EPI

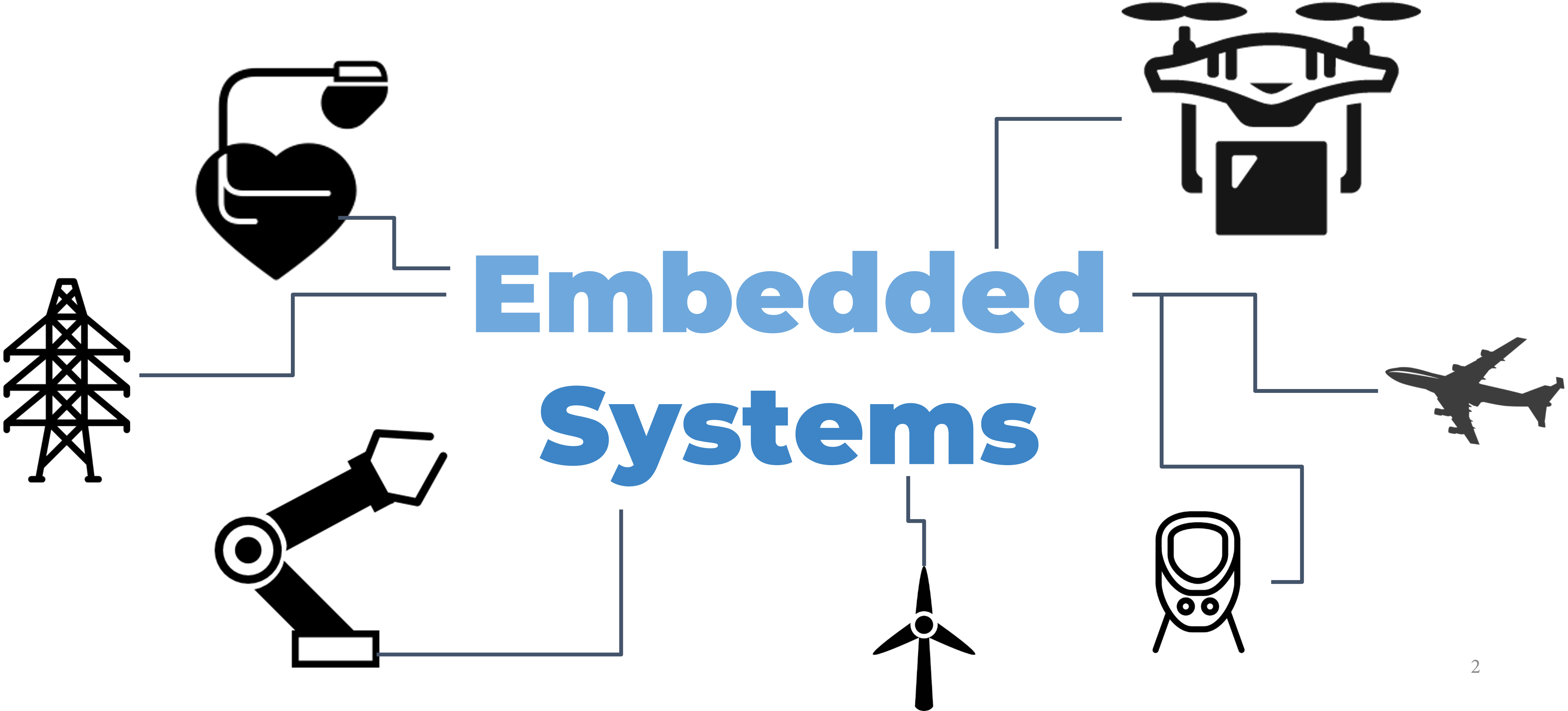
Efficient Pointer Integrity For Securing Embedded Systems

Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Vasileios P. Kemerlis, and Simha Sethumadhavan

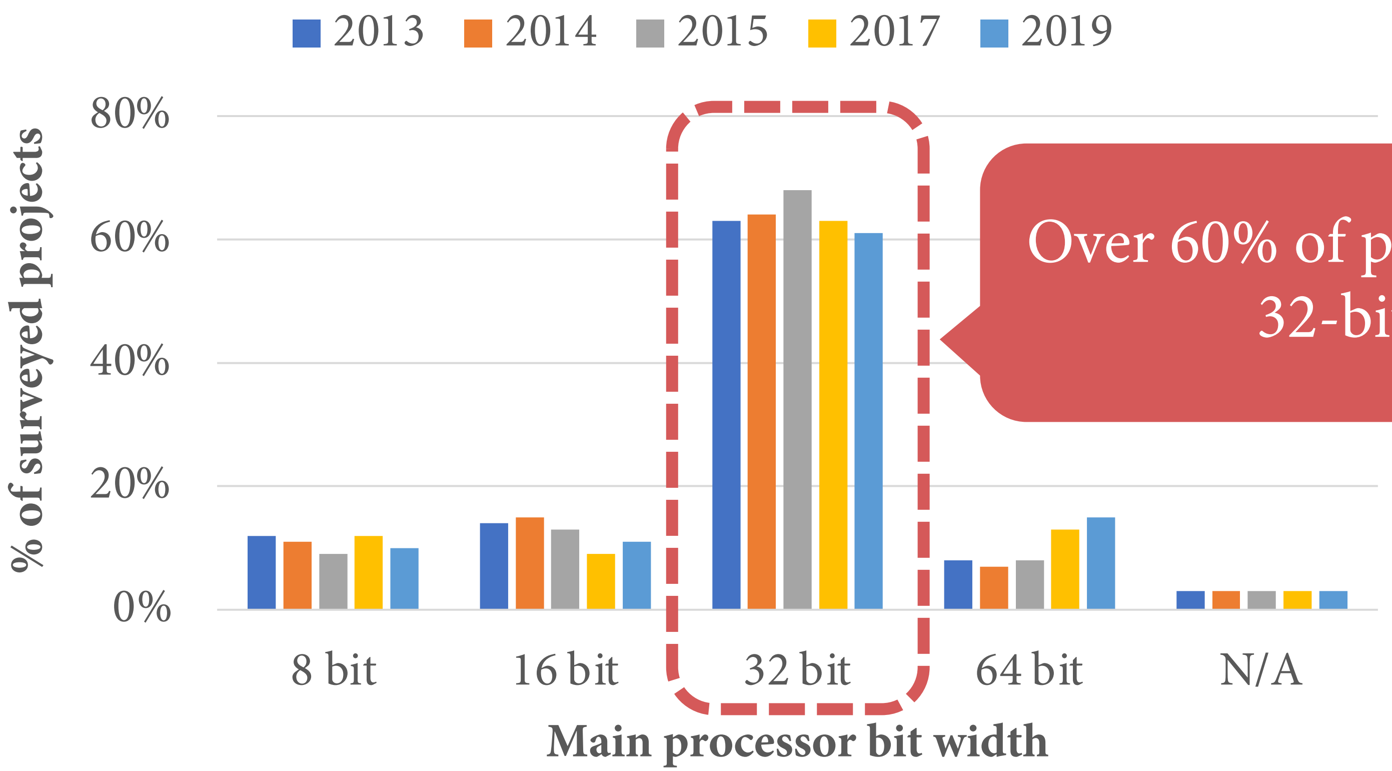
Columbia University
Brown University

09/21/2021

Embedded systems are everywhere!



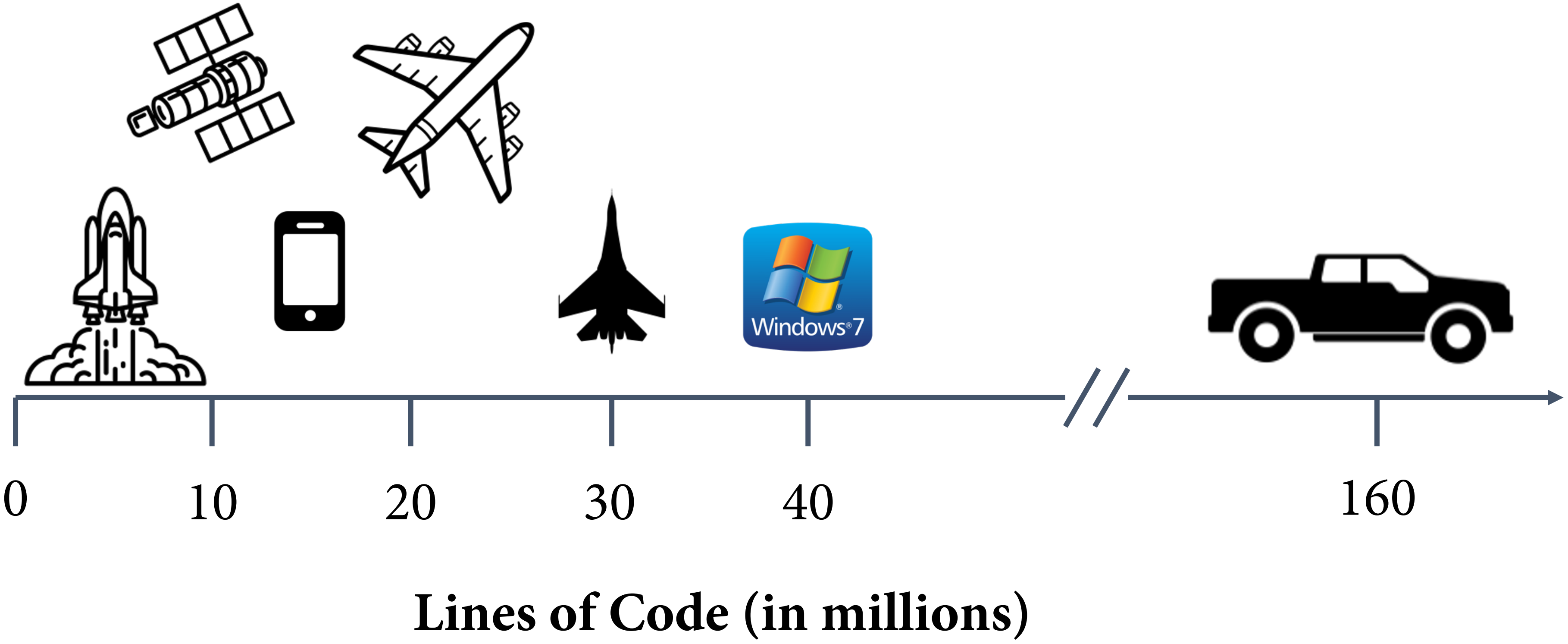
Embedded systems are dominated by 32-bit.



Over 60% of projects are 32-bit!

Why embedded system security is important?

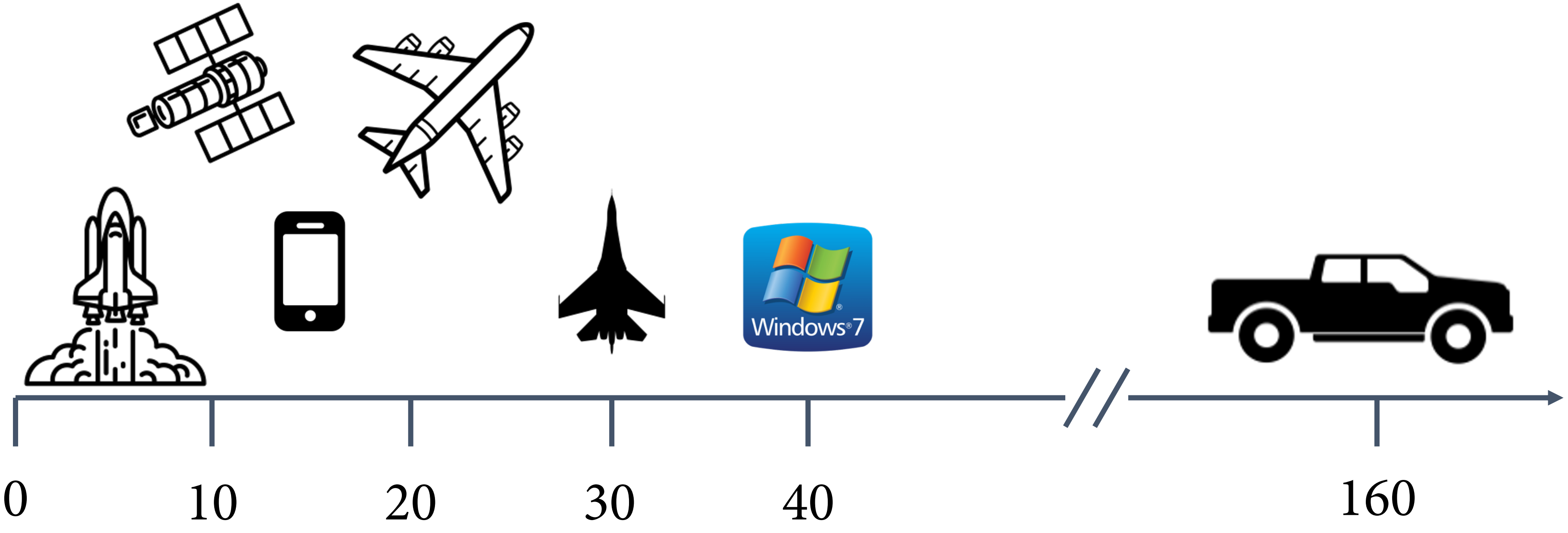
Software has become increasingly complex.



Source: *http://bit.ly/KIB_linescode*

Why embedded system security is important?

Software has become increasingly complex.



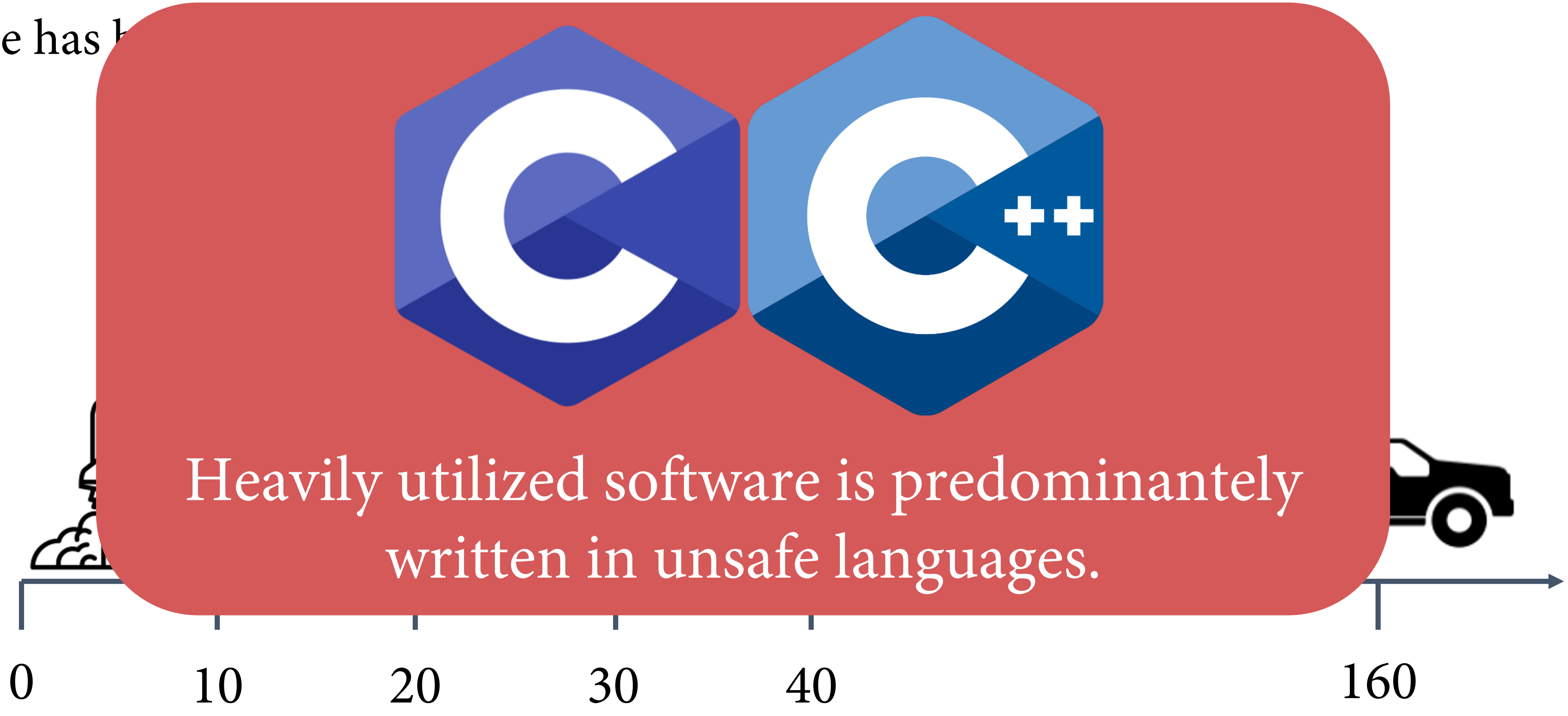
~~Lines of Code (in millions)~~

Number of Bugs

Source: *http://bit.ly/KIB_linescode*

Why embedded system security is important?

