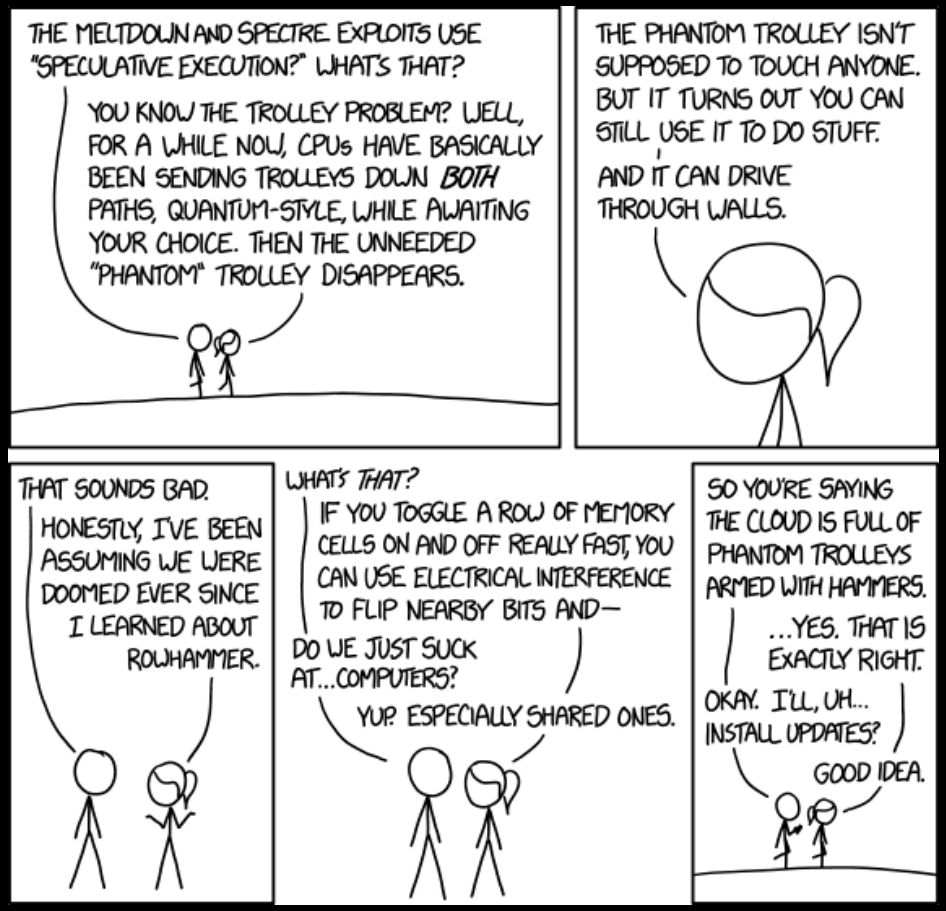
Hardware Security

ECEn 522R

Modern CPU Extensions for Security (Part 1): ARM PAC, BTI, and MTE



Stacey D. Son 20-Nov-2025

ARM Pointer Authentication Code (PAC)

A Brief History of Pointer Authentication

- 2003 PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities
  - XOR'ing pointers with a random value, very weak but interesting idea
- 2015 CCFI: Cryptographically Enforced Control Flow Integrity
  - Essentially a software implementation of PAC, but with high overhead
- ~2016 The idea of a hardware version of CCFI is considered. The core instructions are emulated with bad instruction traps into the kernel.
- 2017 Pointer Authentication Code instructions are added to the ARMv8 spec.
- 2018 Apple is first to deploy PAC in the A12 Bionic chip.
- 2019 Project Zero examines possible PAC weaknesses
- 2022 PACMAN: Attacking ARM Pointer Authentication with Speculative Execution

ARM Pointer Authentication Code (PAC)

- Pointer signature is stored in the “PAC” area of the pointer.
- Strength: (#Trials = log(1 - p)/log(1-2^{-b}): p = 0.5 for average, b = PAC Sig bits)

PAC Size (bits)	Average Number of Guesses Required to Forge a Signature
20	726,817
19	363,408
18	181,704
17	90,852
16	45,426
15	22,713
14	11,356
13	5,678
12	2,839
11	1,419
10	709

ARM Pointer Authentication Code (PAC)

iPhone 17 (A18)

