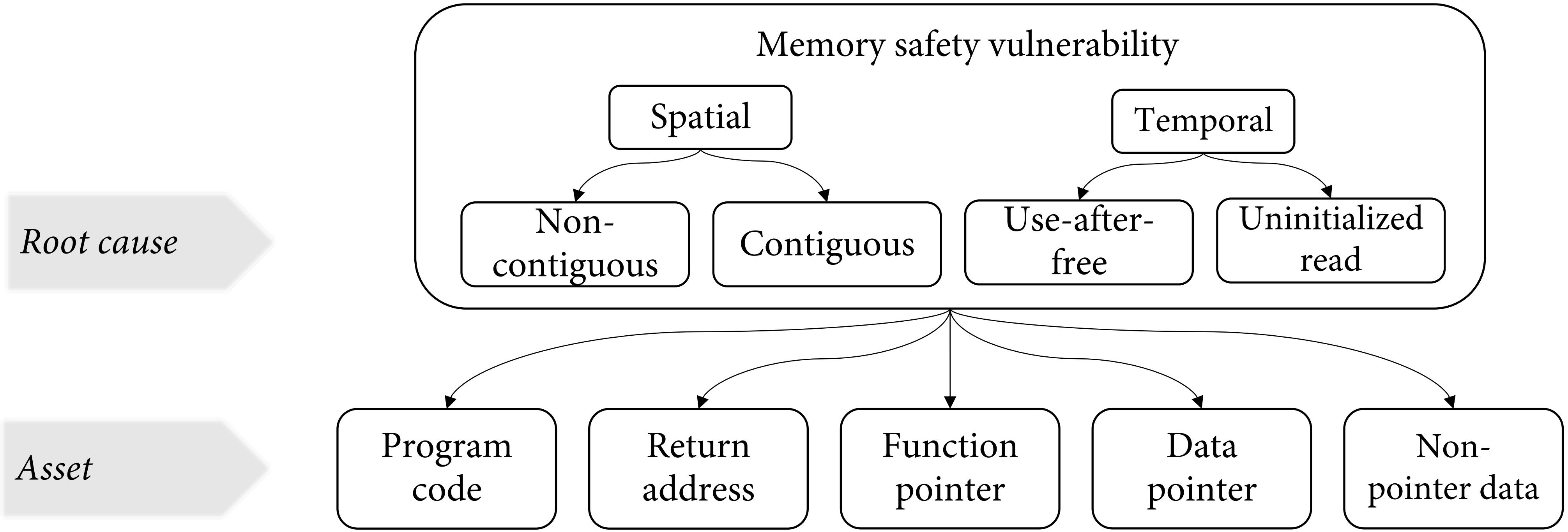
Contiguous

Temporal

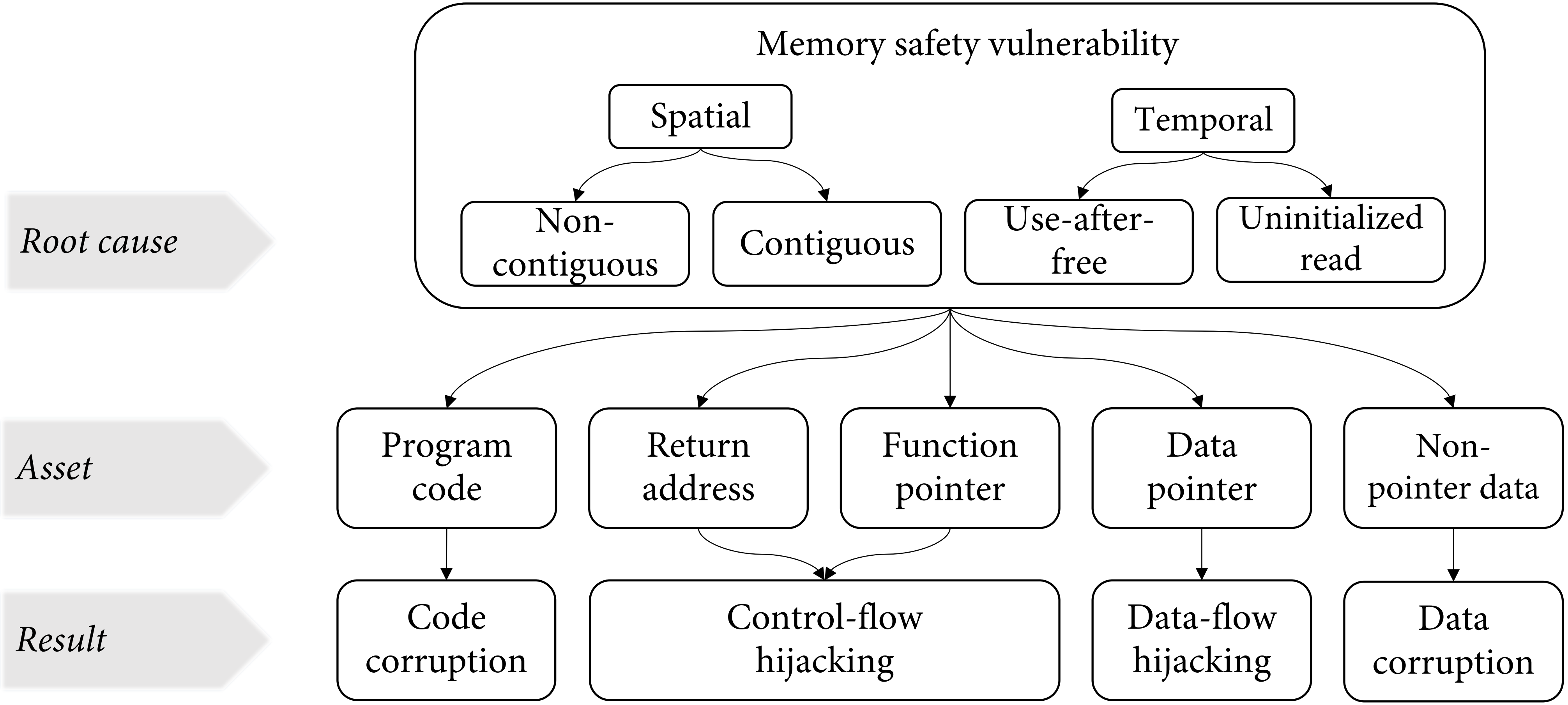
Use-after-free

Uninitialized read

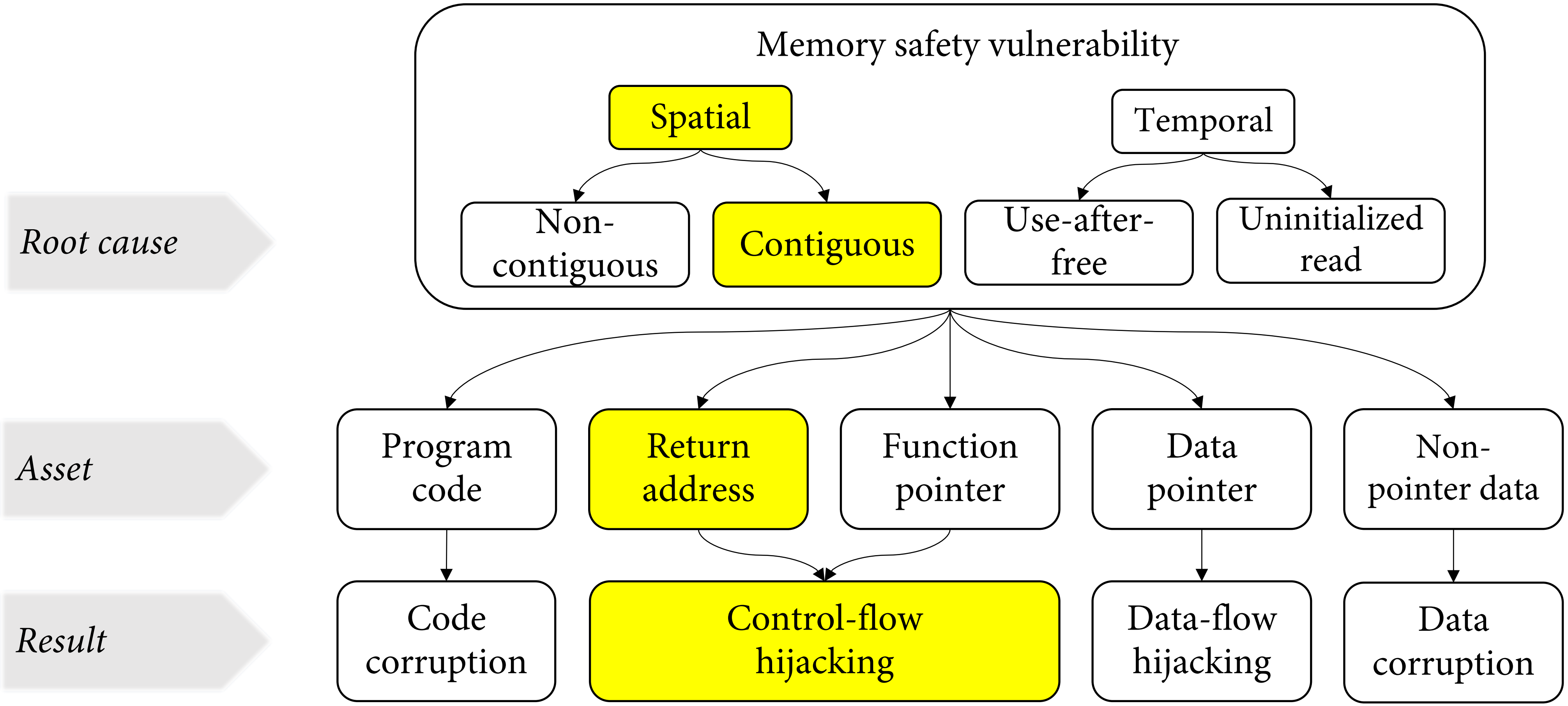
Memory Attacks Taxonomy



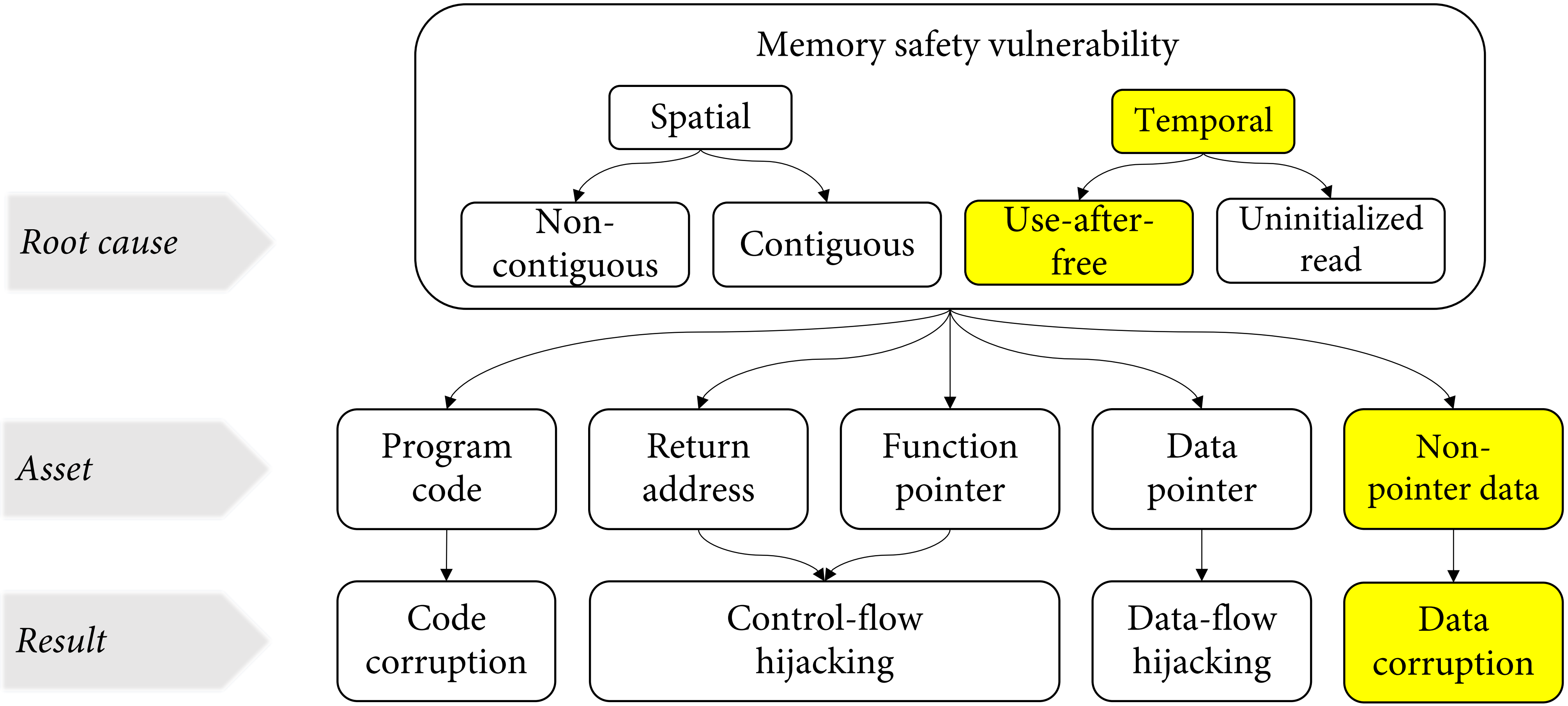
Memory Attacks Taxonomy



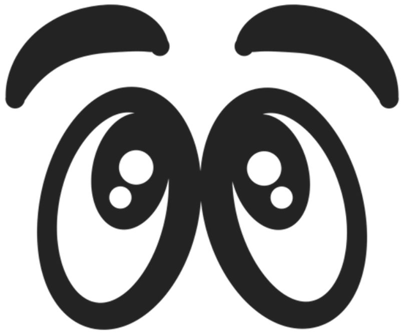
Memory Attacks Taxonomy



Memory Attacks Taxonomy

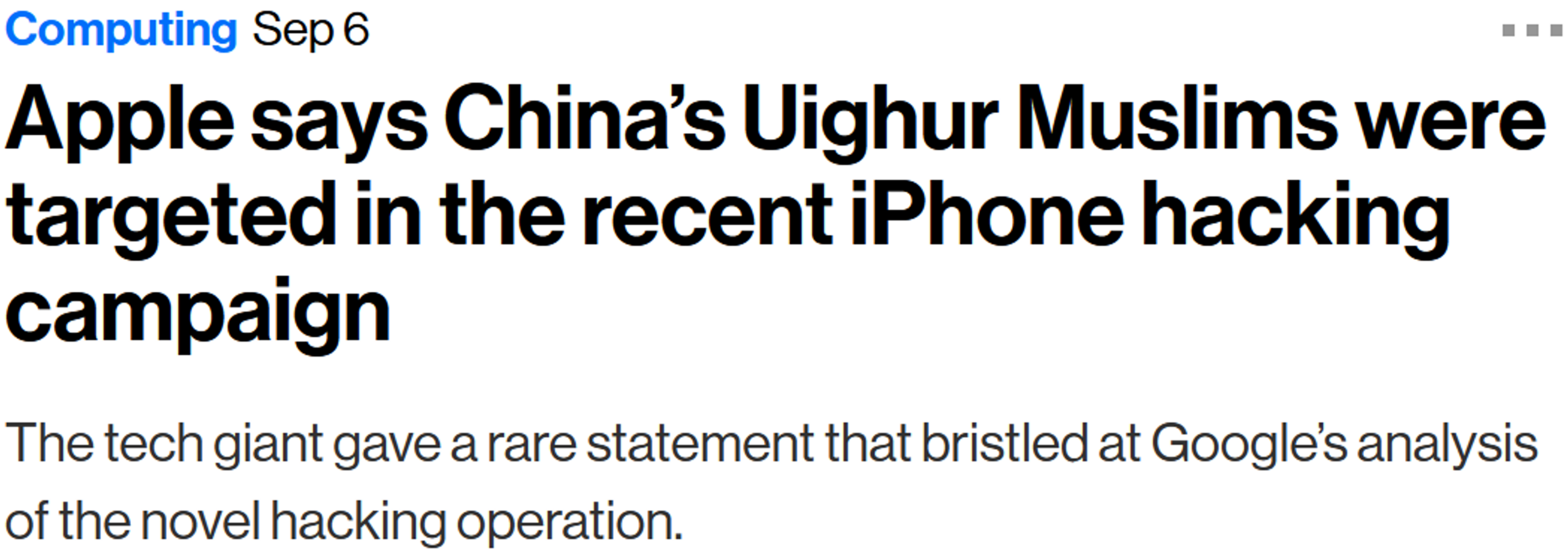


Why is memory safety a concern?



Memory Safety is a serious problem!

Memory Safety is a serious problem!

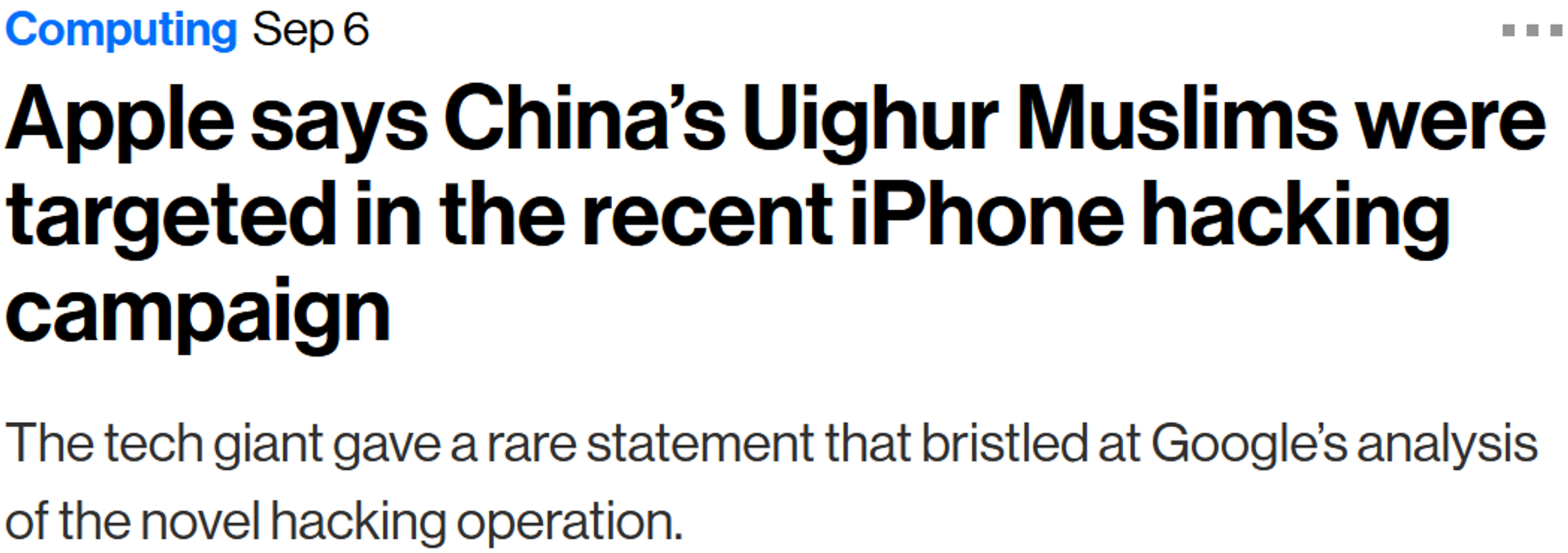


Computing Sep 6

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.

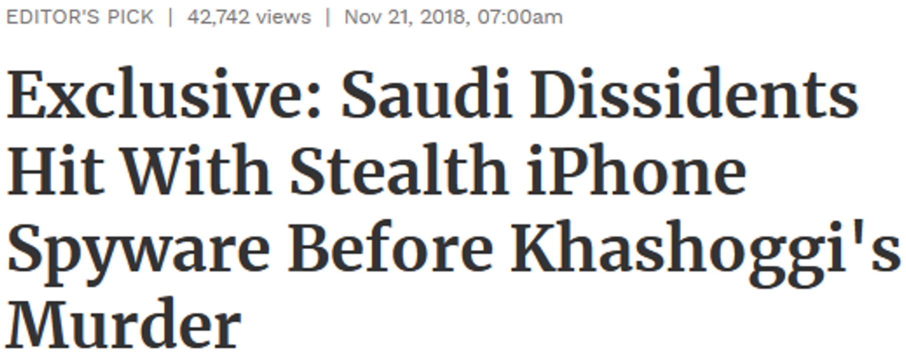
Memory Safety is a serious problem!



Computing Sep 6

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

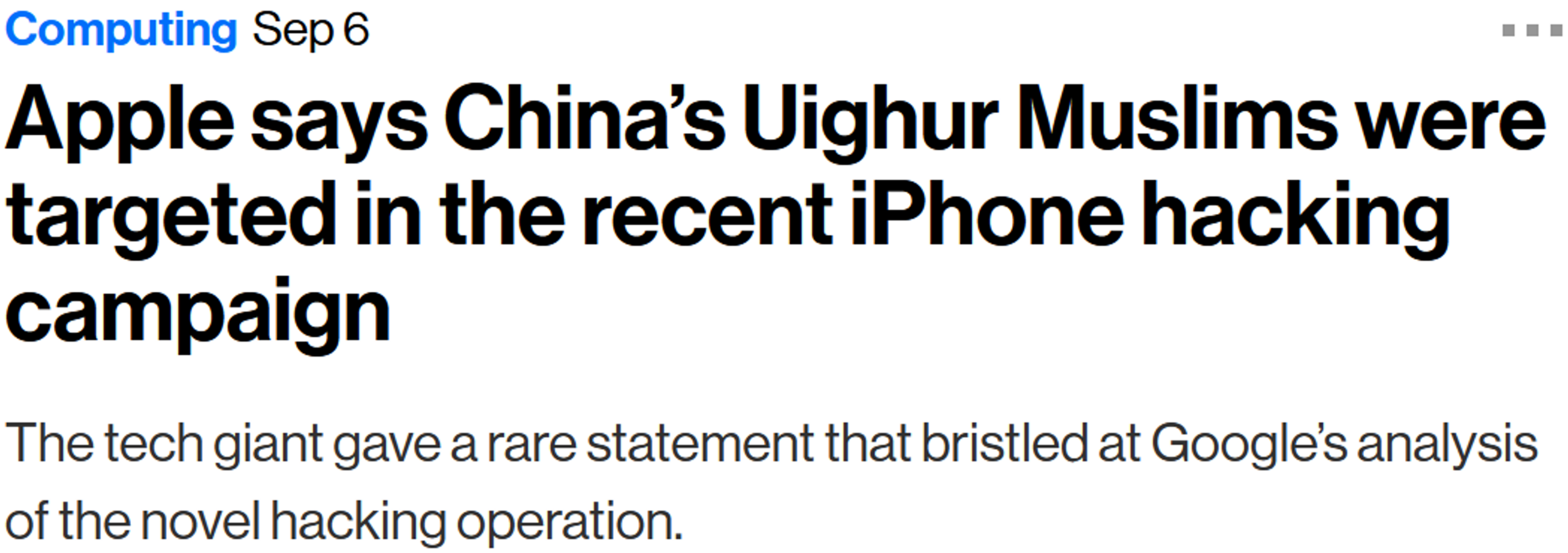
The tech giant gave a rare statement that bristled at Google’s analysis of the novel hacking operation.



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

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

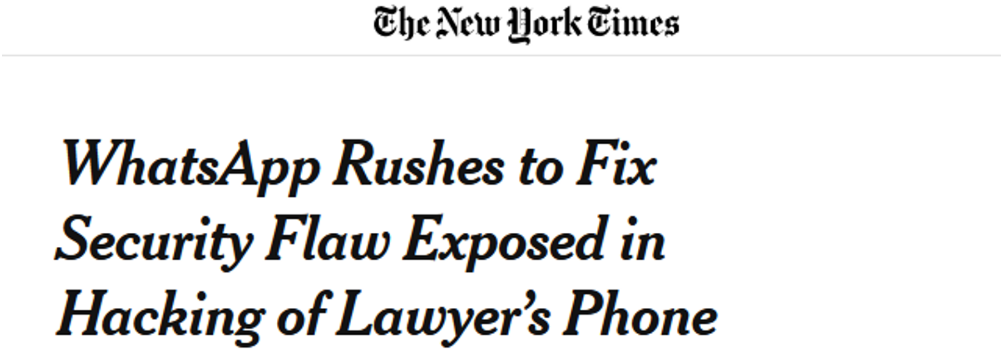
Memory Safety is a serious problem!



Computing Sep 6

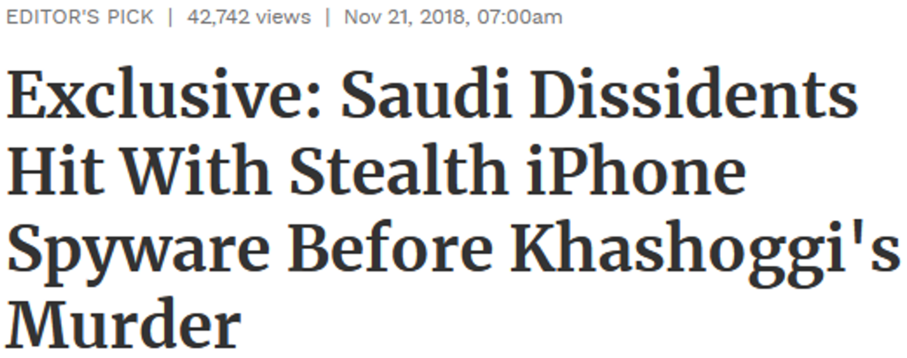
Apple says China's Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google's analysis of the novel hacking operation.



The New York Times

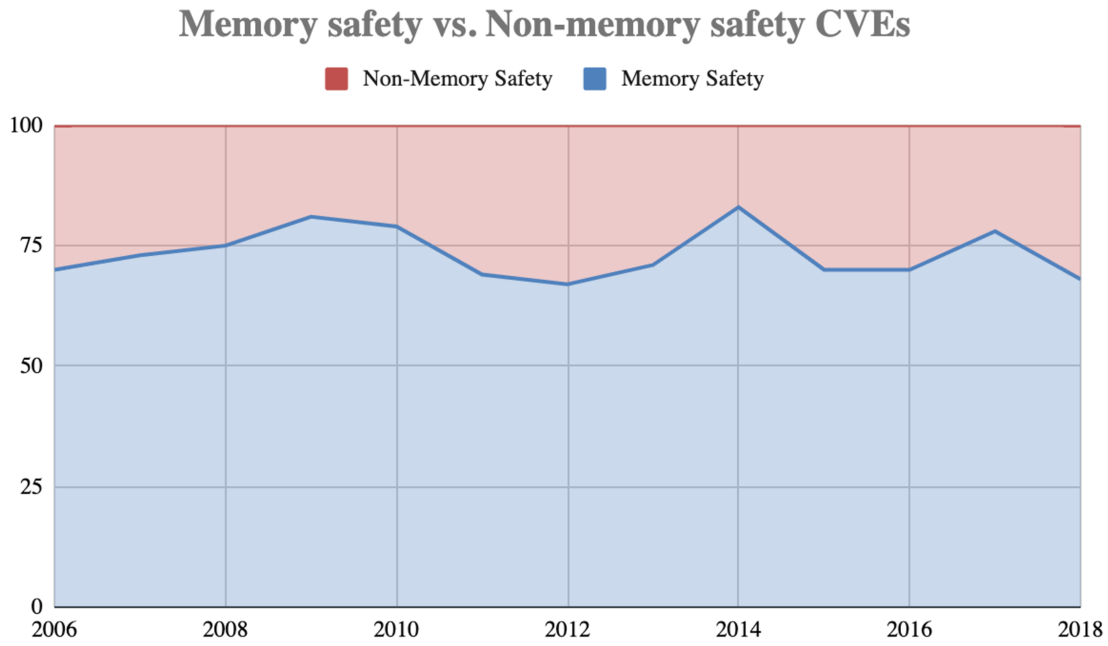
WhatsApp Rushes to Fix Security Flaw Exposed in Hacking of Lawyer's Phone



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

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

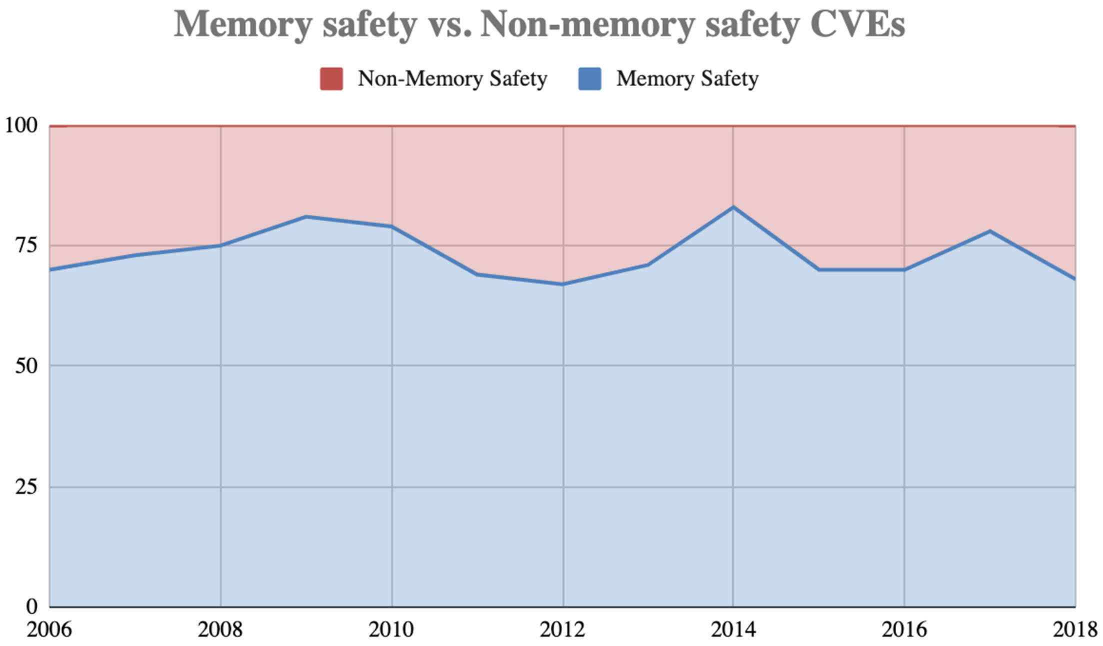
Prevalence of Memory Safety Vulns



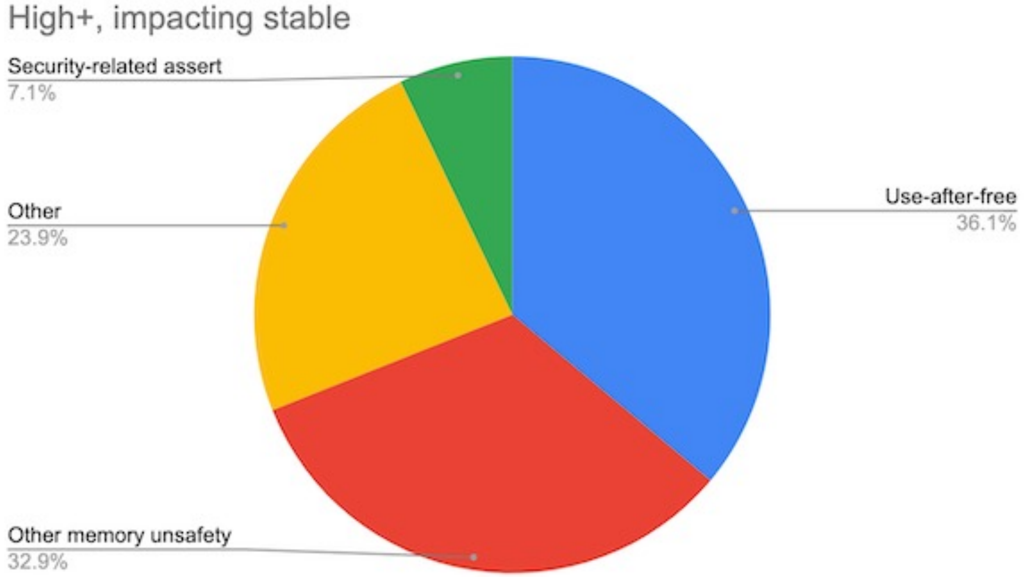
Microsoft Product CVEs
between 2006-2018

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

Prevalence of Memory Safety Vulns



Microsoft Product CVEs between 2006-2018



Chromium high severity security bugs between 2015-2020

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

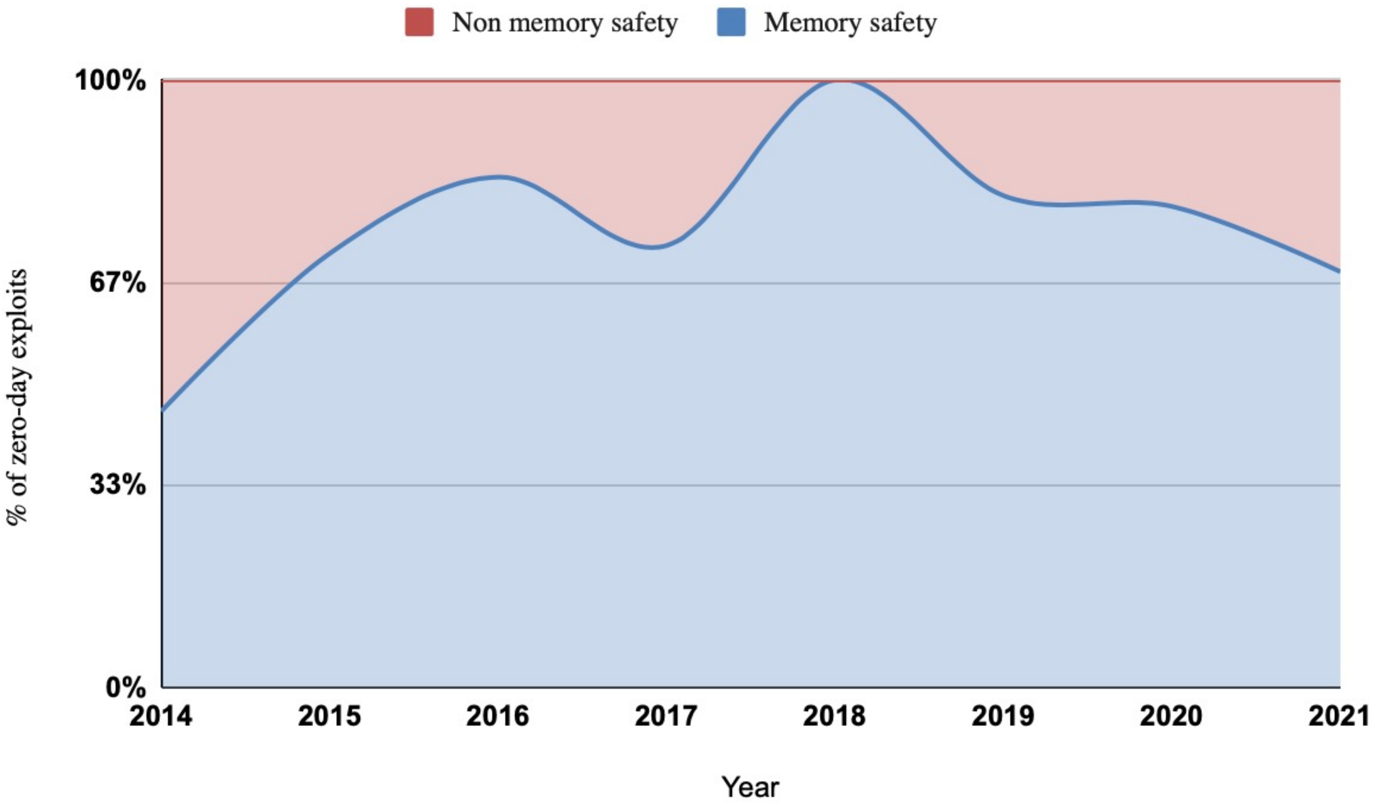
Source: *https://www.chromium.org/Home/chromium-security/memory-safety/*

ATTACKERS

❤

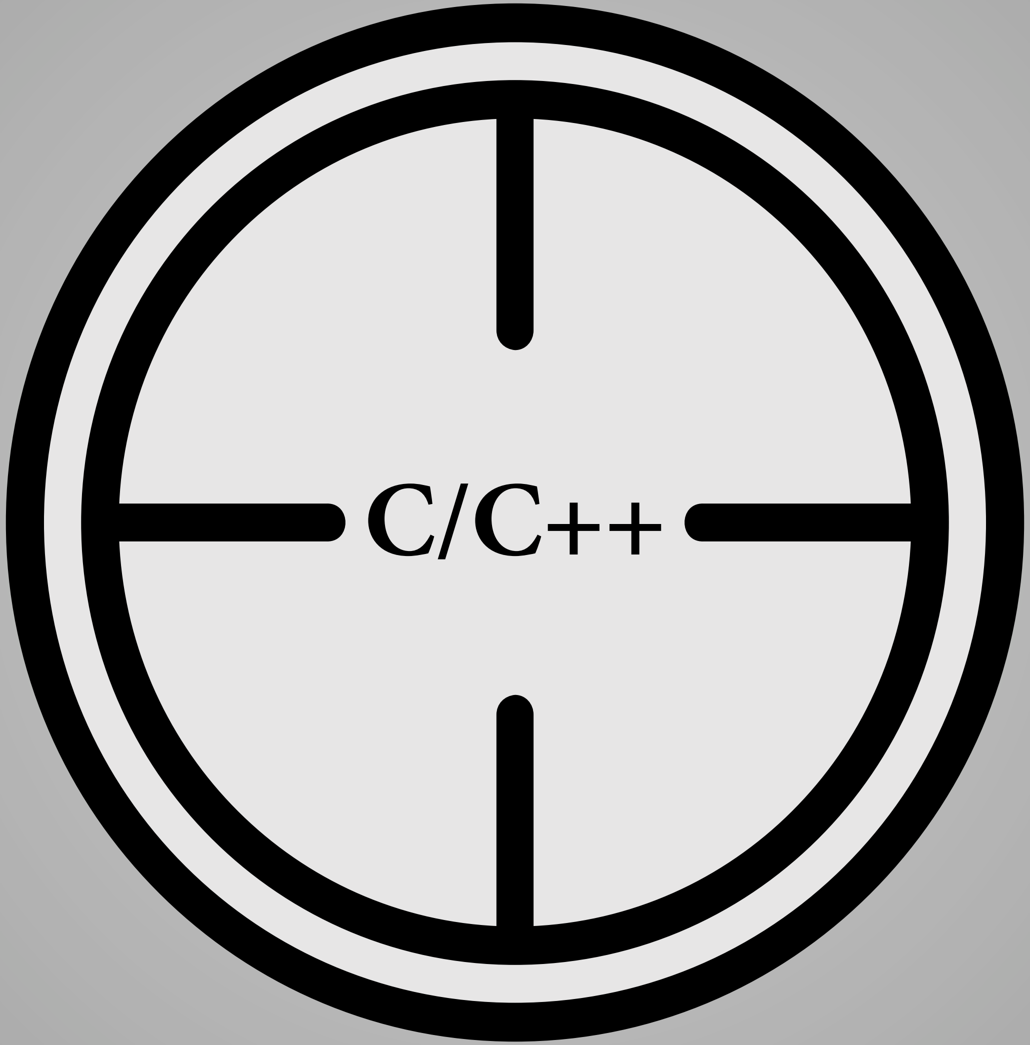
MEMORY SAFETY

Attackers prefer Memory Safety Vulns



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

Source: *Google Project Zero, 0day "In the Wild" spreadsheet. Last updated: January 2022*



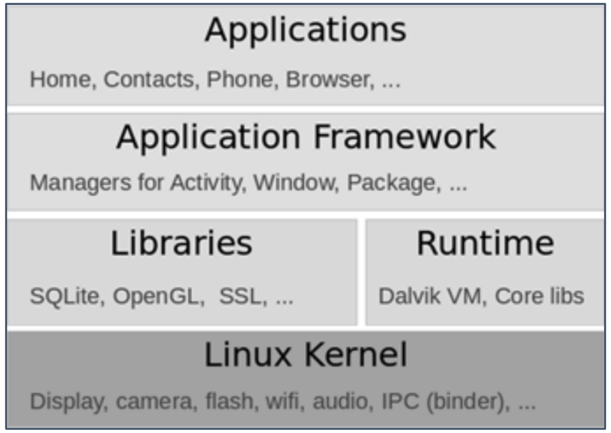
C/C++

C/C++ is here to stay!