Software has b



Heavily utilized software is predominantely written in unsafe languages.

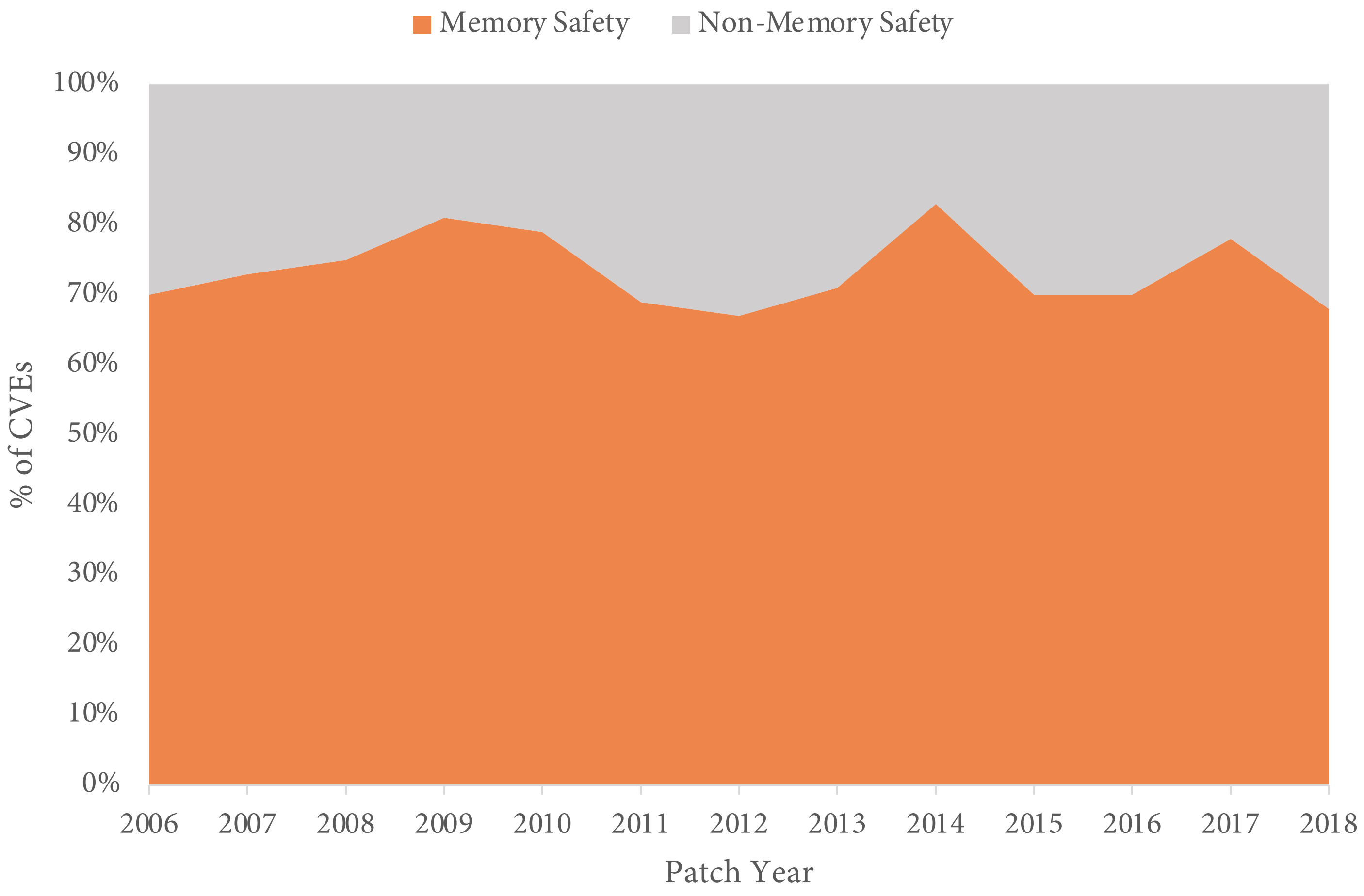
~~Lines of Code (in millions)~~

Number of Bugs

Source: *http://bit.ly/KIB_linescode*

Why Memory Safety?

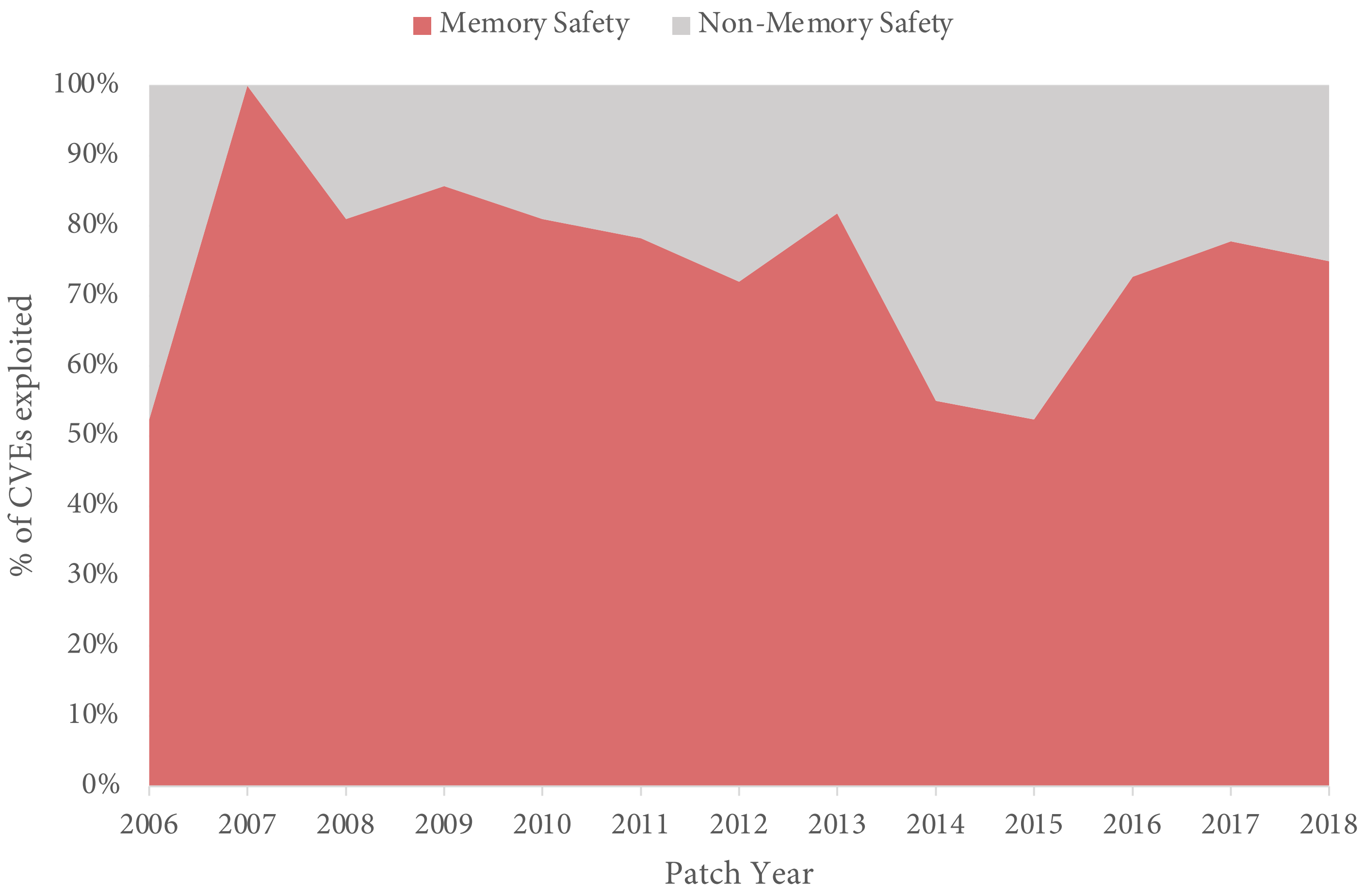
It is the predominant source of vulnerabilities (ie. CVEs).



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

Why Memory Safety?

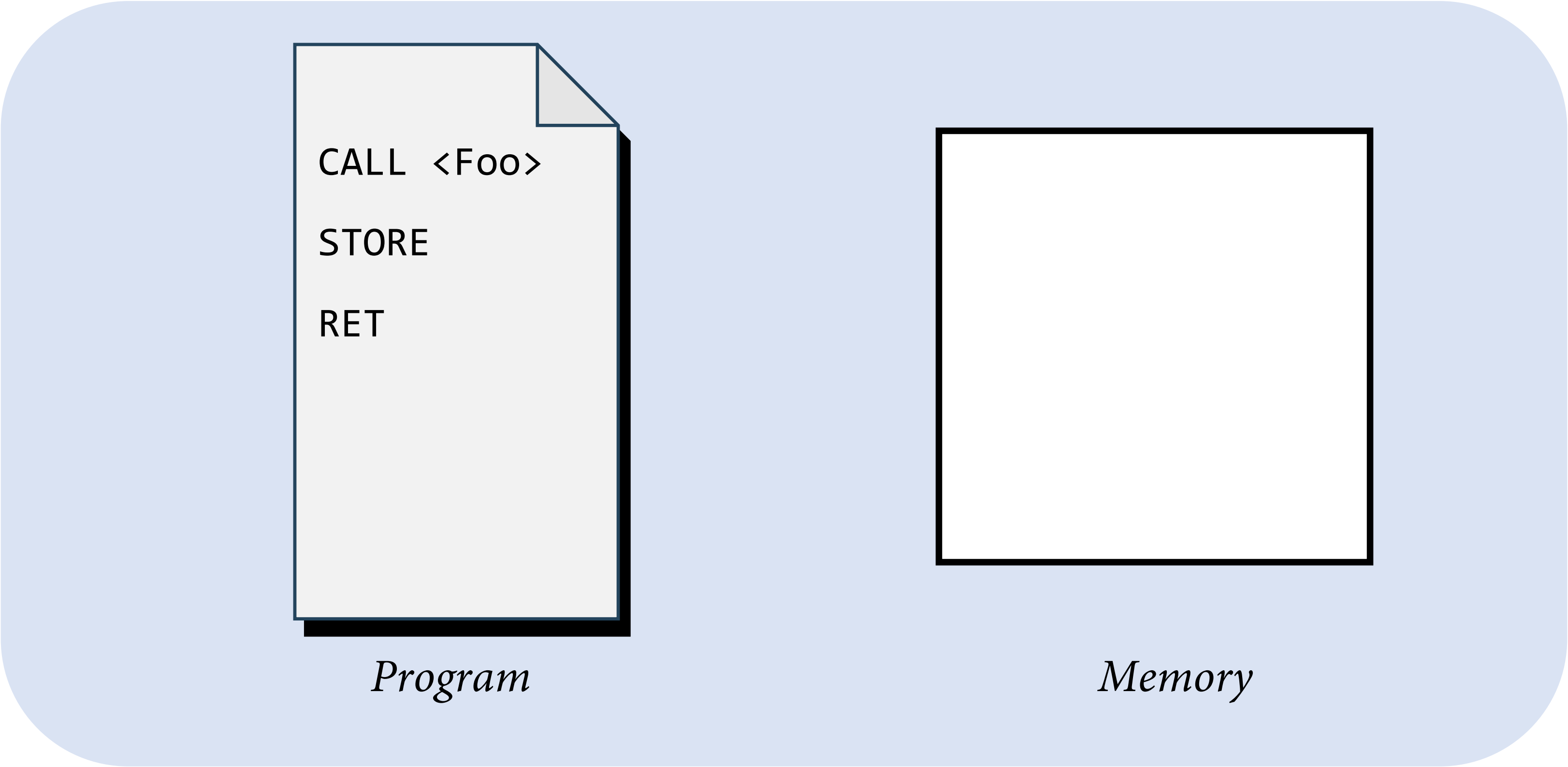
Memory Safety CVEs are heavily exploited.



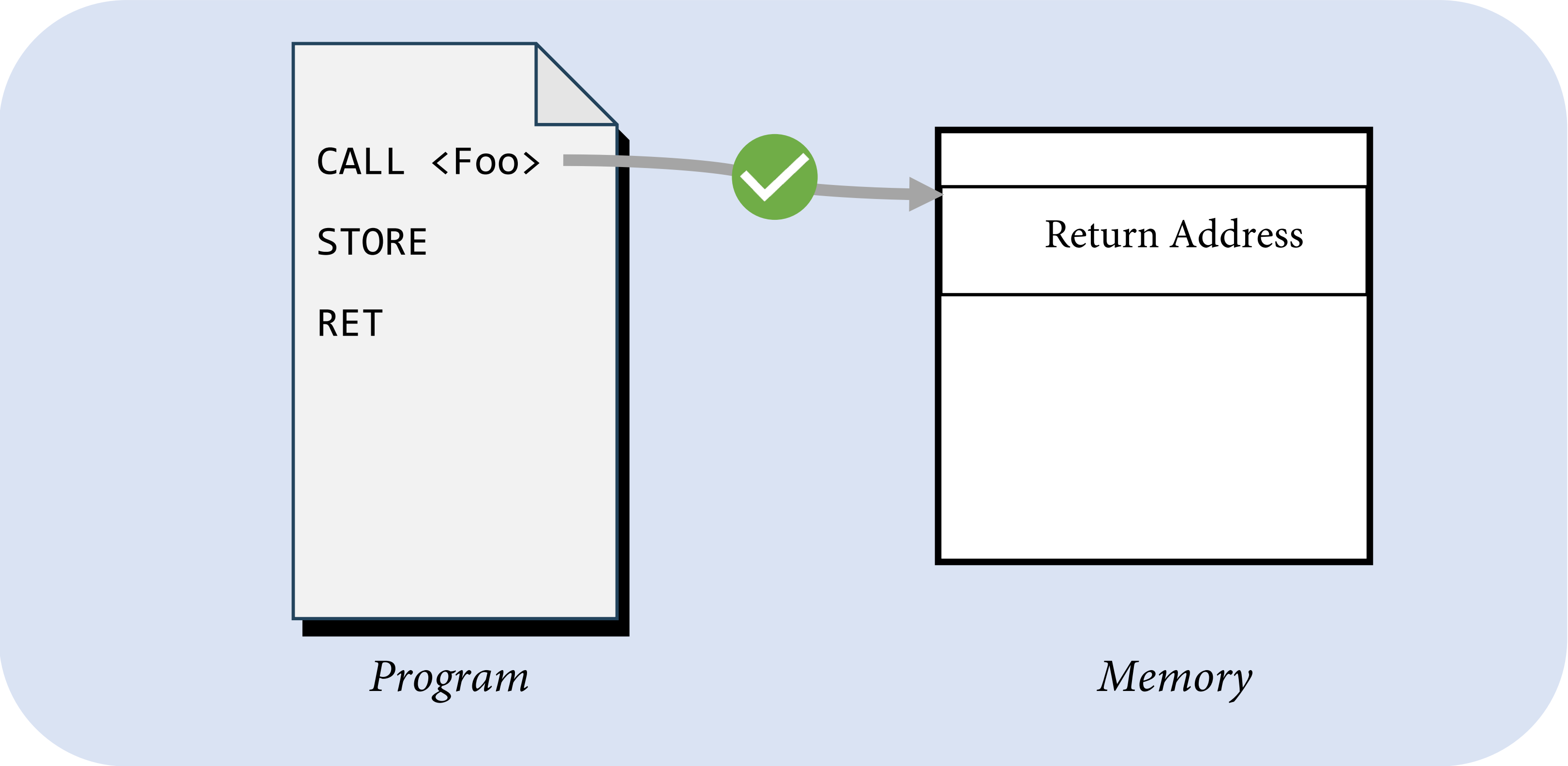
Source: *Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019*