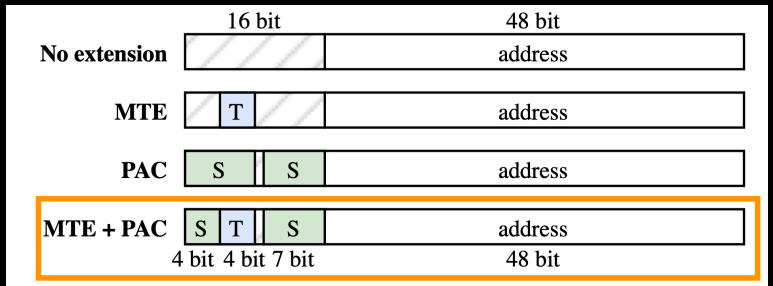


Figure 3. Pointer layout on aarch64 in Linux with and without MTE and PAC enabled.

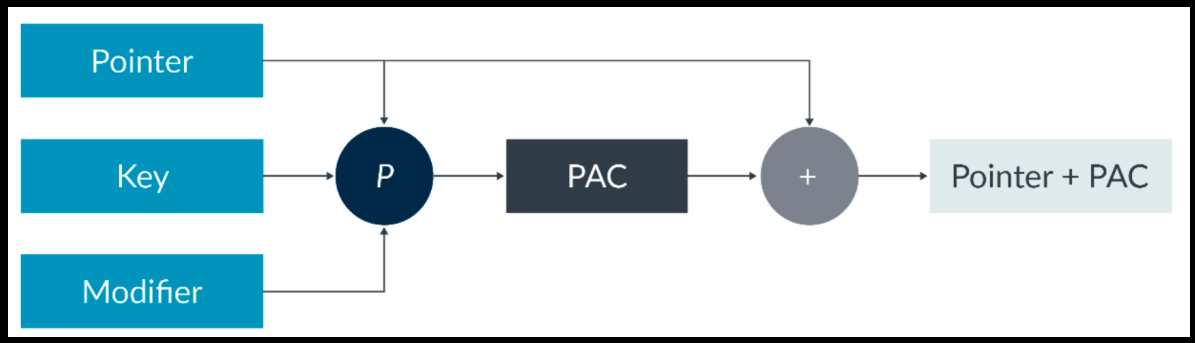
PAC Size (bits)	Average Number of Guesses Required to Forge a Signature
20	726,817
19	363,408
18	181,704
17	90,852
16	45,426
15	22,713
14	11,356
13	5,678
12	2,839
11	1,419
10	709

ARM PAC Keys

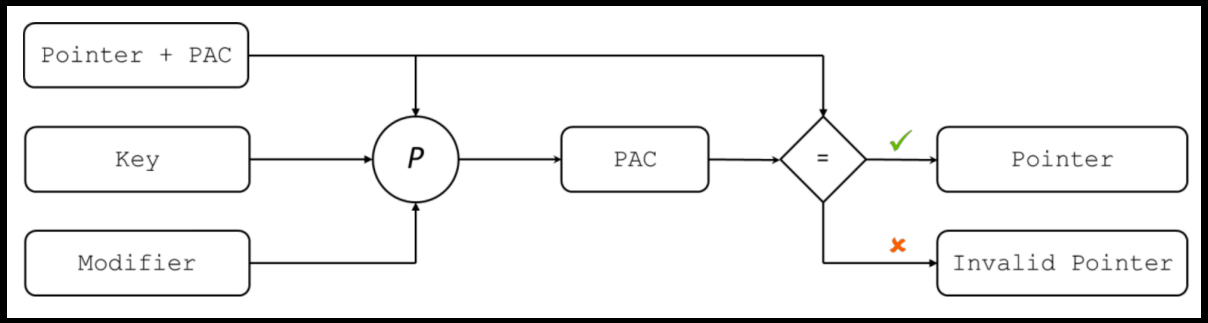
- Four 128-bit keys. Each key is stored in a pair of 64-bit System registers:
  - Two keys, A and B, for instruction pointers
  - Two keys, A and B, for data pointers
  - One General Purpose “modifier”.
- A-Family or global keys that are shared between all processes
- B-Family keys that are unique per-process
- Pointers signed with global keys have been shown to be forgeable in practice, See:
  - Google Project Zero’s Splitting Atoms in XNU
  - DEFCON 27: HackPac: Hacking Pointer Authentication in iOS User Space

ARM Pointer Authentication (PAC) Operations

- Signing (P = hashing algorithm):