C/C++ is here to stay!

C/C++ is here to stay!

C/C++ is here to stay!



Applications

Home, Contacts, Phone, Browser, ...

Application Framework

Managers for Activity, Window, Package, ...

Libraries

SQLite, OpenGL, SSL, ...

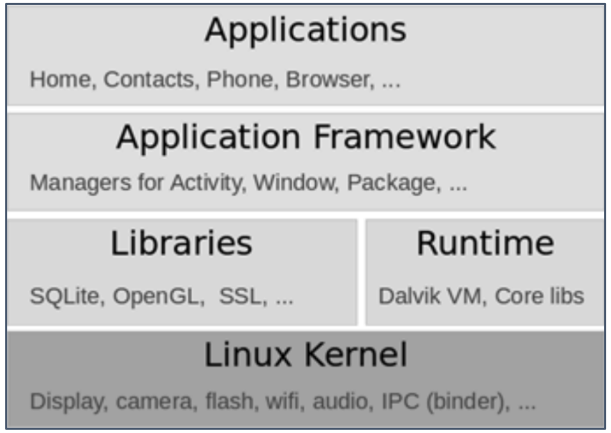
Runtime

Dalvik VM, Core libs

Linux Kernel

Display, camera, flash, wifi, audio, IPC (binder), ...

C/C++ is here to stay!


Applications

Home, Contacts, Phone, Browser, ...

Application Framework

Managers for Activity, Window, Package, ...

Libraries

SQLite, OpenGL, SSL, ...

Runtime

Dalvik VM, Core libs

Linux Kernel

Display, camera, flash, wifi, audio, IPC (binder), ...

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

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

Memory Blocklisting

Memory Permitlisting

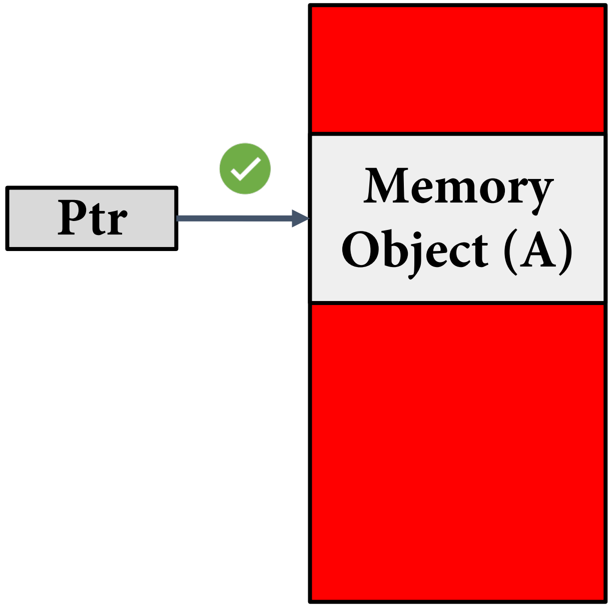
Exploit Mitigation

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

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

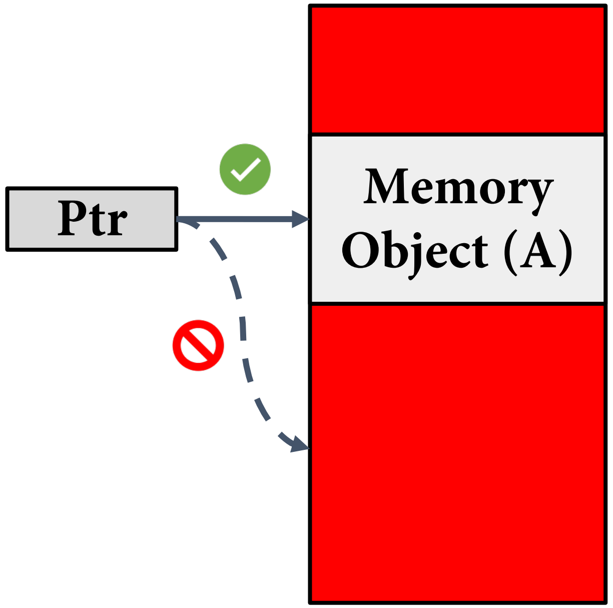


How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

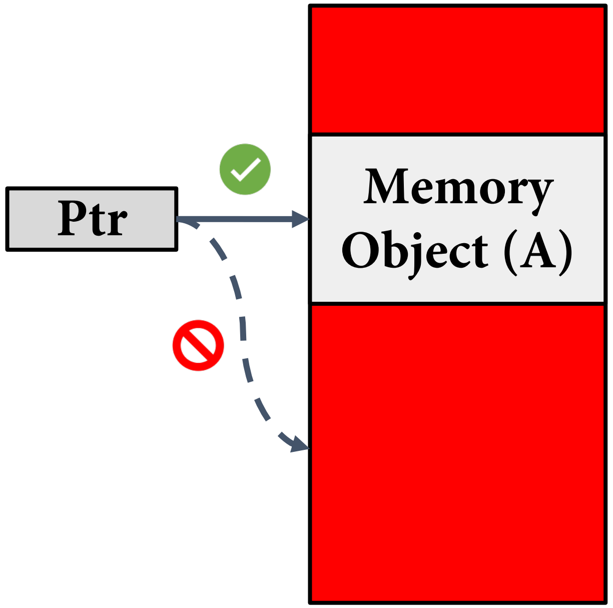


How to fix C/C++ memory (un)safety?

Memory Blocklisting

Memory Permitlisting

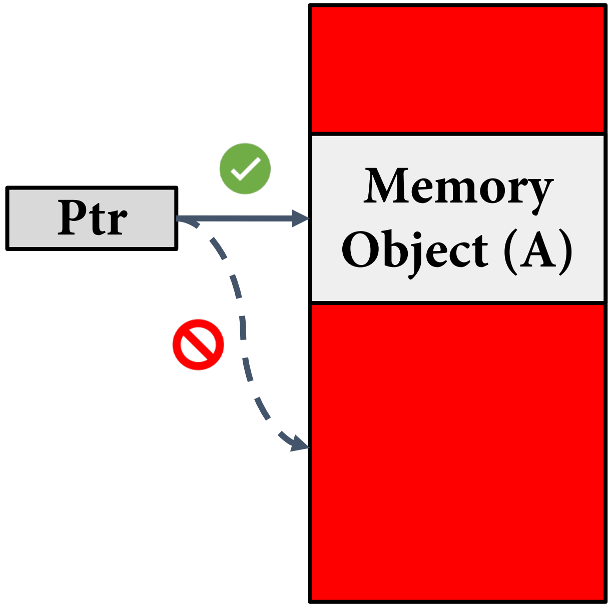
Exploit Mitigation



e.g., Google's Address Sanitizer

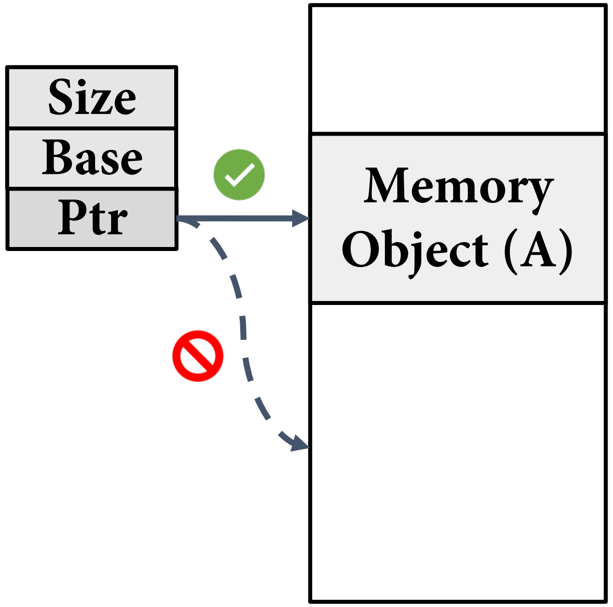
How to fix C/C++ memory (un)safety?

Memory Blocklisting



e.g., Google's Address Sanitizer

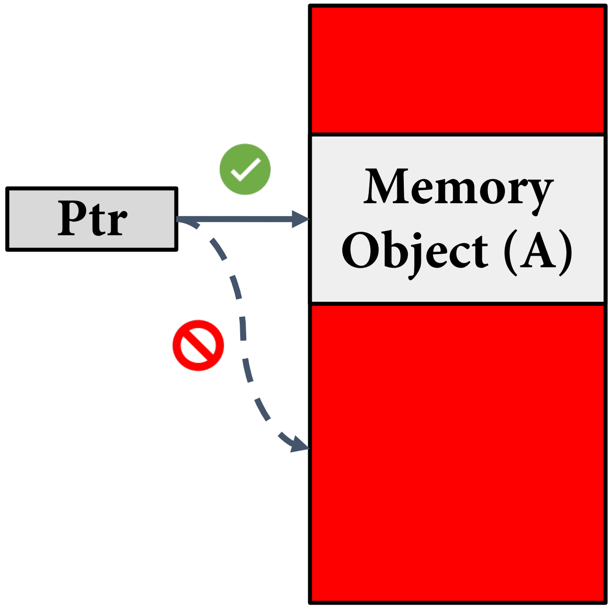
Memory Permitlisting



Exploit Mitigation

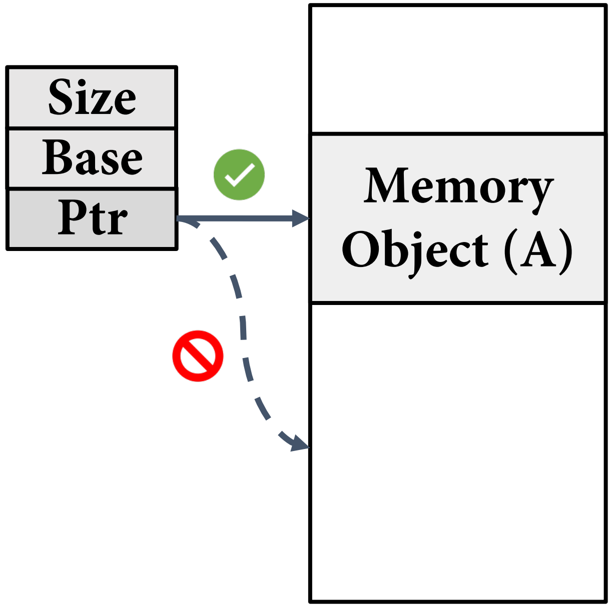
How to fix C/C++ memory (un)safety?

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

Memory Permitlisting

Exploit Mitigation

Ptr
M
Ob
Size

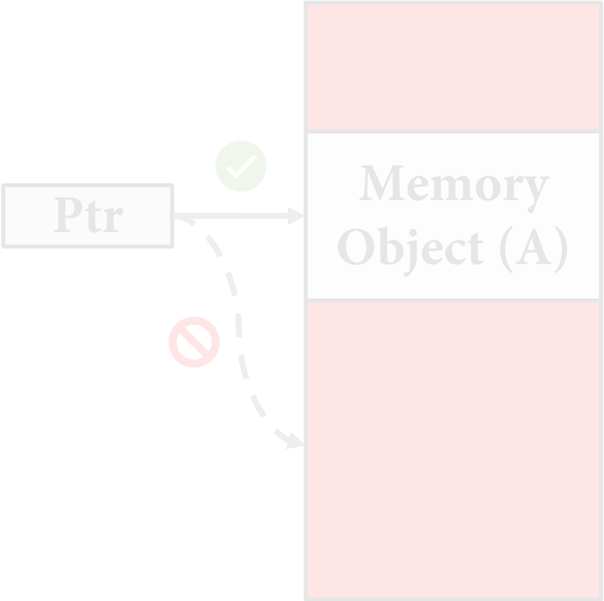
Enforcing strict memory safety rules comes with non-negligible performance costs!

e.g., Google's Address Sanitizer

e.g., Intel's MPX and CHERI

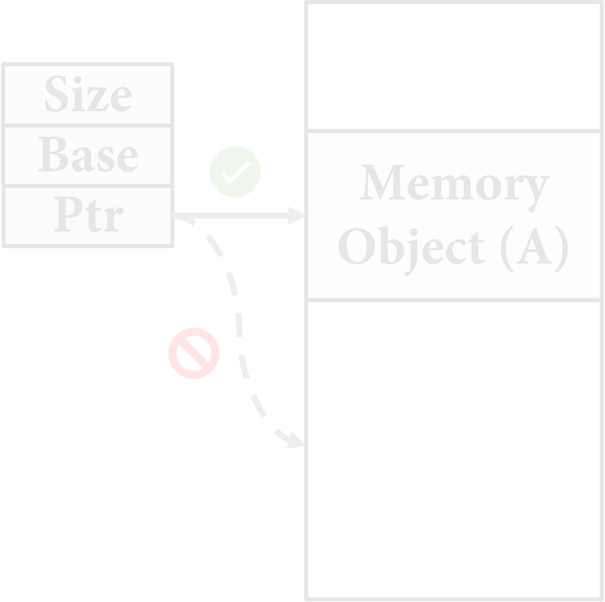
How to fix C/C++ memory (un)safety?

Memory Blocklisting



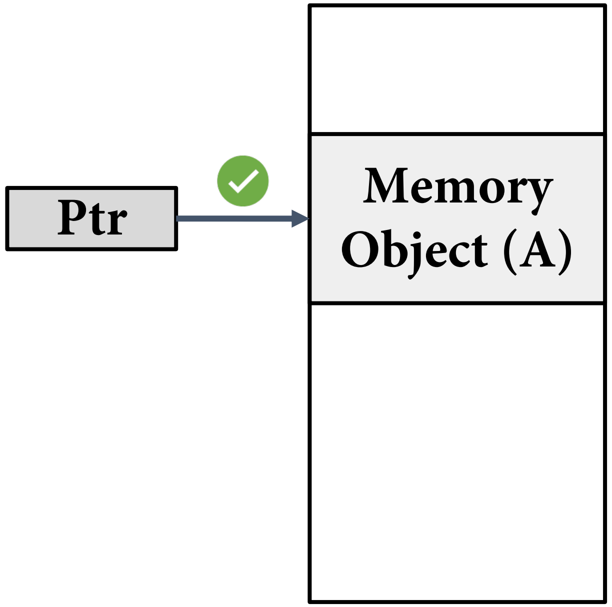
e.g., Google's Address Sanitizer

Memory Permitlisting



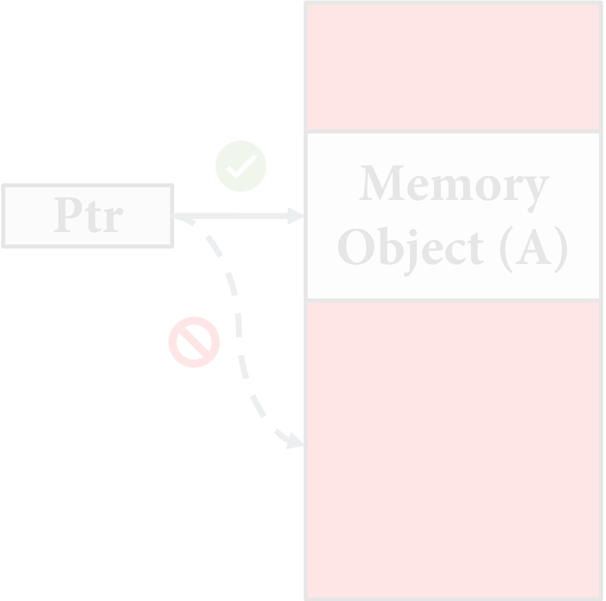
e.g., Intel's MPX and CHERI

Exploit Mitigation



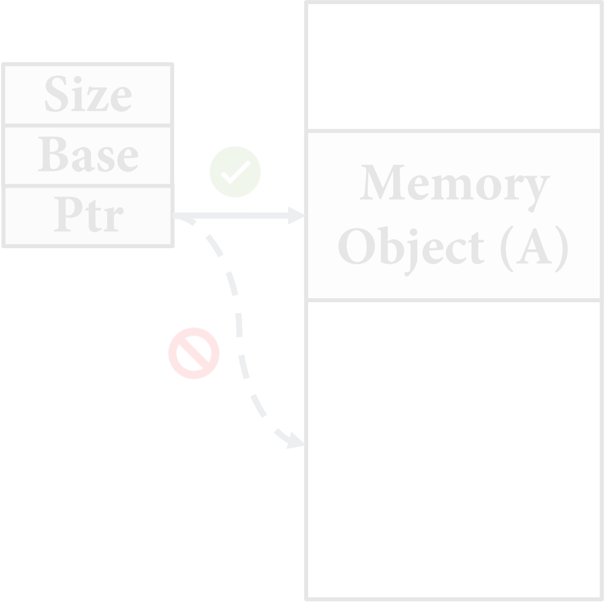
How to fix C/C++ memory (un)safety?

Memory Blocklisting



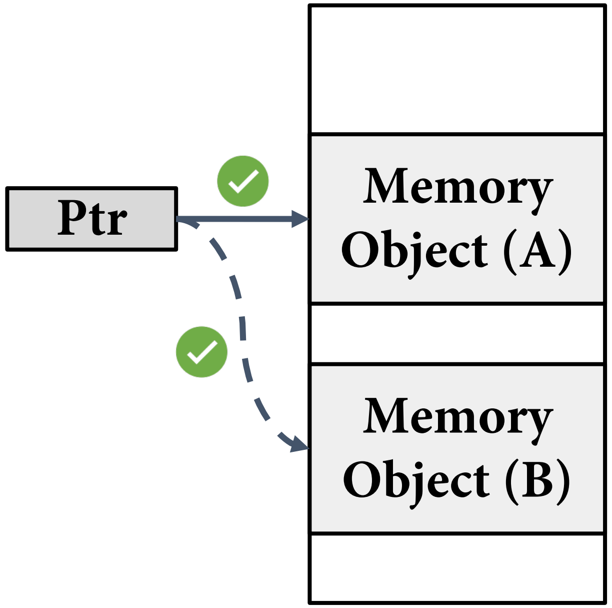
e.g., Google's Address Sanitizer

Memory Permitlisting



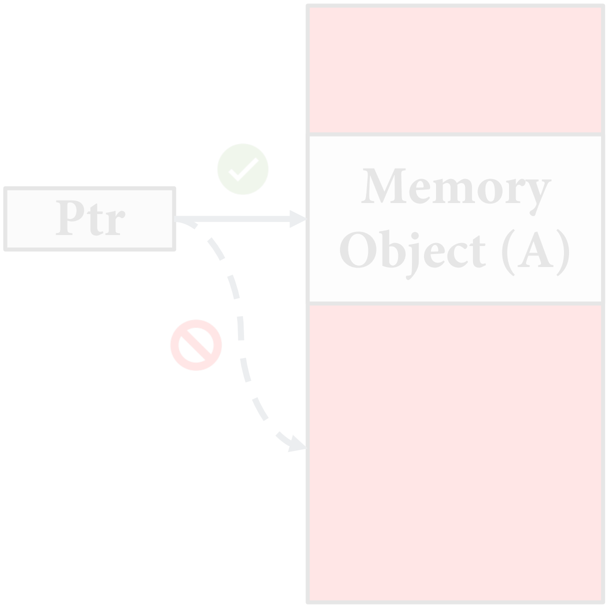
e.g., Intel's MPX and CHERI

Exploit Mitigation



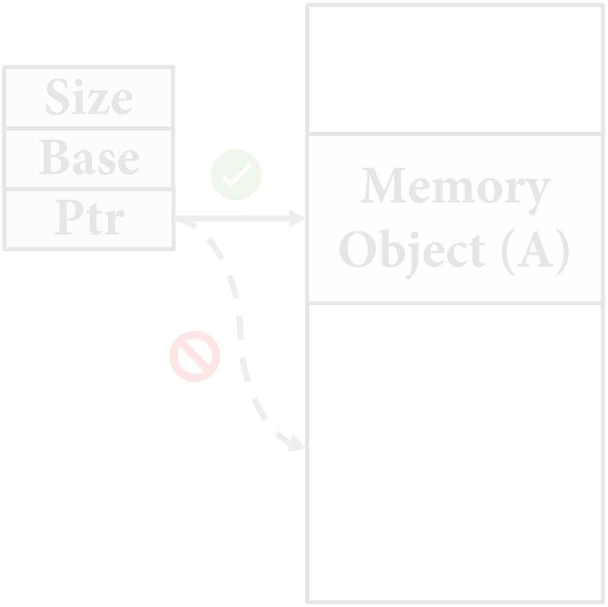
How to fix C/C++ memory (un)safety?

Memory Blocklisting



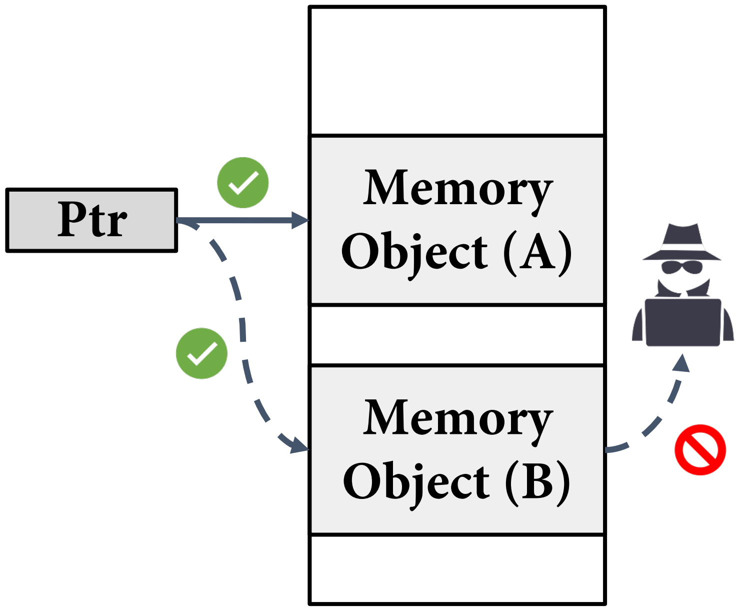
e.g., Google's Address Sanitizer

Memory Permitlisting



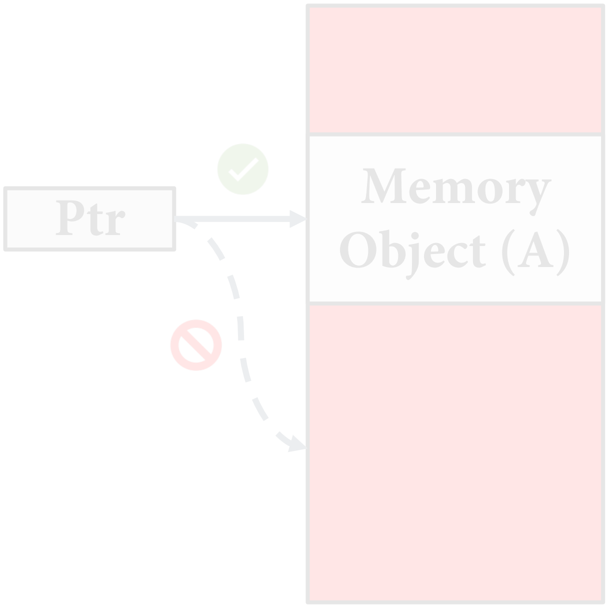
e.g., Intel's MPX and CHERI

Exploit Mitigation



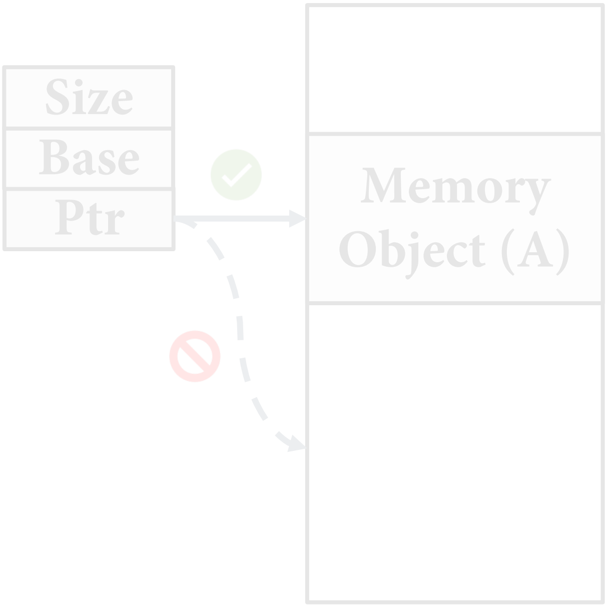
How to fix C/C++ memory (un)safety?

Memory Blocklisting



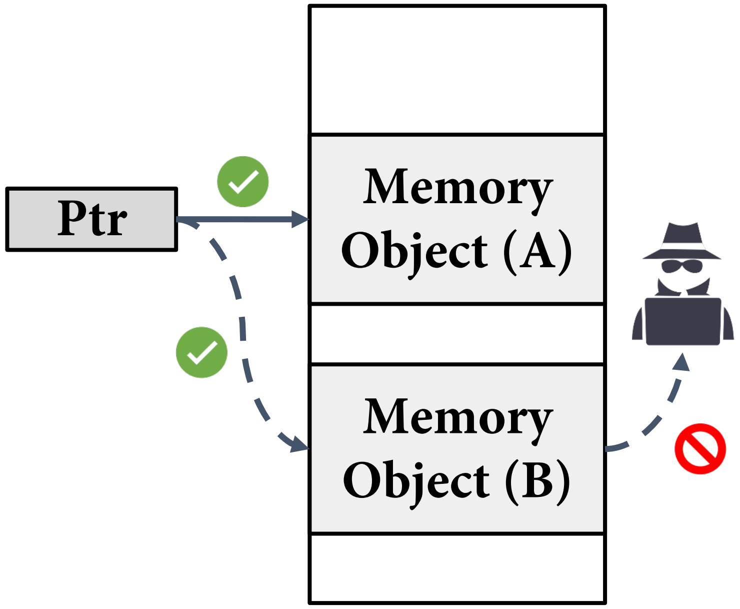
e.g., Google’s Address Sanitizer

Memory Permitlisting



e.g., Intel’s MPX and CHERI

Exploit Mitigation



e.g., ARM’s PAC

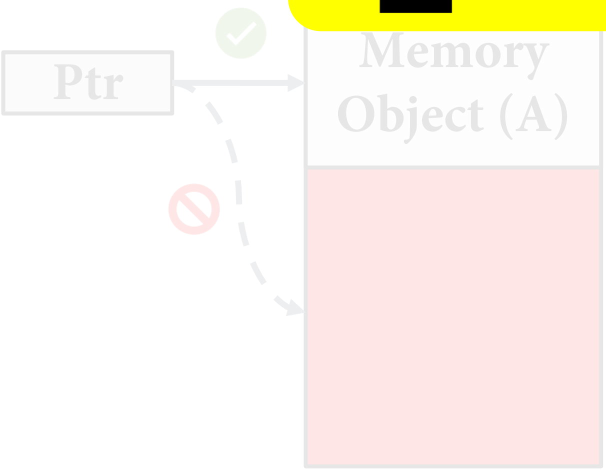
How to fix C/C++ memory (un)safety?

Memory Blocklisting

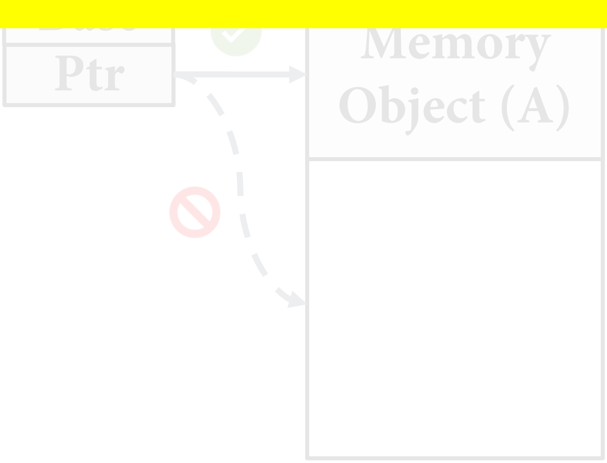
Memory Permitlisting

Exploit Mitigation

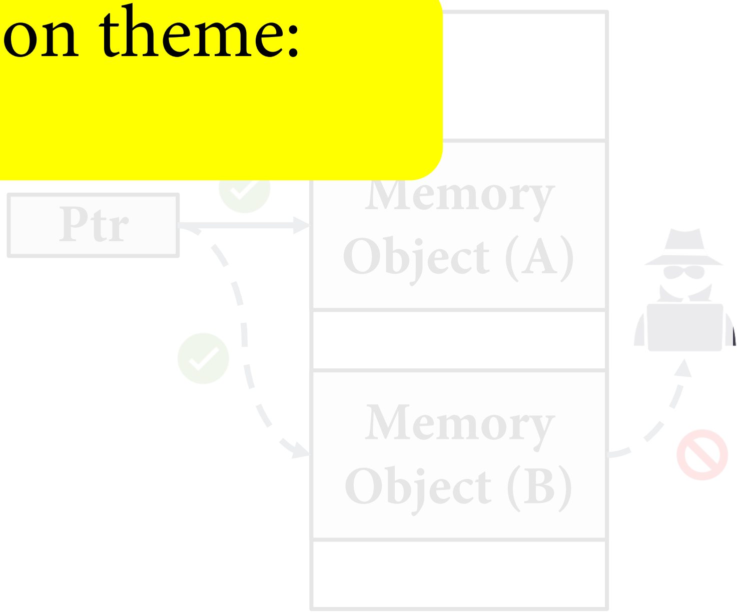
All prior approaches share a common theme:



e.g., Google's Address Sanitizer



e.g., Intel's MPX and CHERI



e.g., ARM's PAC

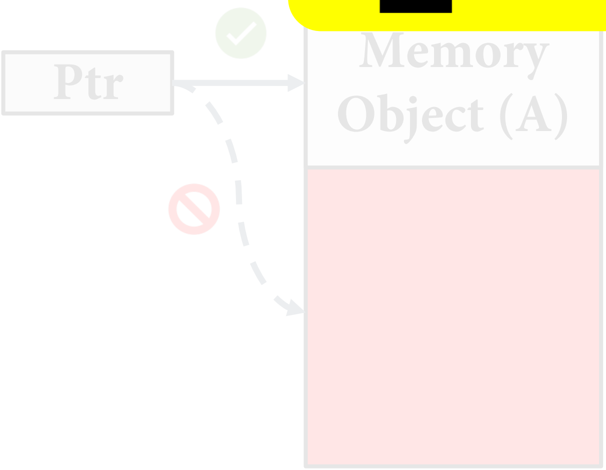
How to fix C/C++ memory (un)safety?