EPI

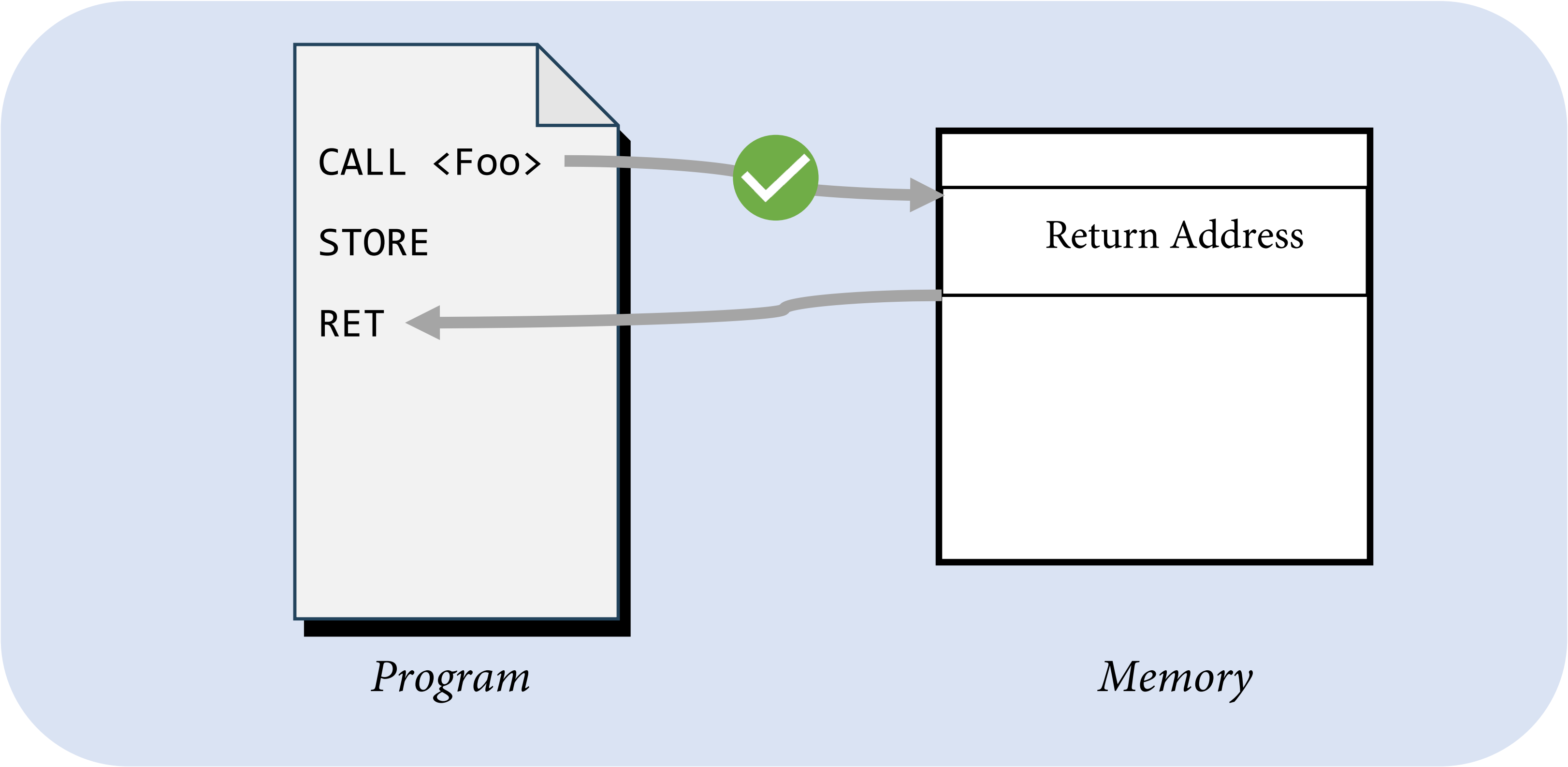
Return Address Integrity



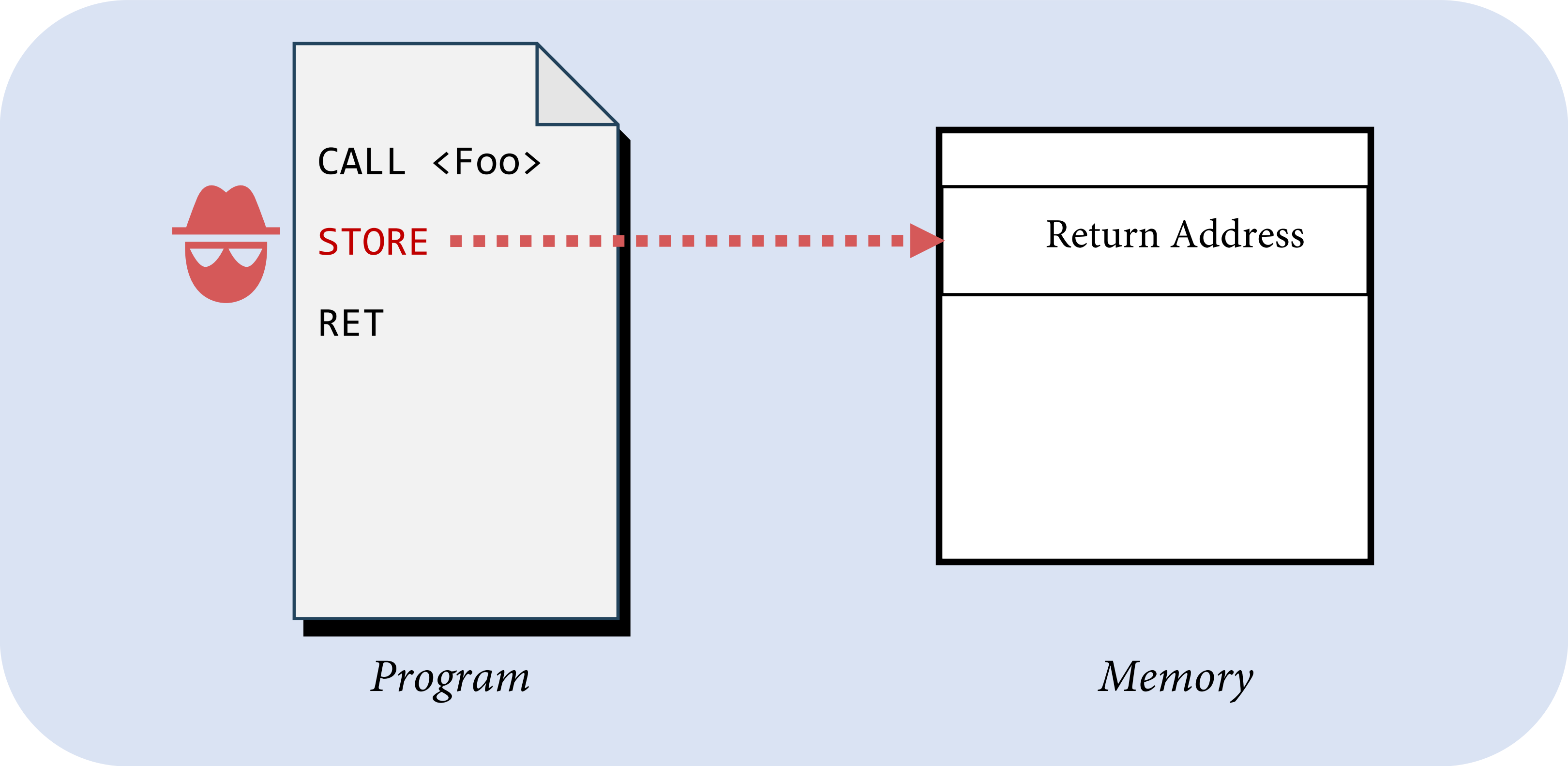
Return Address Integrity



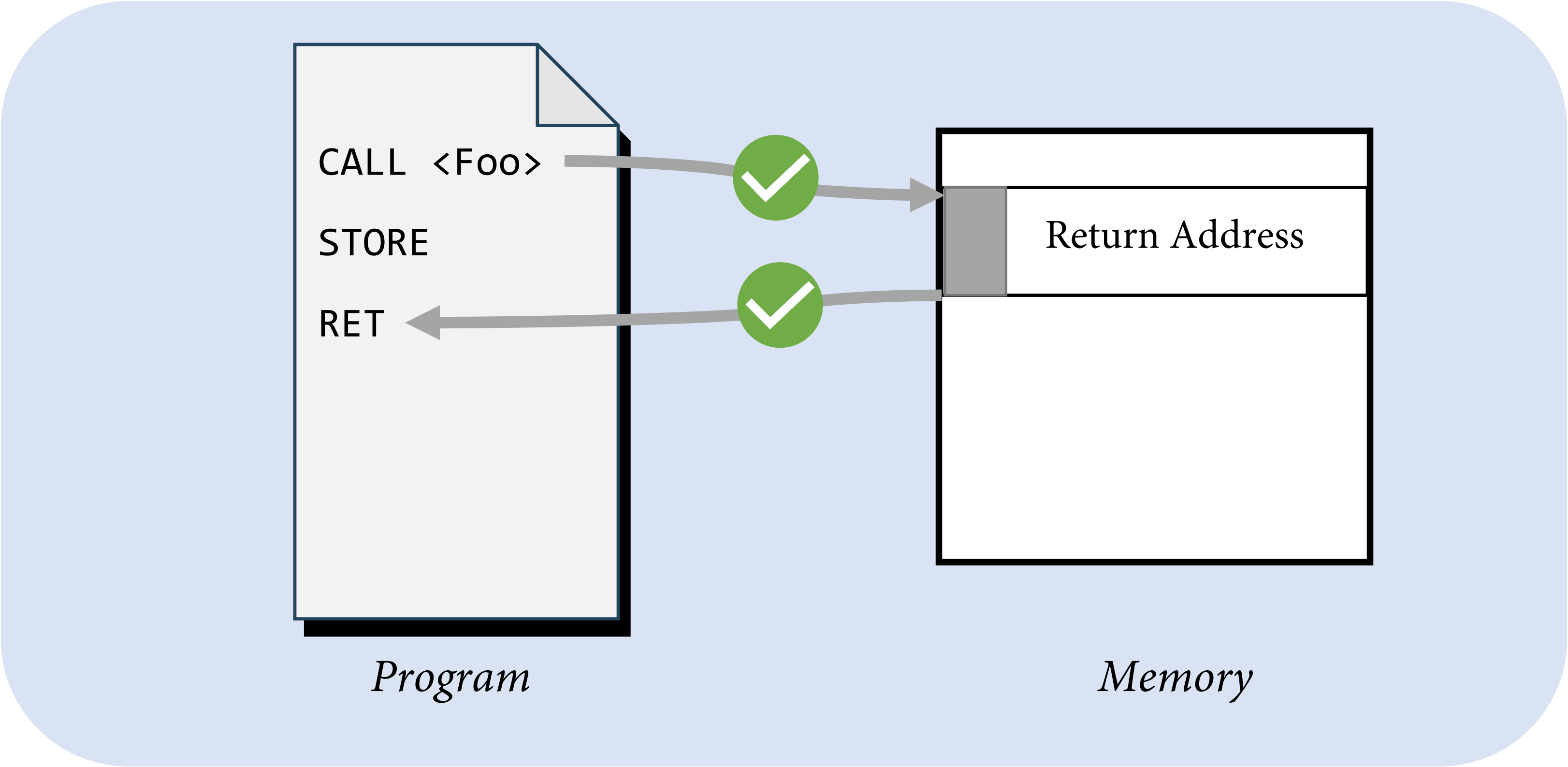
Return Address Integrity



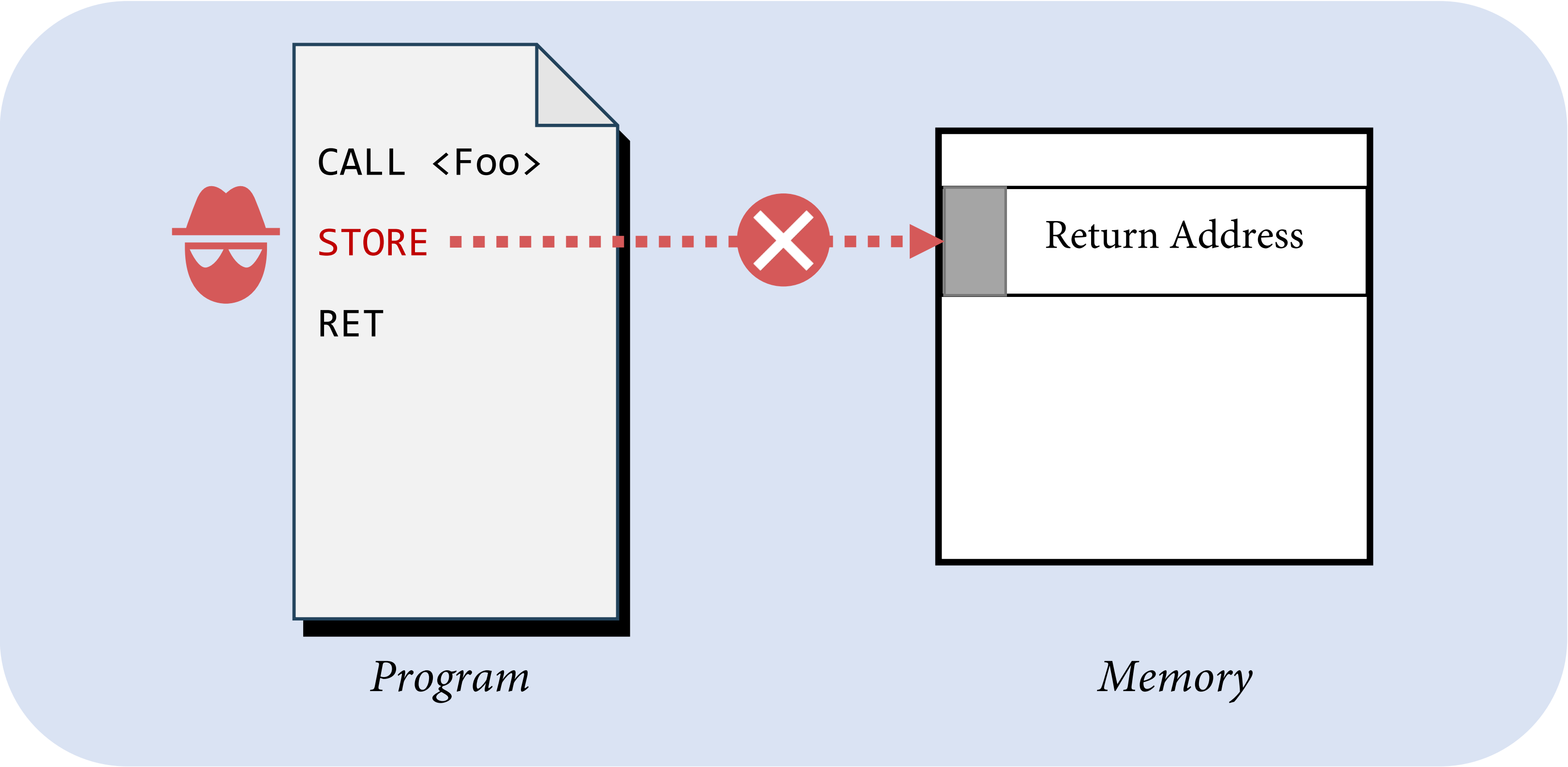
Return Address Integrity