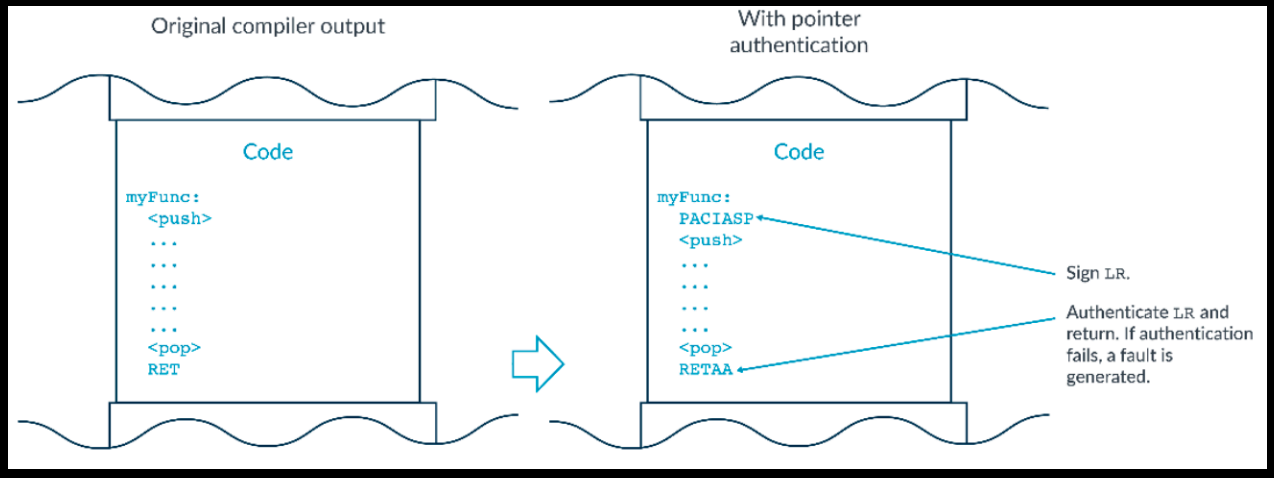
- Signature Validation (P = hashing algorithm):



ARM Pointer Authentication (PAC) Instructions

`PACIxSP`	Sign `LR`, using `SP` as the modifier.
`PACIxZ`	Sign `LR`, using 0 as the modifier.
`PACIx`	Sign `Xn`, using a general-purpose register as modifier.
`AUTIxSP`	Authenticate `LR`, using `SP` as the modifier.
`AUTIxZ`	Authenticate `LR`, using 0 as the modifier.
`AUTIx`	Authenticate `Xn`, using a general-purpose register as modifier.
`BRAx`	Indirect branch with pointer authentication.
`BLRAx`	Indirect branch with link, with pointer authentication.
`RETAx`	Function return with pointer authentication.
`ERETAx`	Exception return with pointer authentication.

In each case, replace x with A or B to select the wanted key.

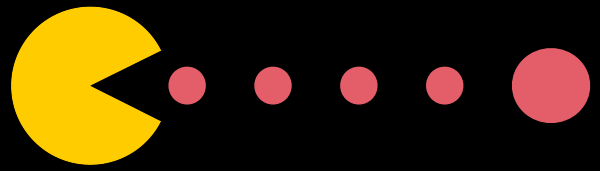


ARM PAC Keys and Modifiers Use on {Mac, i, Watch} OS

Type of Pointer	Key	Modifier
C Function Pointer	A (Global)	“0” (No Modifier)
GOT Entry	A (Global)	Address of Ptr
Objective-C Method Table Entry	A (Global)	Address of Ptr
Block invocation function	A (Global)	Address of Ptr
Block copy/destroy helper function	A (Global)	Address of Ptr
C++ V-table entry (Function Ptr)	A (Global)	(Type hash << 48) |
C++ V-table (data) pointer	A (Global)	“0” (No Modifier)
Function return	B	Address of Ptr
Userspace address passed to kernel	A (Global)	“0” (No Modifier)
ELR_EL1 for return to EL0	B	Address of Ptr
ELR_EL1 for return to EL1	B	Address of Ptr

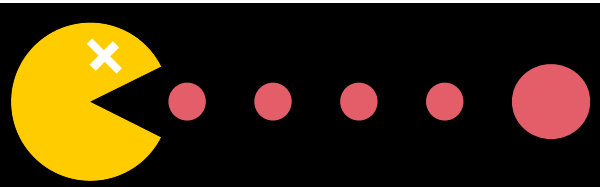
* C Function Pointers in Shared Libraries.