Memory Blocklisting

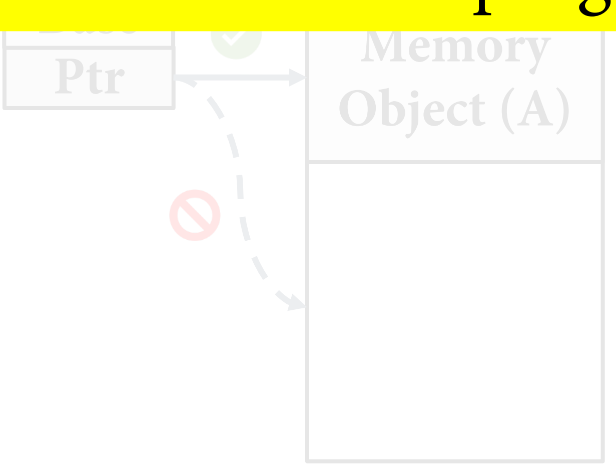
Memory Permitlisting

Exploit Mitigation

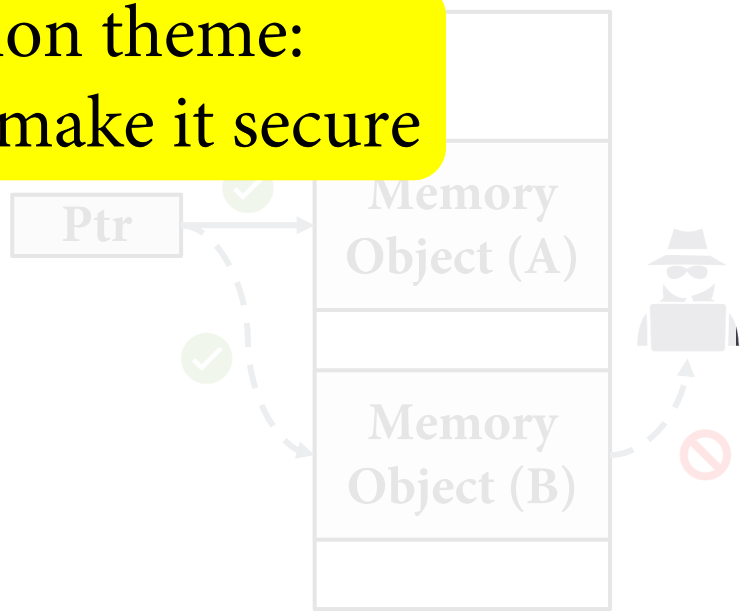
All prior approaches share a common theme:
Adding more features to a program to make it secure



e.g., Google’s Address Sanitizer



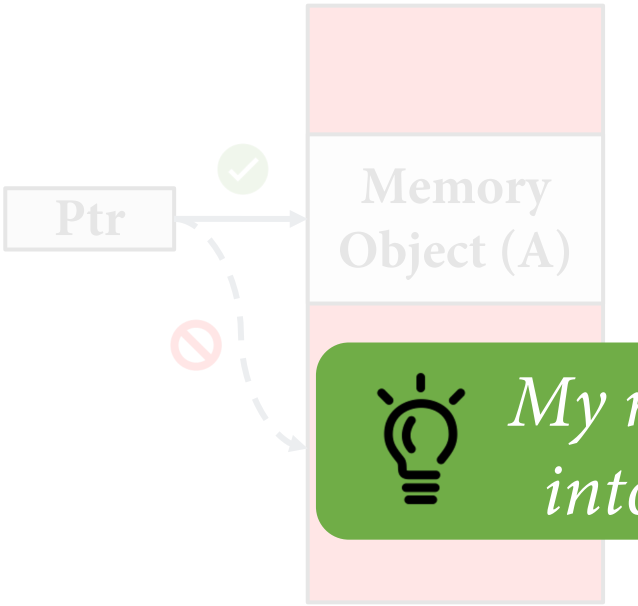
e.g., Intel’s MPX and CHERI



e.g., ARM’s PAC

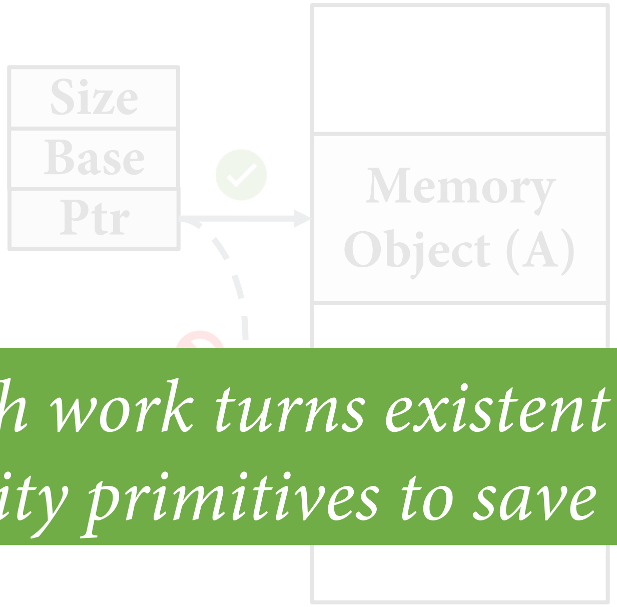
My solutions for C/C++ memory (un)safety

Memory Blocklisting



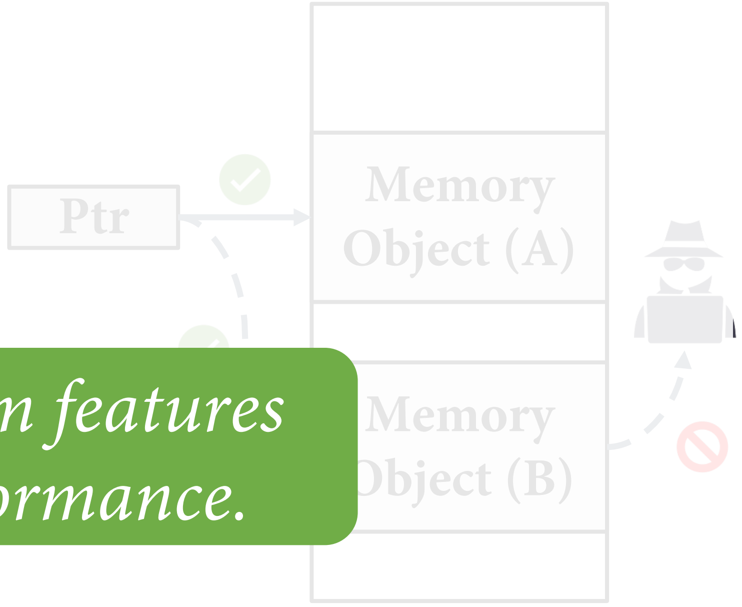
e.g., Google's Address Sanitizer

Memory Permitlisting



e.g., Intel's MPX and CHERI

Exploit Mitigation



e.g., ARM's PAC

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

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

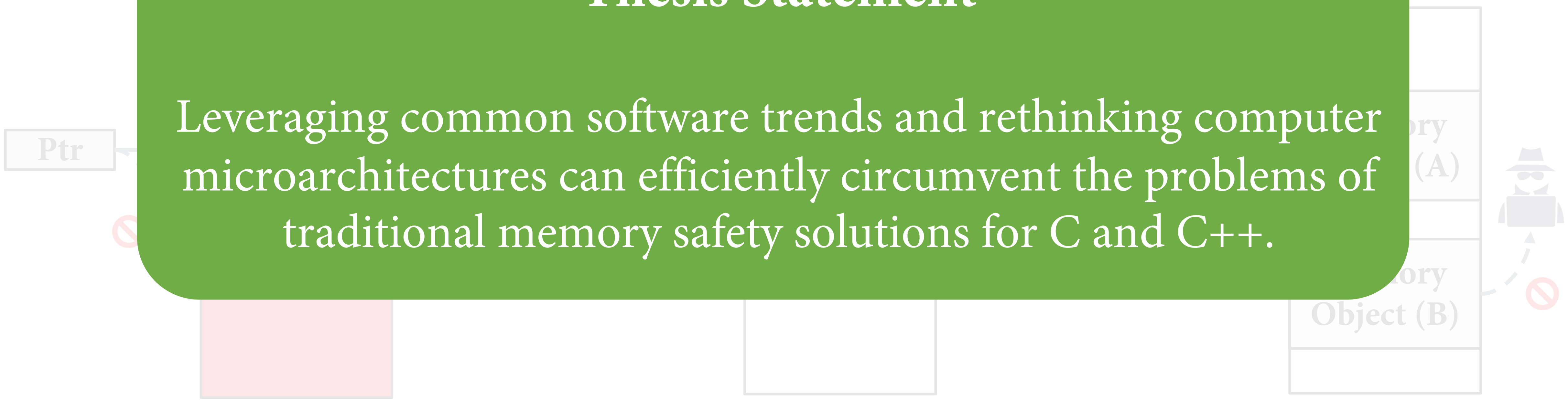
Memory Blocklisting

Memory Permitlisting

Exploit Mitigation

Thesis Statement

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



e.g., Google's Address Sanitizer

e.g., Intel's MPX and CHERI

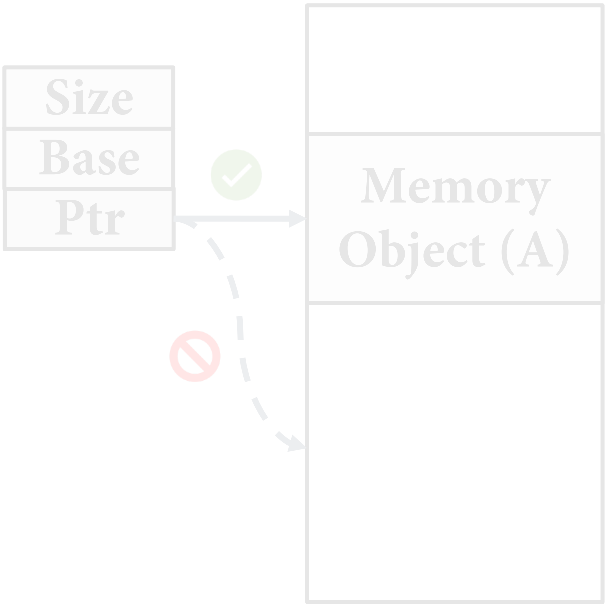
e.g., ARM's PAC

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

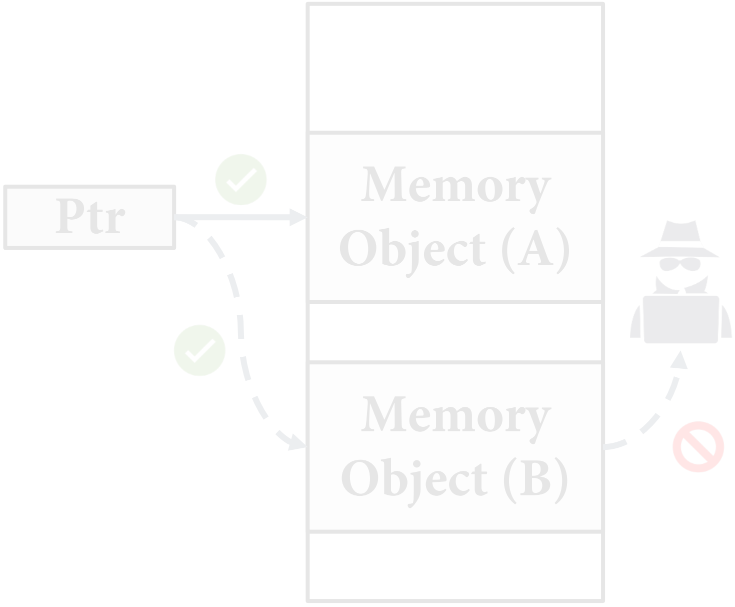
Memory Blocklisting

*Uses **dead bytes** in program memory*

Memory Permitlisting



Exploit Mitigation

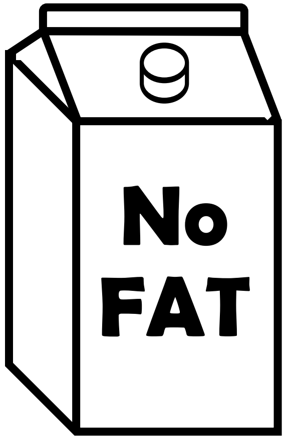


Practical byte-granular memory blacklisting using Califorms. [MICRO 2019]

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

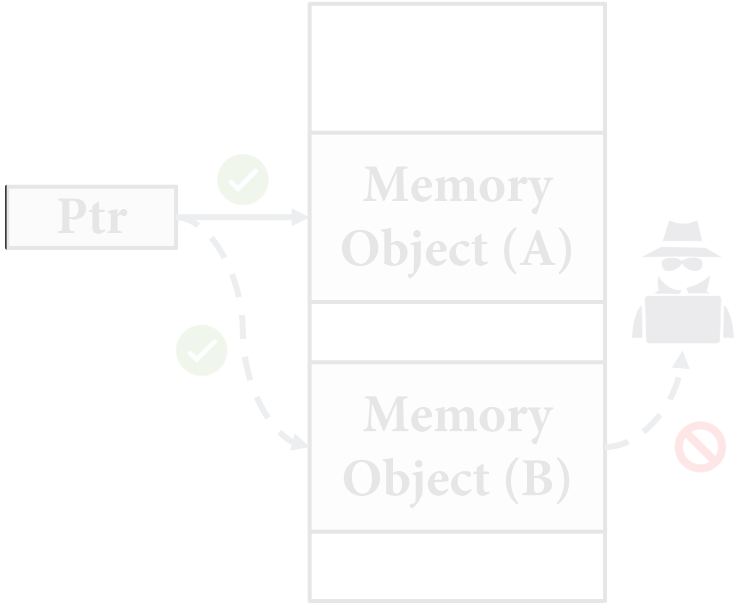
Memory Blocklisting

Memory Permitlisting



Leverages modern software trends

Exploit Mitigation

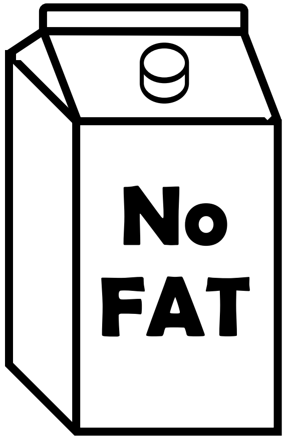


Architectural Support for Low Overhead Memory Safety Checks. [ISCA 2021]

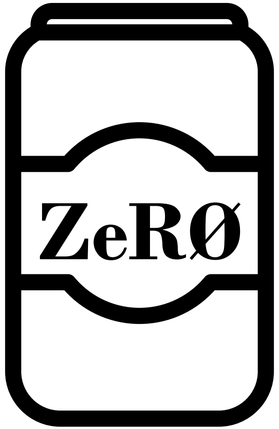
My solutions for C/C++ memory (un)safety

Memory Blocklisting

Memory Permitlisting



Exploit Mitigation



*Mitigates all known exploits with **zero** runtime overheads.*

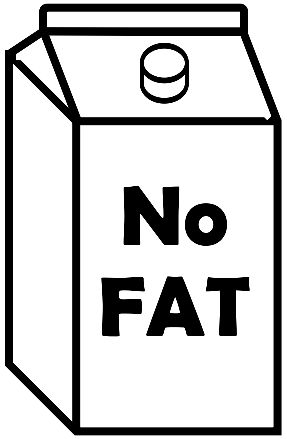
ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks. [ISCA 2021]

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

Memory Blocklisting

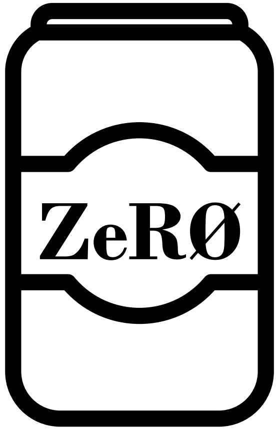
[MICRO 2019]

Memory Permitlisting



[ISCA 2021]

Exploit Mitigation

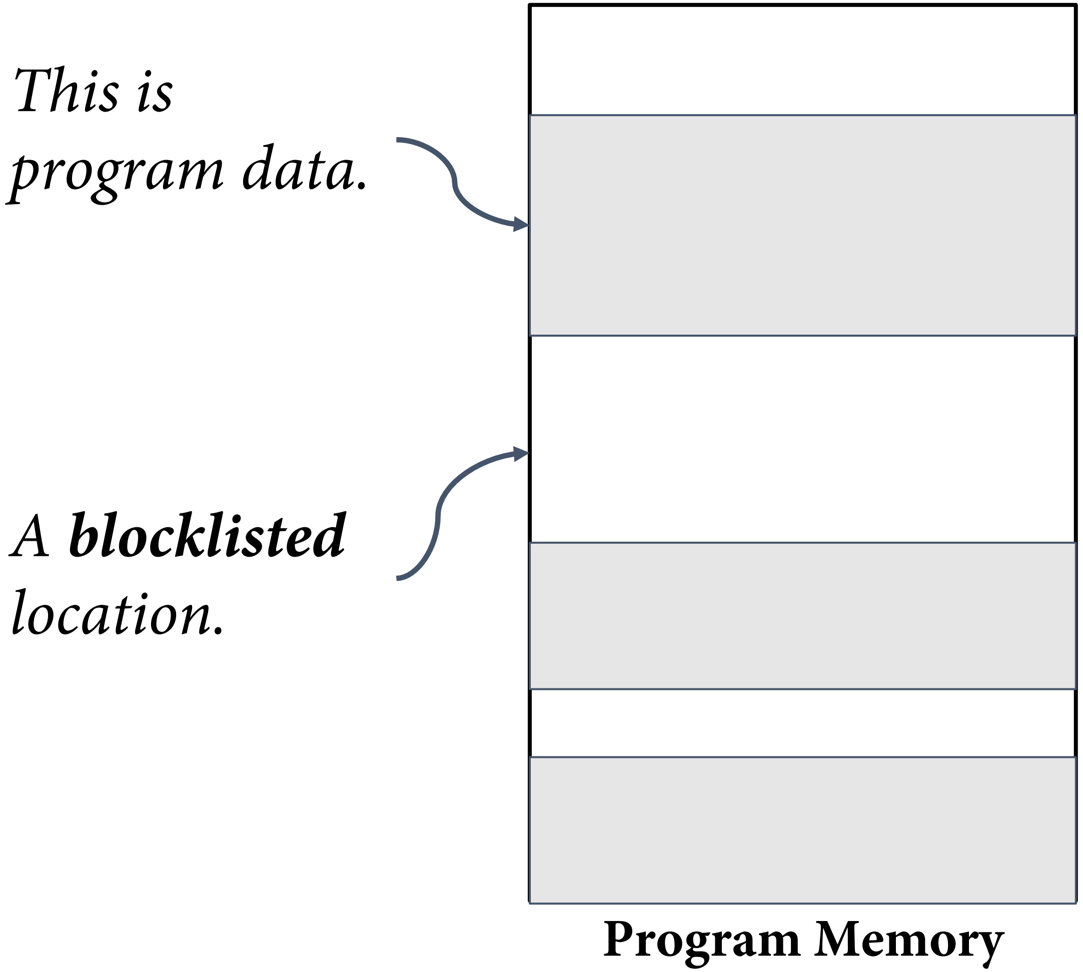


[ISCA 2021]

Cache Line Formats

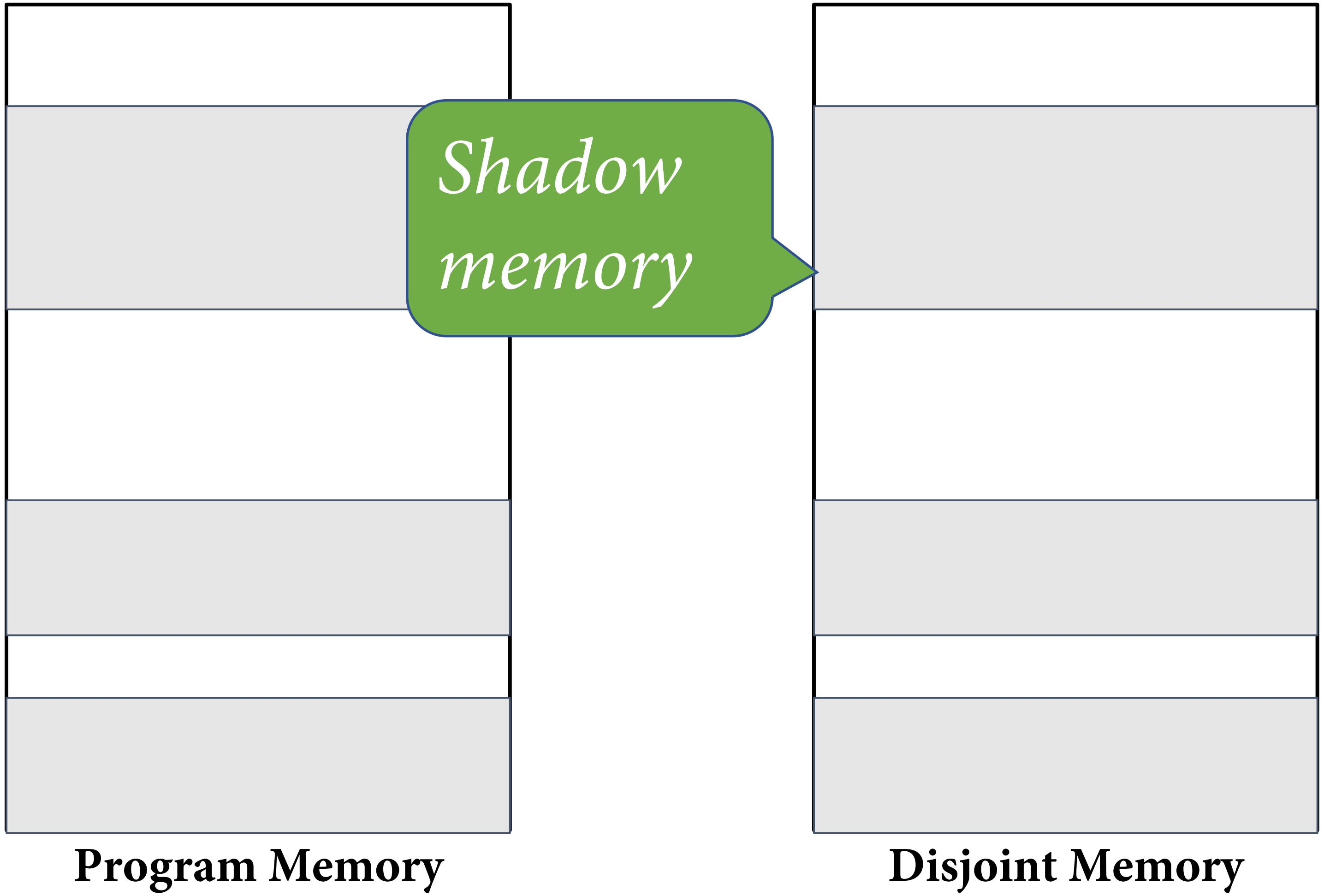
Hiroshi Sasaki, Miguel A. Arroyo, **Mohamed Tarek Ibn Ziad**, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan, Practical byte-granular memory blacklisting using Califorms. [MICRO 2019] [**IEEE Micro Top Picks Honorable Mention**]

CaLiForms Memory Blocklisting

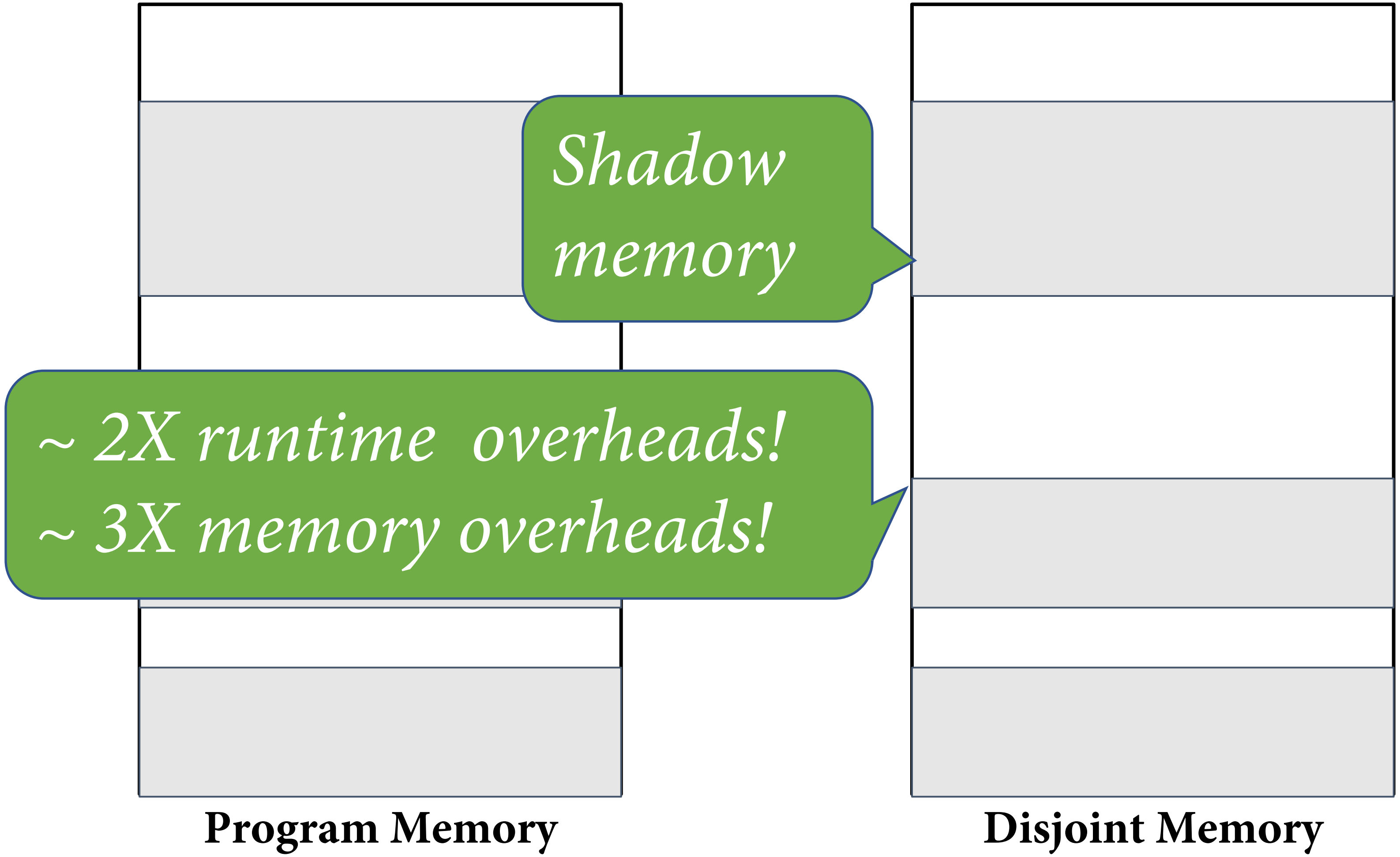


<u>Challenge</u>
How to efficiently track the state of memory locations?

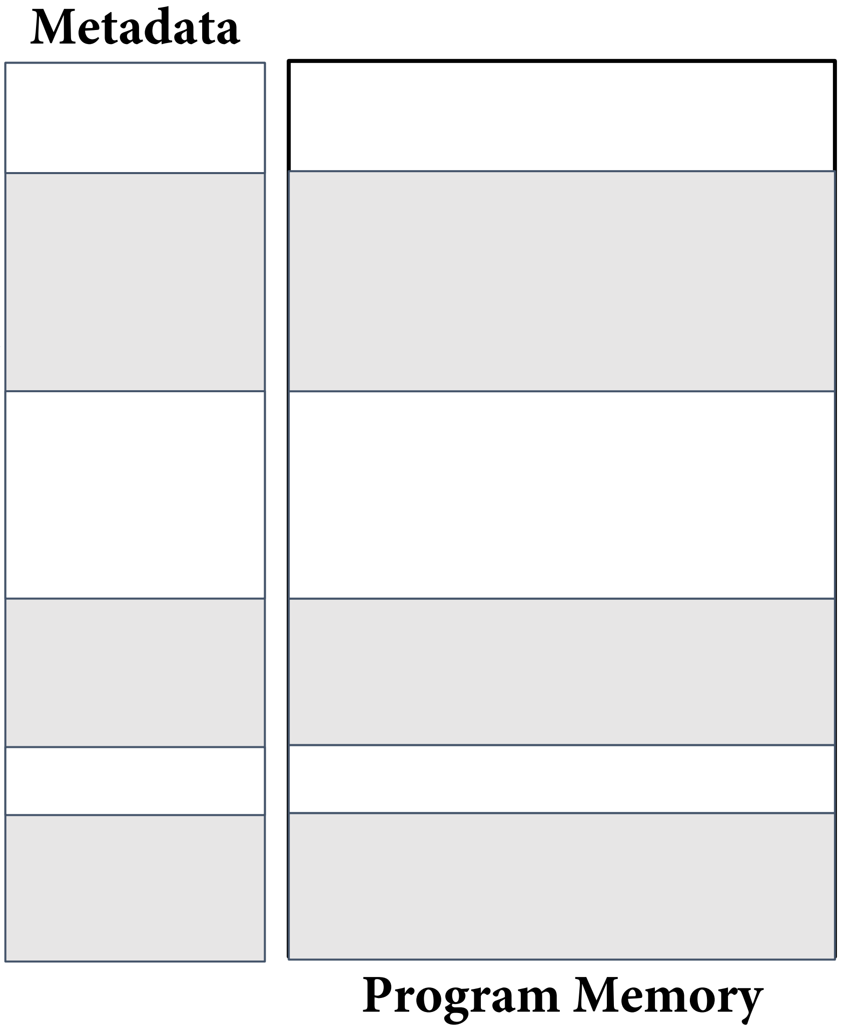
CaLiForms Memory Blocklisting



CaLiForms Memory Blocklisting

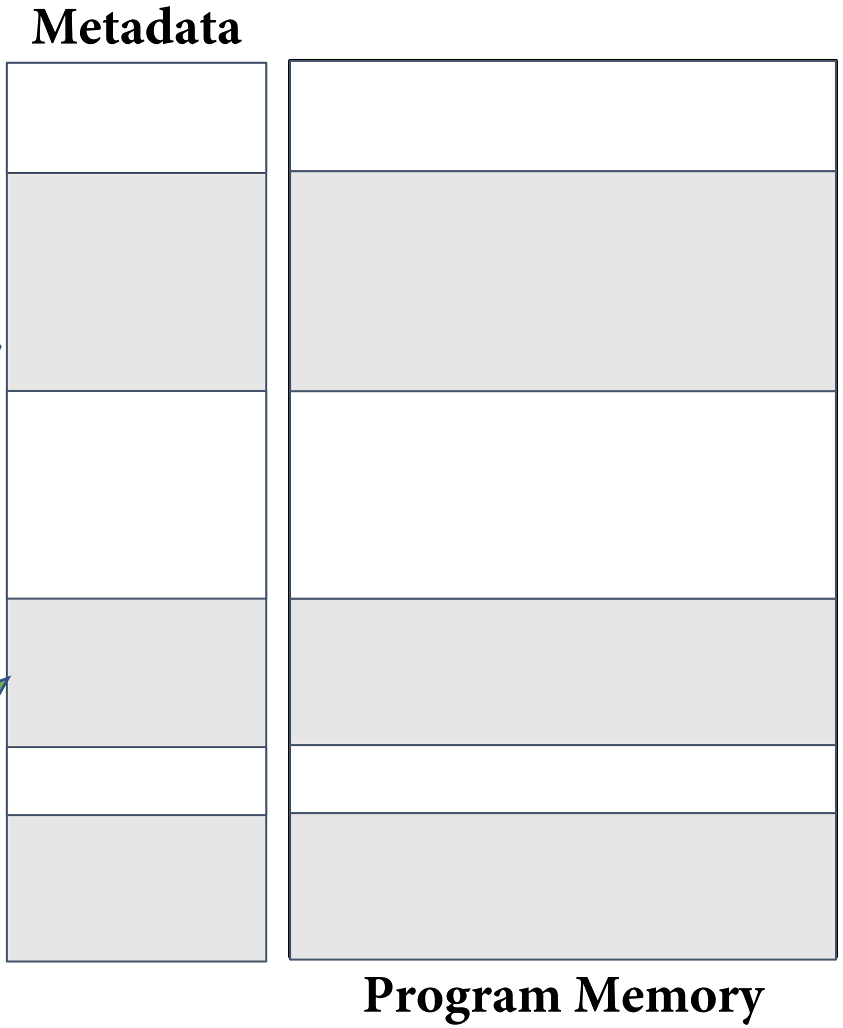


CaLiForms Memory Blocklisting



Memory Tagging
n bits per cache line

CaLiForms Memory Blocklisting

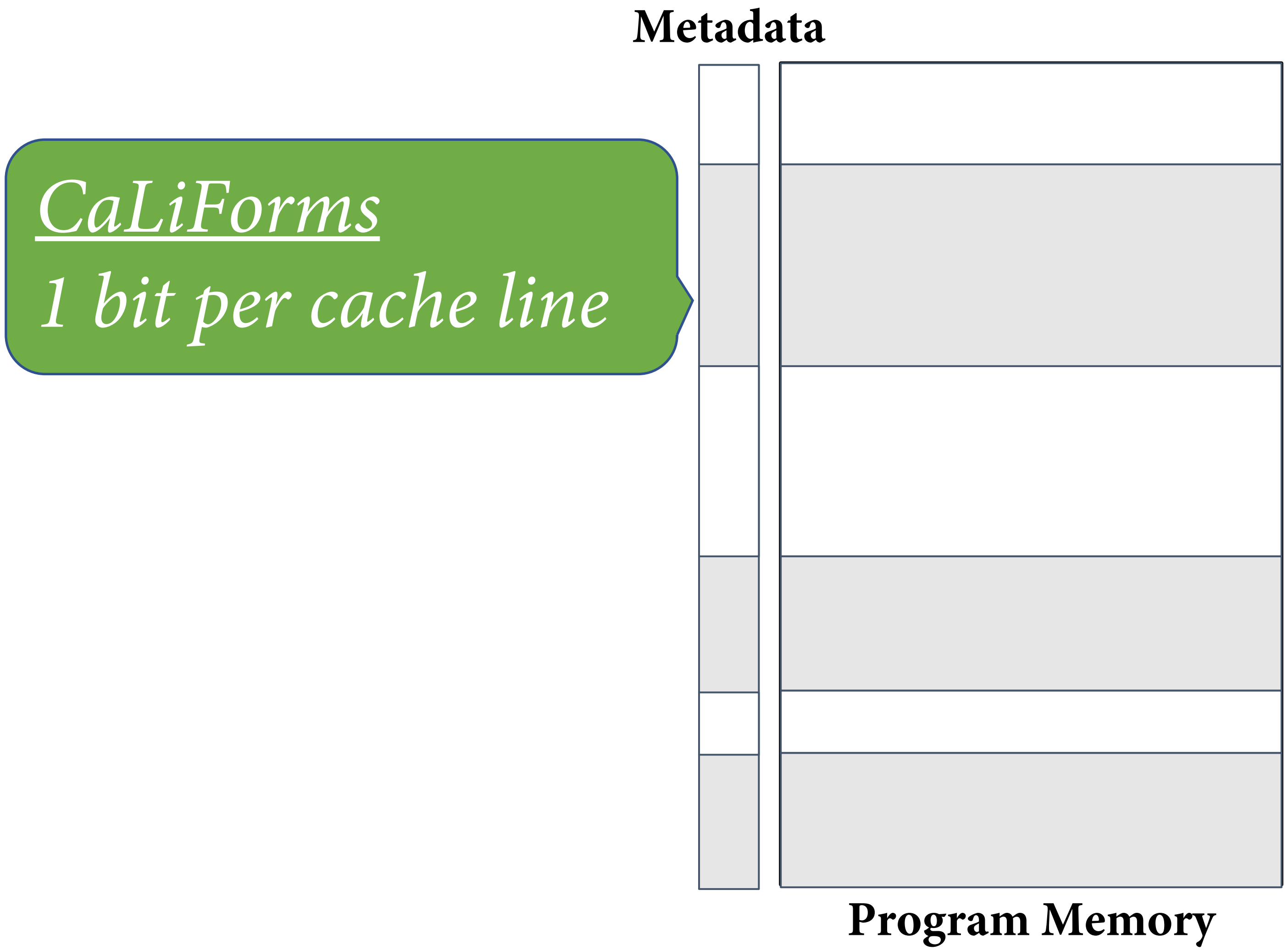


Memory Tagging

n bits per cache line

Limited entropy!