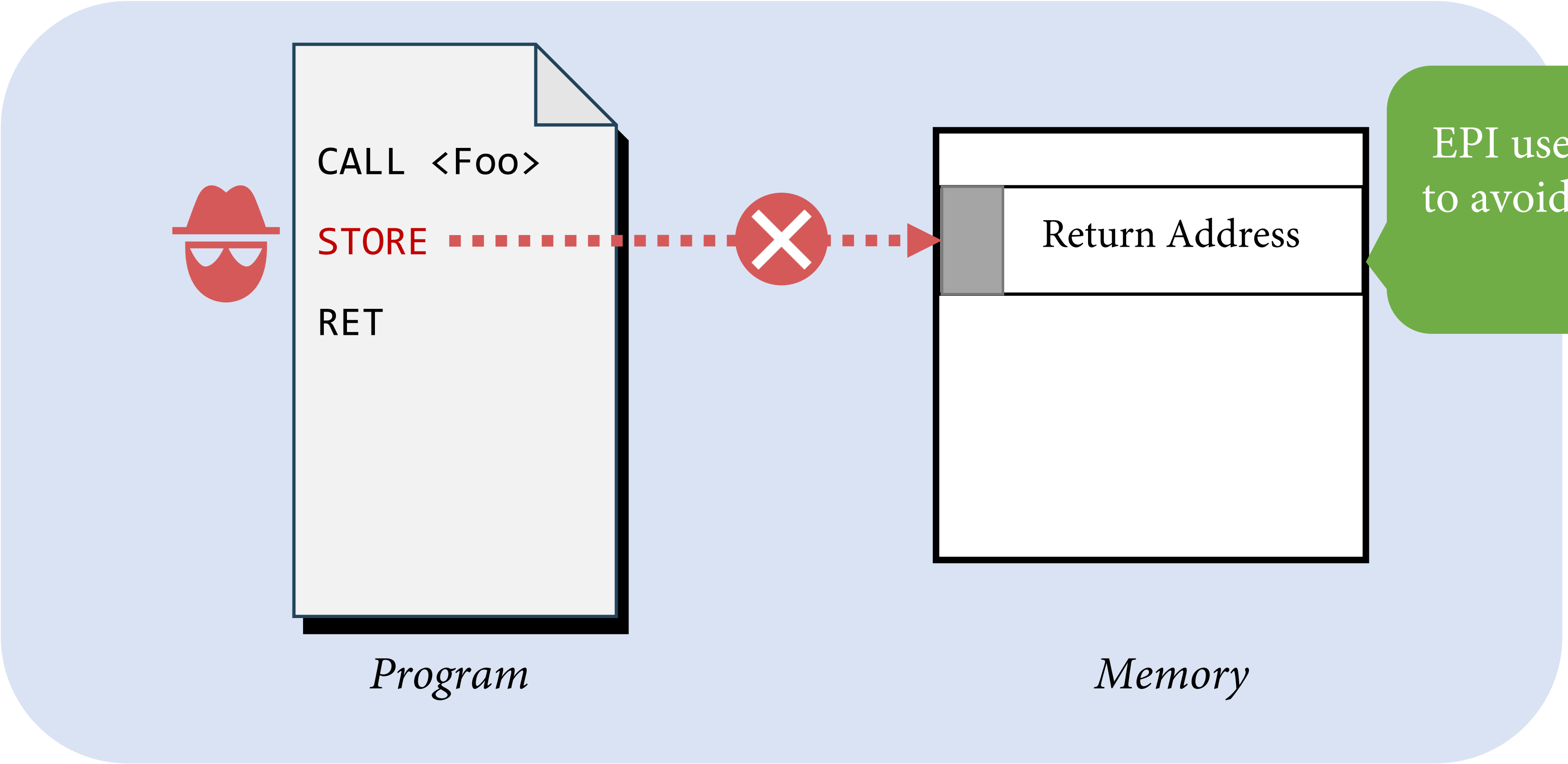
Return Address Integrity



Return Address Integrity

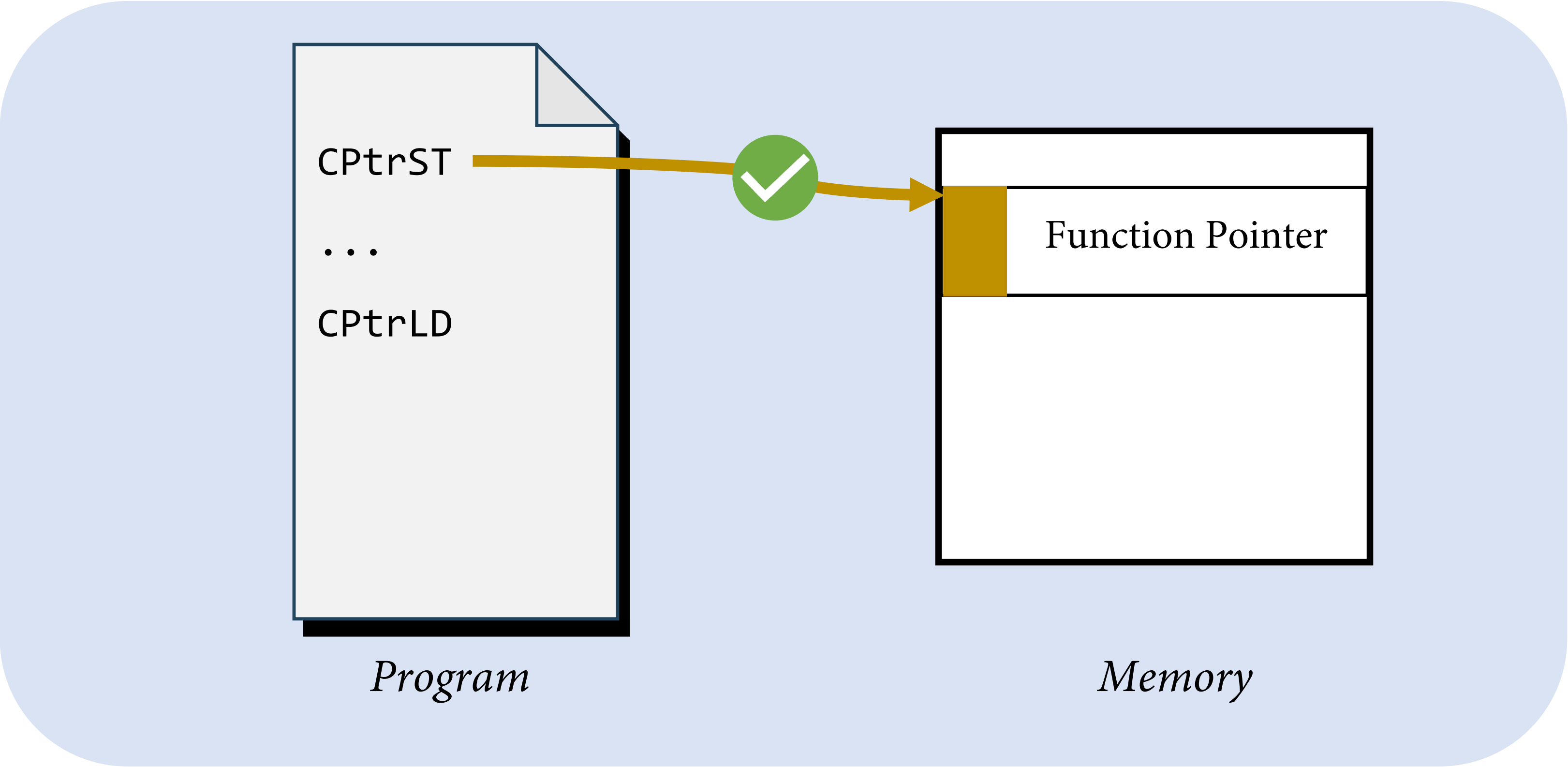


Return Address Integrity

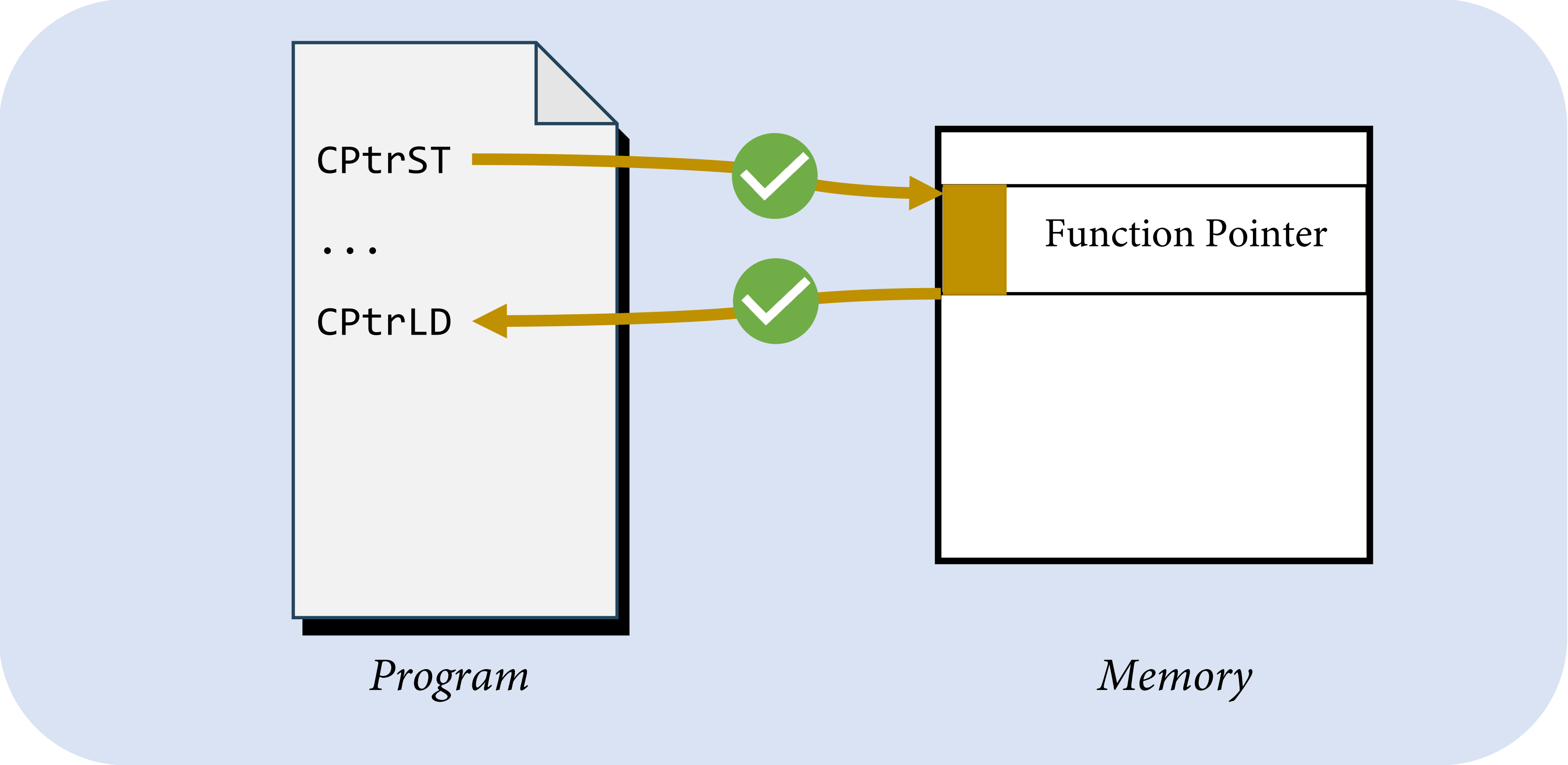


EPI uses advisory exceptions to avoid crashing when under attack.