ARM PAC Attacks

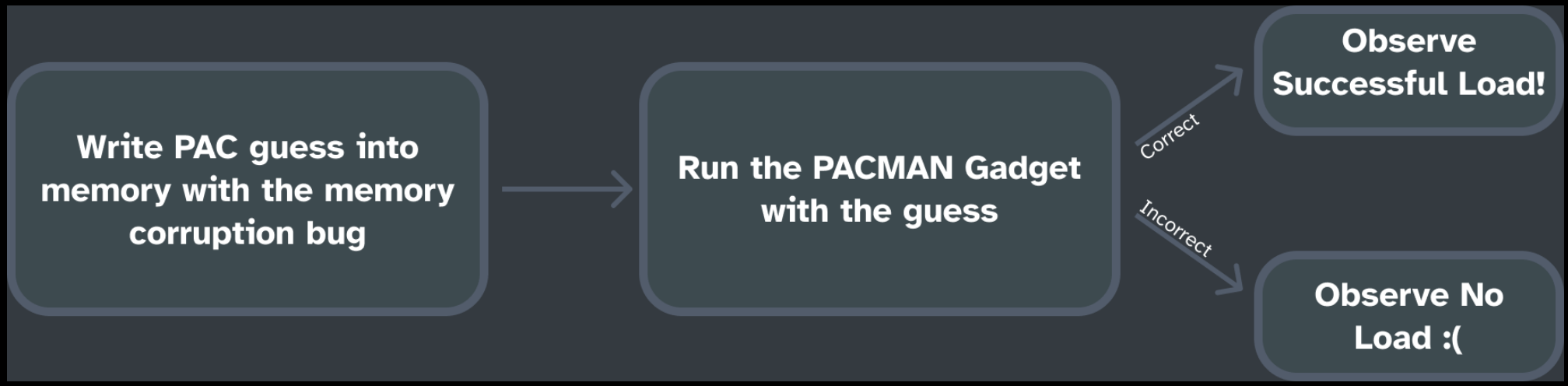


- Simply guessing (16-bit or less) PAC hash
  - Guess wrong and the device may crash application (or reboot device)
- Reuse attacks
  - Harvesting valid pointers from the stack and reusing
    - Stack pointers use SP as the modifier
- Generate Valid Signed Pointers in Another (Attack) Process
  - Only works with global keys (weak modifiers help too); see links above
- Attack the Hashing/Encryption Algorithm (But what is it?)
  - ARM recommends QARMA in Spec (Apple uses a different algorithm; one that encrypts in a single cycle and is low power would be nice)

“PACMan Attack”



- Suppress crashes by performing each PAC guess speculatively on a PACMan Gadget to prevent any crashes. (Uses a microarchitecture side-channel.)



- e.g., Data PACMan Gadget:

```
if (cond):
    verified_ptr <- aut(guess_ptr)
    load(verified_ptr)
```