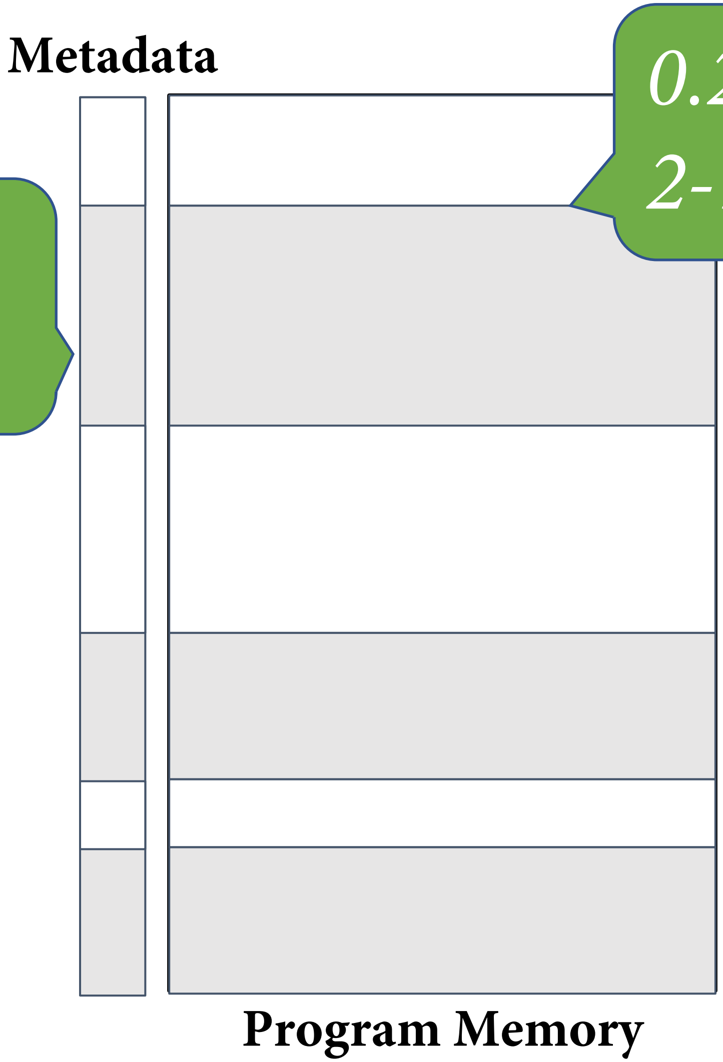
CaLiForms Memory Blocklisting



CaLiForms

1 bit per cache line

CaLiForms Memory Blocklisting



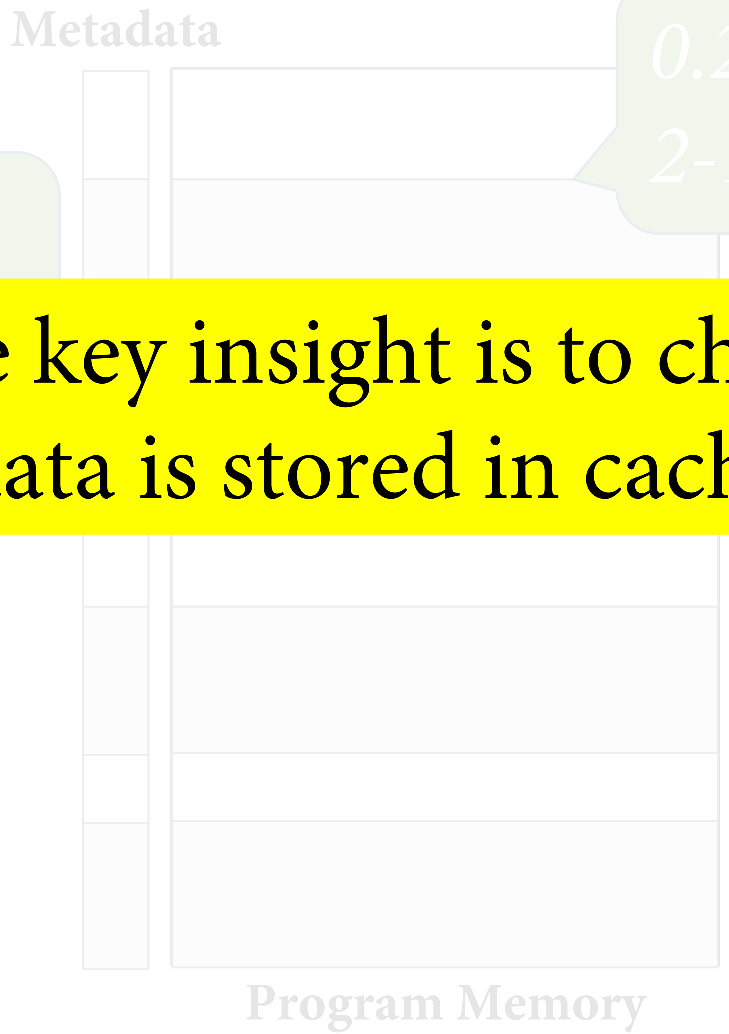
CaLiForms

1 bit per cache line

0.2%   memory overhead!

2-14% runtime overhead!

CaLiForms Memory Blocklisting



0.2% memory overhead!

2-14% runtime overhead!

CaLiForms

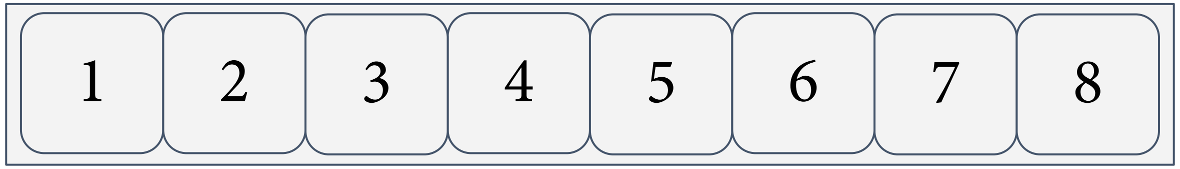
1 bit p

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

CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

Normal



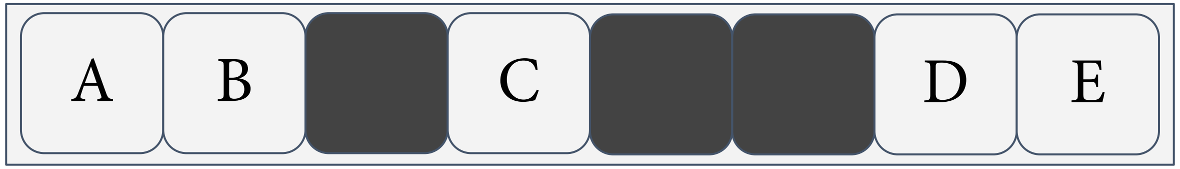
1 2 3 4 5 6 7 8

CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

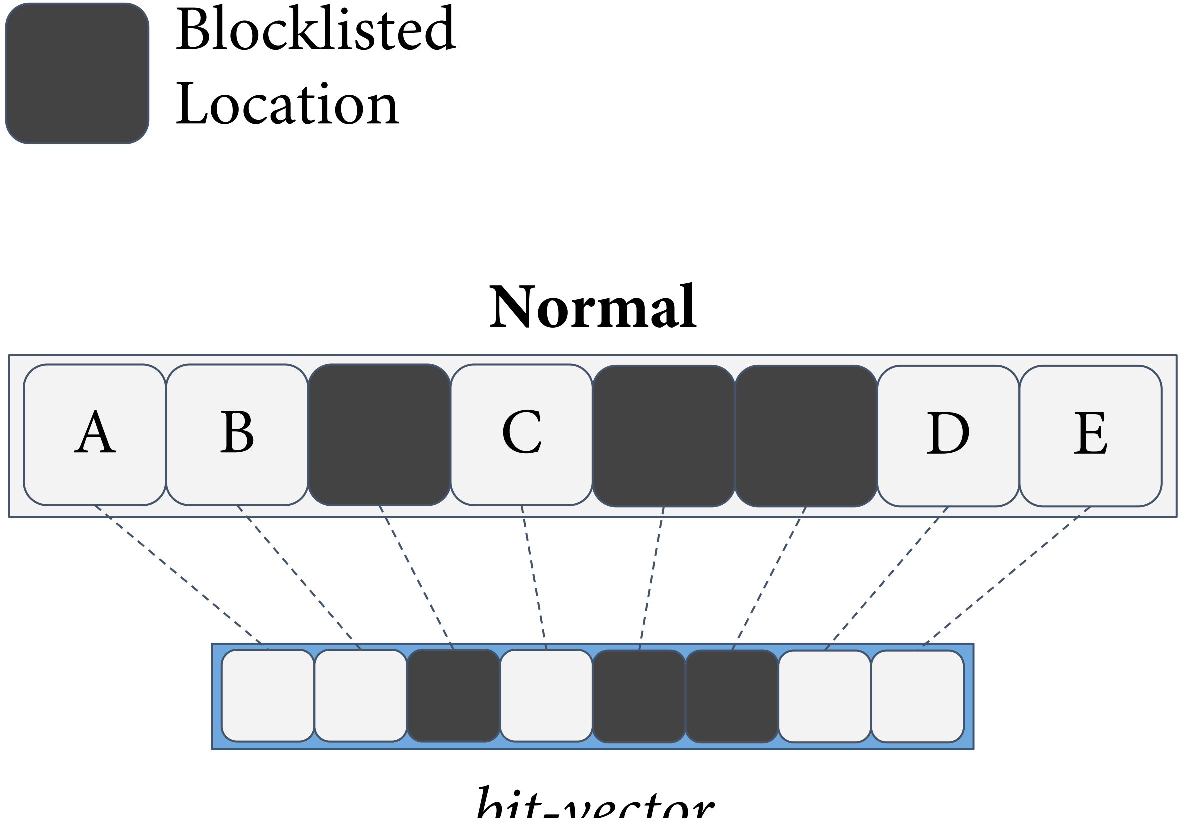
■ Blocklisted Location

Normal



CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

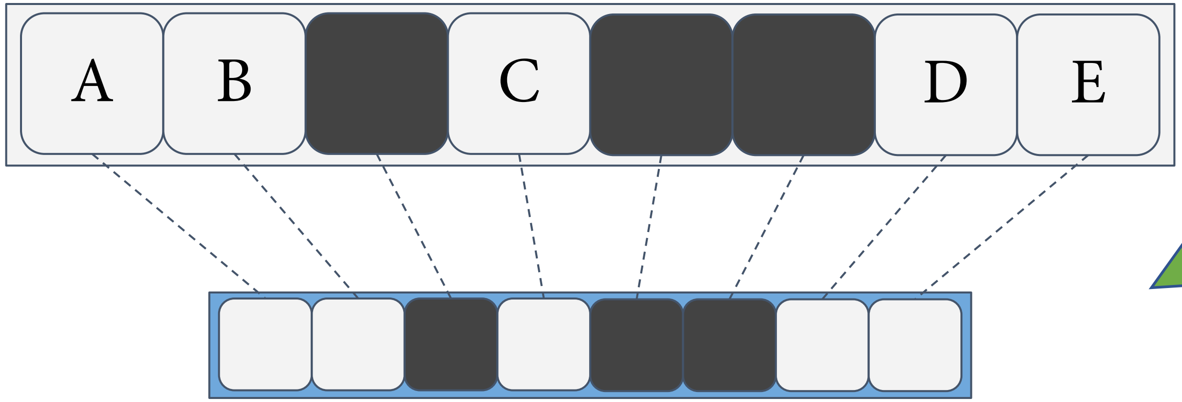


CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

■ Blocklisted Location

Normal



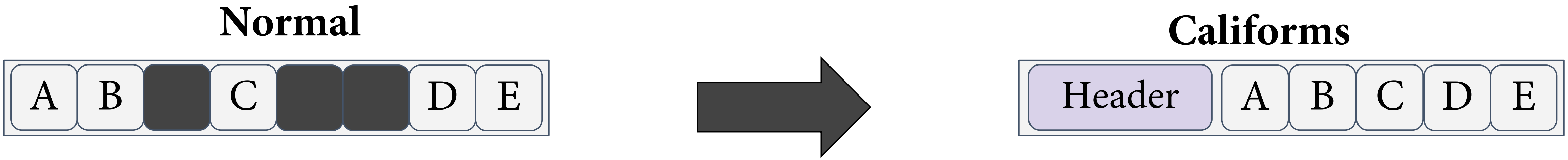
bit-vector

12.5% memory overhead

CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

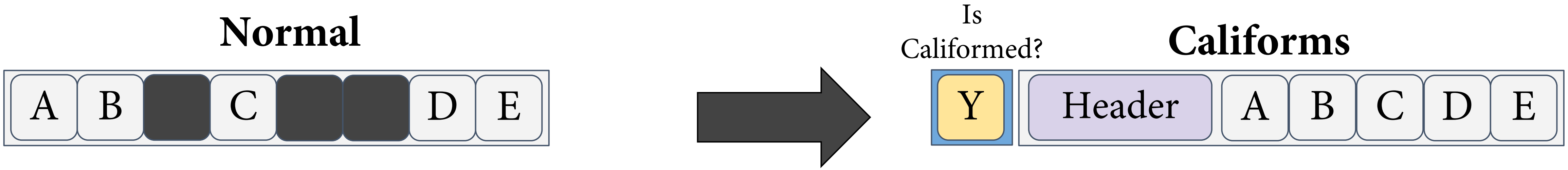
■ Blocklisted Location



CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

■ Blocklisted Location



Normal

A	B	■	C	■	■	D	E

Is Califormed?

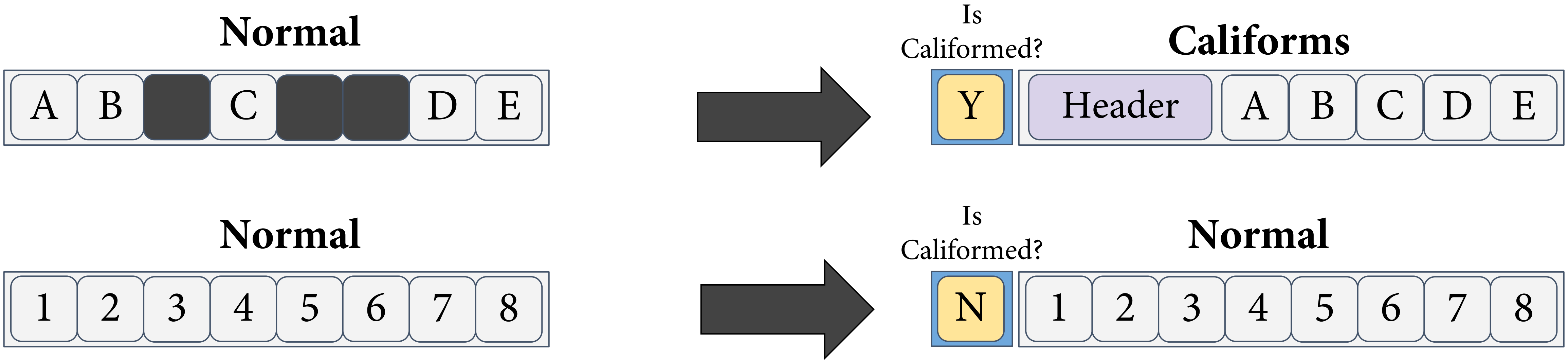
Califorms

Y	Header	A	B	C	D	E

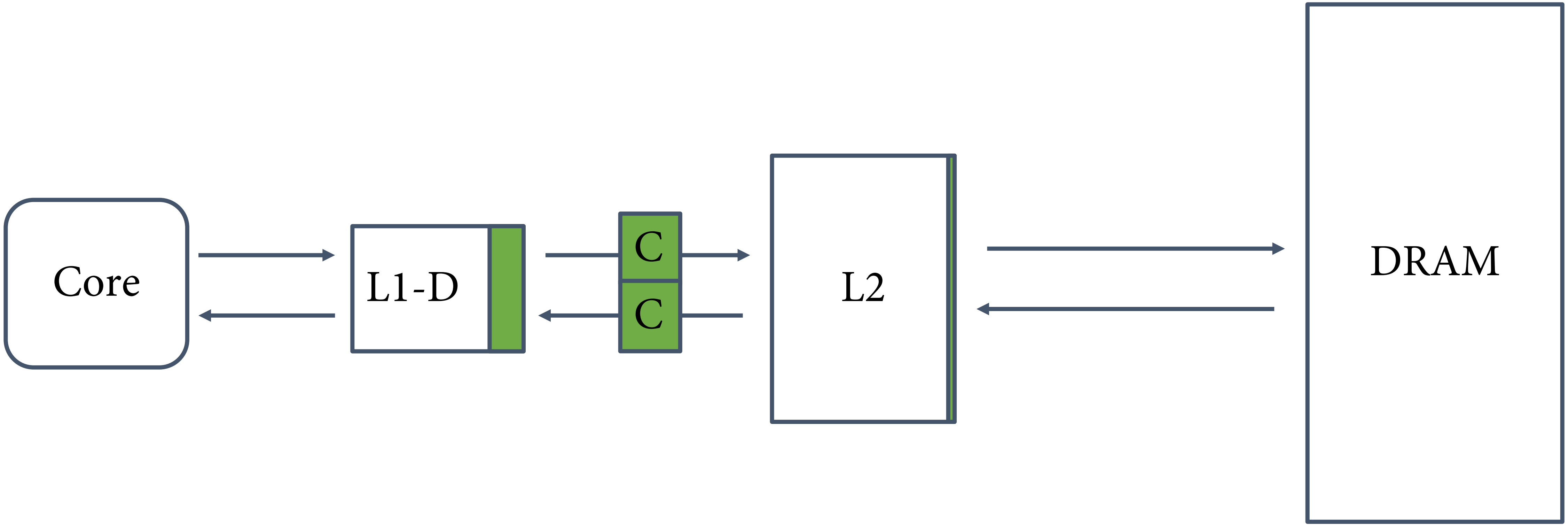
CaLiForms Cache Line Formats

Our Metadata: Encoded within unused data.

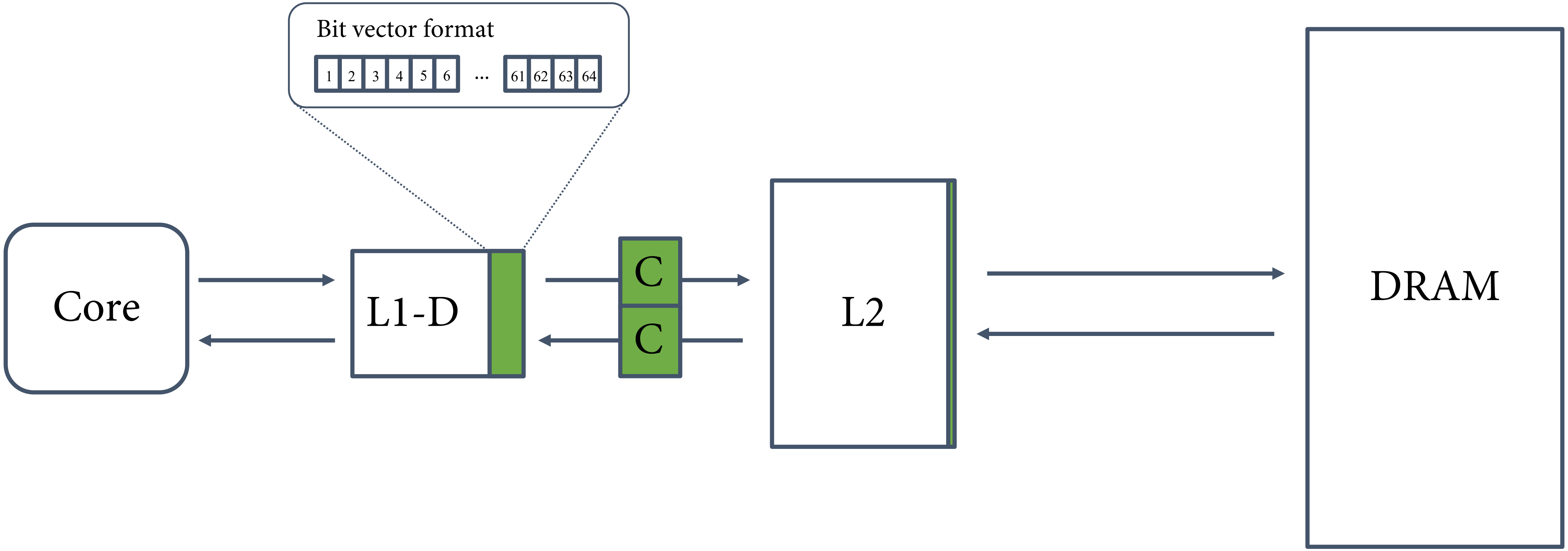
■ Blocklisted
Location



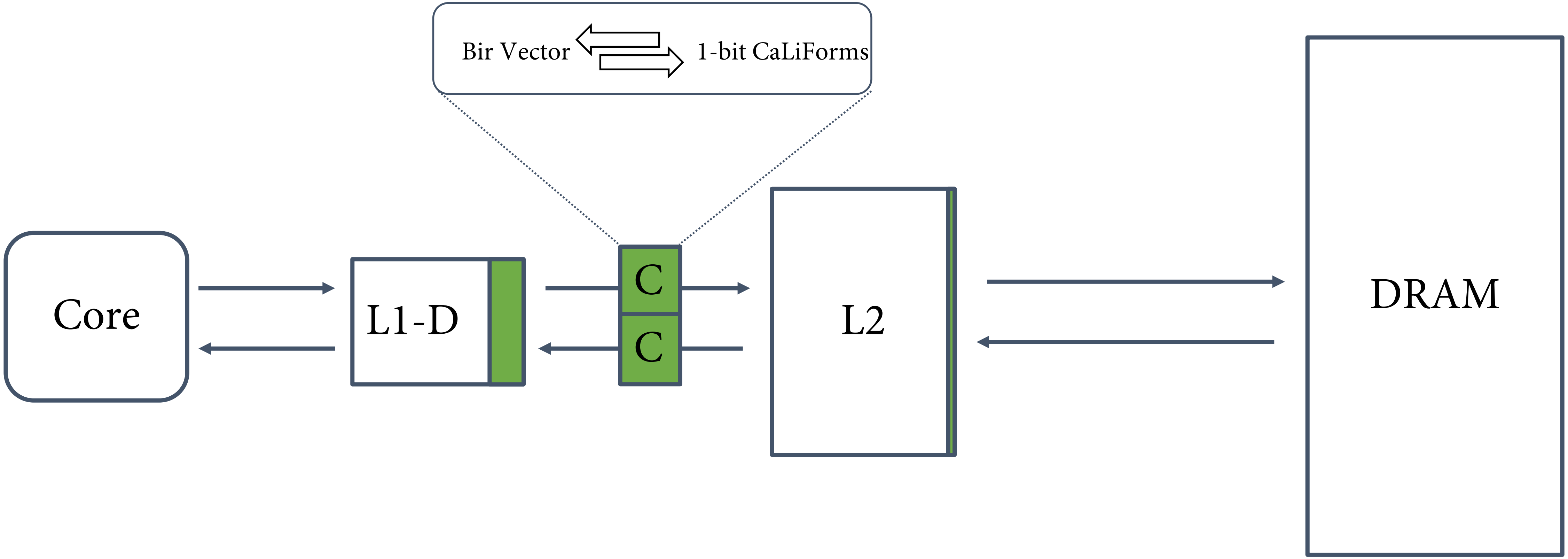
CaLiForms Microarchitectural Overview



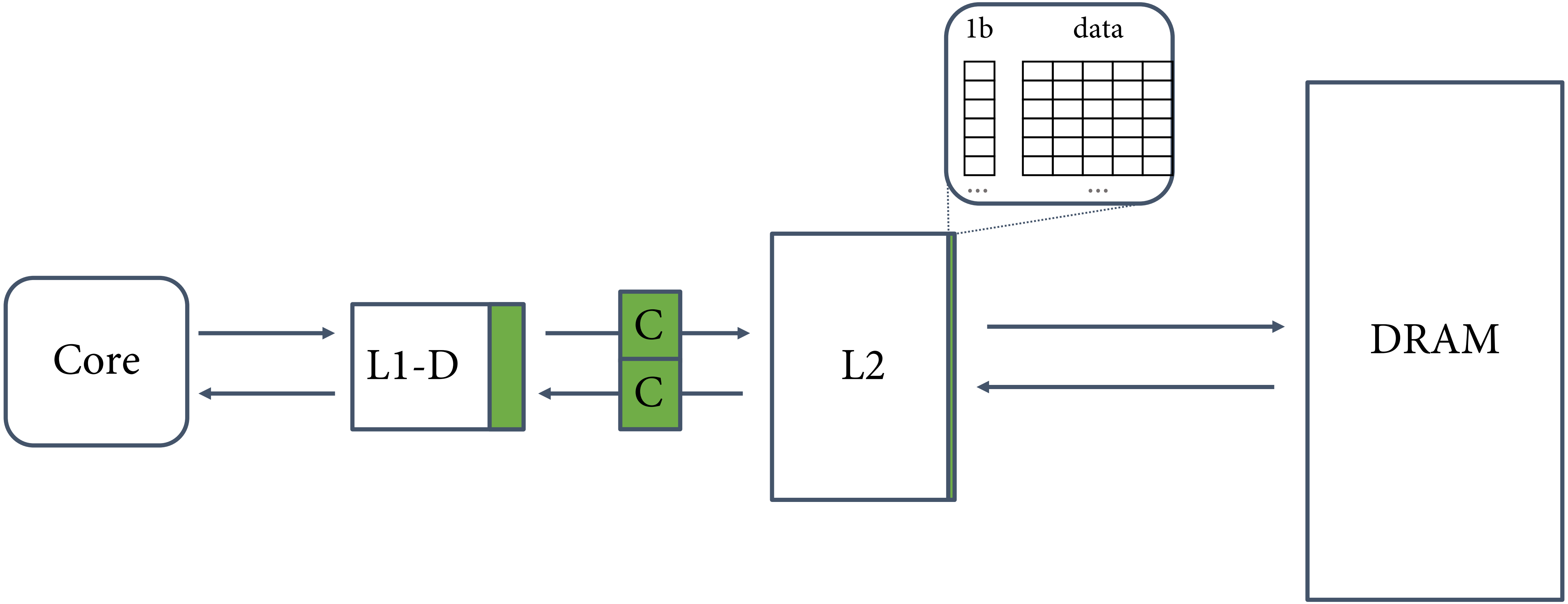
CaLiForms Microarchitectural Overview



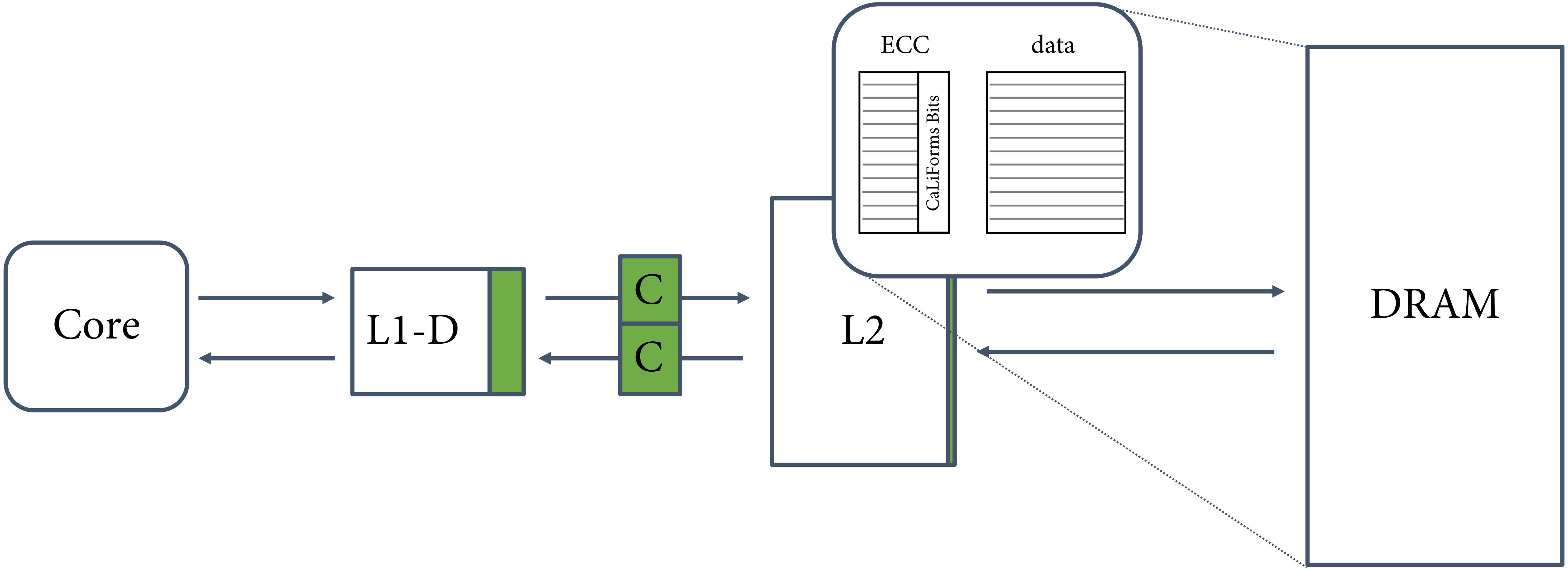
CaLiForms Microarchitectural Overview



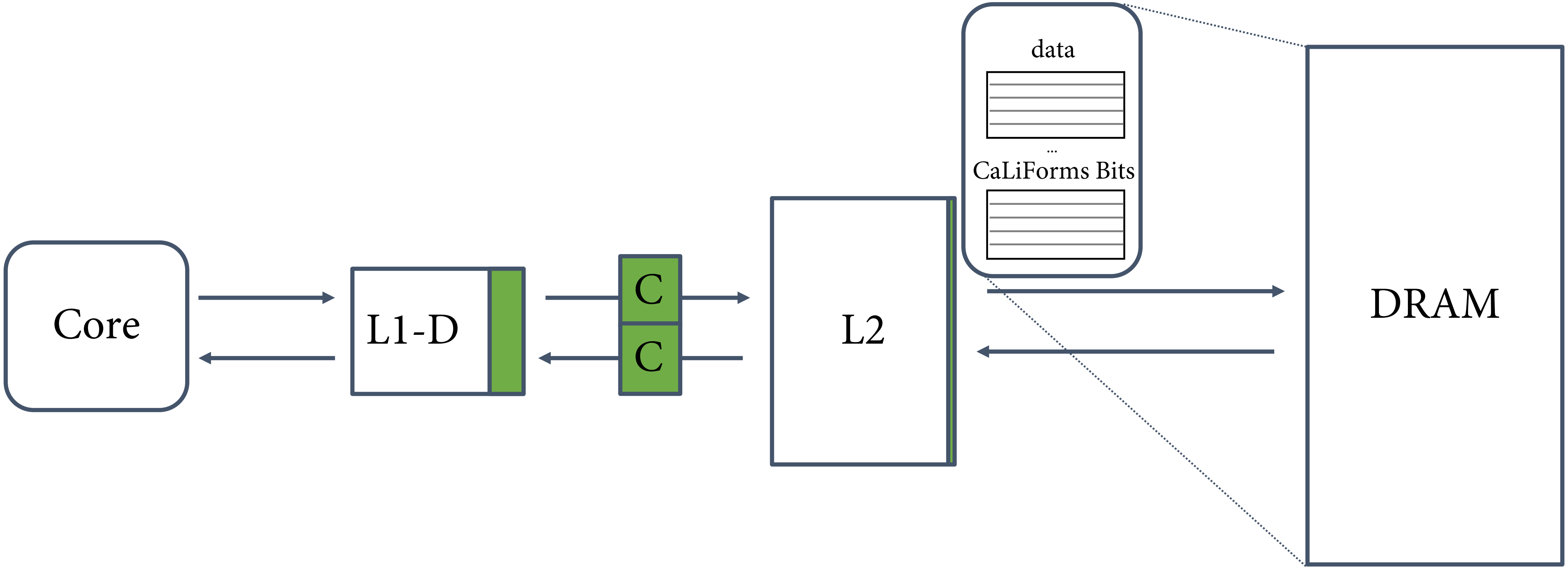
CaLiForms Microarchitectural Overview



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.

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

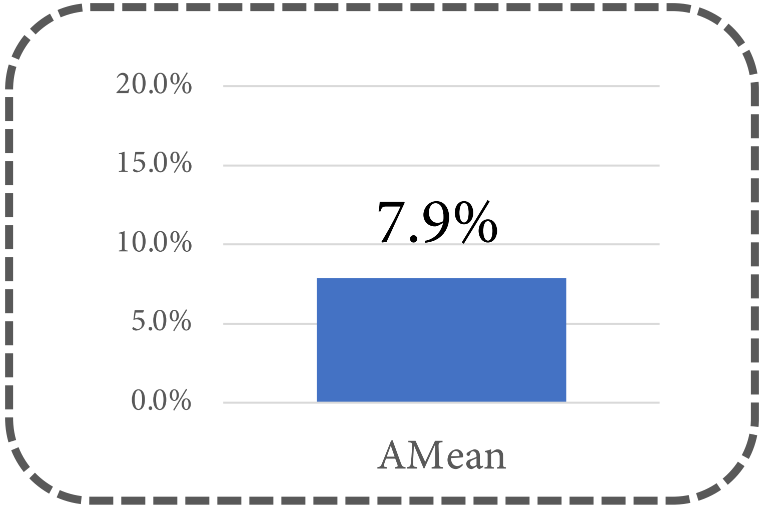
Our measurements show no impact on the cache access latency.

00010010
101001101
00010010
111001001
00010010

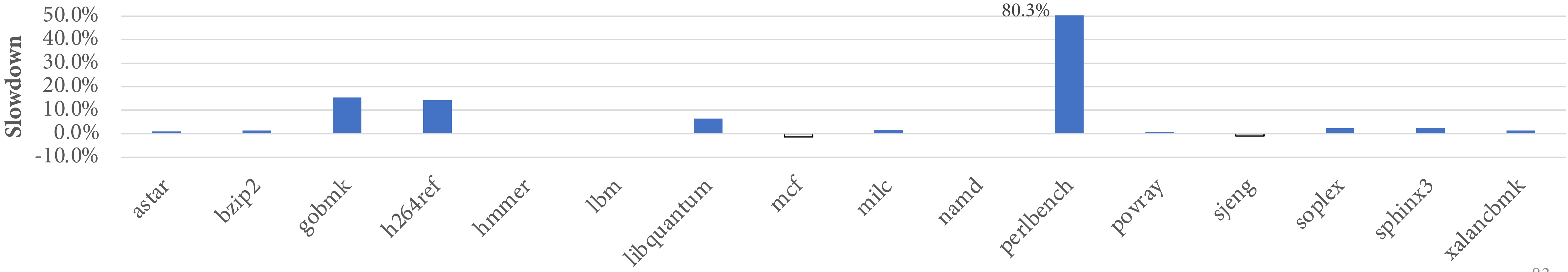
Software Modifications

- 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.