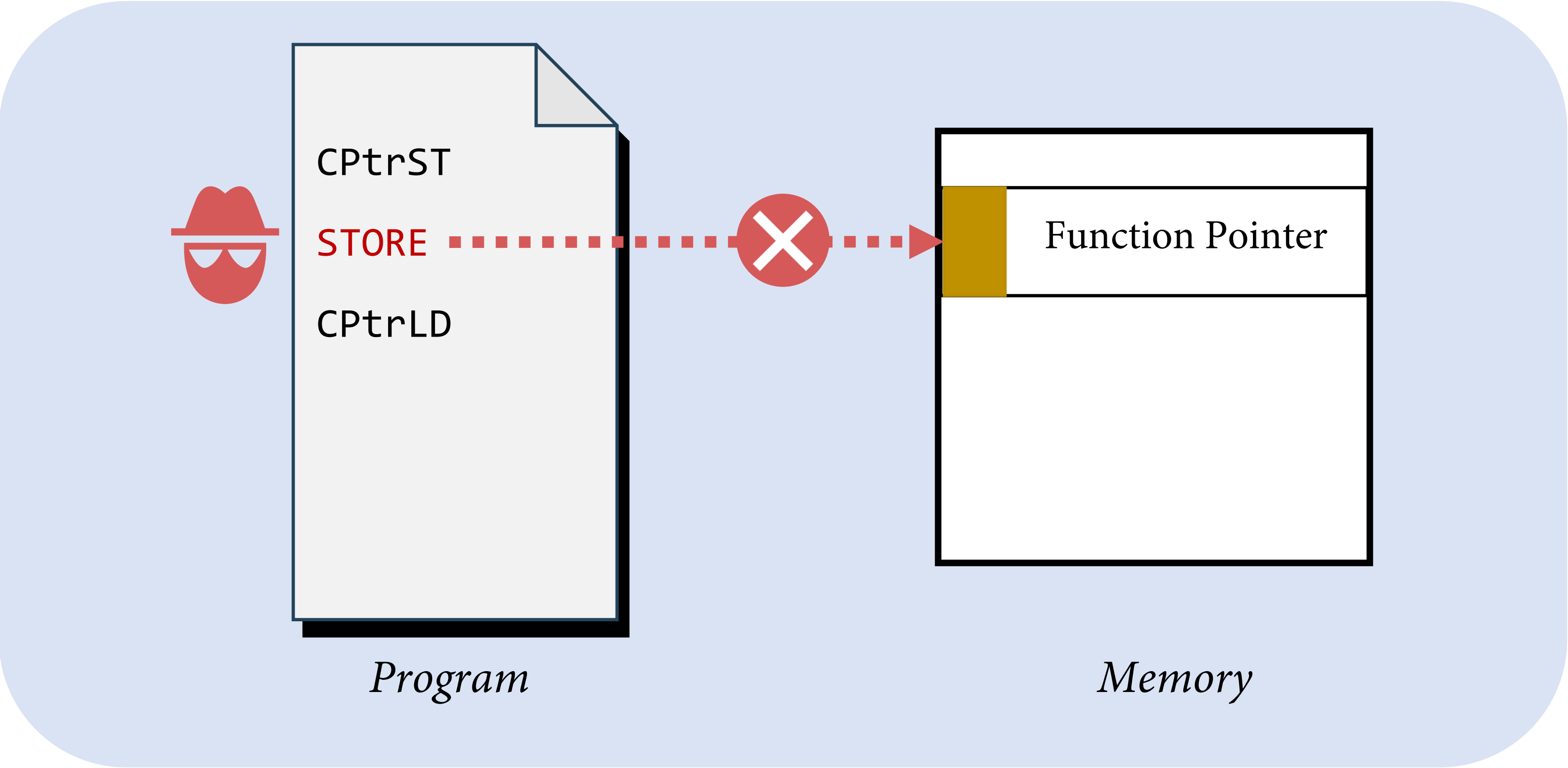
Code Pointer Integrity



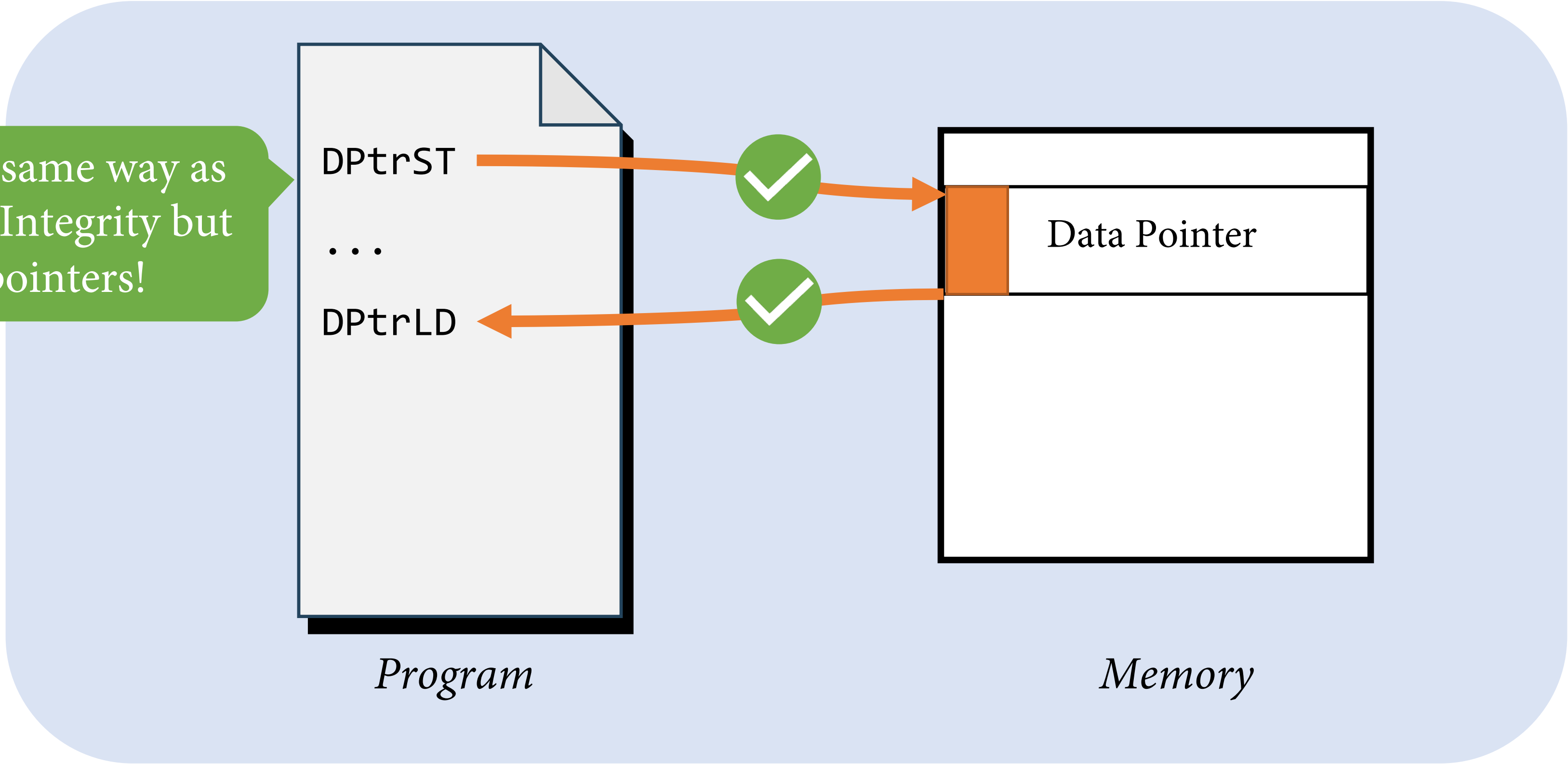
Code Pointer Integrity



Code Pointer Integrity



Data Pointer Integrity

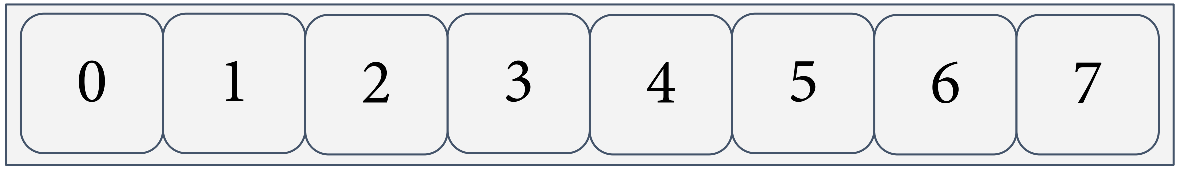


Works in the same way as Code Pointer Integrity but for data pointers!

EPI

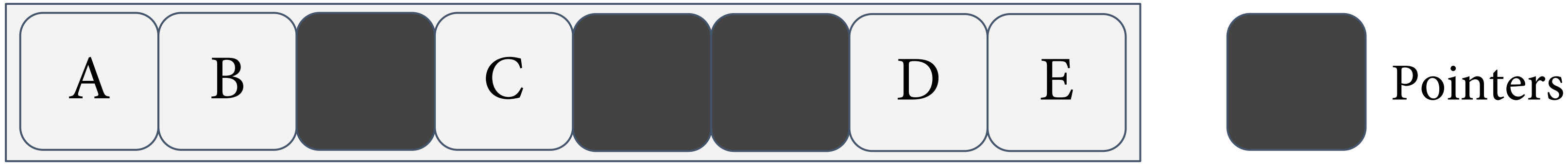
Cache Line Formats

Cache Line Formats



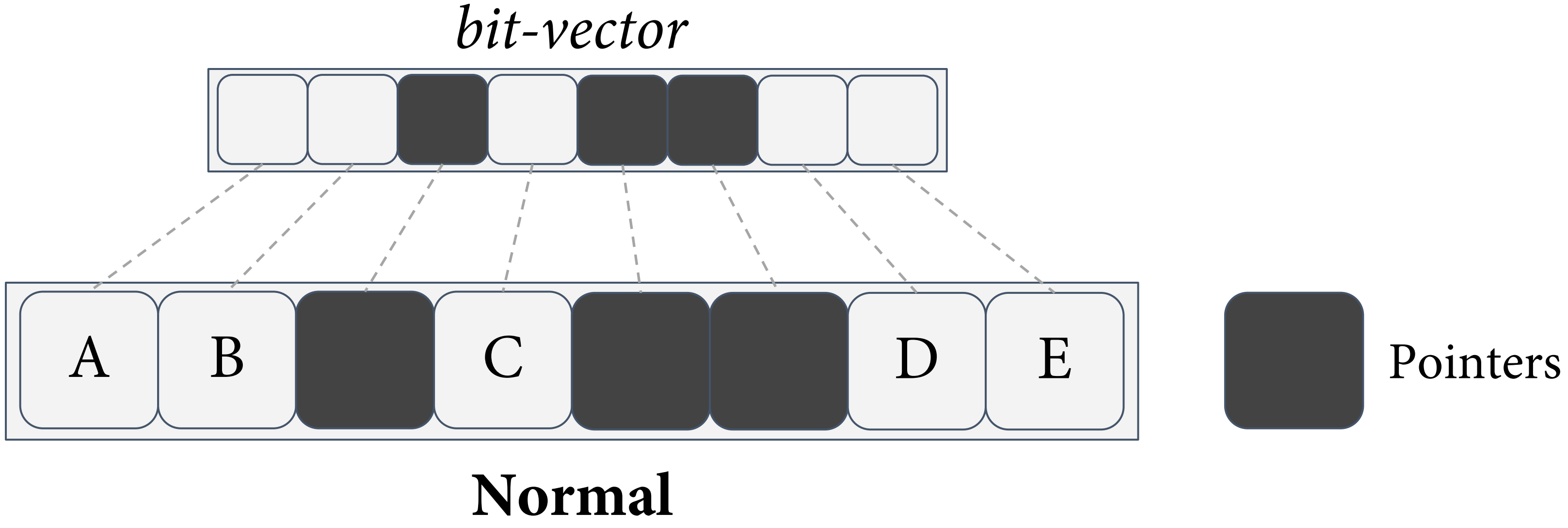
Normal

Cache Line Formats



Normal

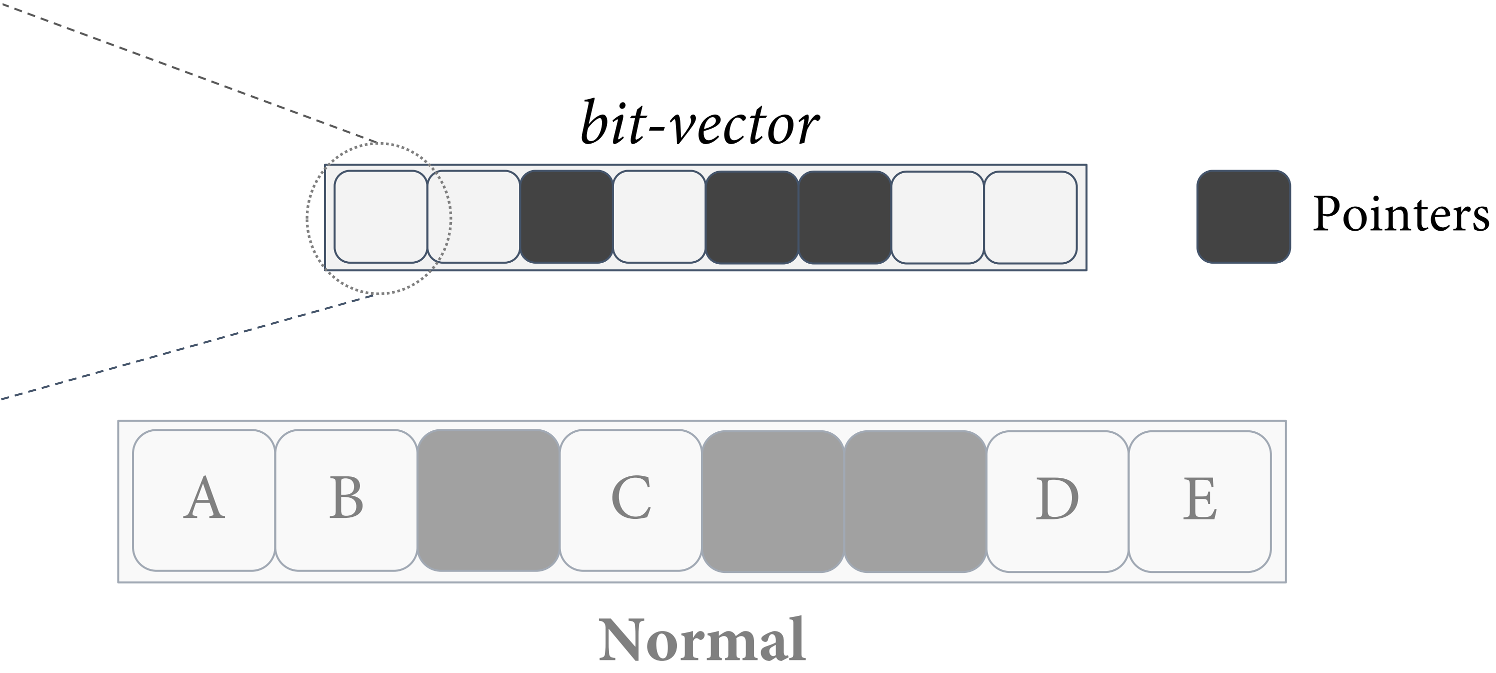
Cache Line Formats



Cache Line Formats

Format Encoding Table

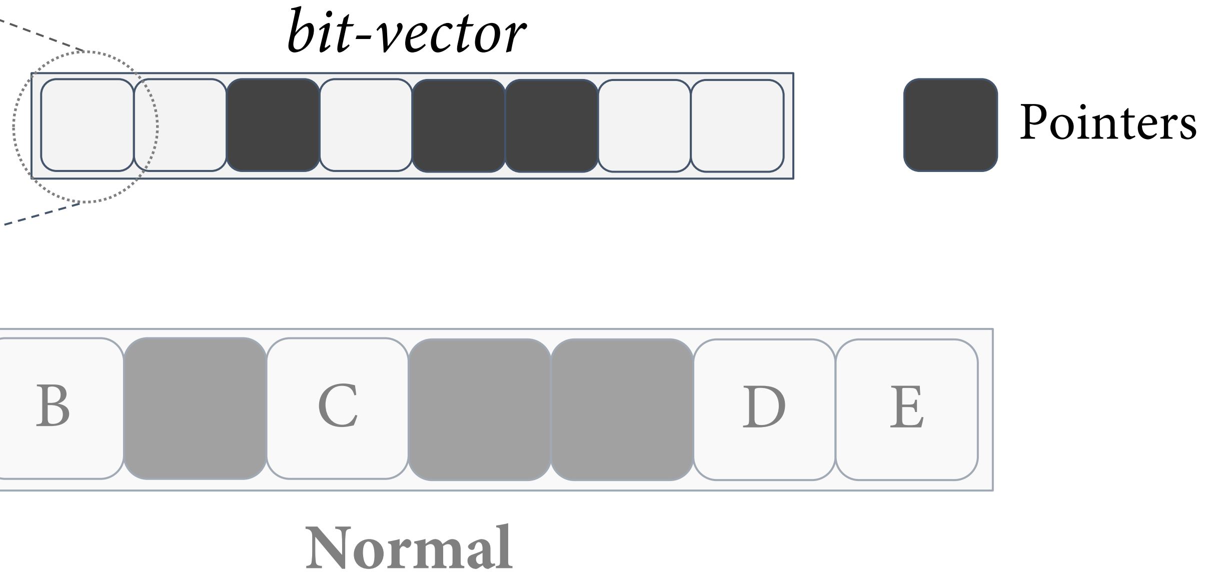
Type	Bits
Return address	01



Cache Line Formats

Format Encoding Table

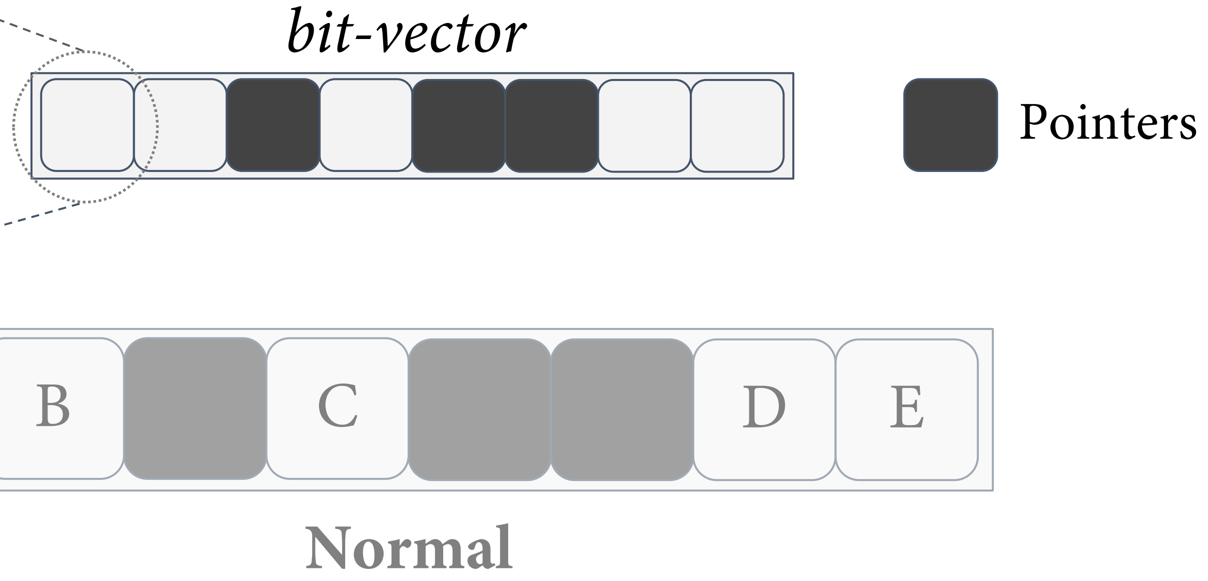
Type	Bits
Return address	01
Function pointer	10