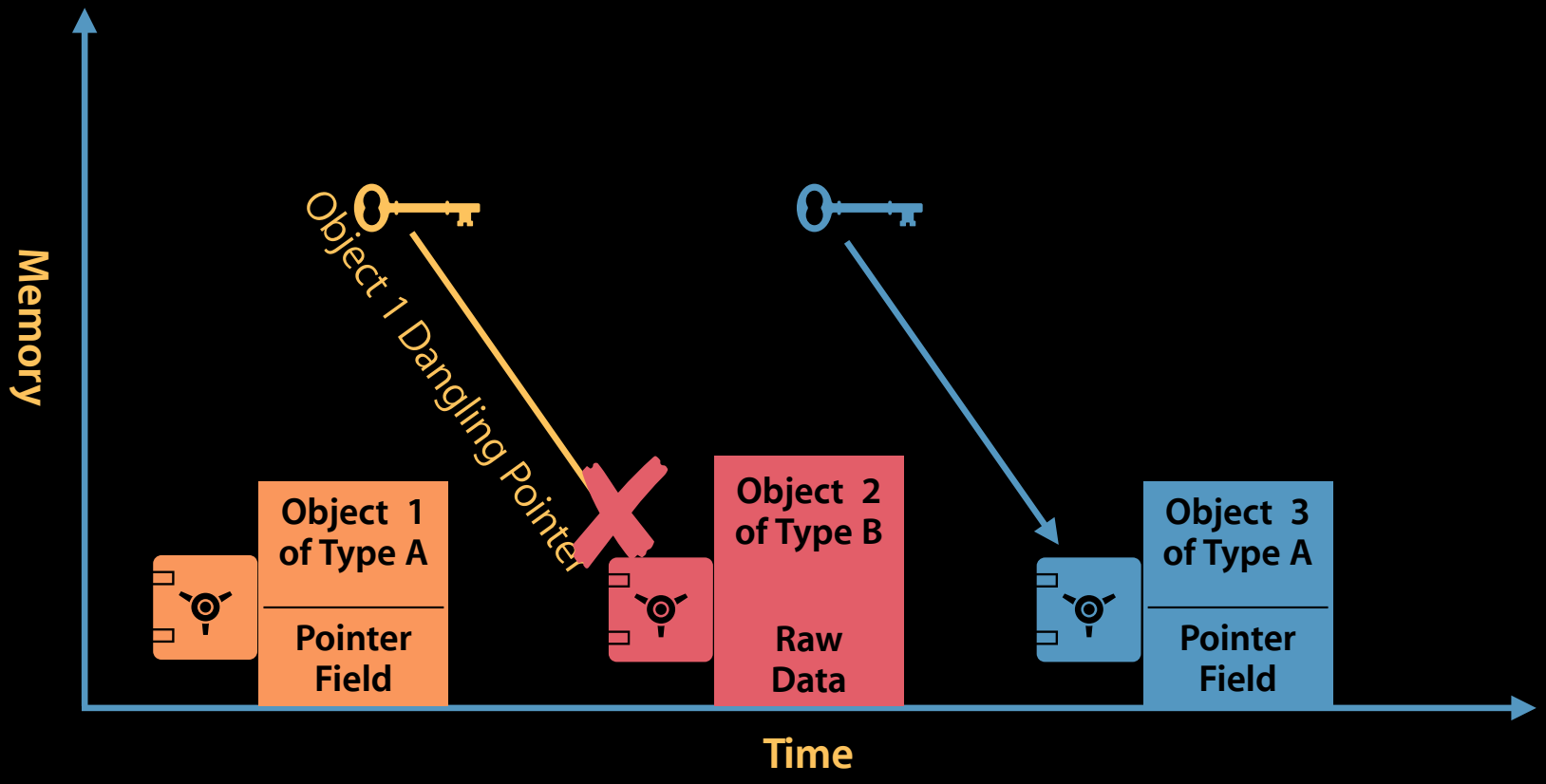
ARM Memory Tagging Extension (MTE)

A Brief History of Memory Tagging

- Originally added to the SUN/Oracle SPARC M7/M8 CPUs around 2016: SPARC Application Data Integrity (ADI)
  - Used the extra bits in ECC RAM for the tag bits. Really only available in their server hardware.
- Around 2018 Google pushed ARM to add it to the ARMv8.5-A spec: ARM Memory Tagging Extension (MTE). Apple evaluates ARM MTE.
- 2019 Apple evaluates MTE and concludes that it is really only good for debugging because performance issues.
- 2022 ARM releases the Enhanced Memory Tagging Extension for ARMv8.9.
- 2023 The Google Pixel 8/8 Pro phones ship with MTE hardware.
- 2025 The Apple iPhone 17 (A18) phones ship with EMTE hardware.