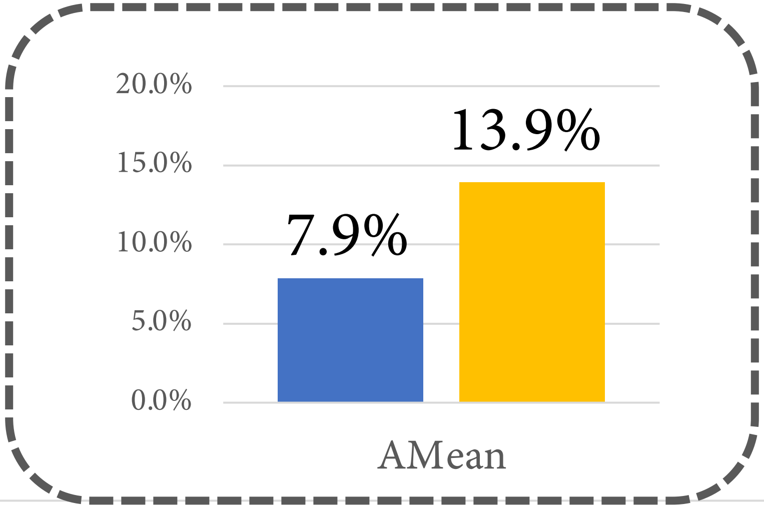
CaLiForms Performance Results (x86_64)

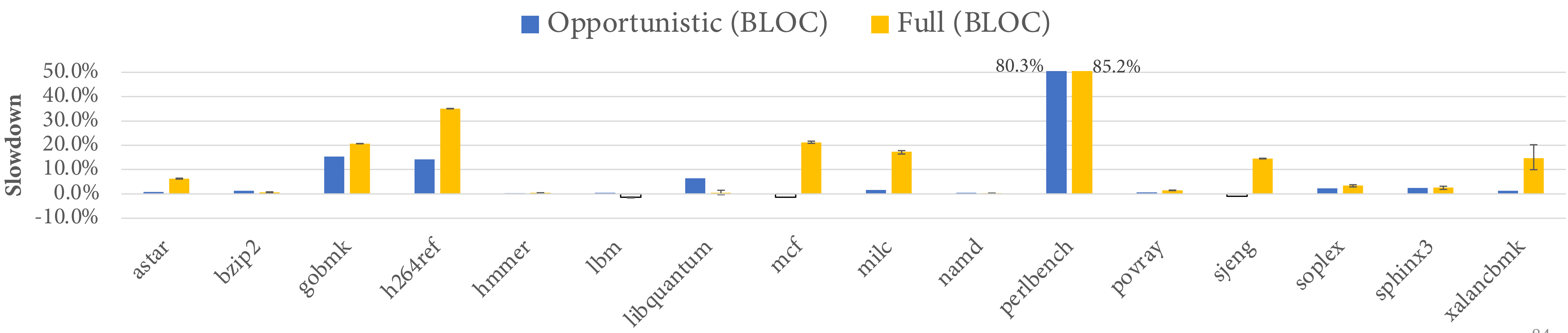


Opportunistic (BLOC)

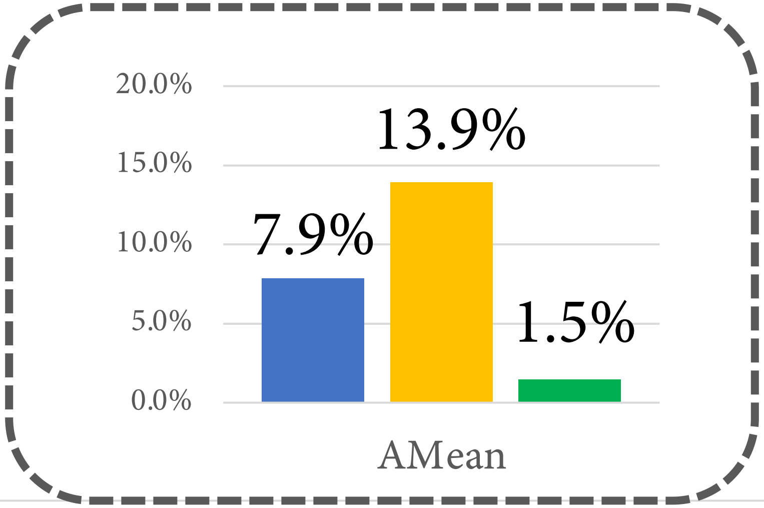


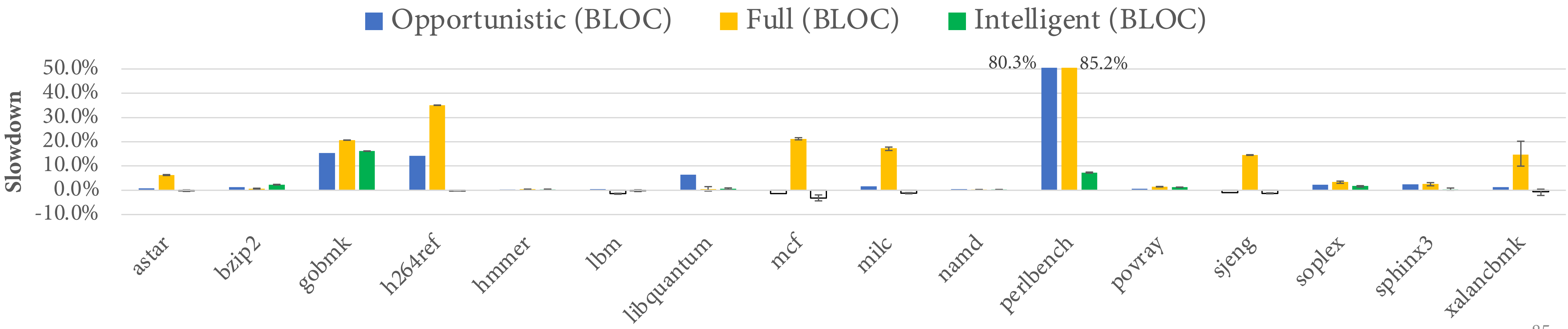
CaLiForms Performance Results (x86_64)





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;
```

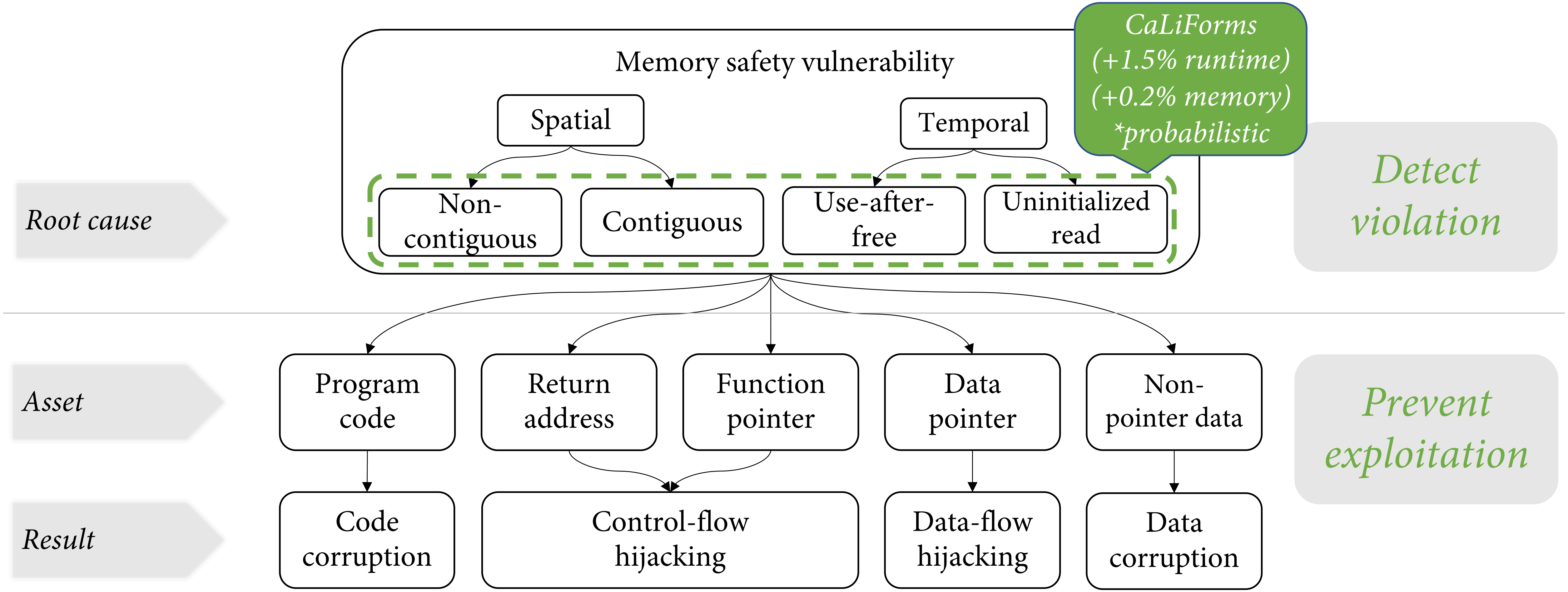
(2) Full

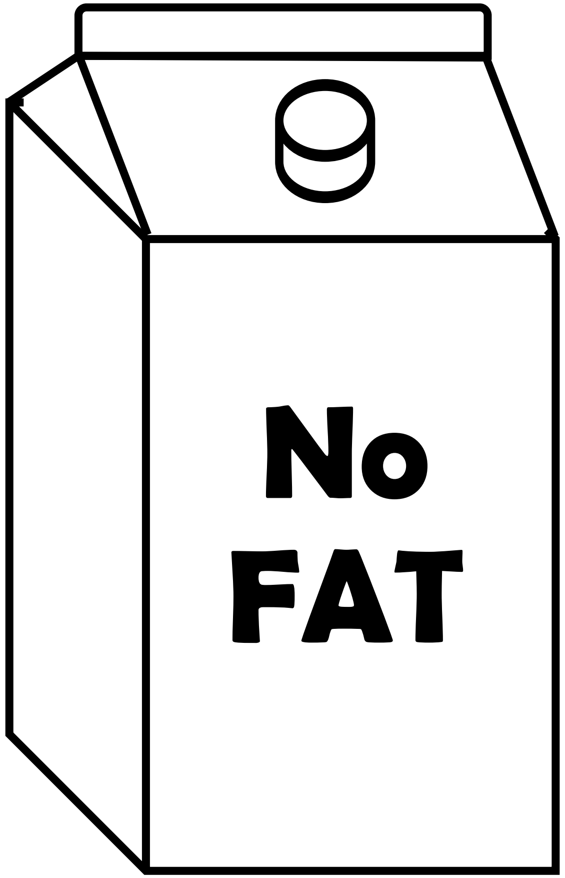
```
struct A_intelligent
{
  char c;
  int i;
  char tripwire[3];
  char buf[64];
  char tripwire[2];
  void (*fp)();
  char tripwire[3];
}
```

(3) Intelligent

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

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]

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.

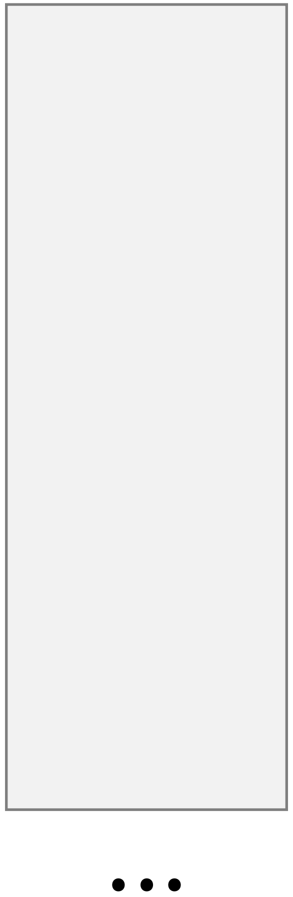
FreeBSD

mi-malloc

tcMalloc

Binning Memory Allocators

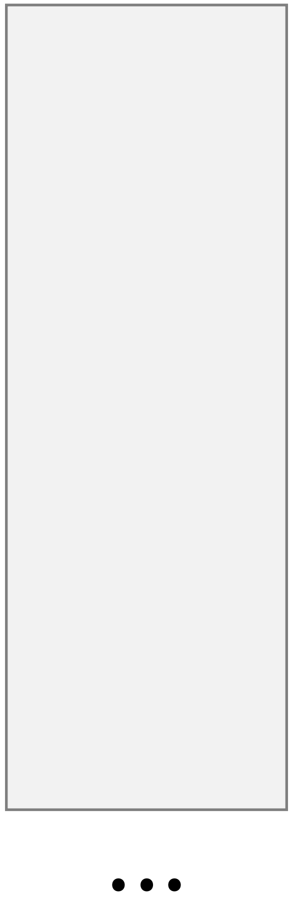
```
int main() {
  char* ptr = malloc(12);
  …
}
```



Virtual Memory

Binning Memory Allocators

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

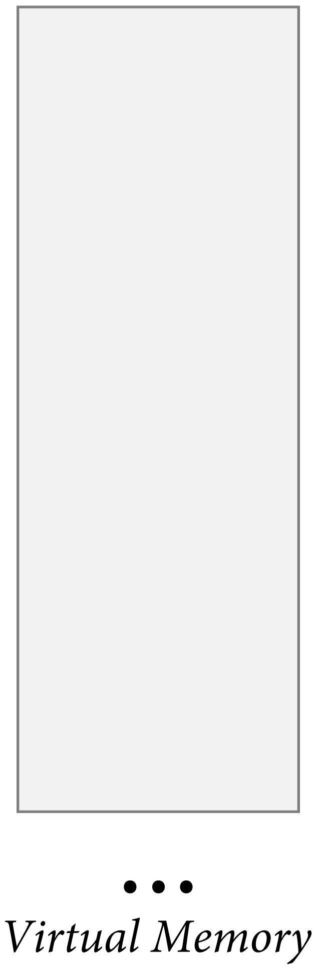


Virtual Memory

Binning Memory Allocators

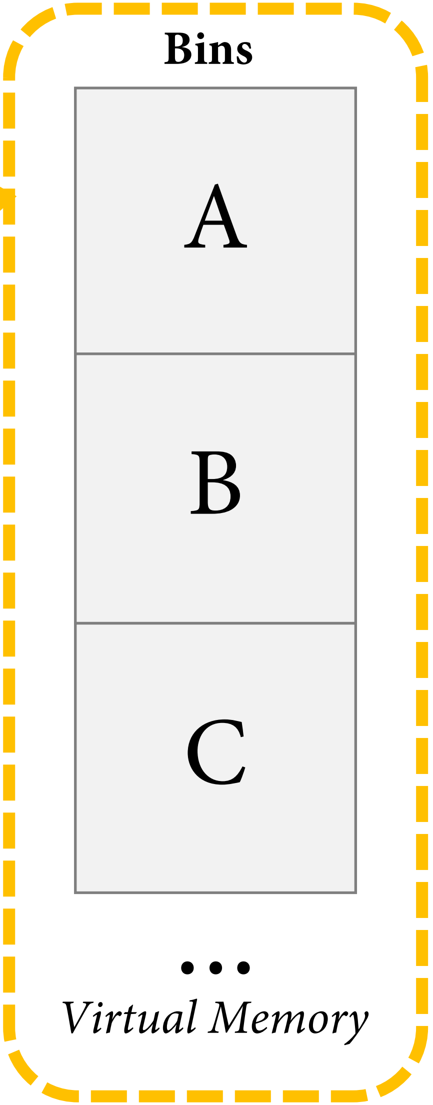
```
int main() {
    char* ptr = malloc(12);
    ...
}
```

Memory is requested by the allocator.



Binning Memory Allocators

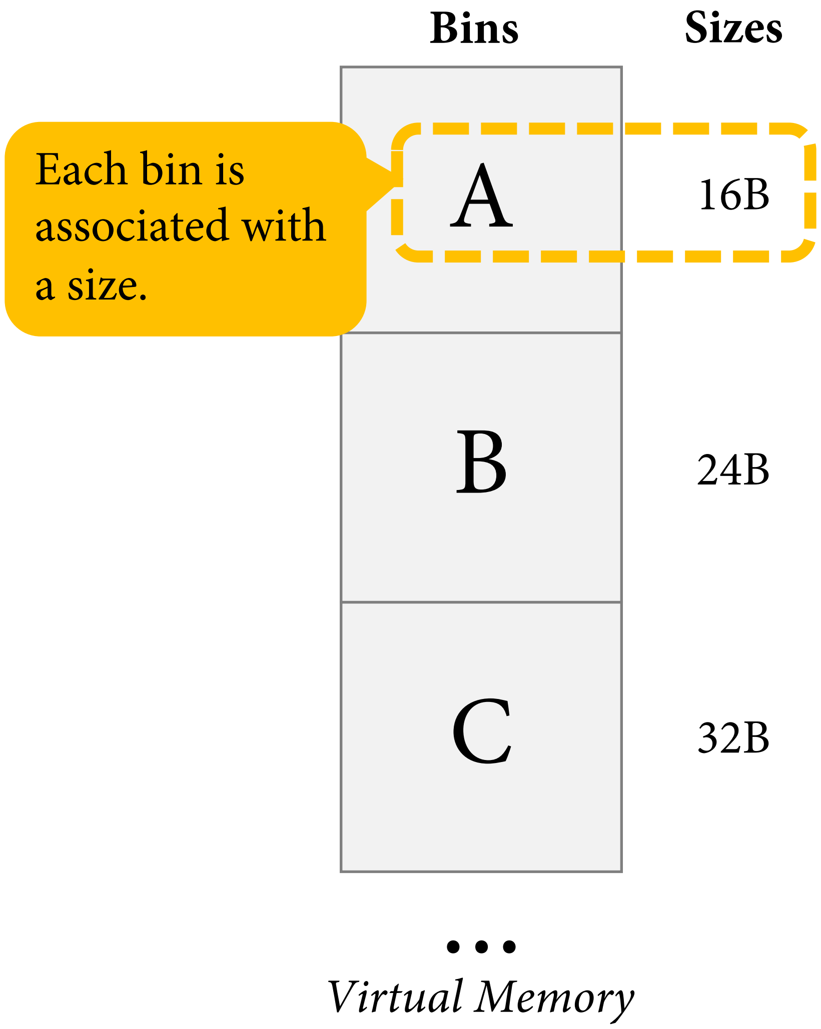
```c
int main() {
  char* ptr = malloc(12);
  …
}
```



Memory is divided into bins.

Binning Memory Allocators

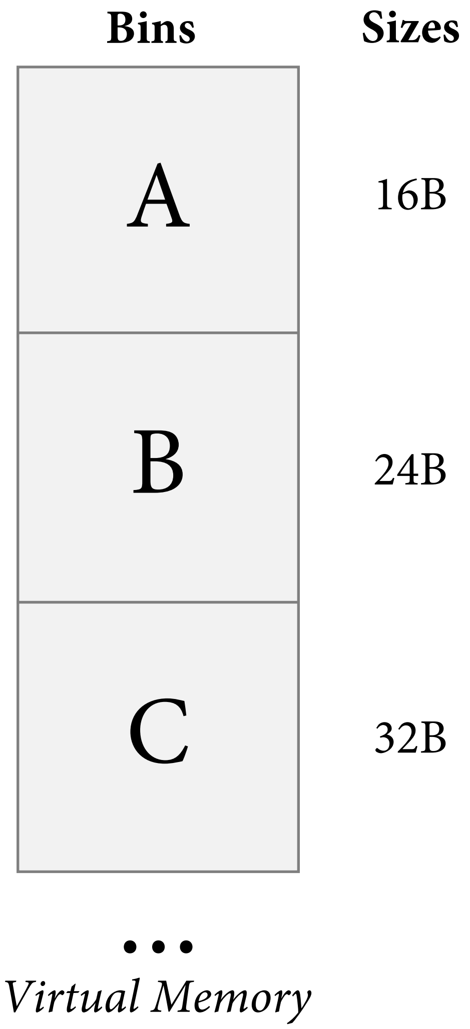
```
int main() {
  char* ptr = malloc(12);
  …
}
```



Each bin is associated with a size.

Binning Memory Allocators

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

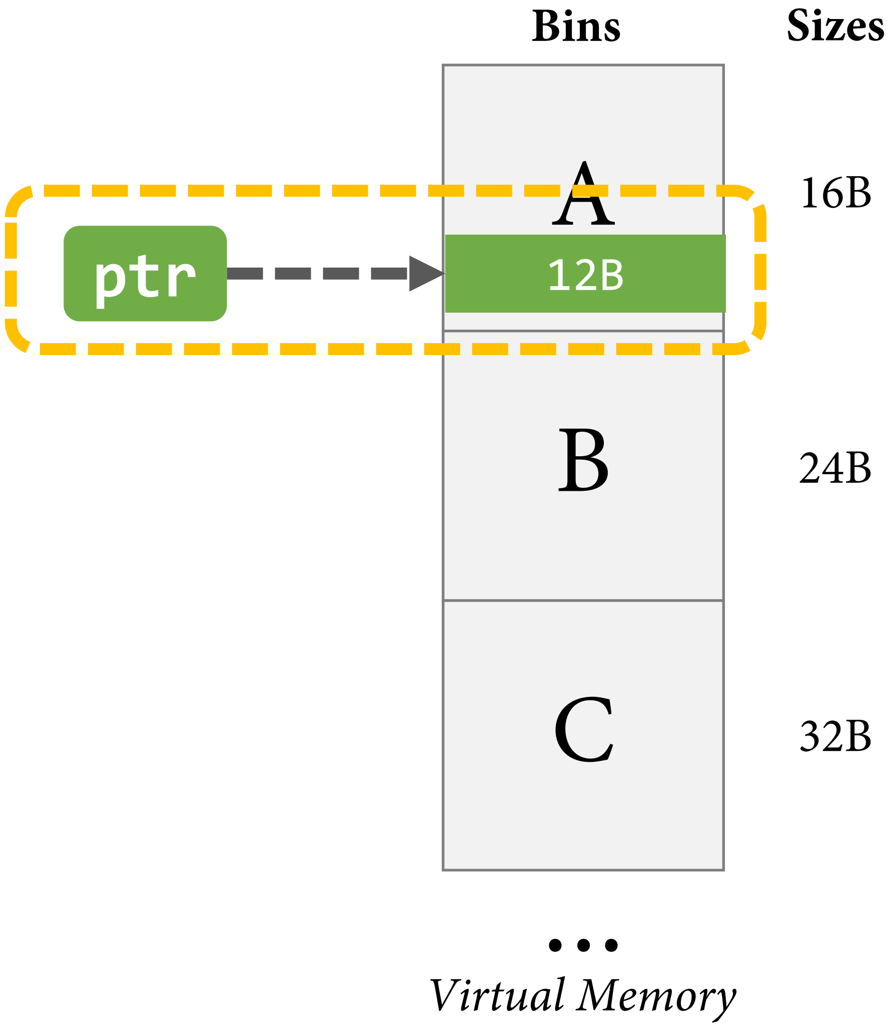


Bins	Sizes
A	16B
B	24B
C	32B
…	

Virtual Memory

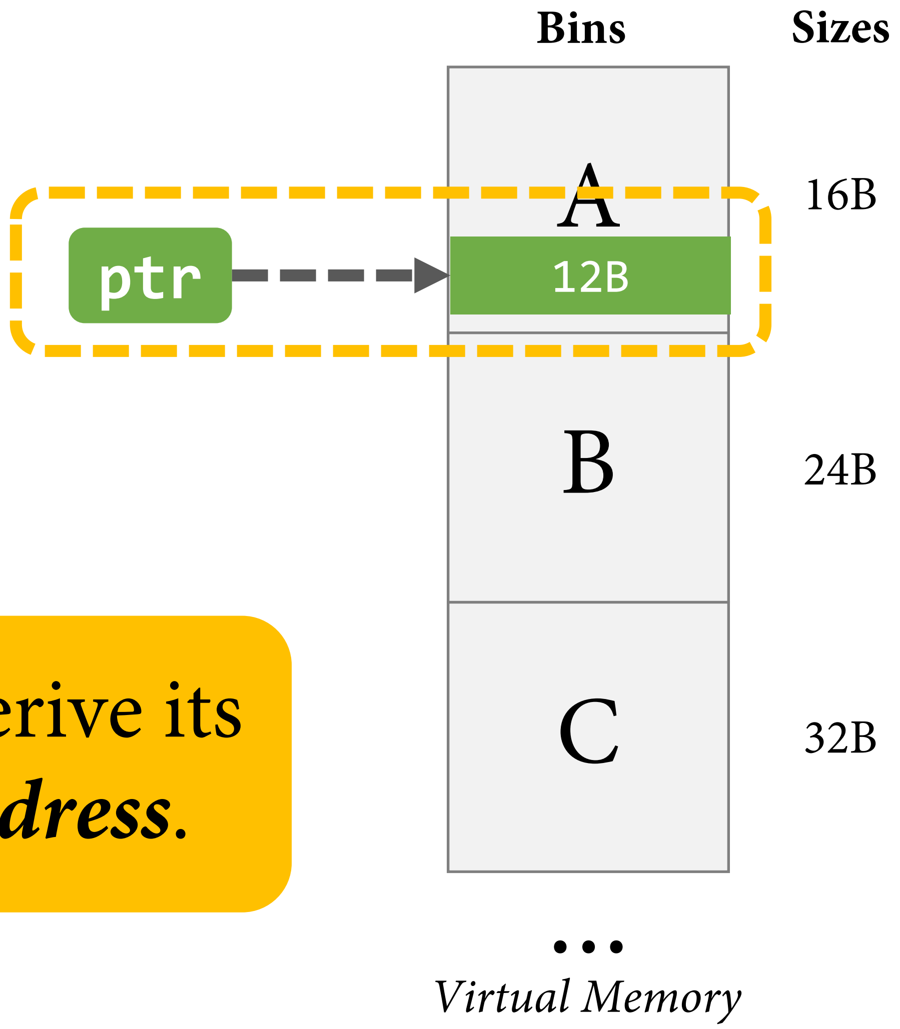
Binning Memory Allocators

```
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***.

How No-FAT Provides Memory Safety

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

How No-FAT Provides Memory Safety

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

How No-FAT Provides Memory Safety

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

→ s_store ptr[1],‘A’,$\text{ptr}_{\text{trusted\_base}}$

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

How No-FAT Provides Memory Safety

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

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

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

How No-FAT Provides Memory Safety

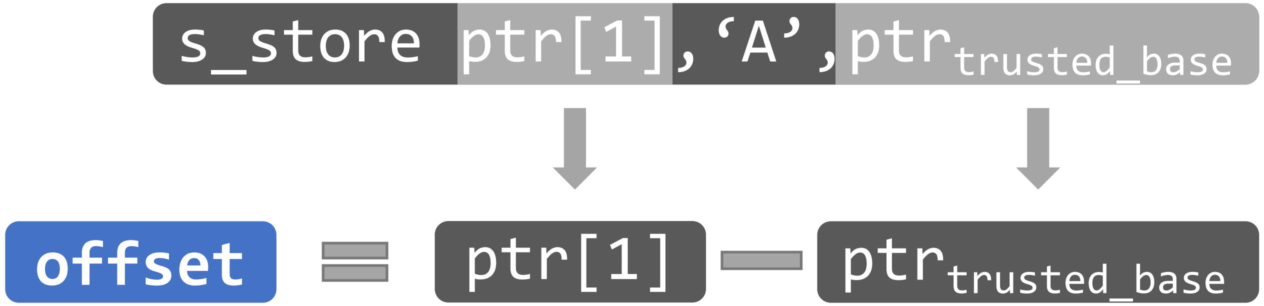
```
int main() {
  char* ptr = malloc(12);
  ptr[1] = ‘A’;
  …
}
```

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

How No-FAT Provides Memory Safety

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

How No-FAT Provides Memory Safety



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

offset = ptr[1] − $\text{ptr}_{\text{trusted\_base}}$

How No-FAT Provides Memory Safety

$\texttt{s\_store ptr[1],‘A’,ptr}_{\texttt{trusted\_base}}$

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

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

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

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

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

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

Bounds Check

$\texttt{offset} < \texttt{size}$?

Temporal Check

$\texttt{ptr[1]}\ \texttt{[63:48]} = \texttt{ptr}_{\texttt{trusted\_base}}\ \texttt{[63:48]}$?

How No-FAT Provides Memory Safety

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

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

Bounds Check

$\texttt{offset} < \texttt{size}$?

Temporal Check

$\texttt{ptr[1]}\ \texttt{[63:48]} = \texttt{ptr}_{\texttt{trusted\_base}}\ \texttt{[63:48]}$?

How No-FAT Provides Memory Safety

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

ptr$_{\text{trusted\_base}}$

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

How No-FAT Provides Memory Safety

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

$\texttt{ptr}_{\texttt{trusted\_base}}$

$\texttt{s\_store ptr[1],‘A’,ptr}_{\texttt{trusted\_base}}$

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

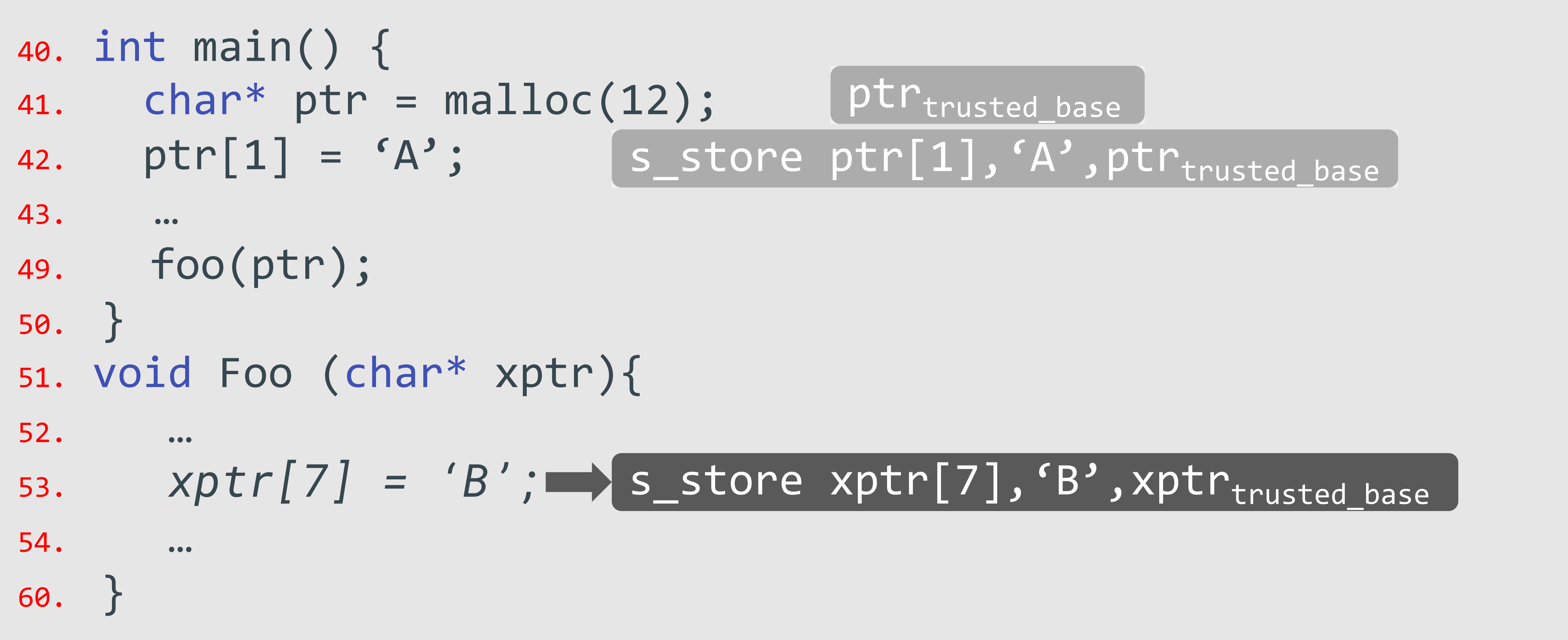
How No-FAT Provides Memory Safety

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

$\mathtt{ptr}_{\mathtt{trusted\_base}}$

$\mathtt{s\_store\ ptr[1],‘A’,ptr}_{\mathtt{trusted\_base}}$

How No-FAT Provides Memory Safety



```
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],‘B’,$xptr_{trusted\_base}$

How No-FAT Provides Memory Safety

```
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],‘B’,$xptr_{trusted\_base}$

How do we get this?

How No-FAT Provides Memory Safety

```
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],‘B’,$xptr_{trusted\_base}$

How No-FAT Provides Memory Safety

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

How No-FAT Provides Memory Safety

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

$\texttt{Bin} == \texttt{xptr} >> \texttt{log}_2\texttt{(S)}$ where S is the size of the bins.

How No-FAT Provides Memory Safety

$\texttt{xptr}_{\texttt{trusted base}}$ ← compBase(xptr[7])

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

size = getSize(Bin)

How No-FAT Provides Memory Safety

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

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

$$\mathtt{size} = \mathtt{getSize(Bin)}$$

$$\mathtt{xptr}_{\mathtt{trusted\_base}} = \left\lfloor \mathtt{xptr} \times (1/\,\mathtt{size}\,) \right\rfloor \times \mathtt{size}$$

How No-FAT Provides Memory Safety

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

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

$\mathtt{size} = \mathtt{getSize(Bin)}$

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

Base pointer is **implicitly** derived!

How No-FAT Provides Memory Safety

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

$\mathrm{ptr}_{\mathrm{trusted\_base}}$

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

How No-FAT Provides Memory Safety

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

$\texttt{ptr}_{\texttt{trusted\_base}}$

$\texttt{s\_store ptr[1],‘A’,ptr}_{\texttt{trusted\_base}}$

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

How No-FAT Provides Memory Safety

```
int main() {
  char* ptr = malloc(12);
  ptr[1] = ‘A’;
  ptr = ptr + 100;