Cache Line Formats

Format Encoding Table

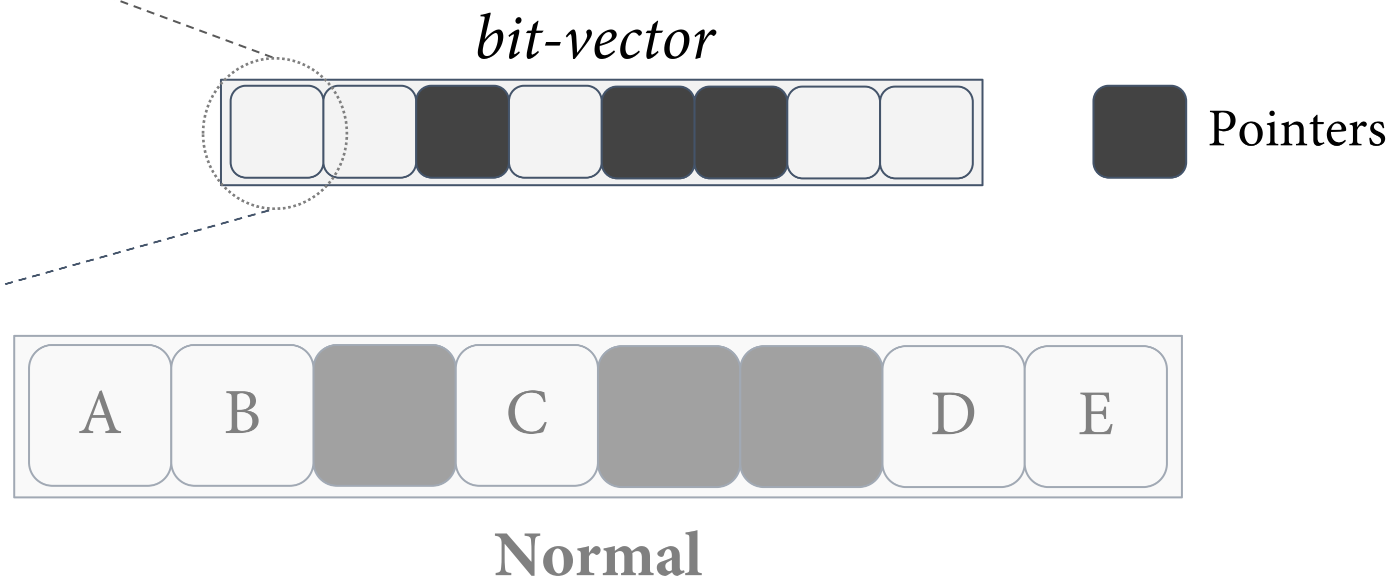
Type	Bits
Return address	01
Function pointer	10
Data pointer	11



Cache Line Formats

Format Encoding Table

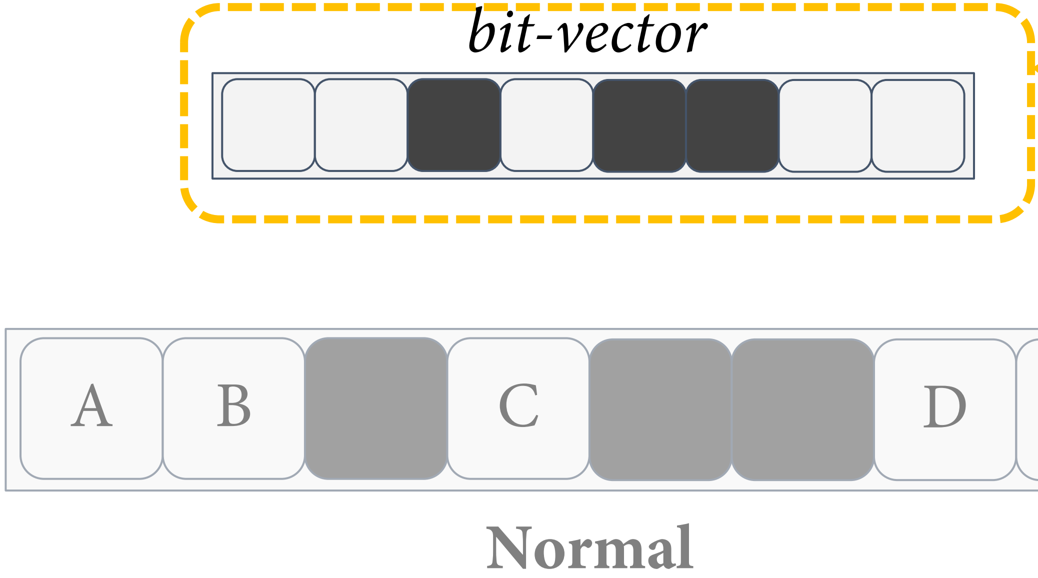
Type	Bits
Regular data	00
Return address	01
Function pointer	10
Data pointer	11



Cache Line Formats

Format Encoding Table

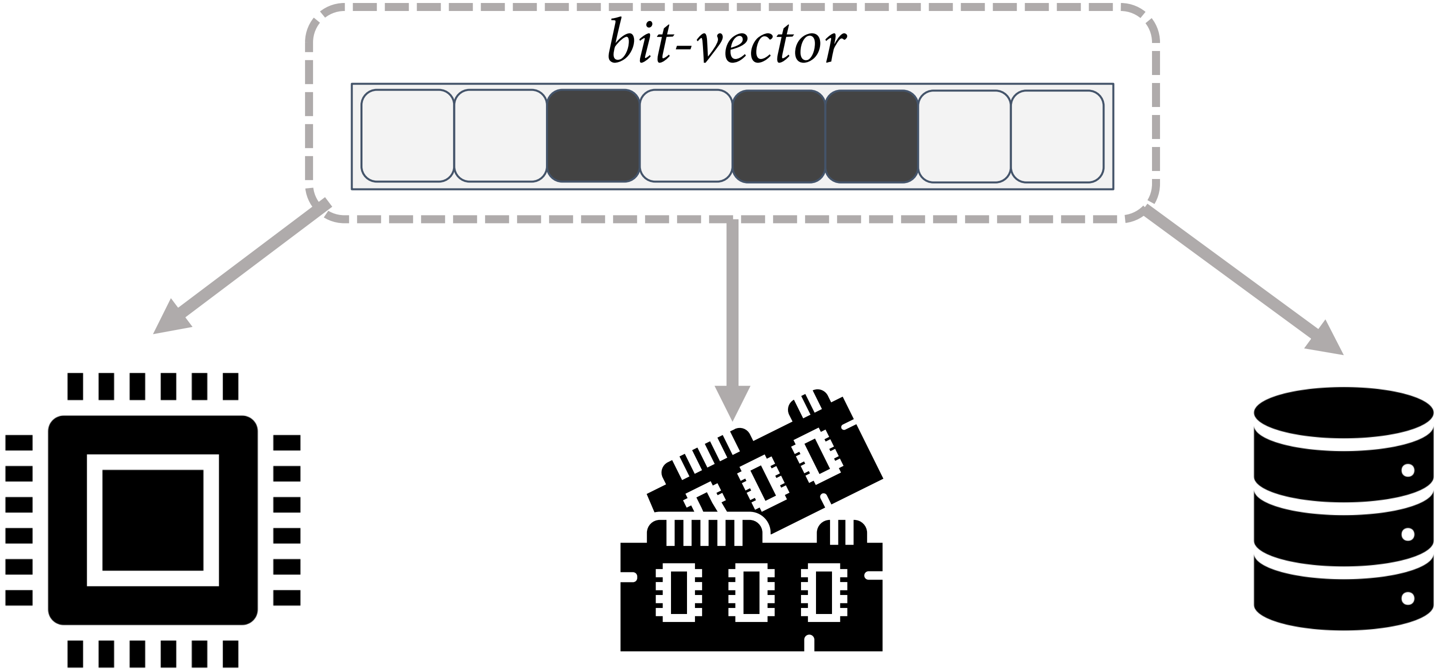
Type	Bits
Regular data	00
Return address	01
Function pointer	10
Data pointer	11



This introduces a 6.25% area overhead.

Cache Line Formats

Using a bit-vector throughout the memory hierarchy is **inefficient!**

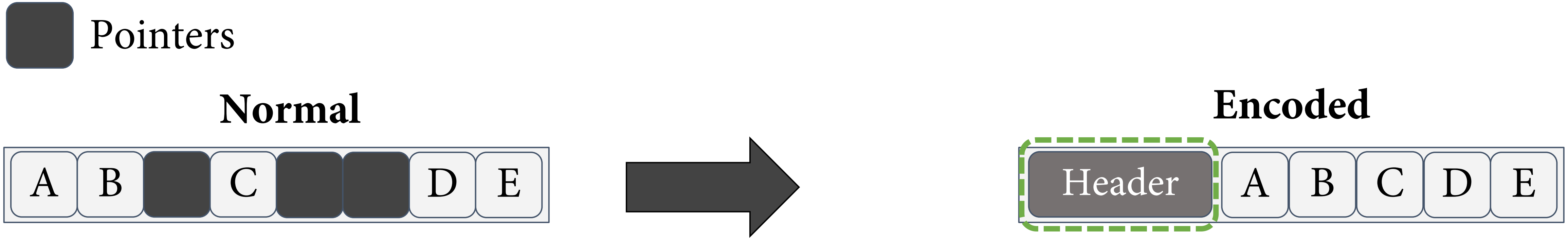


Cache Line Formats

With EPI, we encode metadata **within** unused pointer bits.

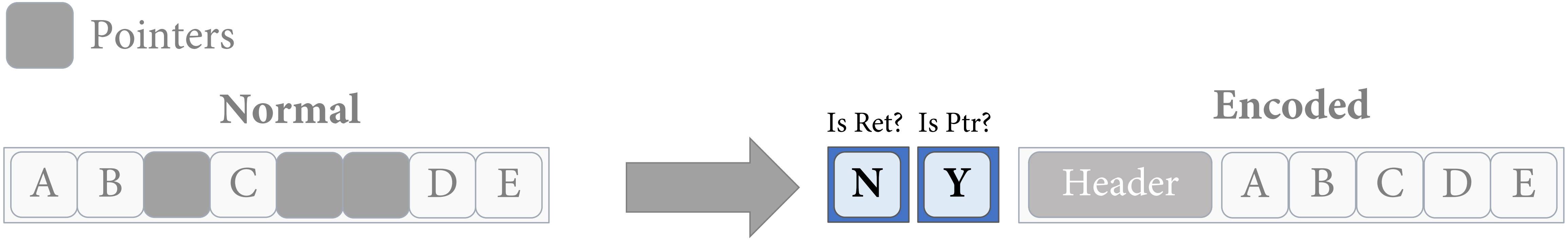
Cache Line Formats

With EPI, we encode metadata **within** unused pointer bits.



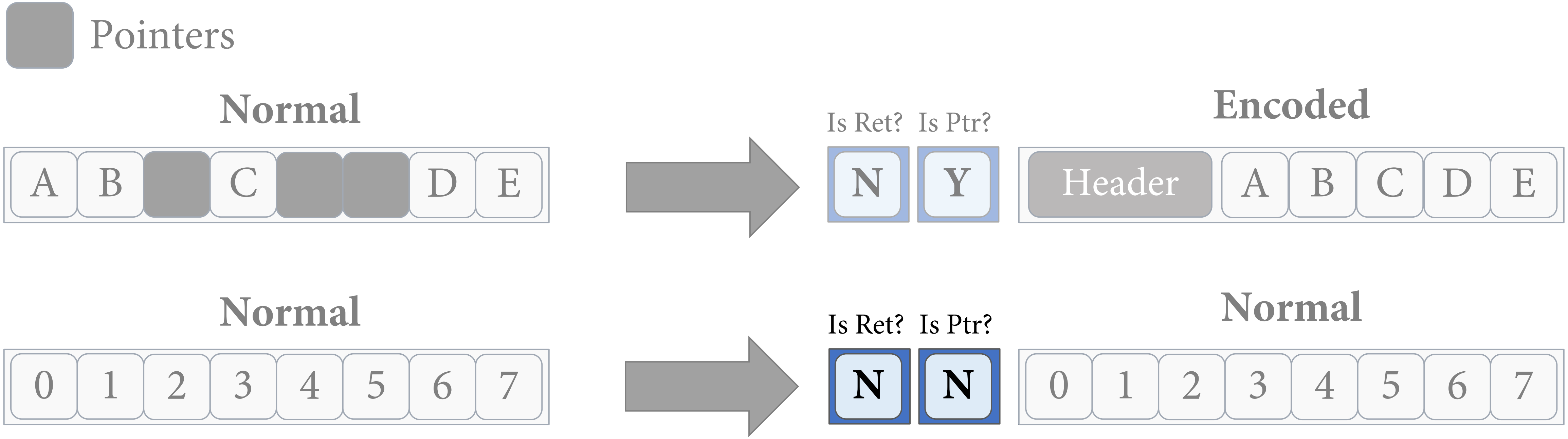
Cache Line Formats

With EPI, we encode metadata **within** unused pointer bits.



Cache Line Formats

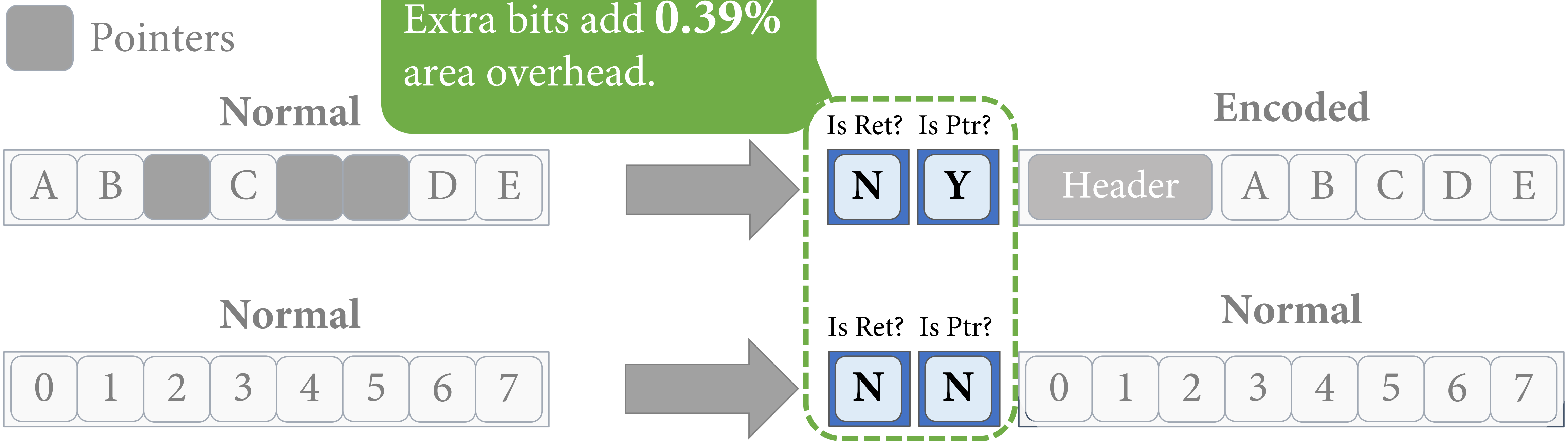
With EPI, we encode metadata **within** unused pointer bits.



Cache Line Formats

With EPI, we encode metadata **within** unused pointer bits.

Extra bits add **0.39%** area overhead.