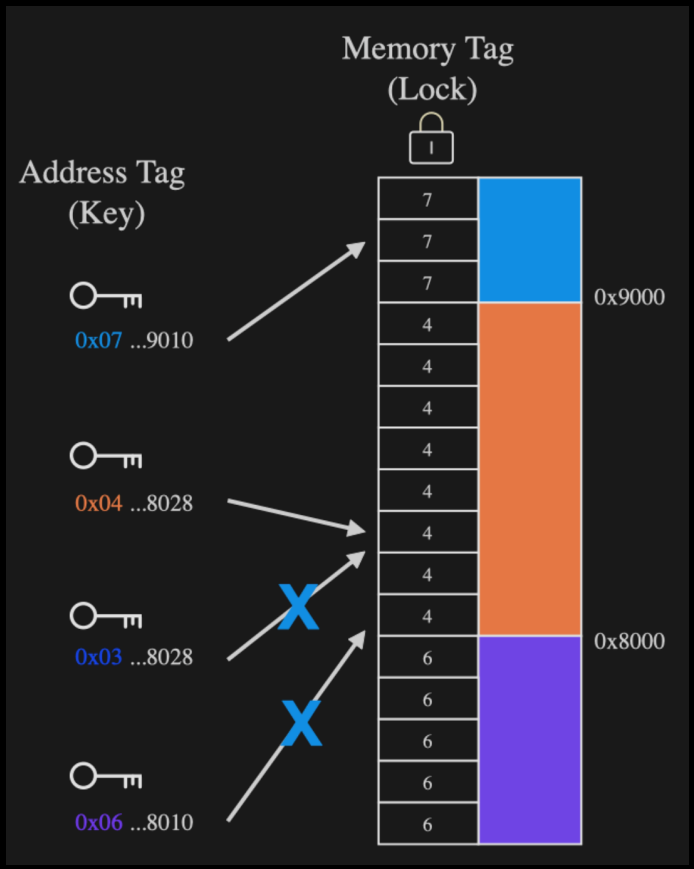
Lock (Memory Object) and Keys (Pointers) Mitigation

Assumption: Change the Lock and Keys No Longer Work



Memory Tagging: Lock and Key

- 16 byte blocks of memory have a 4-bit tag (Lock)
- Pointers have a 4-bit tag (Key)
- Data accesses to a memory region that has a tag is checked with the pointer's tag unless the pointer tag is 0 or 15. Also, the following is unchecked:
  - Instruction fetches
  - Translation table walks, including hardware updates of the Access Flag or Dirty state
  - Data cache maintenance operations
  - Accesses to the Allocation tags



Memory Tagging: Pointer Encoding & Instructions

Pointer Encoding:

MTE Instructions:

- **IRG** - Generates a random tag value and inserts it to a pointer
- **STG** - Sets the tag value for a 16-byte block memory
- **STZG** - Sets the tag value for a block of memory, and zeros corresponding memory location
- **LDG** - Reads the tag value for a block of memory