  …
  foo(ptr);
}
```

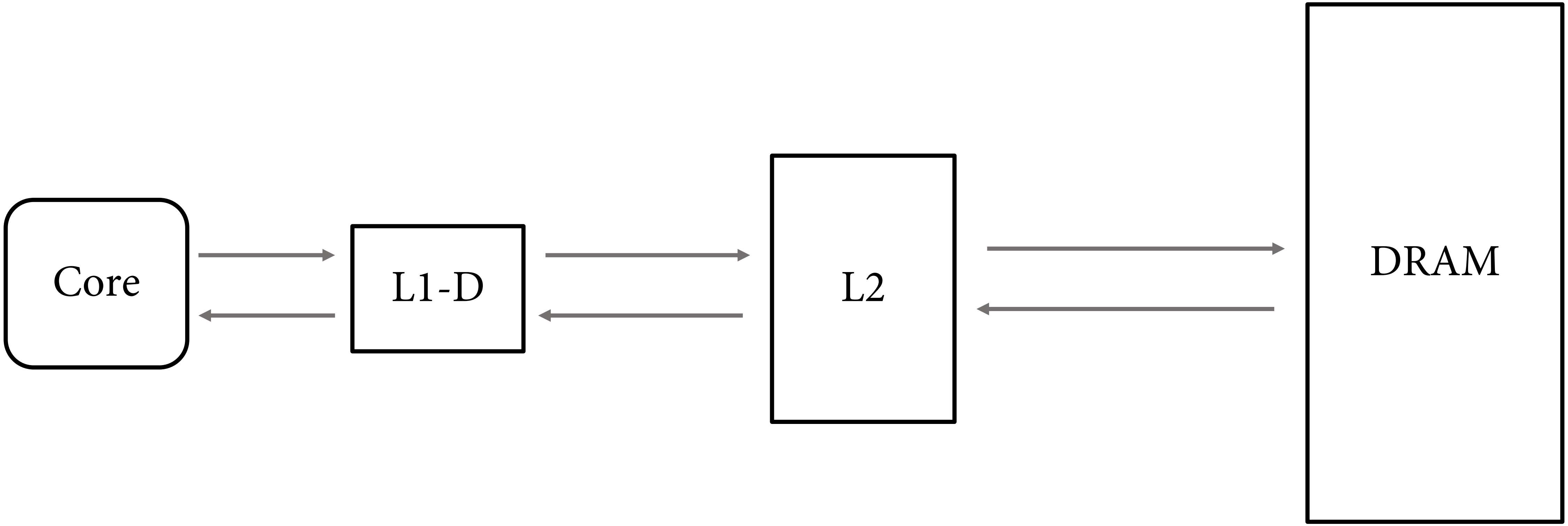
$ptr_{trusted\_base}$

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

verifyBounds ptr,$ptr_{trusted\_base}$

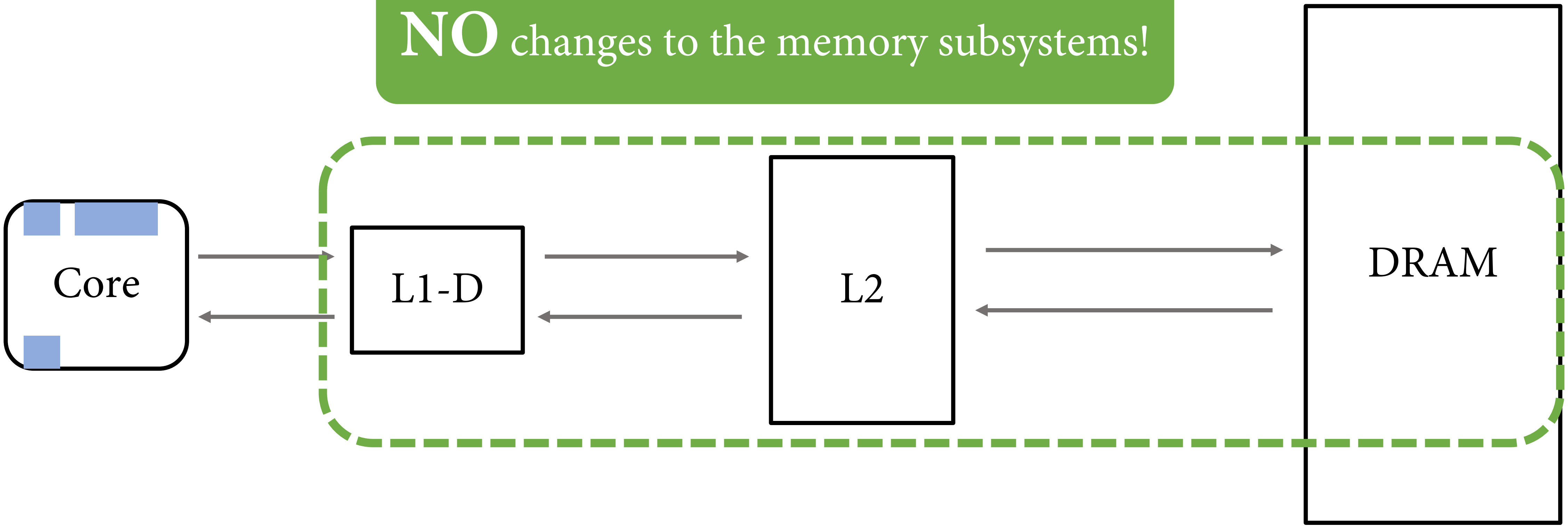
Verify the bounds of all pointers that escape to memory (or another function).

No-FAT Microarchitectural Overview

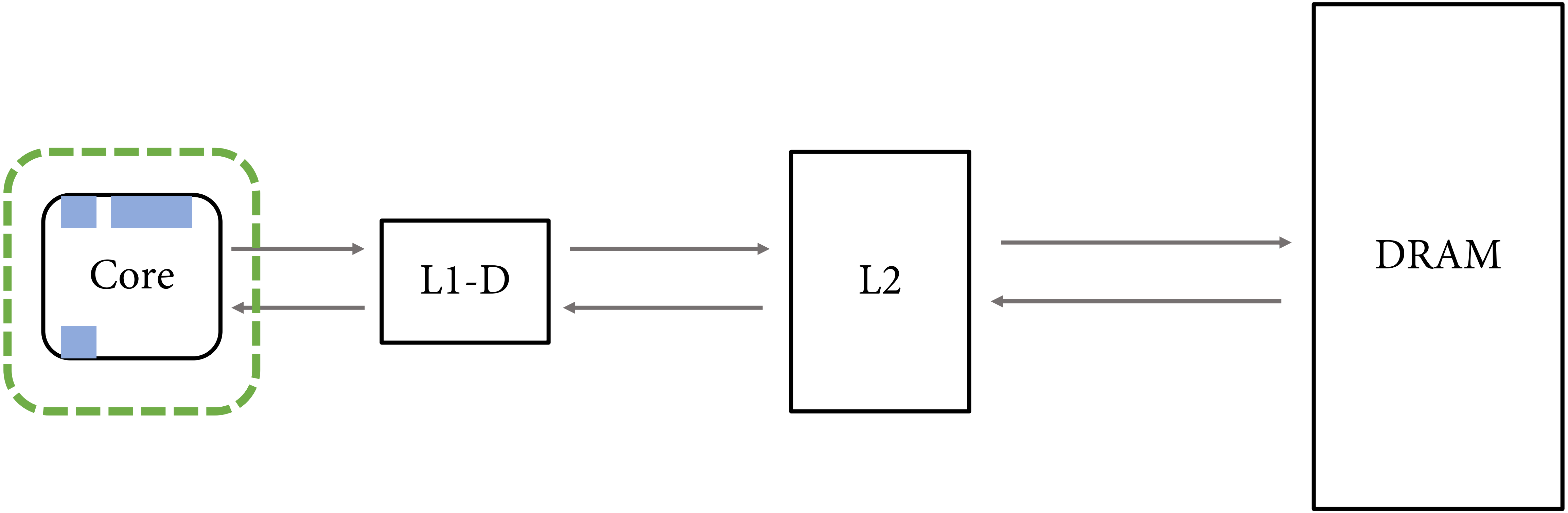


No-FAT Microarchitectural Overview

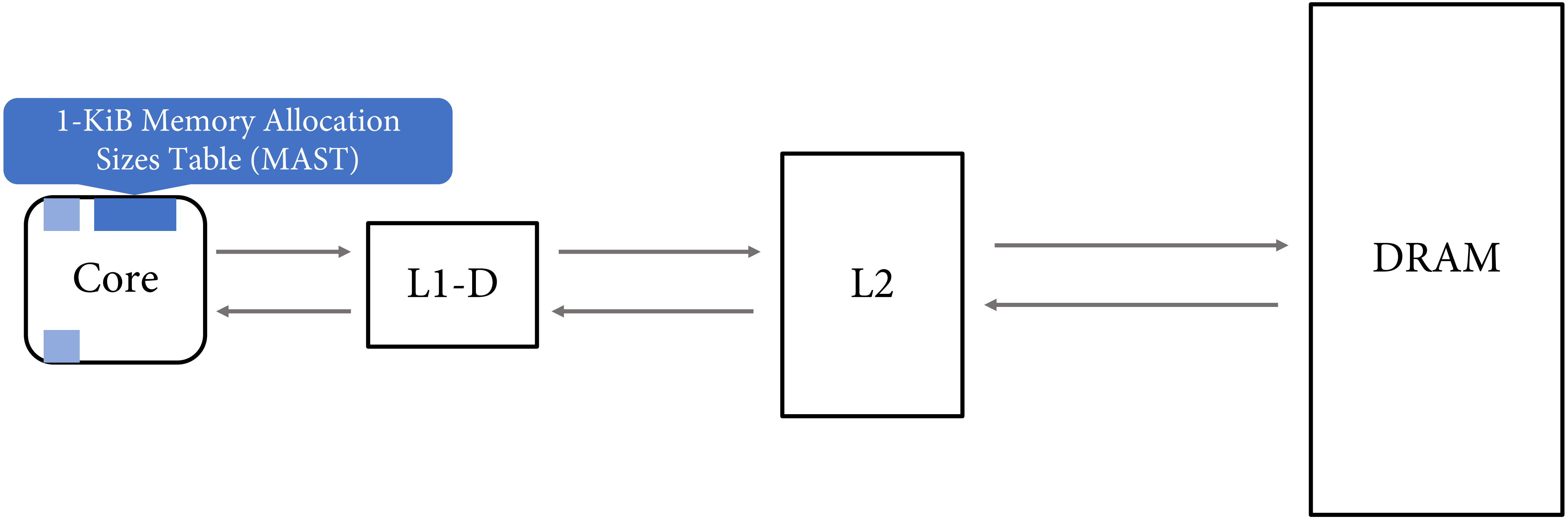
NO changes to the memory subsystems!



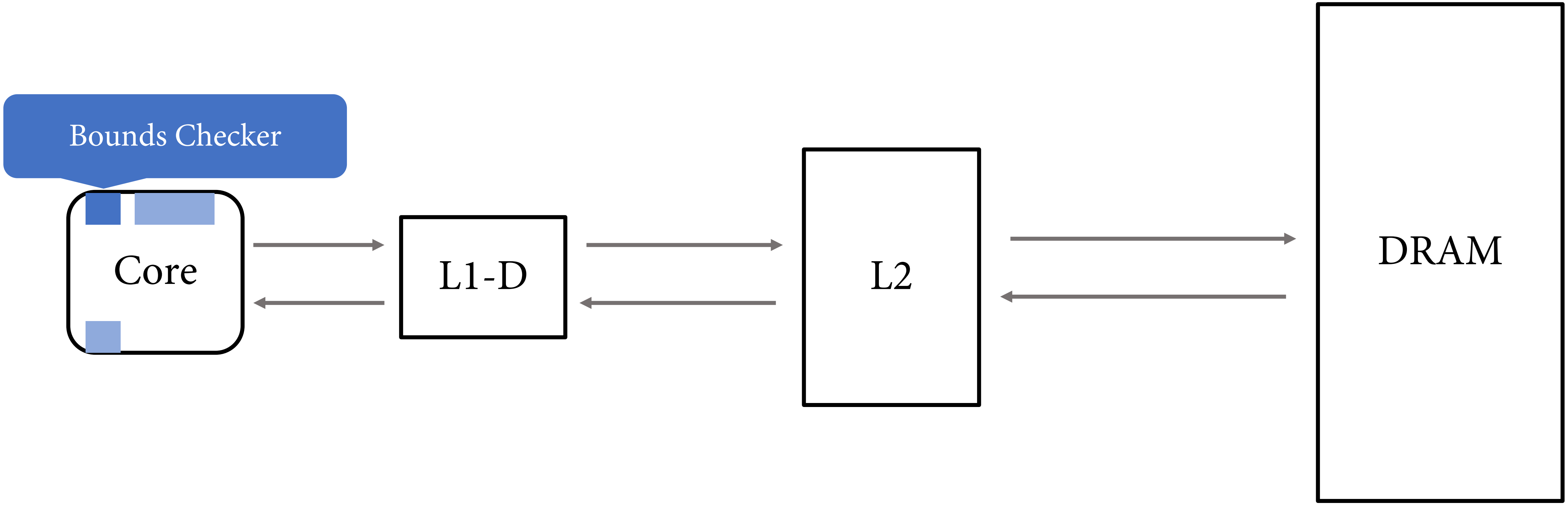
No-FAT Microarchitectural Overview



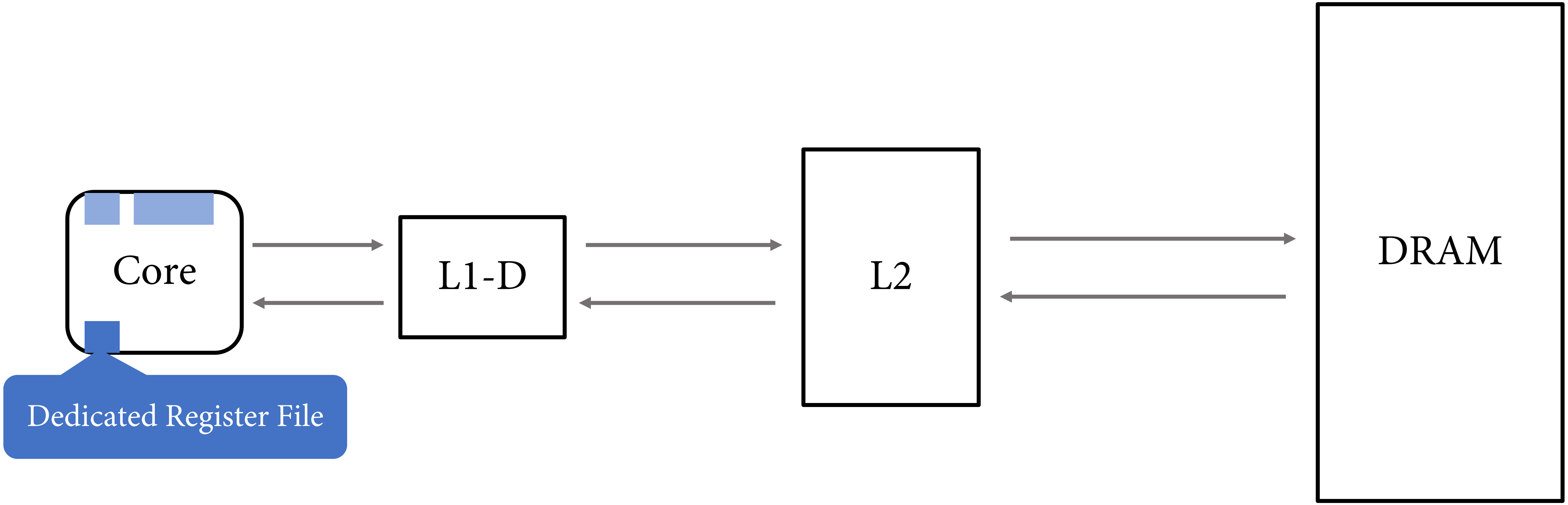
No-FAT Microarchitectural Overview



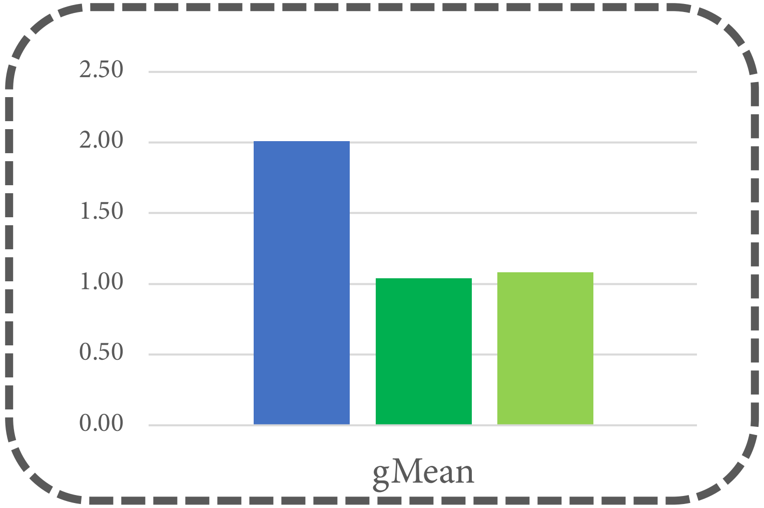
No-FAT Microarchitectural Overview

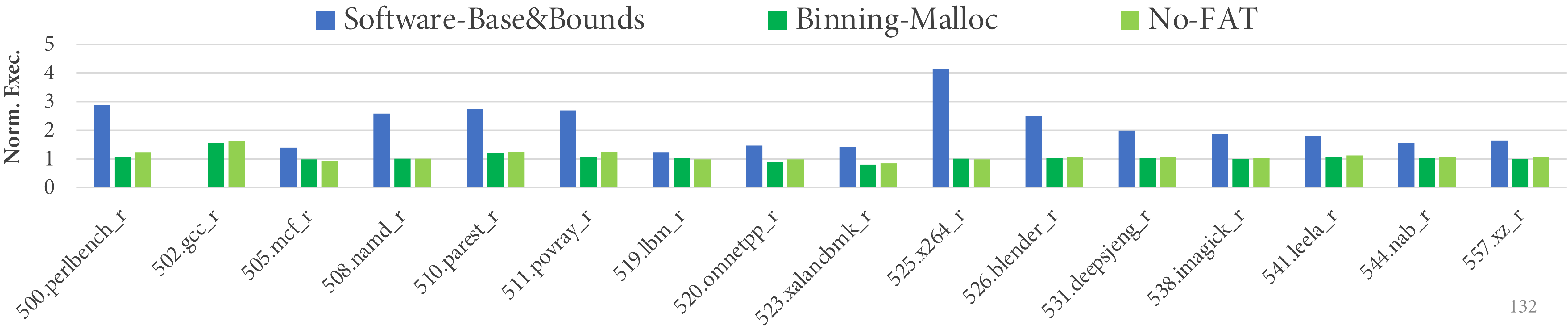


No-FAT Microarchitectural Overview

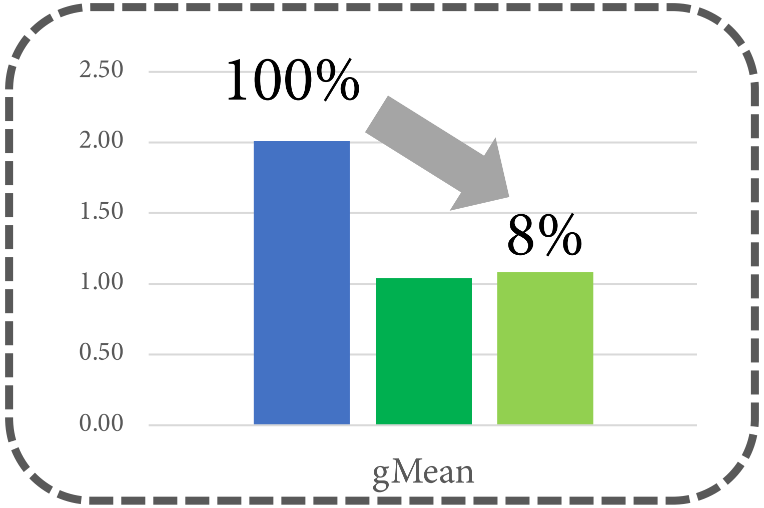


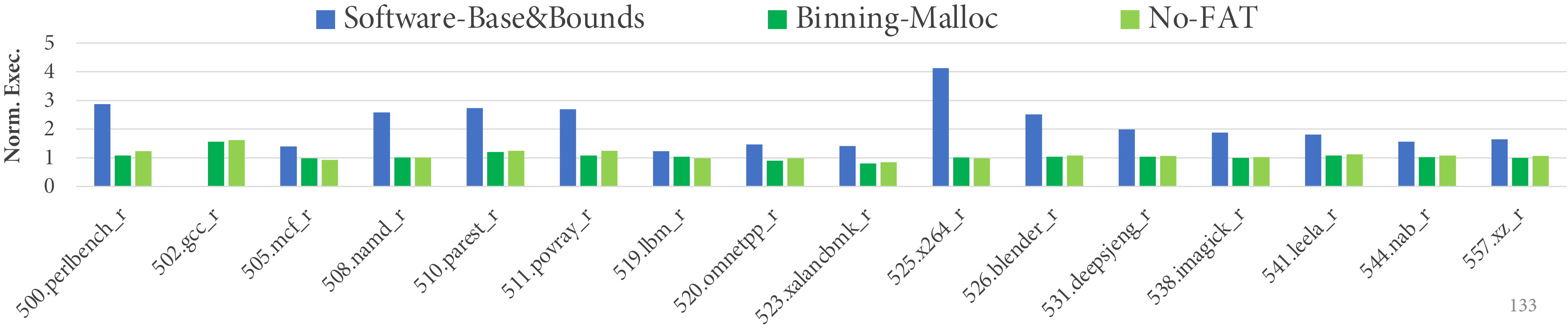
No-FAT Performance Results (x86_64)



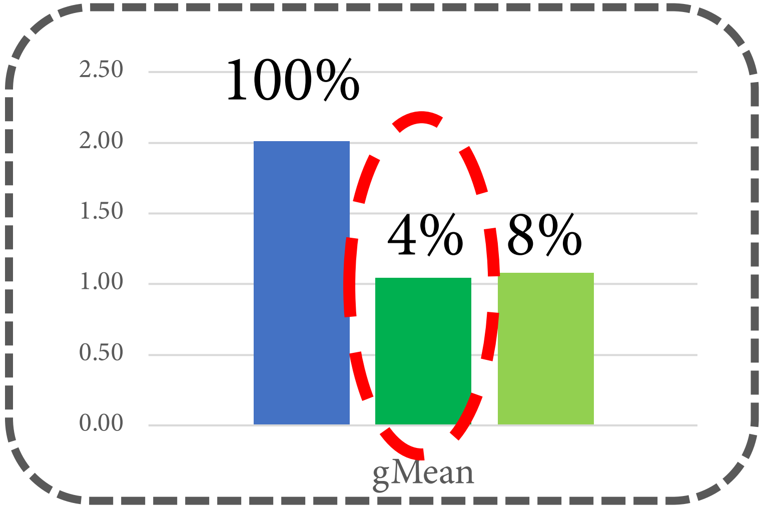


No-FAT Performance Results (x86_64)





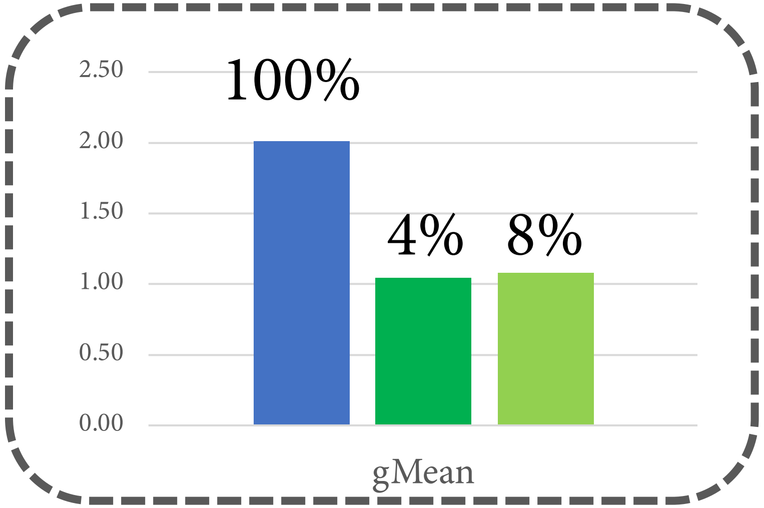
No-FAT Performance Results (x86_64)



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

- The binning memory allocator, and

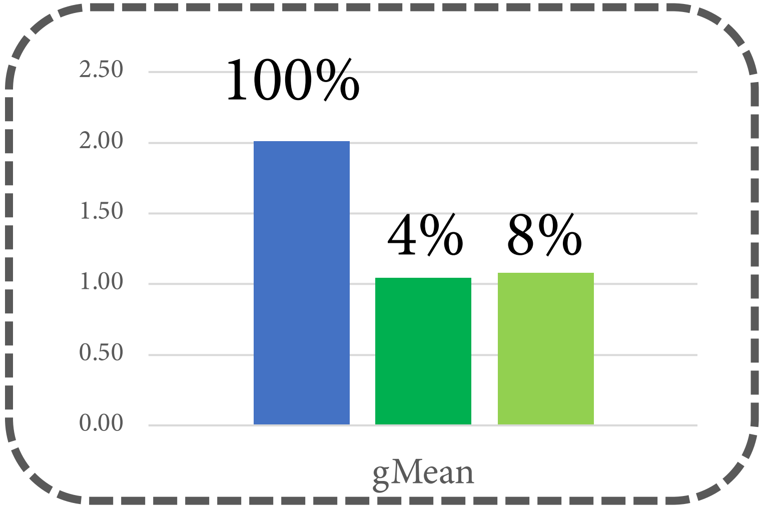
No-FAT Performance Results (x86_64)



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

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

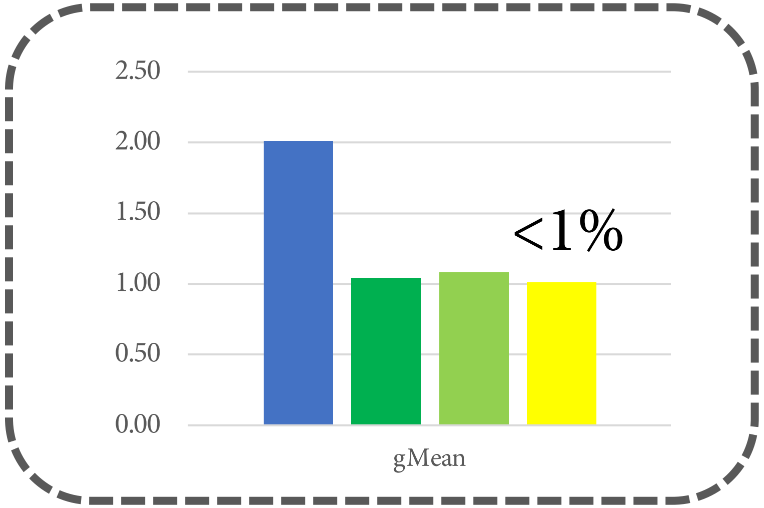
No-FAT Performance Results (x86_64)



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

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

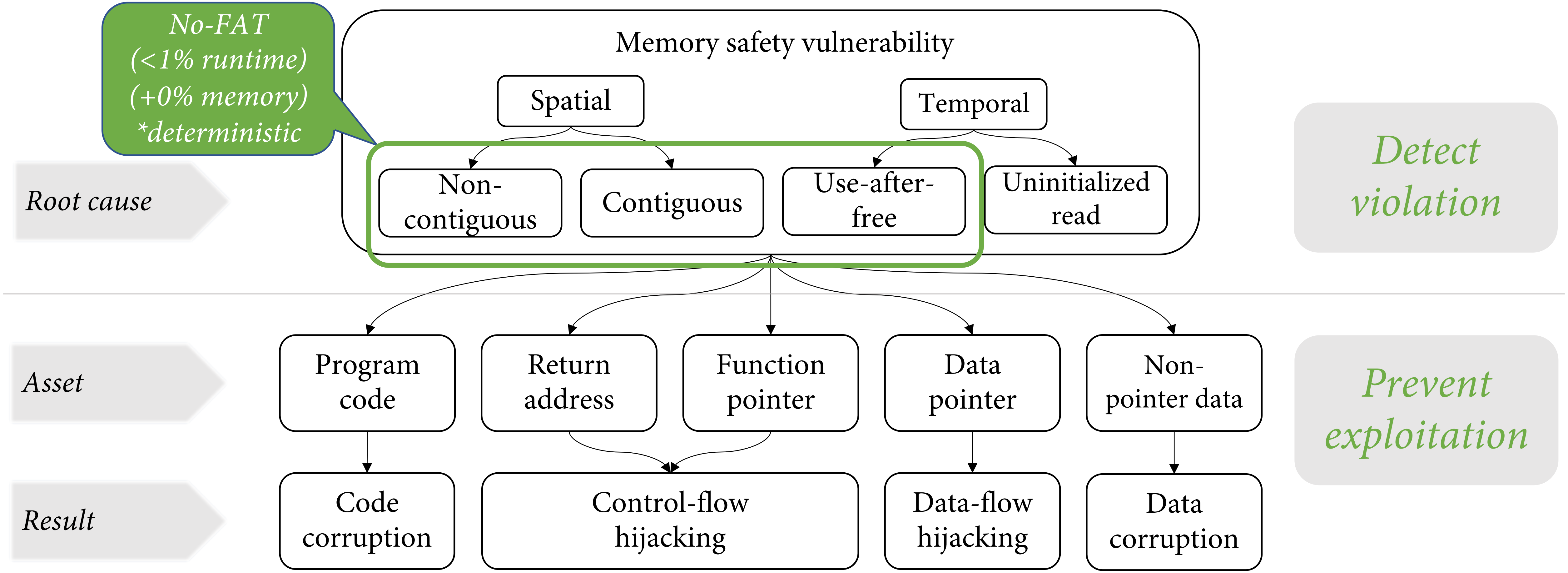
No-FAT Performance Results (x86_64)



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

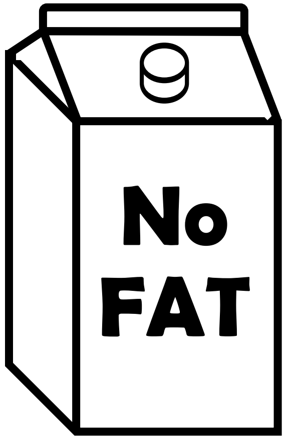


My solutions for C/C++ memory (un)safety

Memory Blocklisting

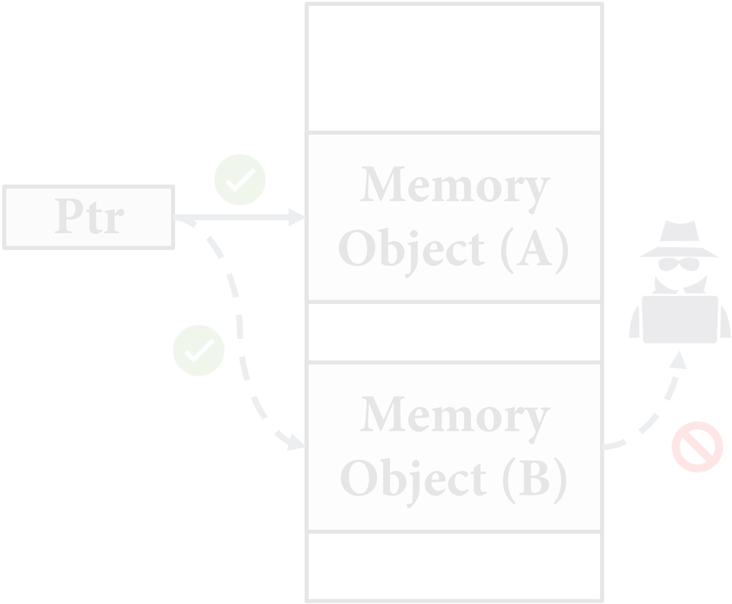
[MICRO 2019]

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

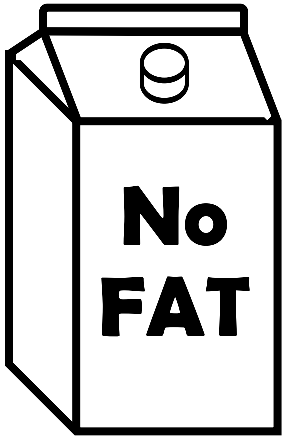
	Metadata	Concerns
Memory Tagging	N bits per pointer & allocation	Spatial & temporal safety limited by tag width
Tripwires	N bits per allocation	Susceptible to non-adjacent overflows
CaLiForms	1 bit per cache line	Provides probabilistic guarantees
Explicit Base & Bounds	N bits per pointer or allocation	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

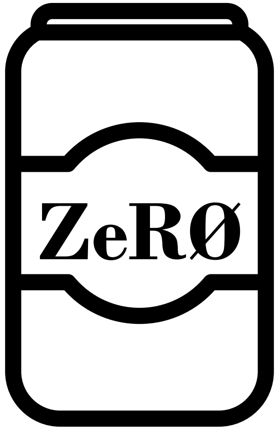
[MICRO 2019]

Memory Permitlisting

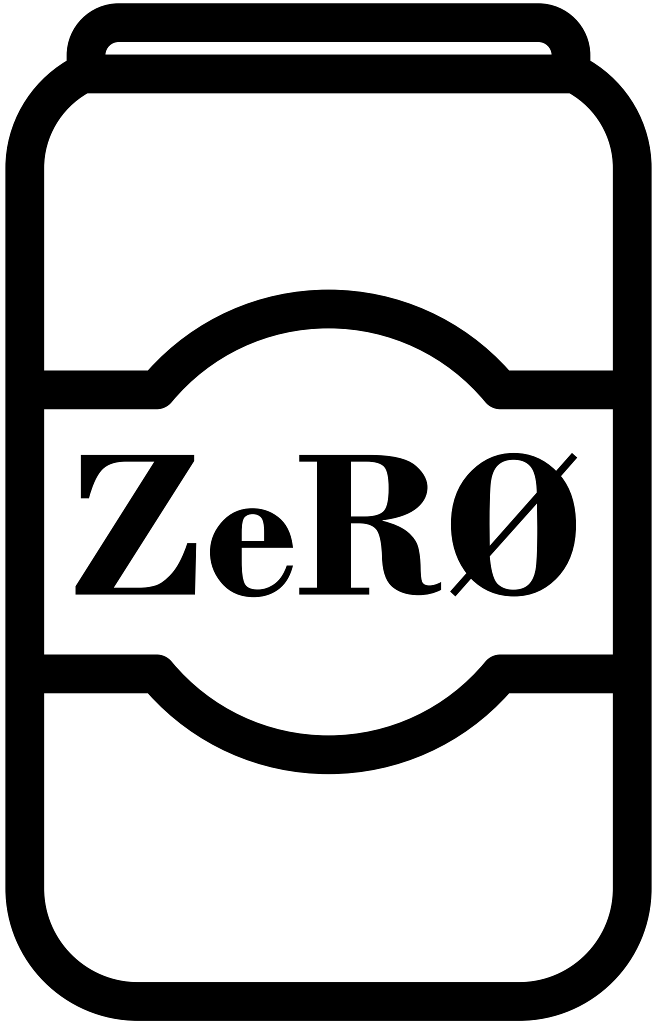


[ISCA 2021]

Exploit Mitigation

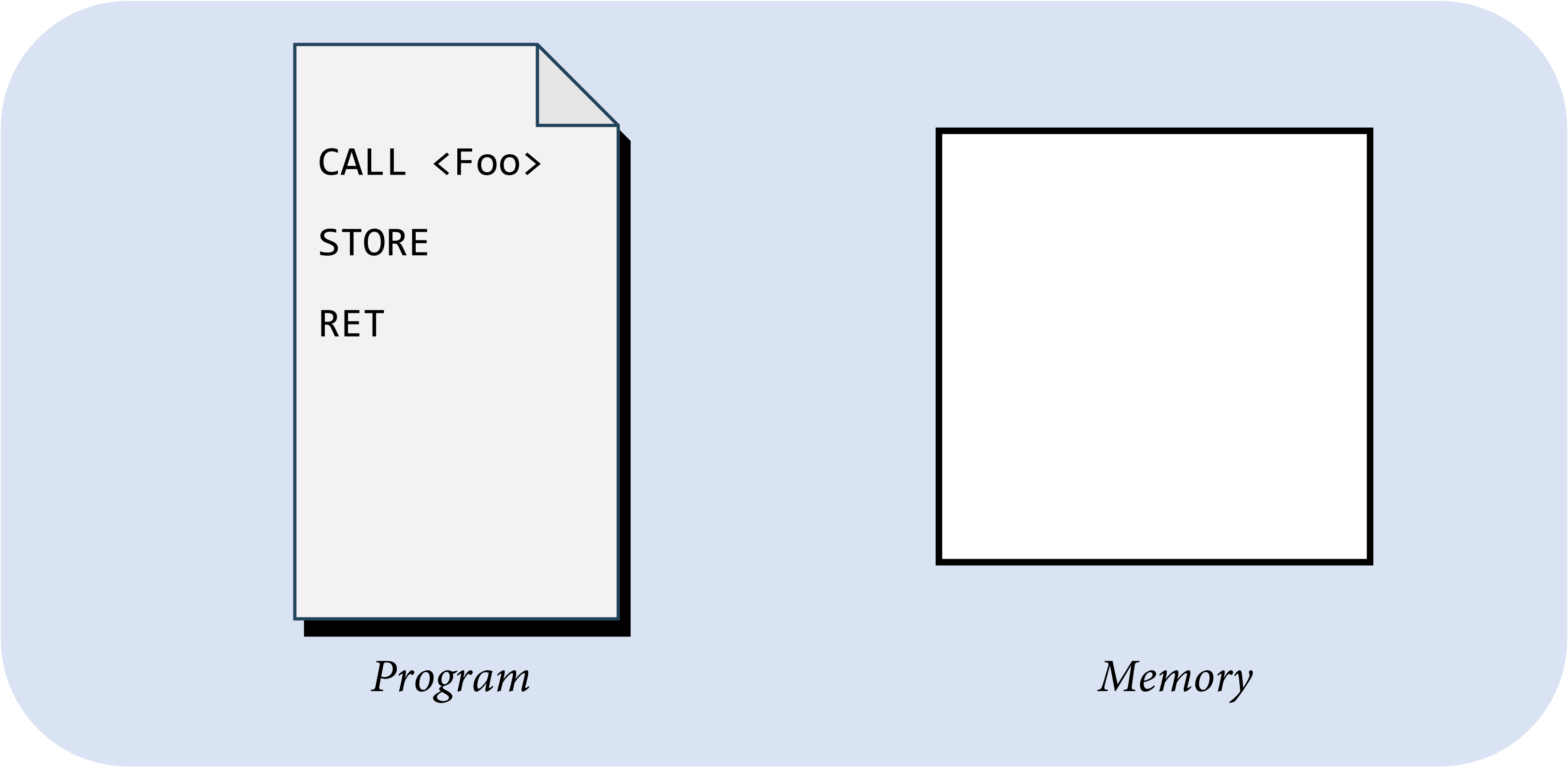


[ISCA 2021]



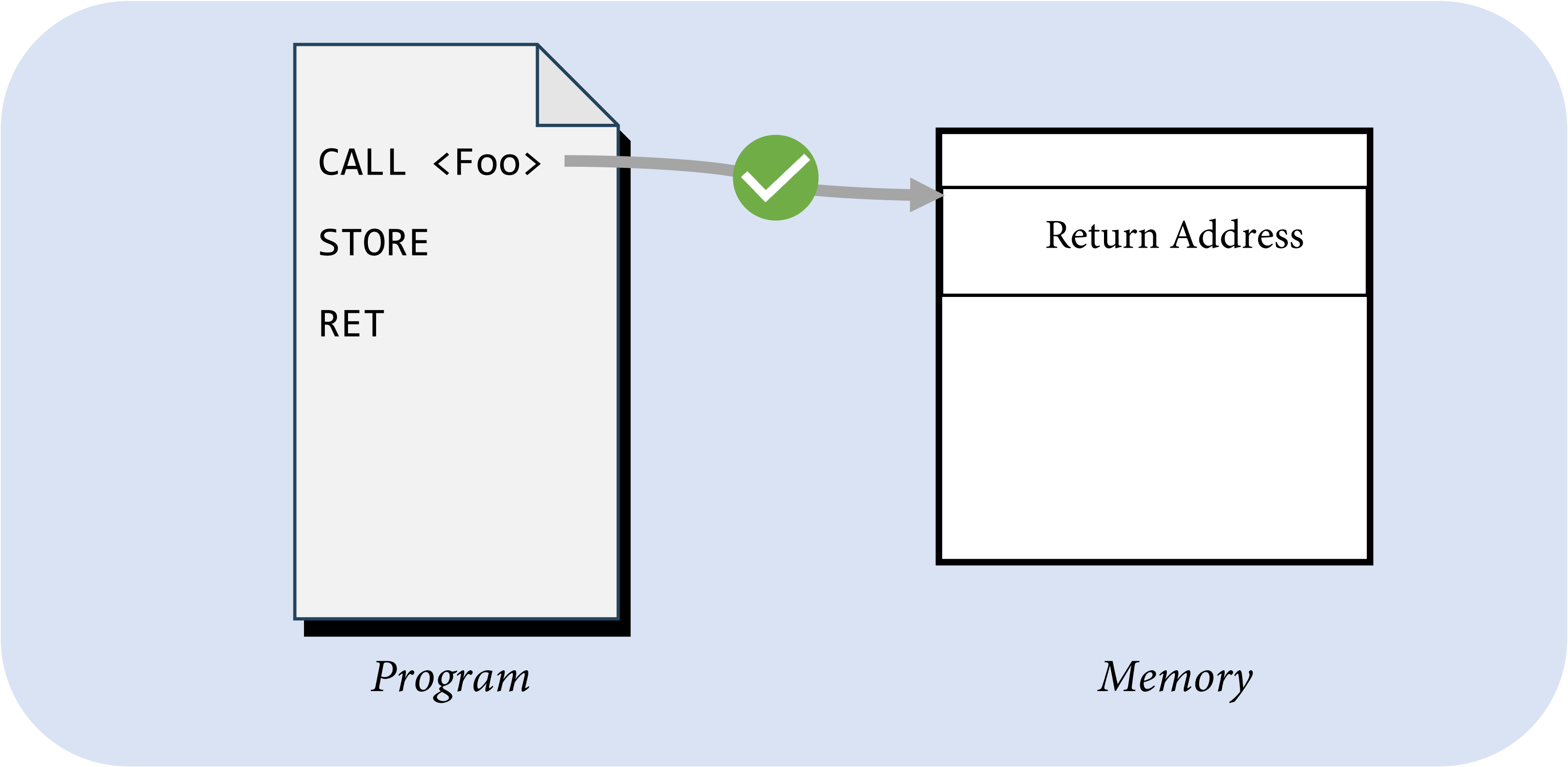
Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan,
ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks. [ISCA 2021]

Return Address Protection with ZeRØ

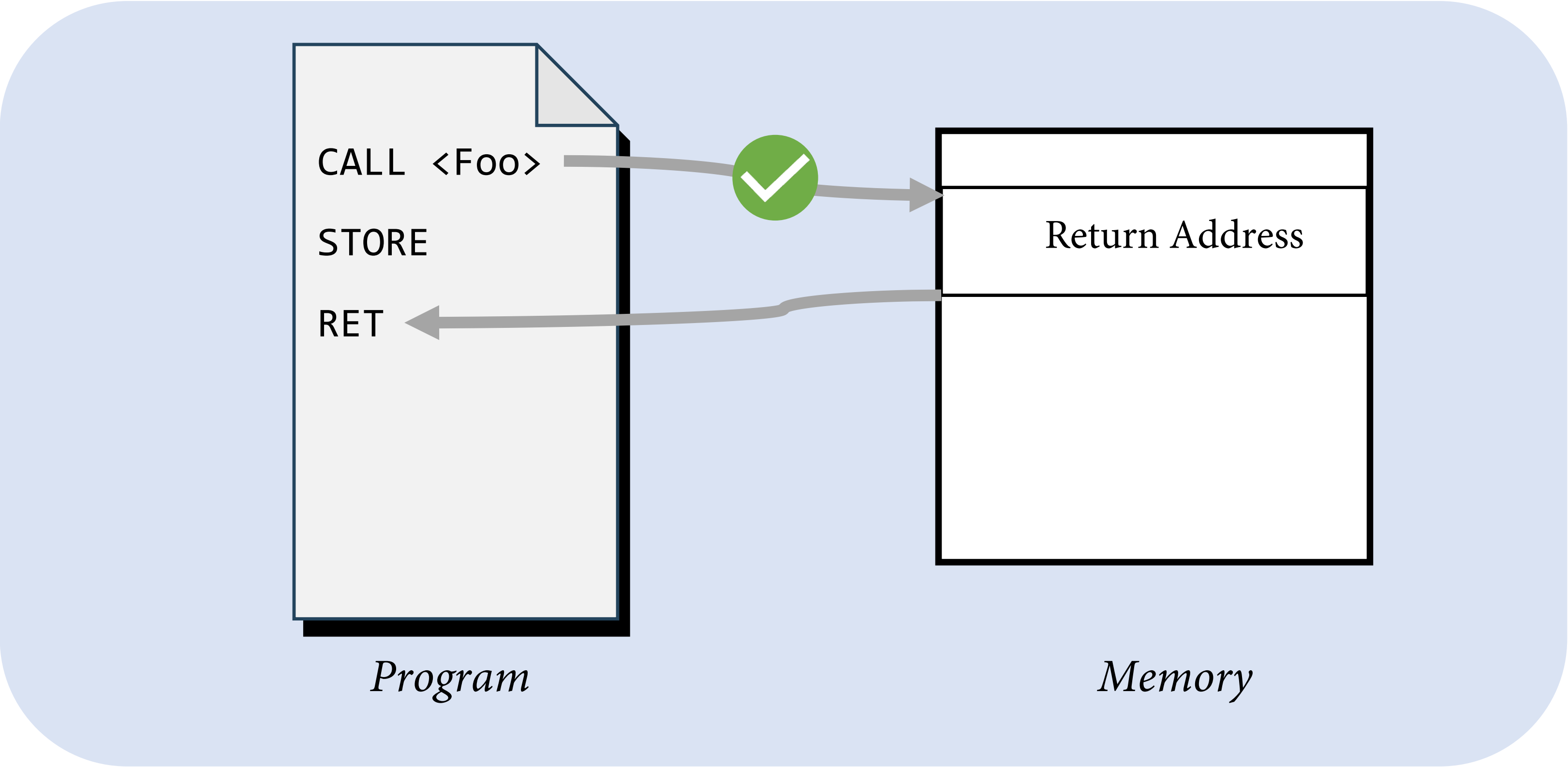