▨ Pointers

Normal

A	B	▨	C	▨	▨	D	E

Is Ret?	Is Ptr?
N	**Y**

Encoded

Header	A	B	C	D	E

Normal

0	1	2	3	4	5	6	7

Is Ret?	Is Ptr?
N	**N**

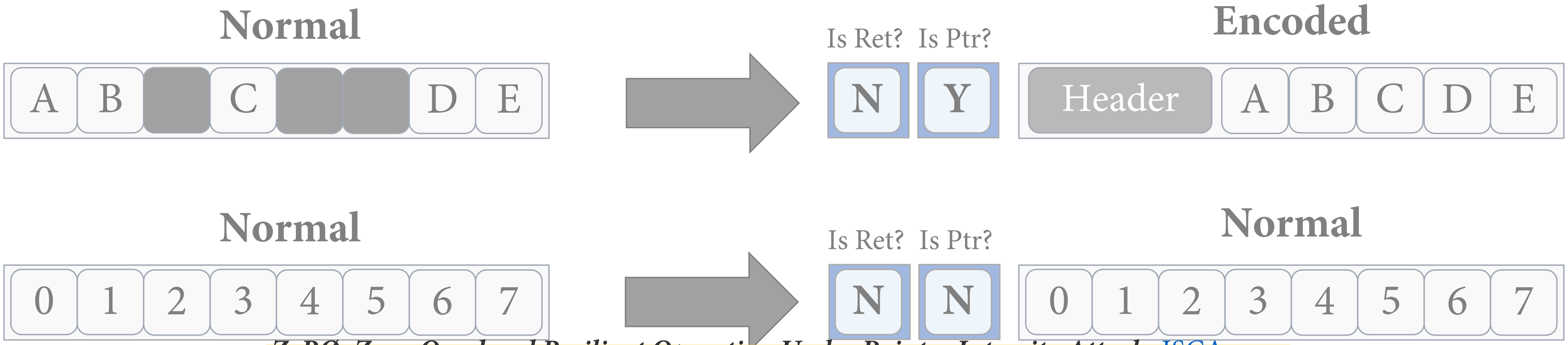
Normal

0	1	2	3	4	5	6	7

Cache Line Formats

A novel variant of ZeRØ & Califorms

▨ Pointers



Normal

A B ▨ C ▨ ▨ D E

Is Ret? Is Ptr?

N Y

Encoded

Header A B C D E

Normal

0 1 2 3 4 5 6 7

Is Ret? Is Ptr?

N N

Normal

0 1 2 3 4 5 6 7

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

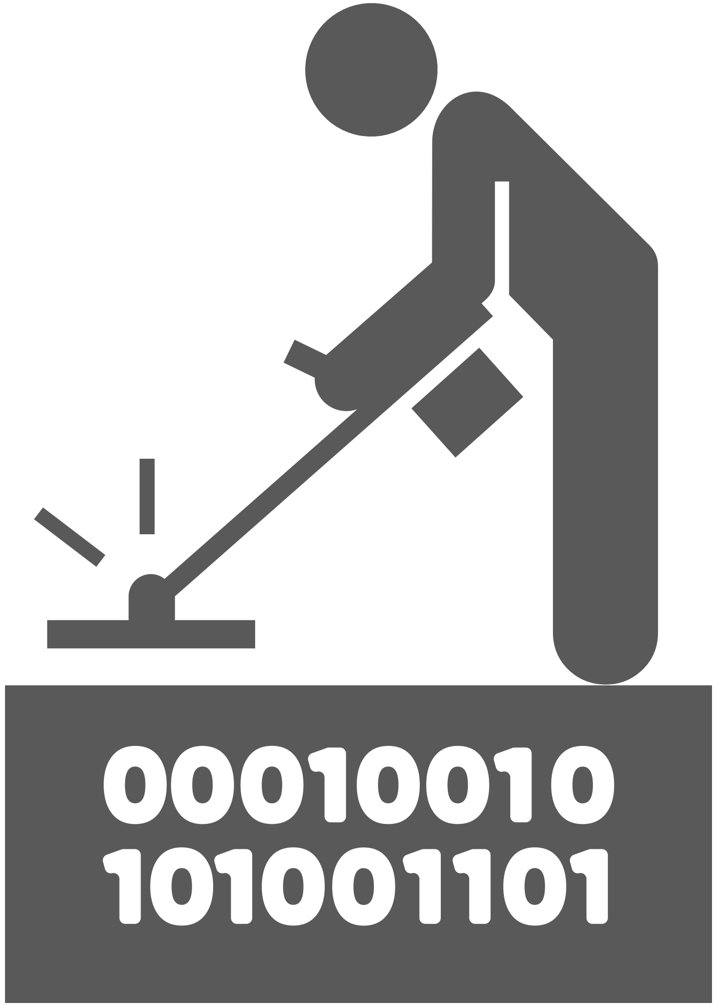
Practical Byte-Granular Memory Blacklisting using Califorms MICRO 2019

Cache Line Formats

With EPI, we encode metadata **within** unused pointer bits.

What unused pointer bits?

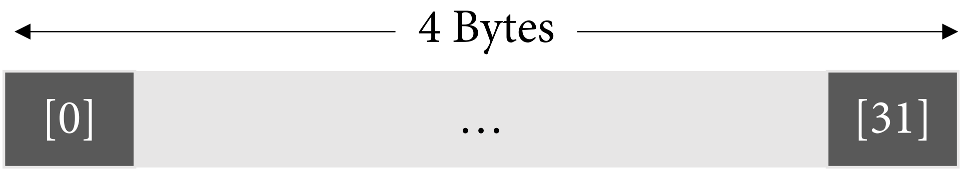
Harvesting Unused Pointer Bits



Common software properties allow us harvest extra bits from pointers on 32-bit architectures.

Harvesting Unused Pointer Bits

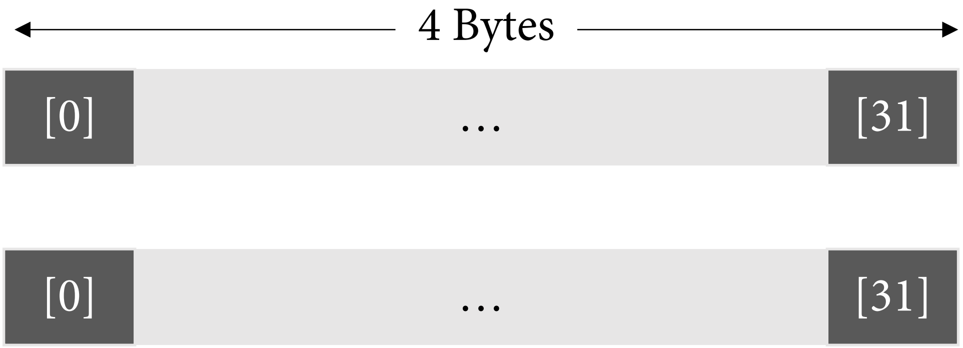
Regular Data



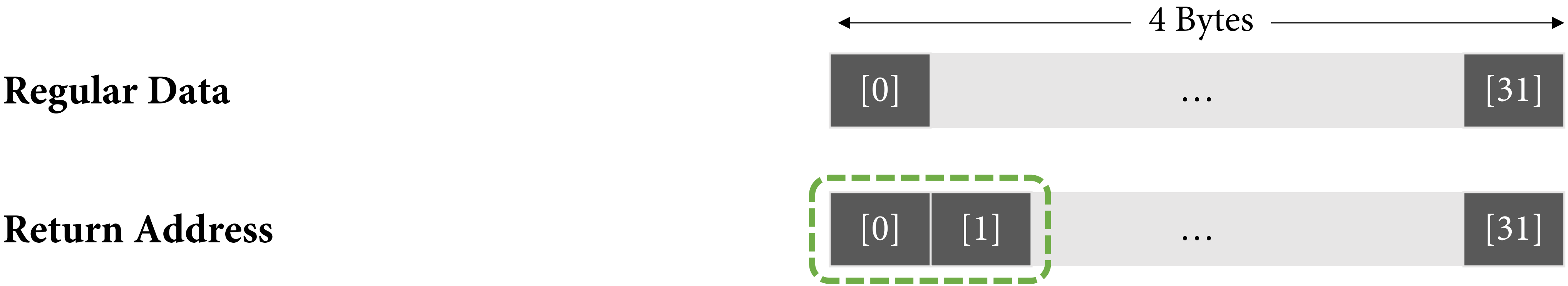
Harvesting Unused Pointer Bits

Regular Data

Return Address

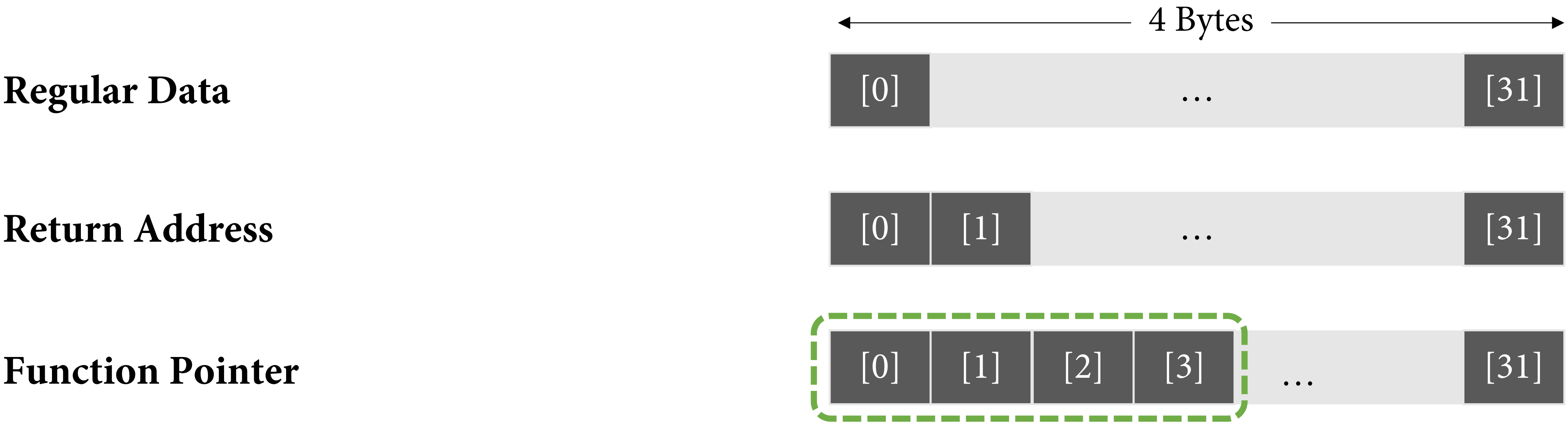


Harvesting Unused Pointer Bits



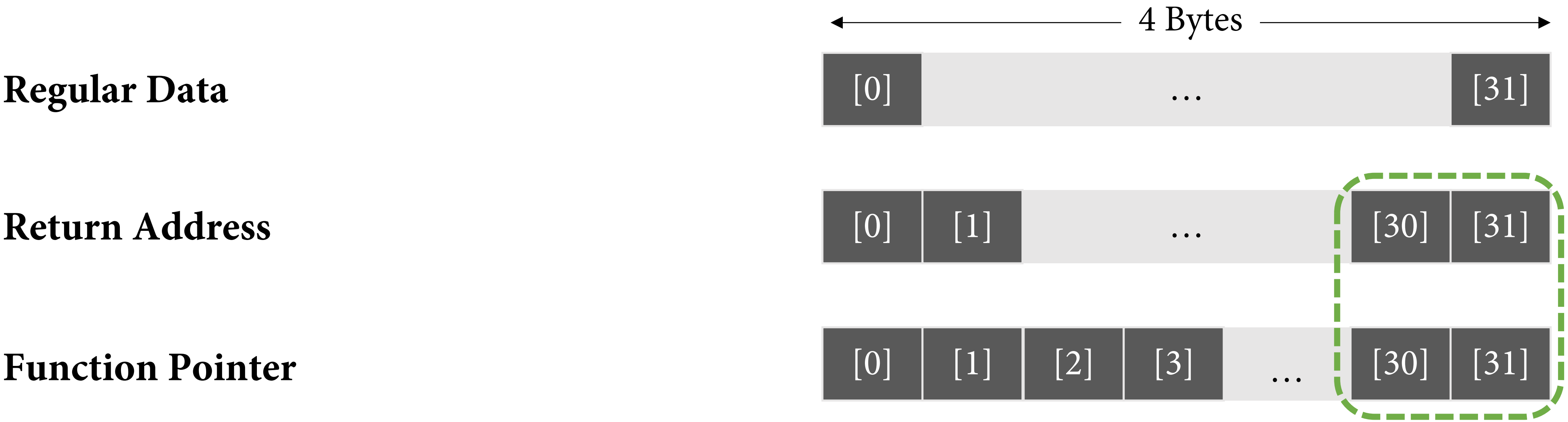
Fixed-width instructions on RISC architectures allow us to harvest the 2 LSBs.

Harvesting Unused Pointer Bits



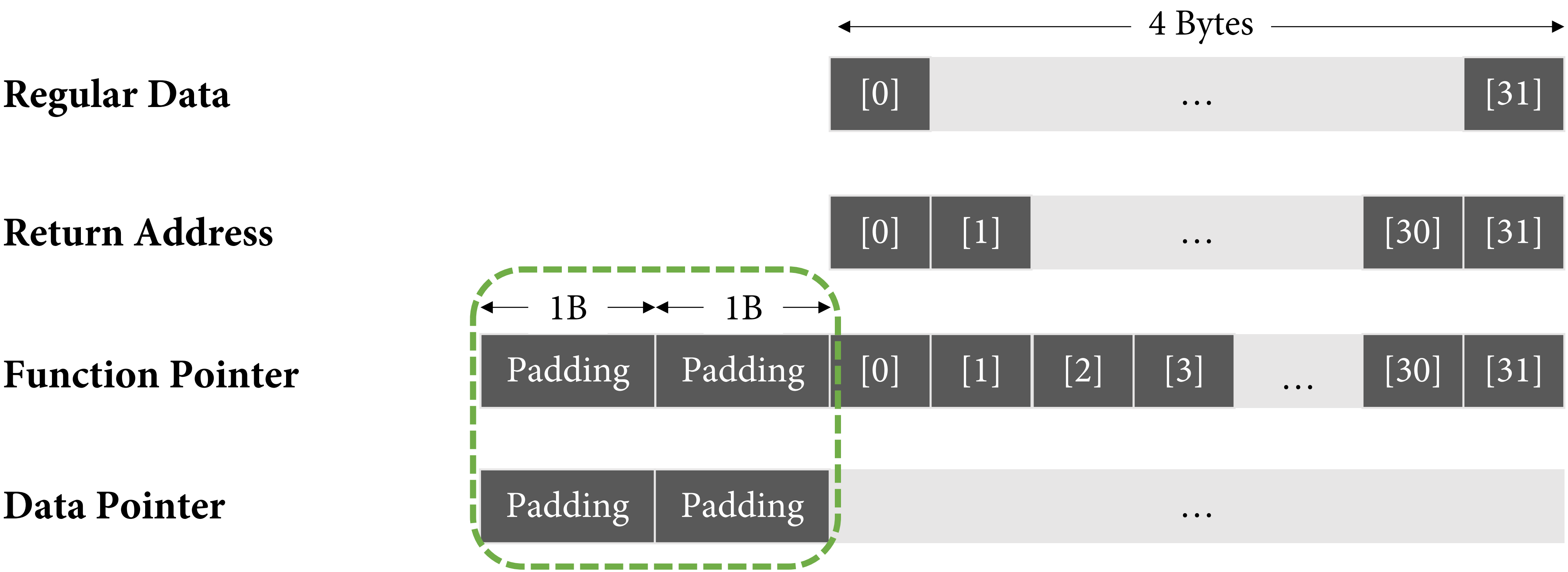
Aligning functions (e.g. `-falign-functions`) allows to harvest the 4 LSBs.

Harvesting Unused Pointer Bits



Compacting the code address space allows us to harvest 2 MSBs.

Harvesting Unused Pointer Bits



4 Bytes

Regular Data: [0] … [31]

Return Address: [0] [1] … [30] [31]

Function Pointer: 1B 1B Padding Padding [0] [1] [2] [3] … [30] [31]

Data Pointer: Padding Padding …

Inserting padding bytes allows us to store a per-pointer ID.

EPI

Performance

EPI Performance Overheads

Hardware Modifications

EPI Performance Overheads

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

EPI Performance Overheads

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

00010010
101001101
00010010
111001001
00010010

Software Modifications

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

EPI Performance Overheads

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

Software Modifications

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

EPI Performance Overheads

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

Software Modifications

- Our special load/stores do not change the binary size.
- The `ClearMeta` instructions are only called on memory deallocation.
- Padding bytes are added to pointers only.