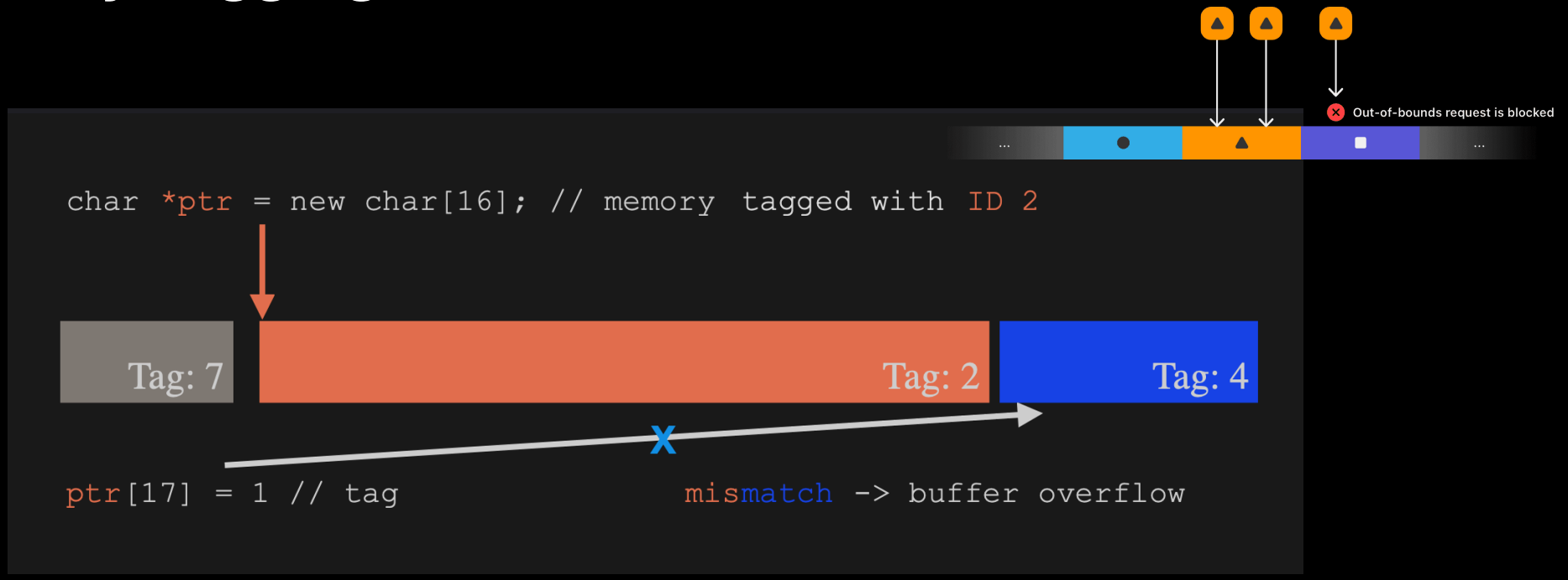
Memory Tagging: Modes of Operation

The architecture provides three operational modes:

- Synchronous checking makes debugging simpler, because it allows you to identify the precise instruction and address that caused the failure. However, synchronous checking typically has a significant performance impact. (High performance overhead. Debugging Mode)
- Asynchronous checking provides less precise information on where the tag comparison failure occurred but can provide some mitigation and be used for profiling. (Moderate performance overhead. Production Mode but with a race condition)
- Asymmetric checking is something between balancing security and performance.

* See Memory Tagging and how it improves C/C++ memory safety

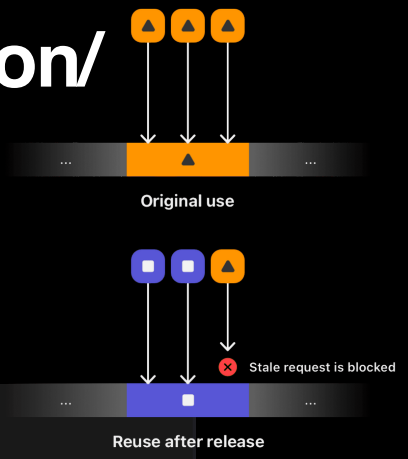
Memory Tagging: Buffer Overflow Detection/Prevention

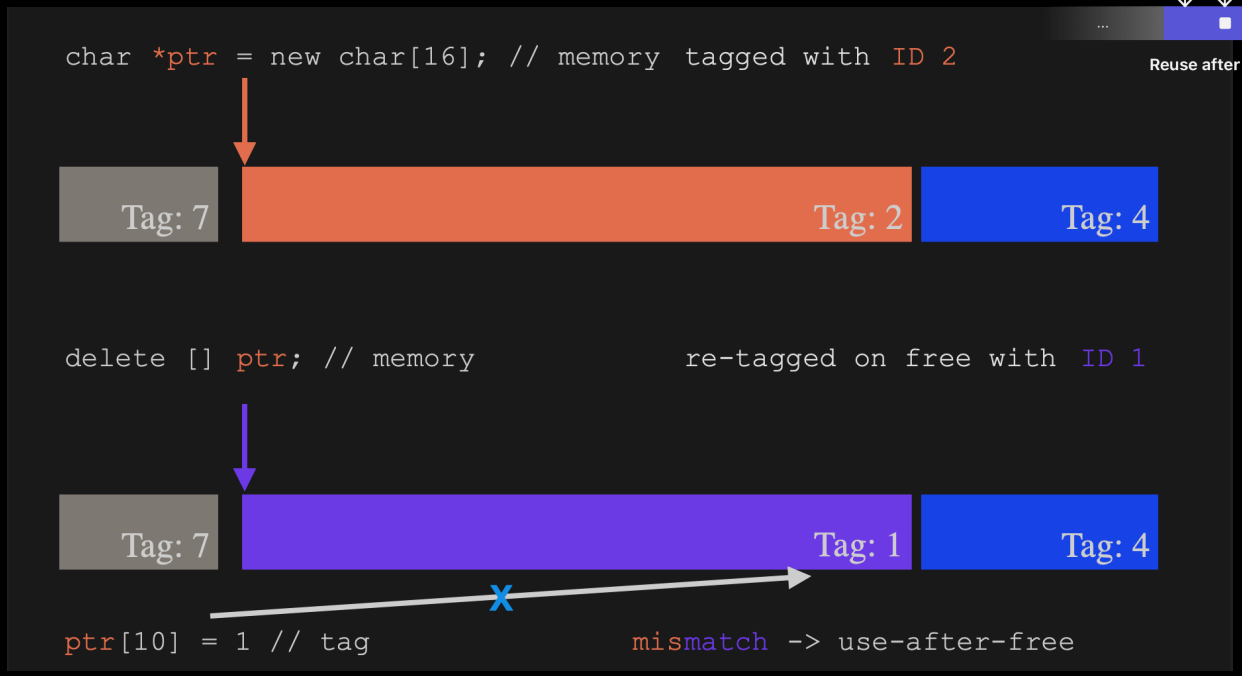


- Each side of a memory region is tagged with a different memory tag.
- If a pointer is used to reference a memory region with a different tag an exception is raised.