```
CALL <Foo>
STORE
RET
```

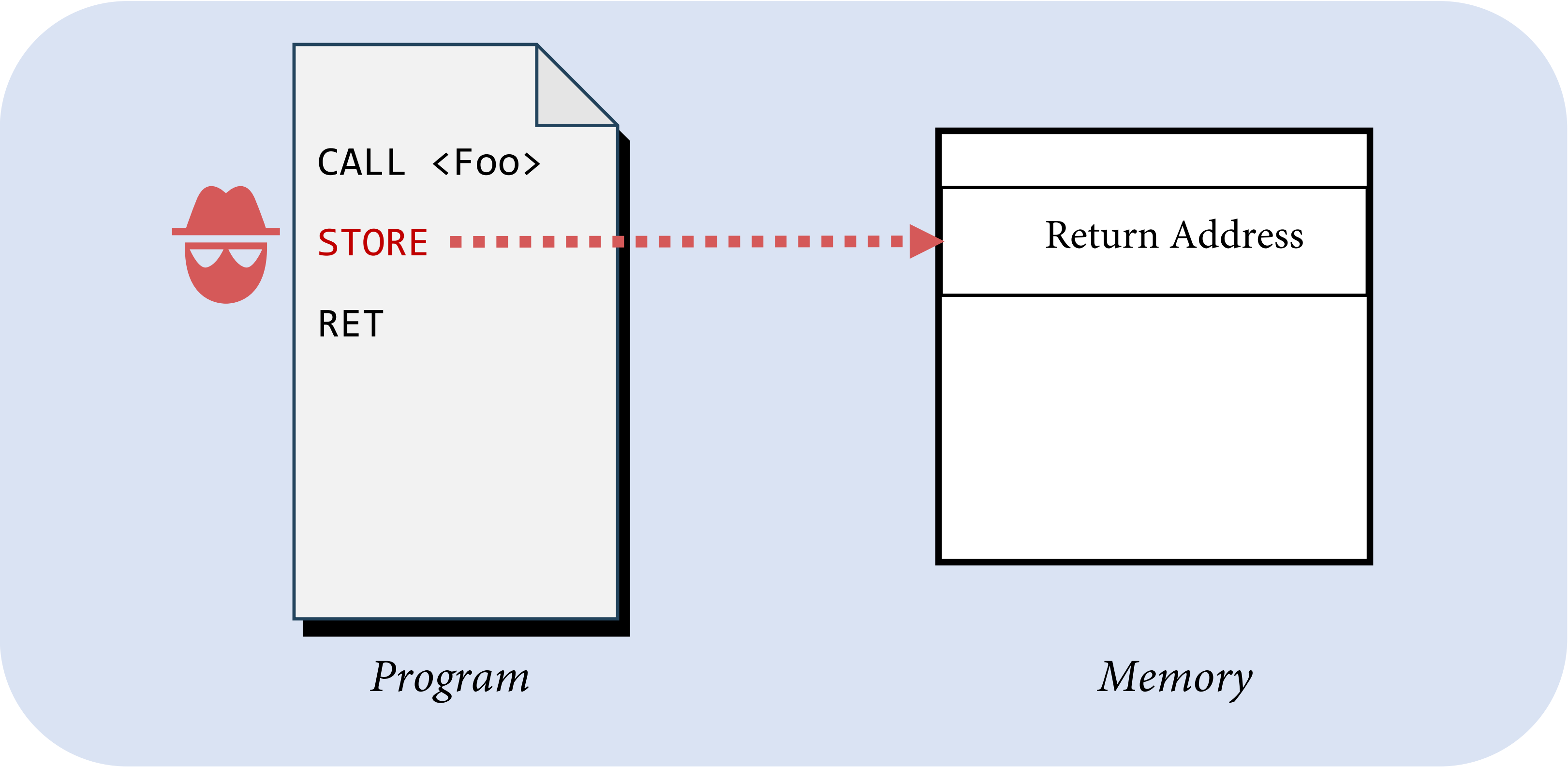
Return Address Protection with ZeRØ



Return Address Protection with ZeRØ

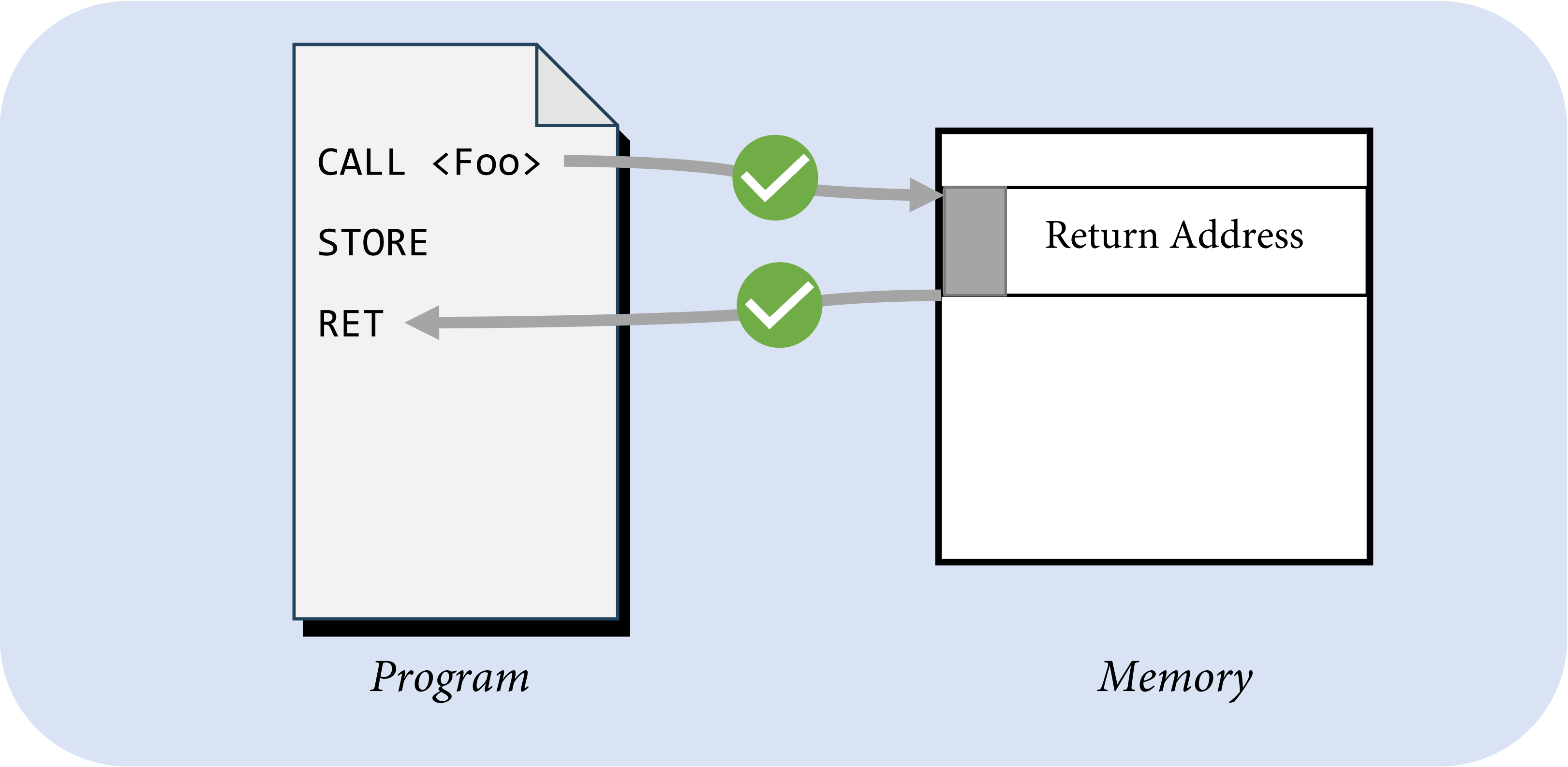


Return Address Protection with ZeRØ

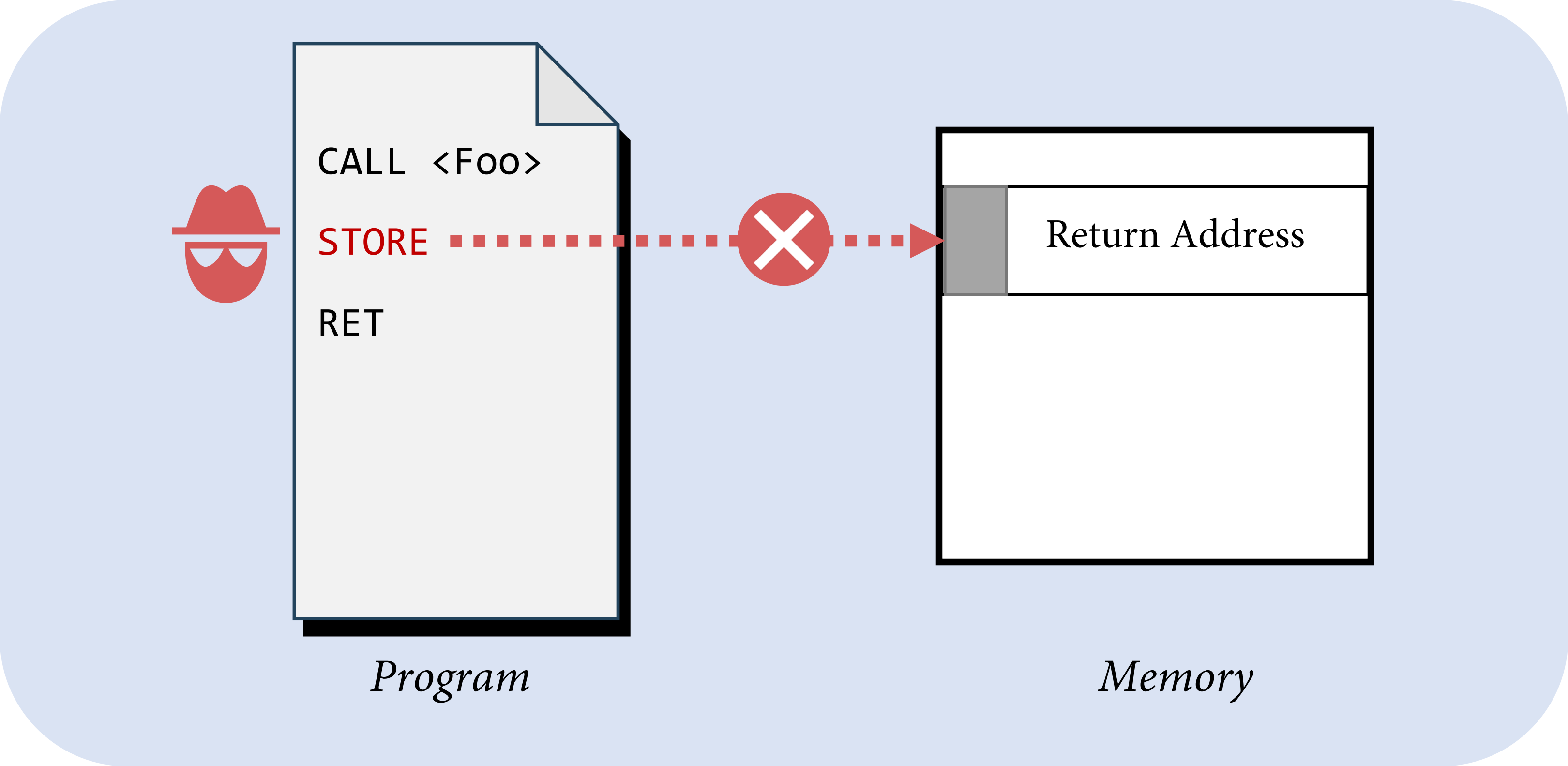