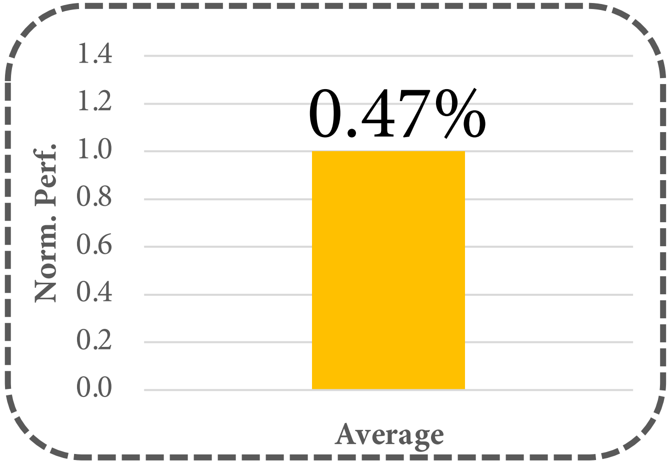
Performance Results

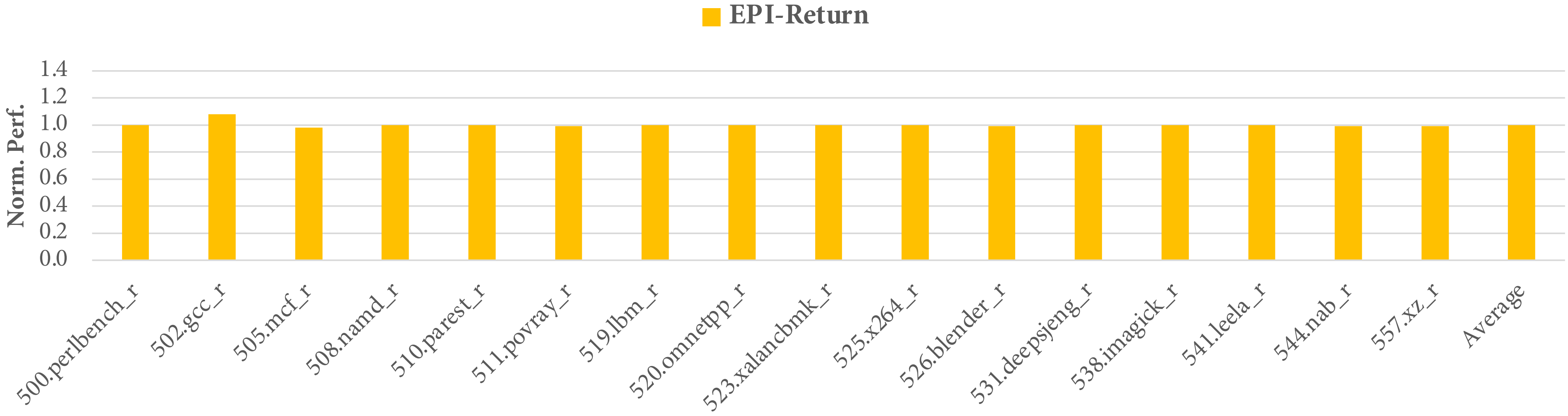
Experimental Setup

We use emulate EPI on x86_64 by modifying LLVM to emit new instructions.

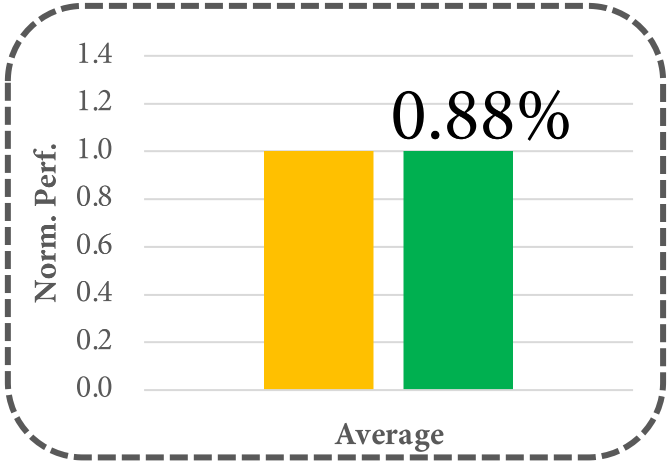
- `ClearMeta` is emulated using dummy stores.
- Padding bytes & necessary LD/ST emulate extra memory utilization.

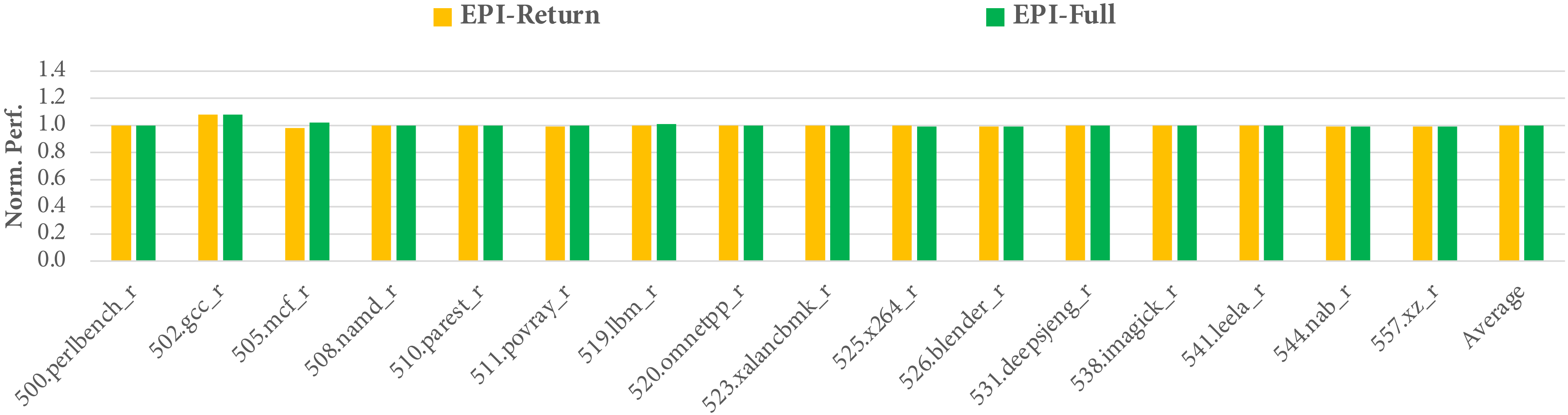
Performance Results



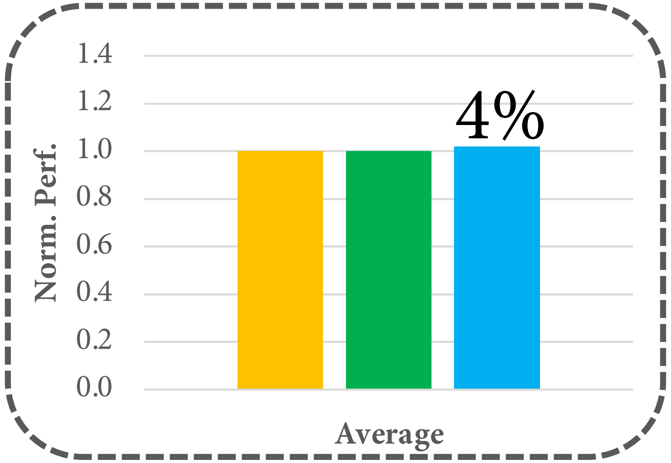


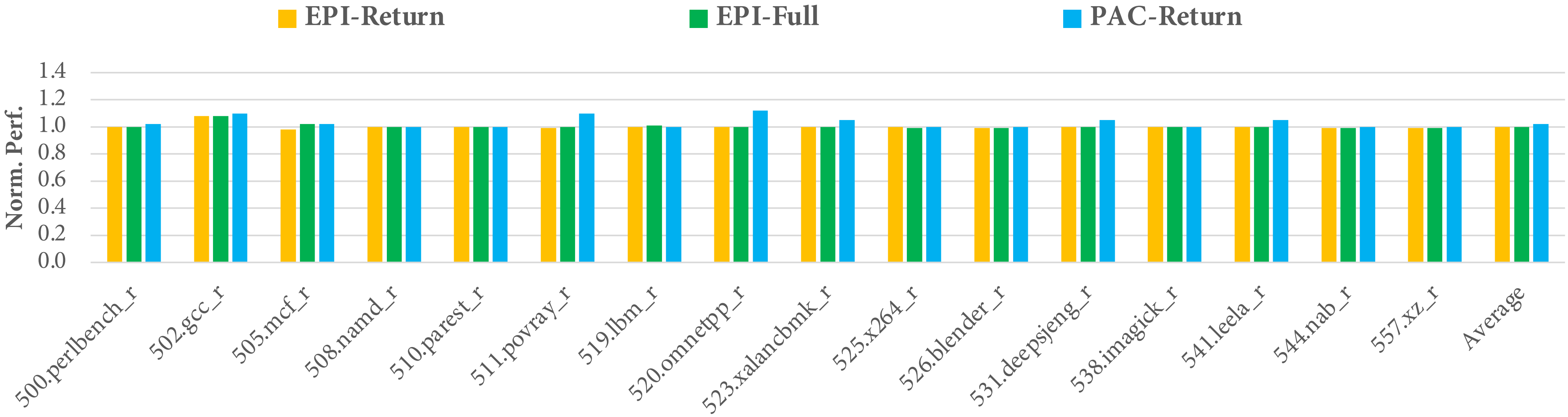
Performance Results



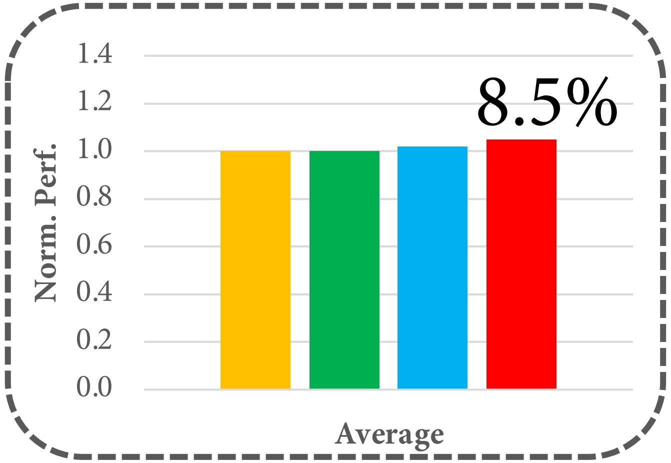


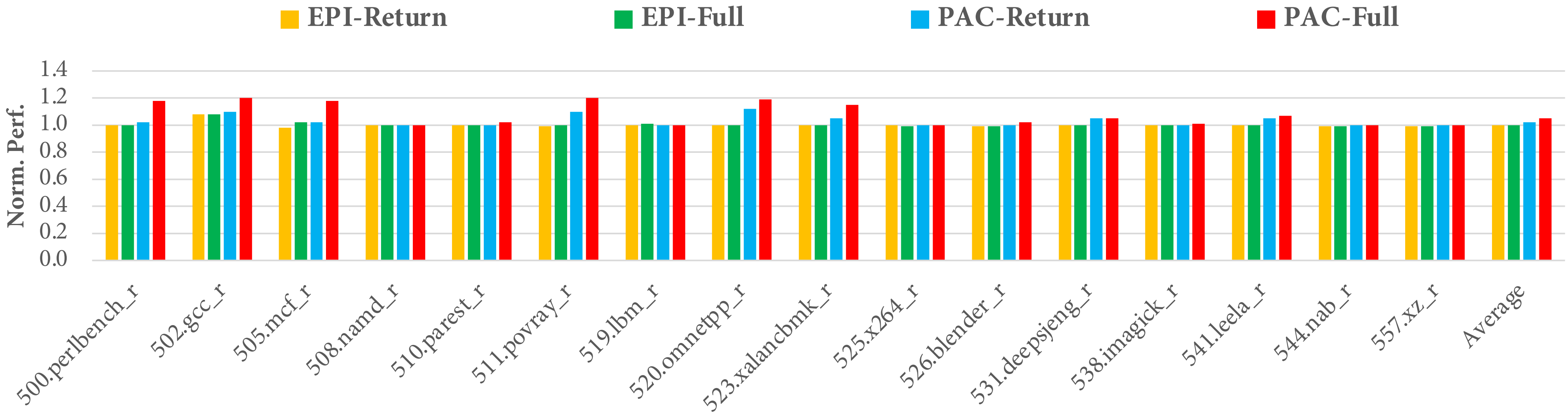
Performance Results



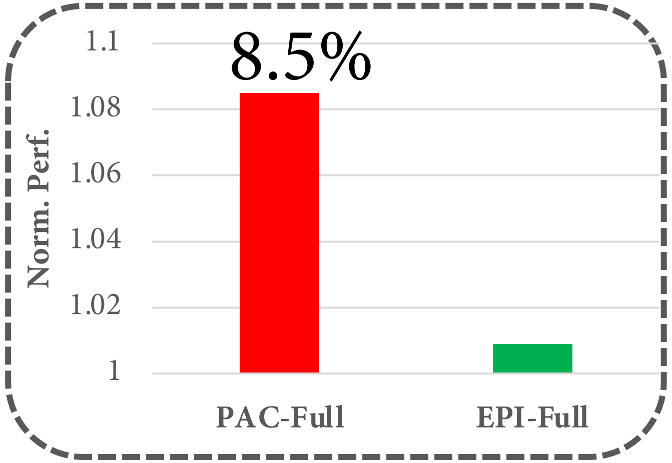


Performance Results





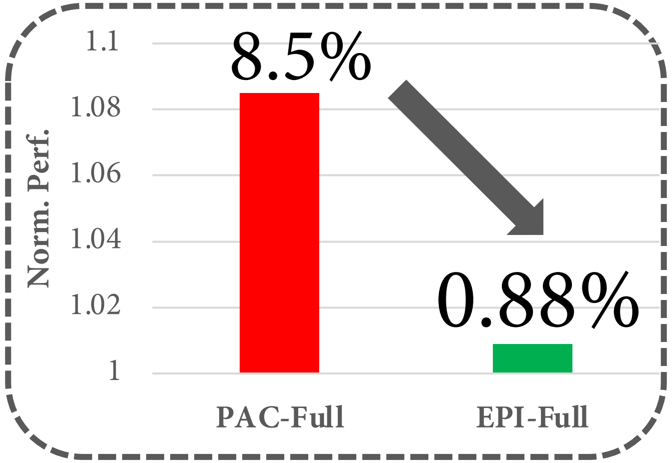
Performance Results



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

- loads/stores
- usages

Performance Results



EPI reduces the average runtime overheads of pointer integrity from 8.5% to 0.88%!

EPI does not compromise on security

No Pointer Manipulation
Protects against all known pointer manipulation attacks (e.g. ROP, JOP/COP, COOP, DOP).

Handling Security Violations

Advisory Exceptions

- Skip faulty instructions.
- Do NOT crash the running process.

Handling Security Violations

Advisory Exceptions

- Skip faulty instructions.
- Do NOT crash the running process.

Permit List

- Initialized during program startup

Handling Security Violations

Advisory Exceptions

- Skip faulty instructions.
- Do NOT crash the running process.

Permit List

- Initialized during program startup
- Avoid false alarms for non-type aware functions (e.g., `memcpy` and `memmove`)

Handling Third Party Code

We can pick from the following options:

Handling Third Party Code

We can pick from the following options:

1

Compile with EPI
Compile third party code with EPI support.

Handling Third Party Code

We can pick from the following options:

1. **Compile with EPI**
   Compile third party code with EPI support.

2. **Add to Permit List**
   Add to a permit list during program initialization.

Handling Third Party Code

We can pick from the following options:

1 **Compile with EPI**
Compile third party code with EPI support.

2 **Add to Permit List**
Add to a permit list during program initialization.

3 **Invoke `ClearMeta`**
`ClearMeta` is inserted before passing pointers to external libraries.

Limitations

Non-pointer Data Corruption
These attacks require a full memory safety solution.

An efficient pointer integrity mechanism

Specifically tailored for 32-bit embedded systems.

✓ **Offers Robust Security**
✓ **Easy to Implement**
✓ **Minimal Runtime Overheads**
✓ **Low Power**
✓ **Increased Reliability**