```
CALL <Foo>
STORE
RET
```

Return Address Protection with ZeRØ

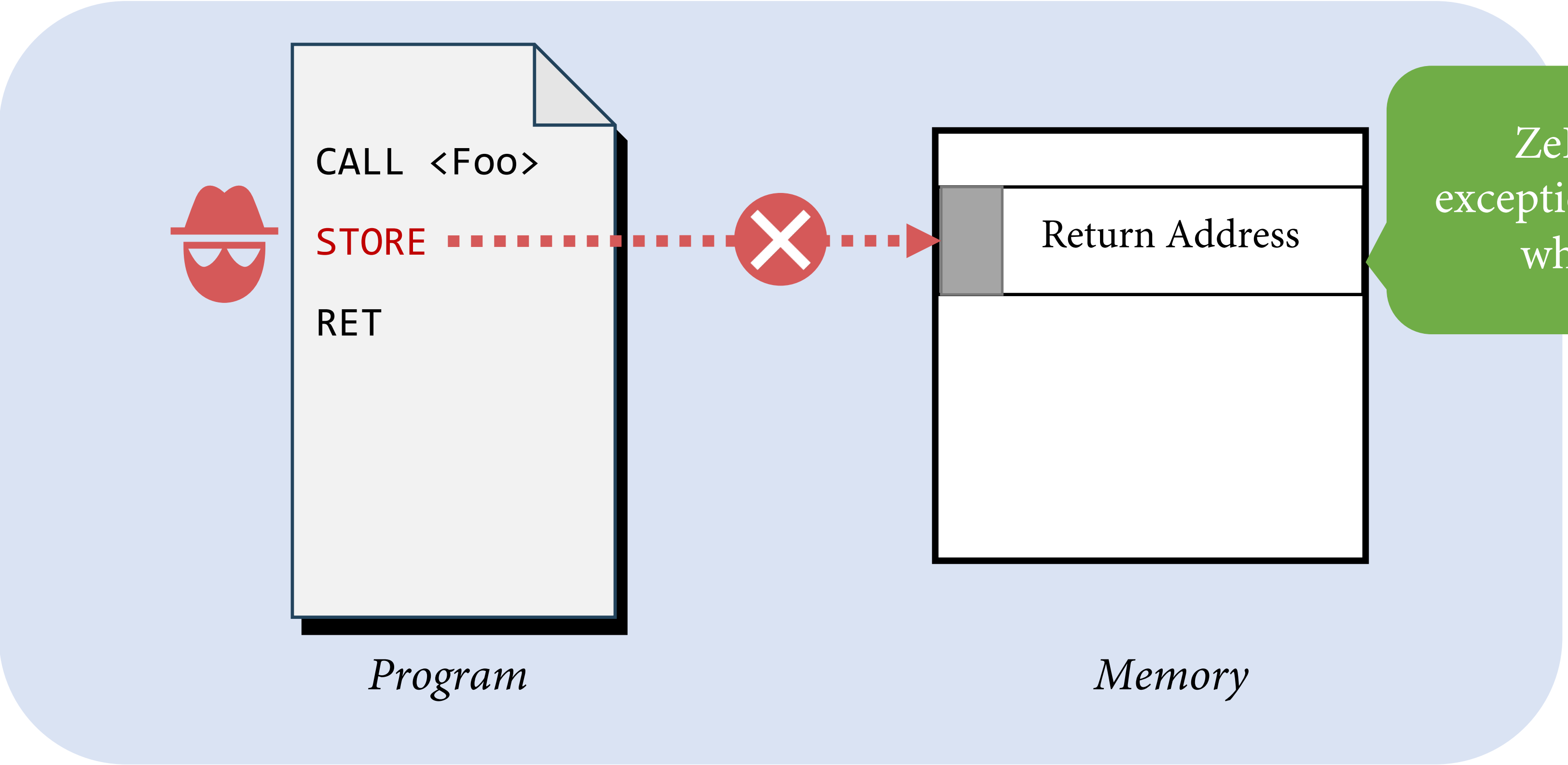


Return Address Protection with ZeRØ



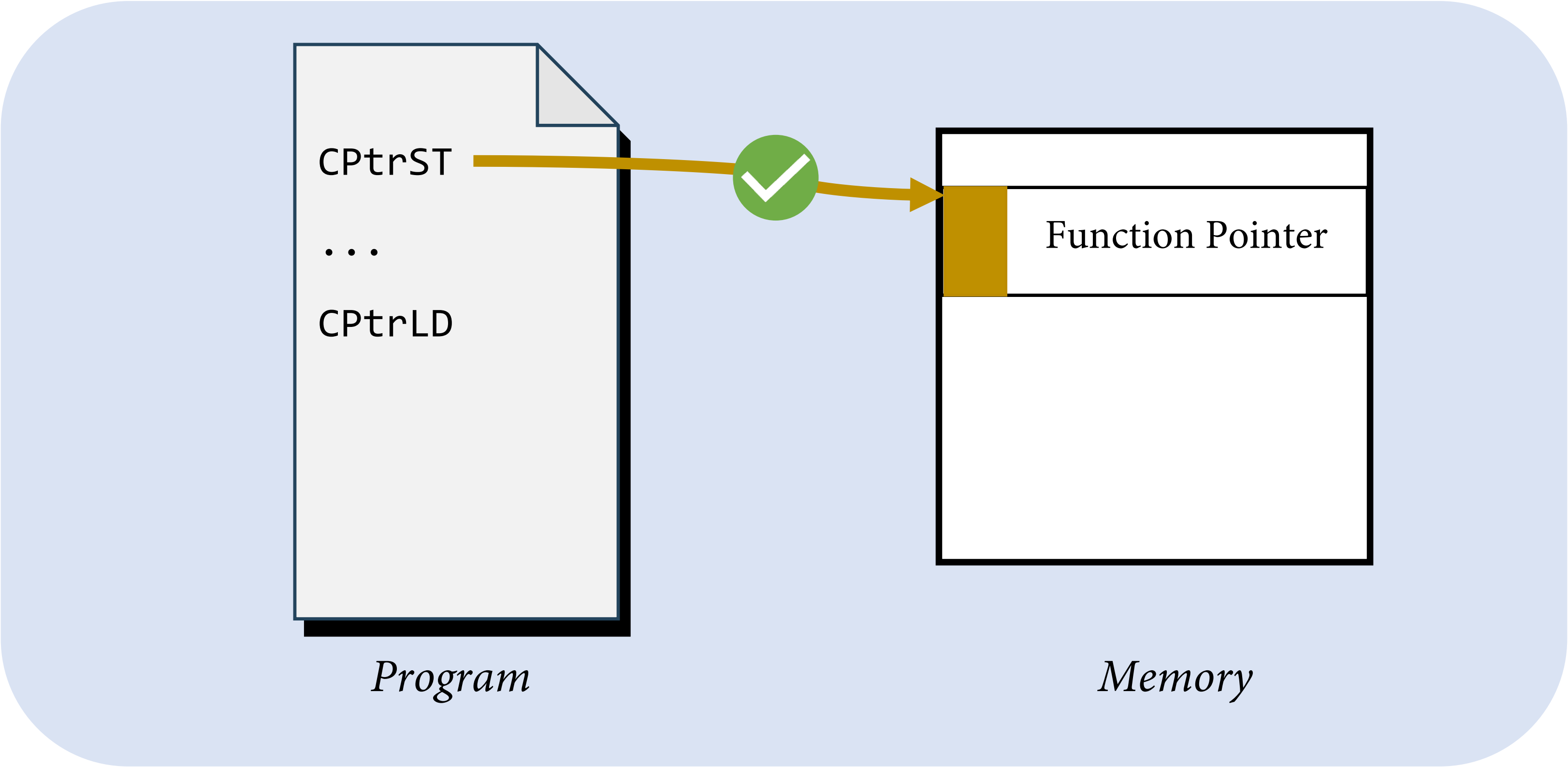
```
CALL <Foo>
STORE
RET
```

Return Address Protection with ZeRØ

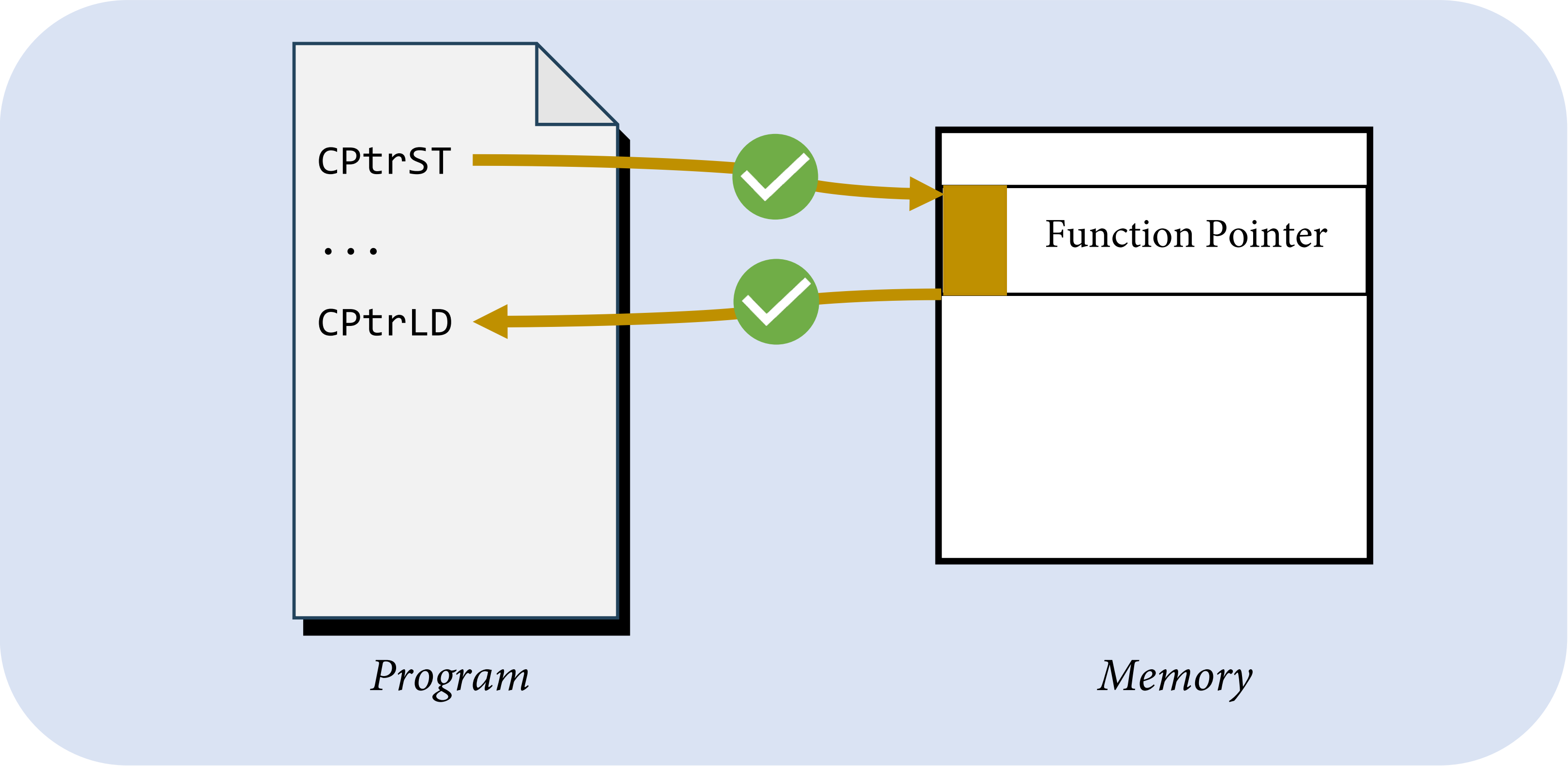


ZeRØ uses advisory exceptions to avoid crashing when under attack.

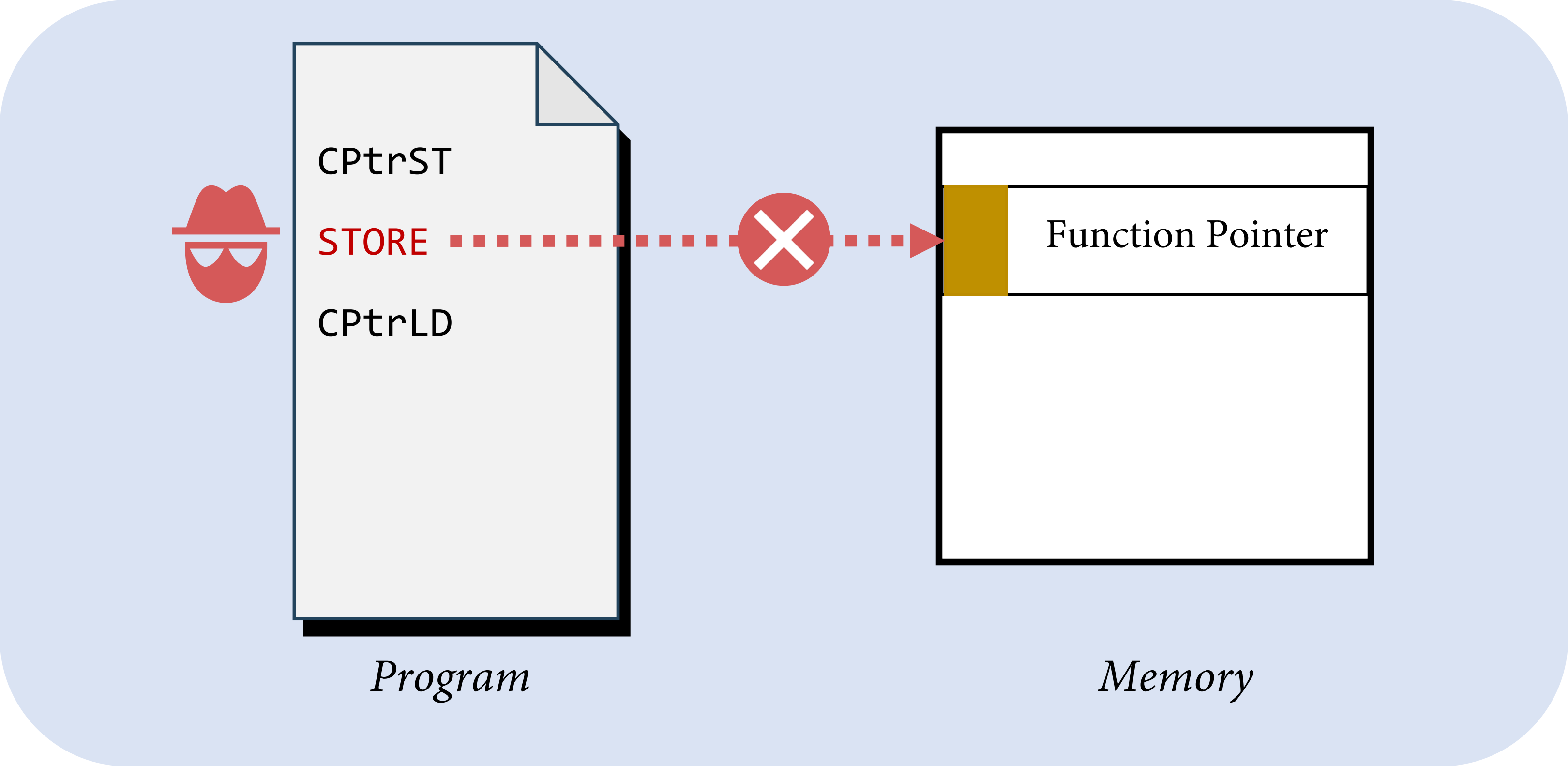
Code Pointer Integrity with ZeRØ



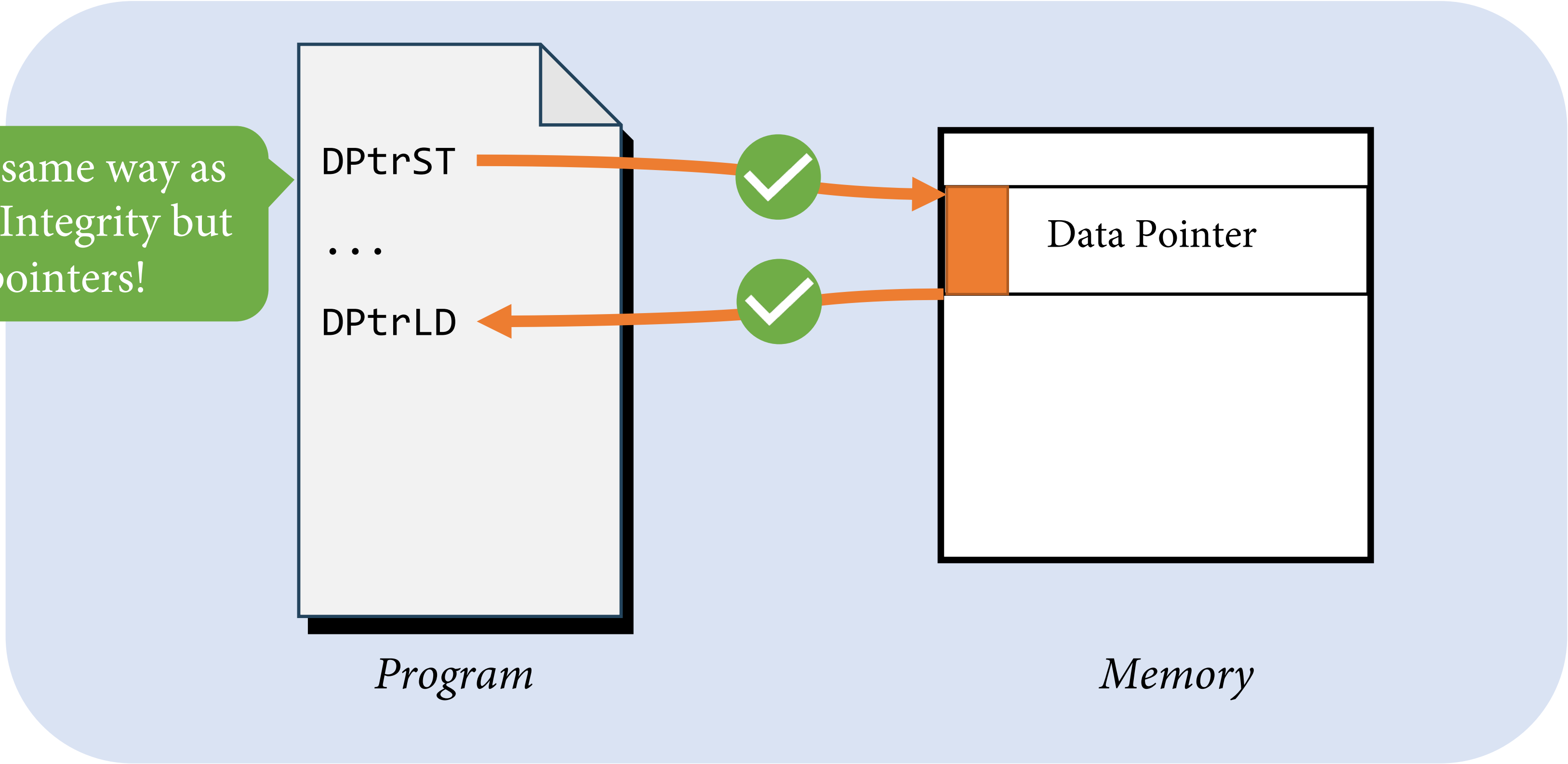
Code Pointer Integrity with ZeRØ



Code Pointer Integrity with ZeRØ

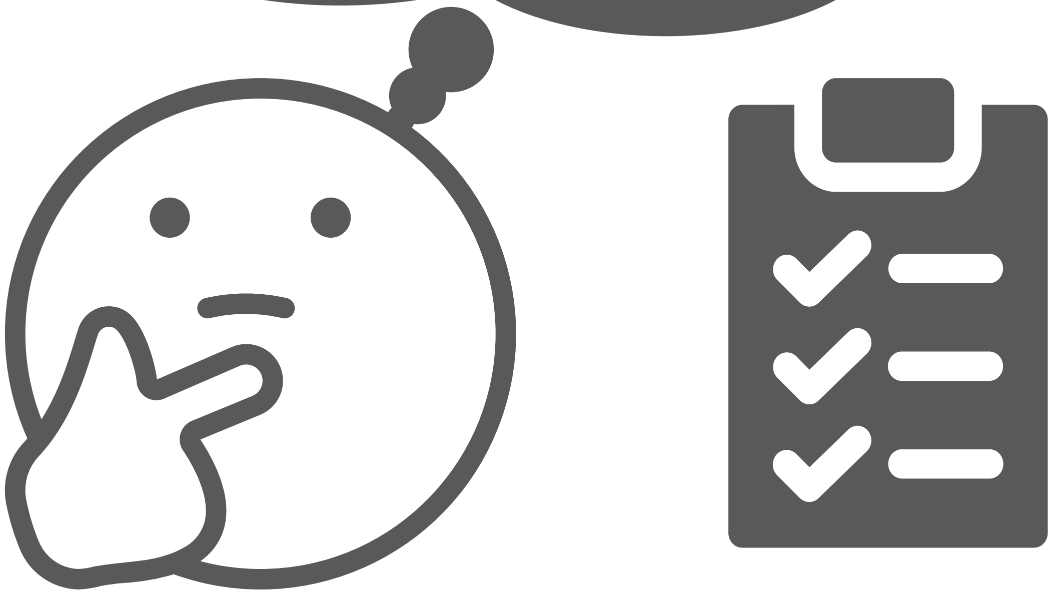


Data Pointer Integrity with ZeRØ



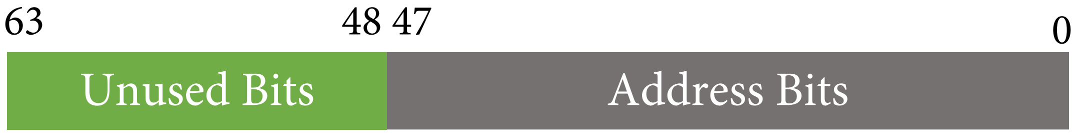
Works in the same way as Code Pointer Integrity but for data pointers!

How can we keep track of ZeRØ bits?



Efficiently Tracking Metadata

In ZeRØ, we encode metadata **within** unused pointer bits.



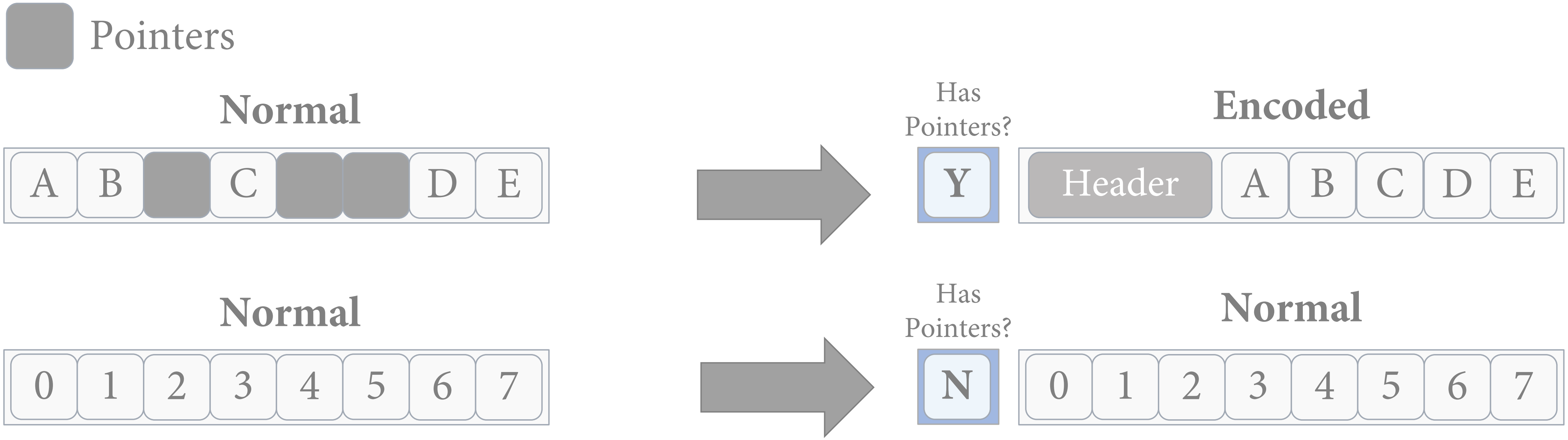
63 48 47 0

Unused Bits | Address Bits

64-bit Pointer

Efficiently Tracking Metadata

We use a novel variant of CaLiForms



■ Pointers

Normal

A	B	■	C	■	■	D	E

Has Pointers? Y

Encoded

Header	A	B	C	D	E

Normal

0	1	2	3	4	5	6	7

Has Pointers? N

Normal

0	1	2	3	4	5	6	7

ZeRØ Performance Overheads

Hardware Modifications

Our measurements show no impact on the cache access latency.

ZeRØ Performance Overheads

Hardware Modifications

Our measurements show no impact on the cache access latency.

00010010
101001101
00010010
111001001
00010010

Software Modifications

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

ZeRØ Performance Overheads

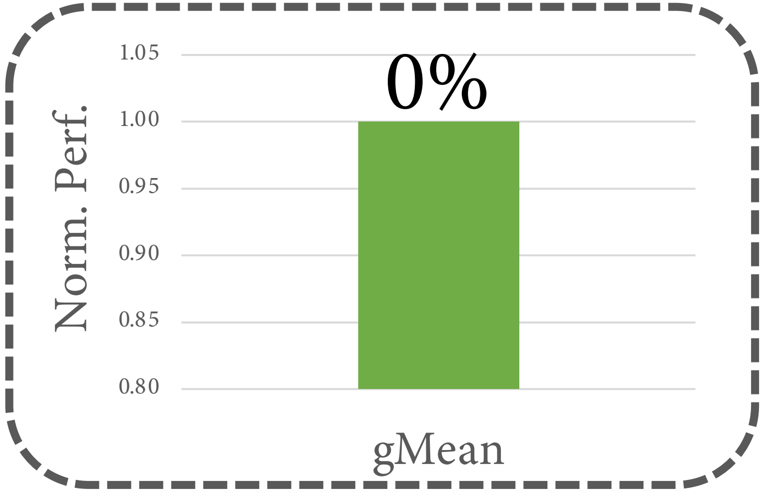
Hardware Modifications

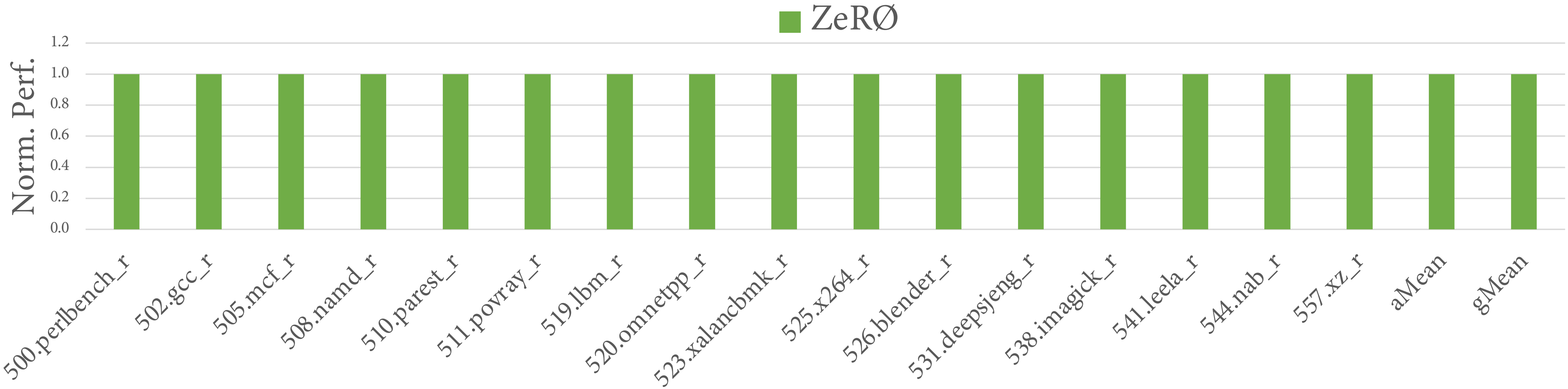
Our measurements show no impact on the cache access latency.

Software Modifications

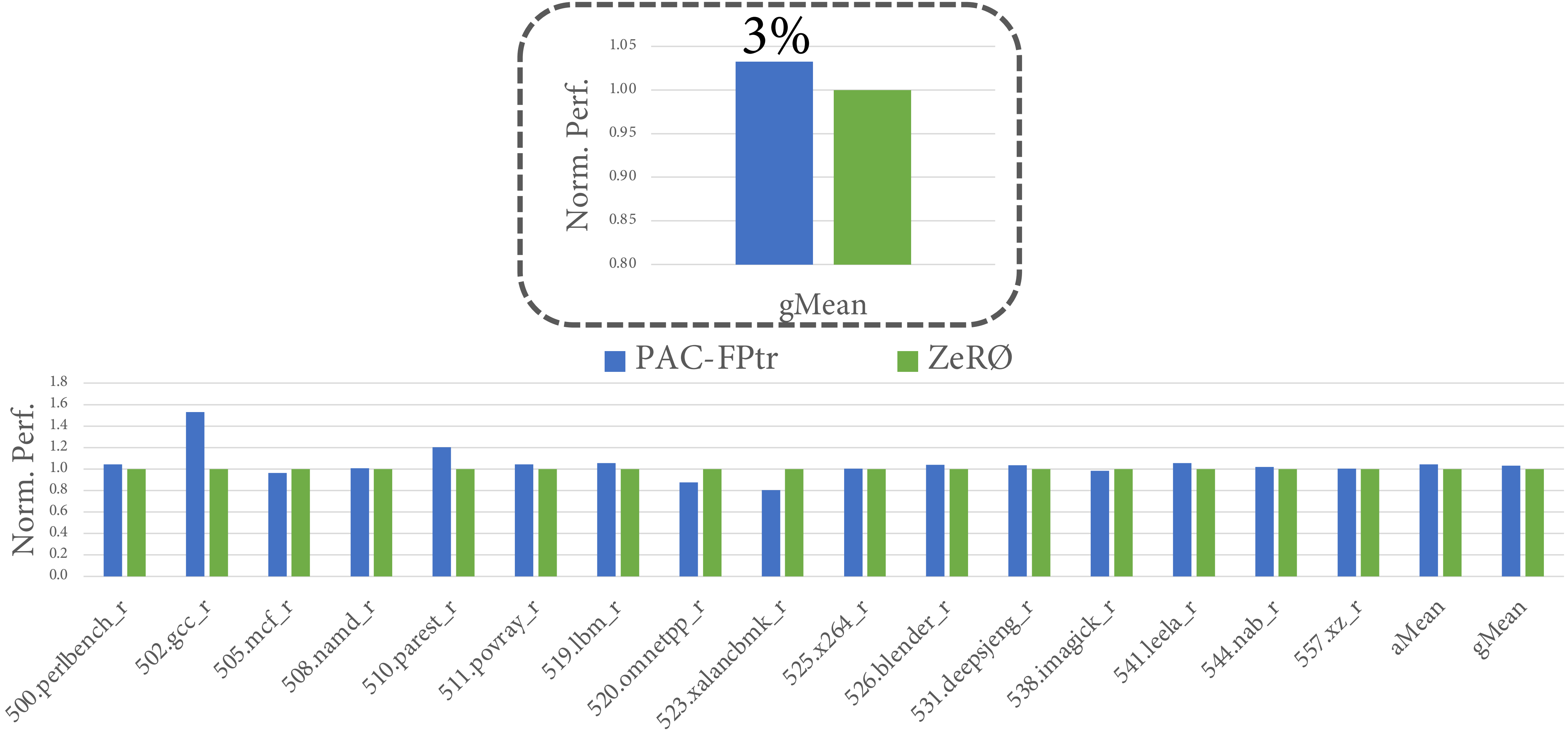
- Our special load/stores do not change the binary size.
- The `ClearMeta` instructions are only called on memory deletion.

ZeRØ Performance Results (x86_64)

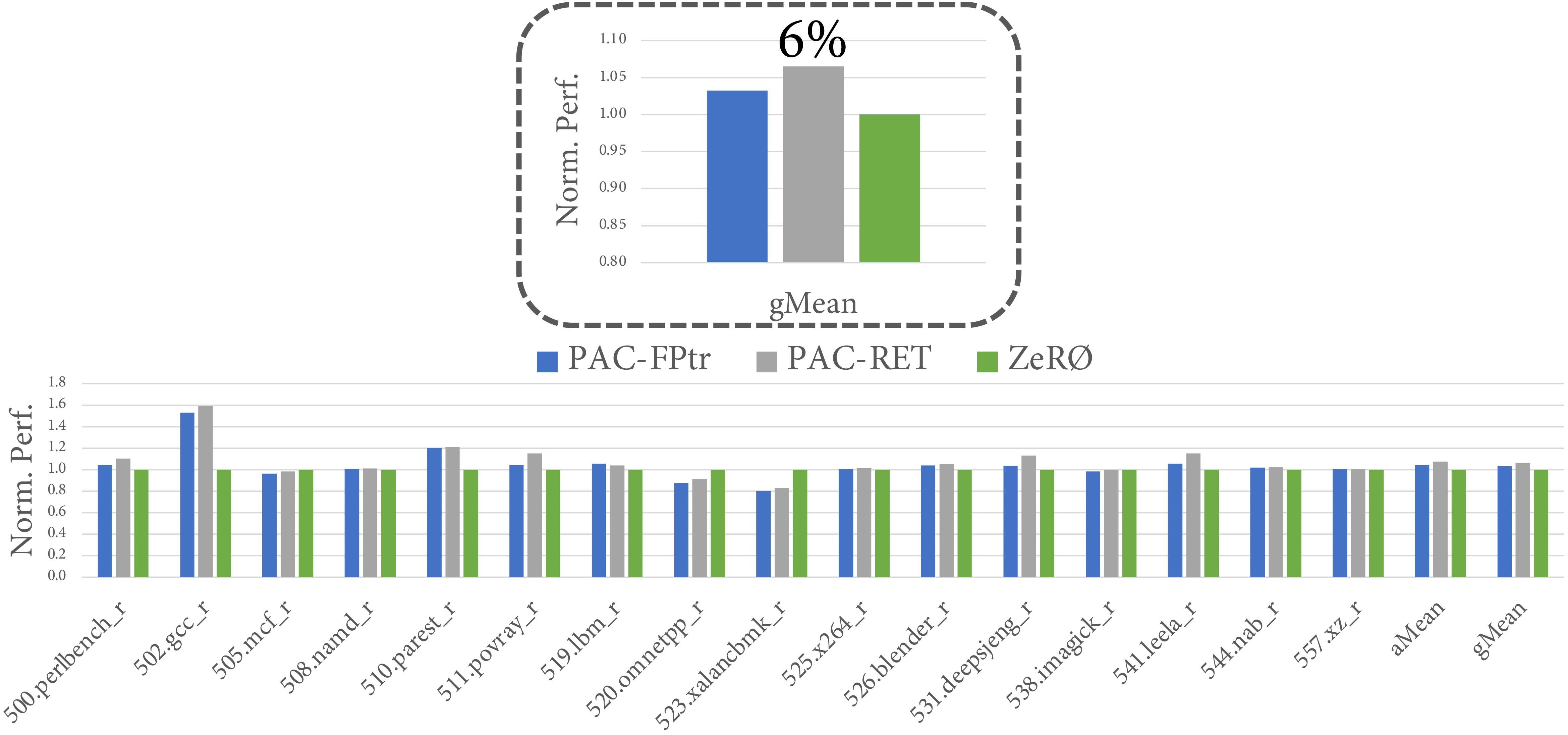




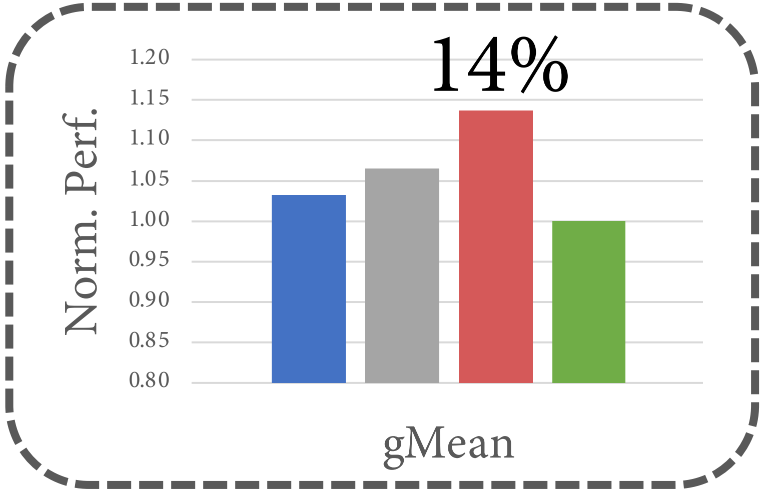
ZeRØ Performance Results (x86_64)

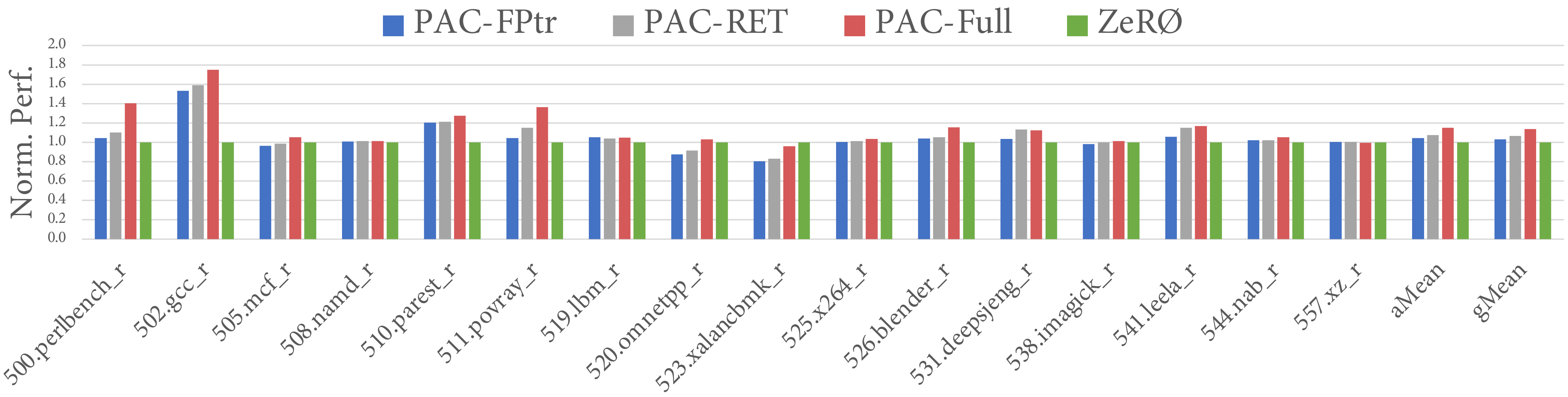


ZeRØ Performance Results (x86_64)

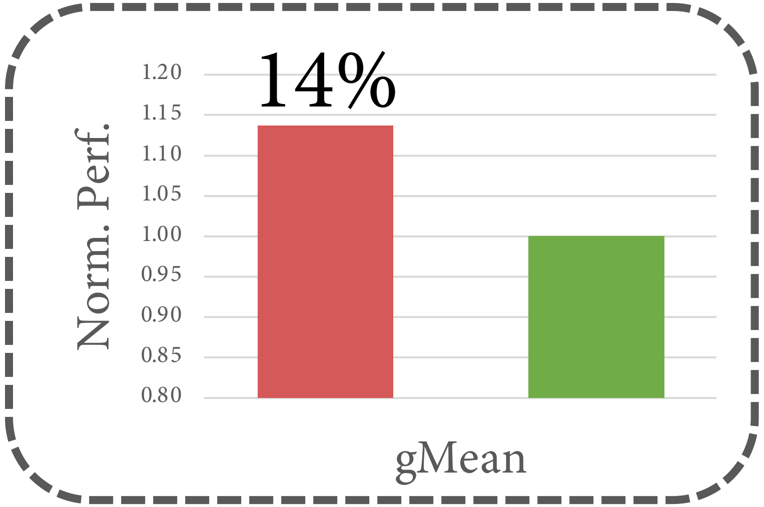


ZeRØ Performance Results (x86_64)





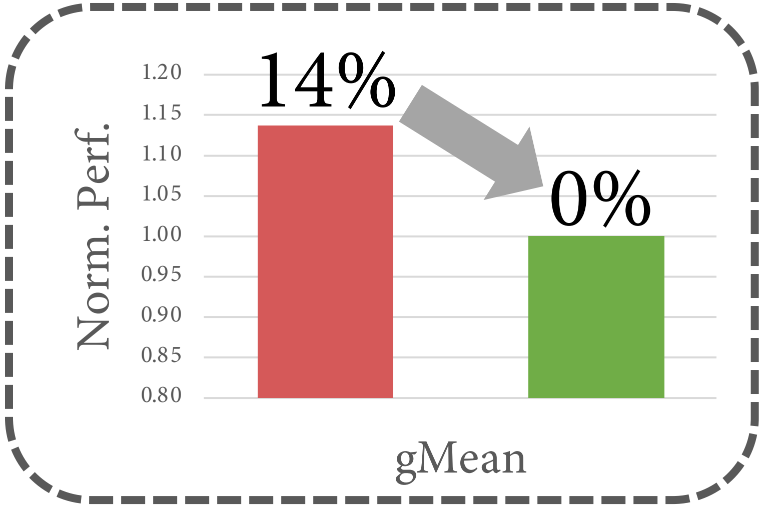
ZeRØ Performance Results (x86_64)



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

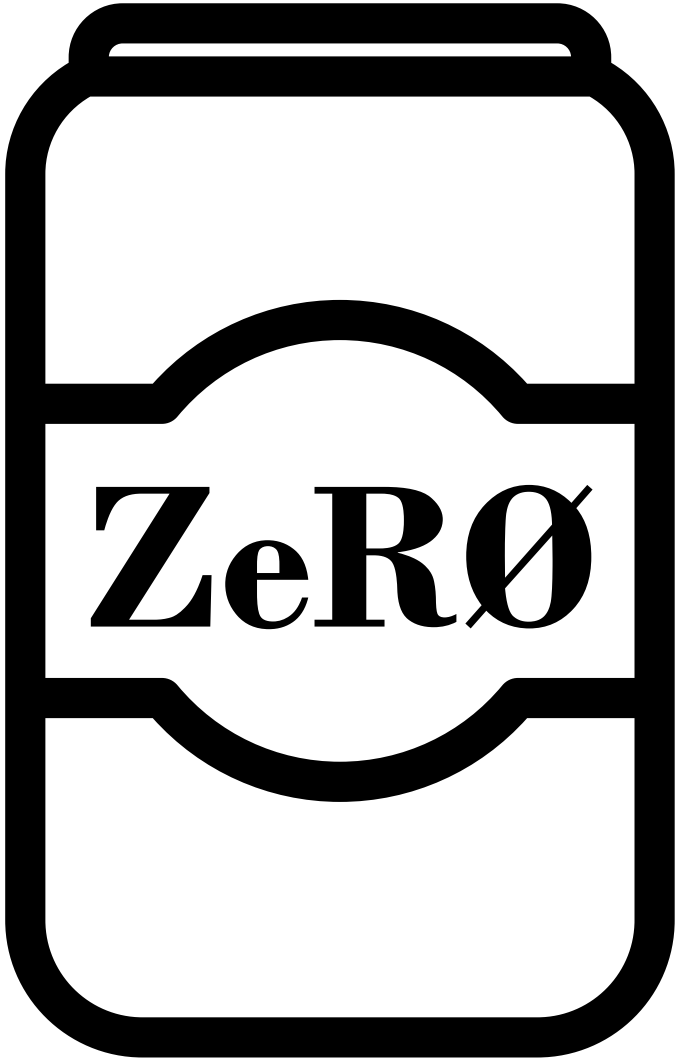
- loads/stores
- usages

ZeRØ Performance Results (x86_64)



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

An efficient pointer integrity mechanism



An ideal candidate for end-user deployment.

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

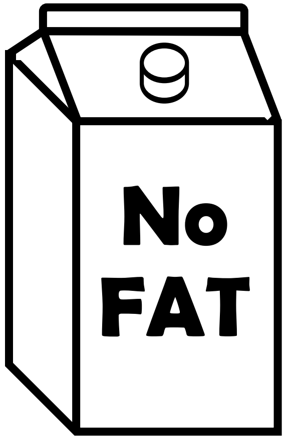
A drop-in replacement for ARM’s PAC

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

Memory Blocklisting

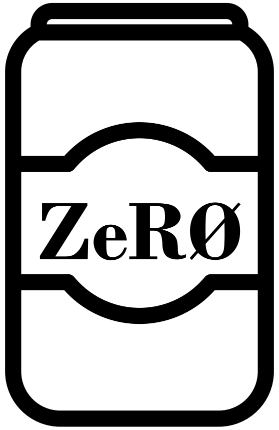
[MICRO 2019]

Memory Permitlisting



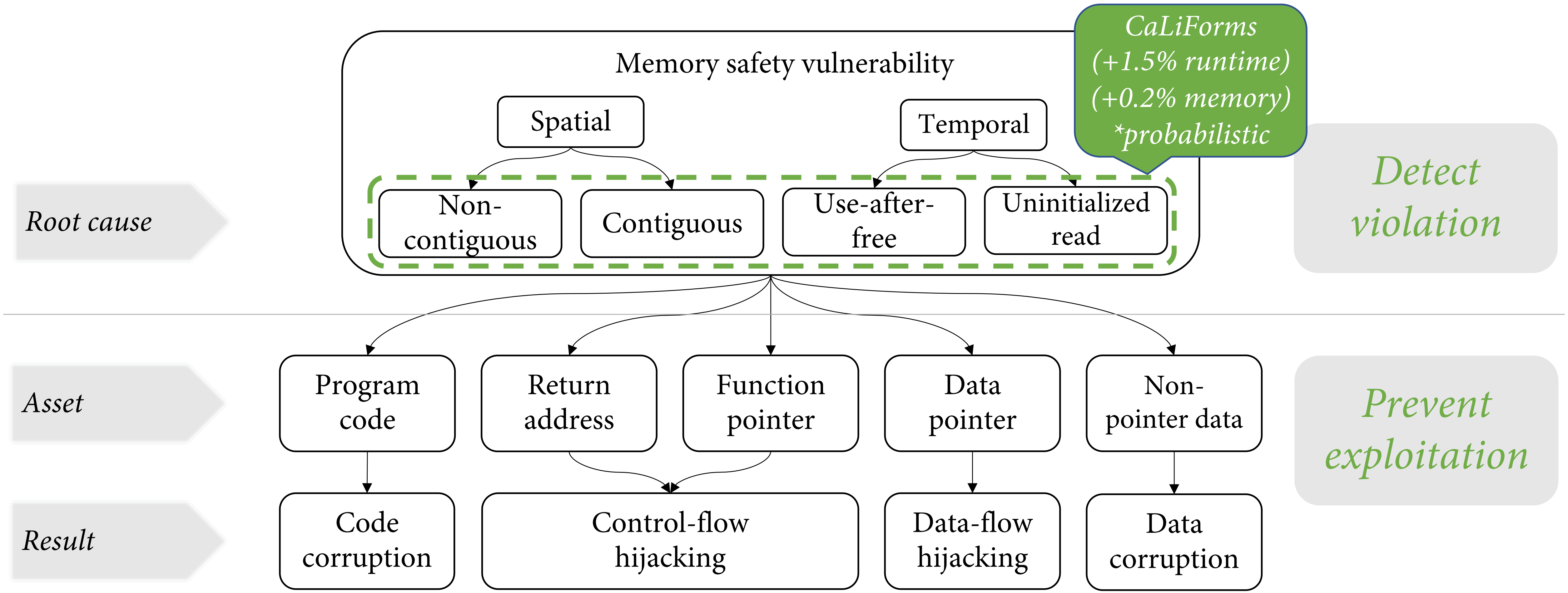
[ISCA 2021]

Exploit Mitigation

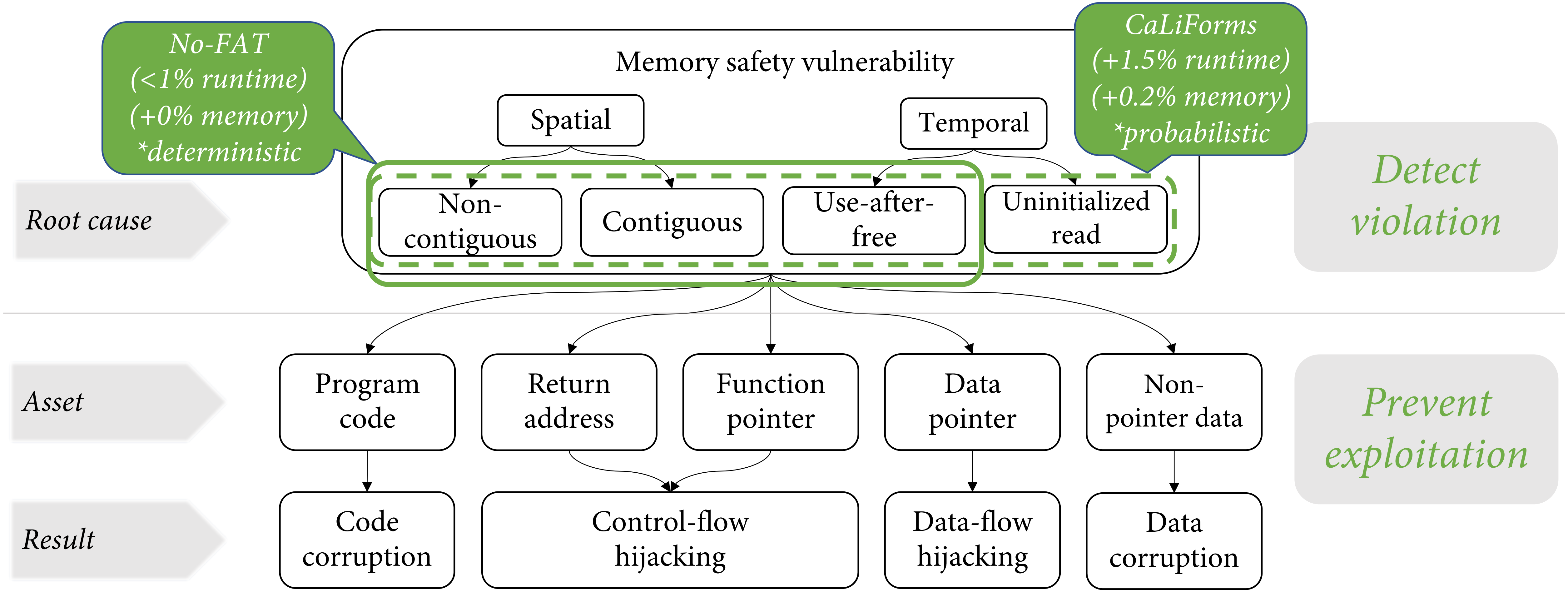


[ISCA 2021]

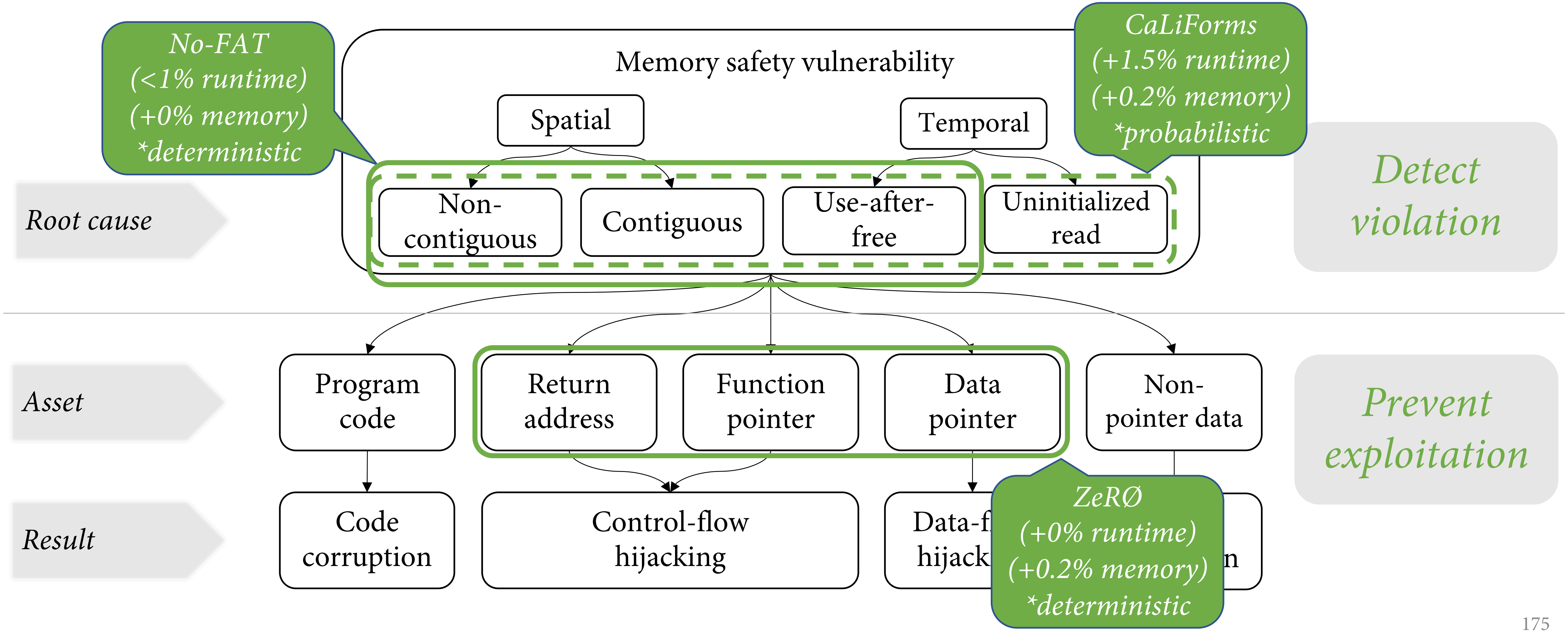
Memory Attacks Taxonomy



Memory Attacks Taxonomy



Memory Attacks Taxonomy



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

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

Root cause

Memory safety vulnerability

Spatial

Temporal

Non-contiguous

Contiguous

Use-after-free

Uninitialized read

Detect violation

Asset

Program code

Return address

Function pointer

Data pointer

Non-pointer data

Prevent exploitation

Result

Code corruption

Control-flow hijacking

Data-f
hijack

n

ZerØ
(+0% runtime)
(+0.2% memory)
*deterministic

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

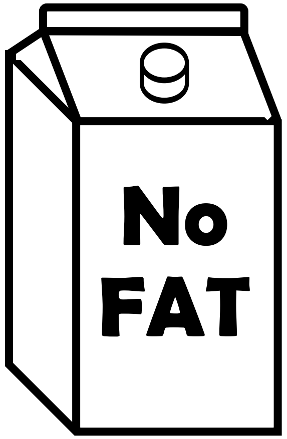
Tokyo Institute of Technology

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

Memory Blocklisting

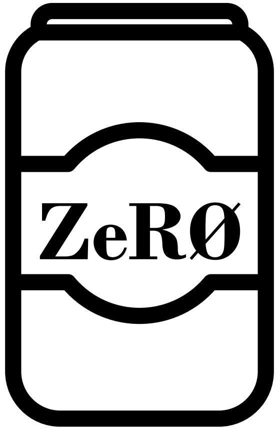
[MICRO 2019]

Memory Permitlisting



[ISCA 2021]

Exploit Mitigation



[ISCA 2021]