Memory Tagging: Use-After-Free (UAF) Detection/ Prevention

- *new* allocates a region a tags it 2. Each side is tagged 7 and 4.
- *delete* frees the region and re-tagged it 2.
- If a dangling pointer using the tag 2 tries to access the memory region (tagged 1) that was freed then an exception is raised.





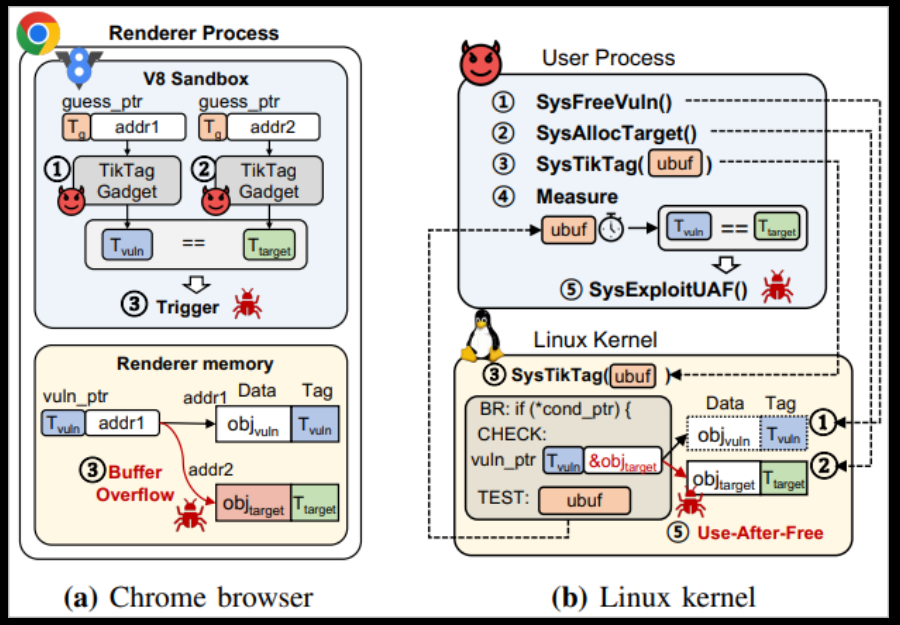
Memory Tagging: Weaknesses and Attacks

- Probabilistic Bug Detection: In guessing the tag in memory there is a 1 in 14 (7.14%) chance (tags “0” and “15” are reserved).
- The address tag is stored in the upper pointer bits, potentially allow the attacker to change the address tag via an integer overflow on the address. Should be coupled with a pointer integrity mechanism like ARM PAC.
- Pointers can be forged on little-endian processors by just changing the lower bits and not the tag in the upper byte.
- Can be used for bounds checking but is a 16 byte granularity so may not be suitable for fine-grained bounds checking/protection.

TIKTAG: Breaking ARM's Memory Tagging Extension with Speculative Execution

Two types of TIKTAG attacks demonstrated:

- TIKTAG-v1 exploits the speculation shrinkage in branch prediction and data prefetching behaviors of the CPU to leak MTE tags. TIKTAG-v1 gadgets work well to exploit the kernel.
- TIKTAG-v2 exploits the store-to-load forwarding behavior in speculative execution, a sequence where a value is stored to a memory address and immediately loaded from the same address.



ARM PAC and MTE Performance Overhead

(From Cage: Hardware-Accelerated Safe WebAssembly using a Google Pixel 8 with an ARMv9 Tensor G3 chip)

Inst	Cortex-X3		Cortex-A715		Cortex-A510	
	Tp	Lat	Tp	Lat	Tp	Lat
	MTE					
irg	1.34	1.99	1.00	2.00	0.50	3.00
addg	2.01	1.99	3.81	1.00	2.22	2.00
subg	2.01	1.99	3.81	1.00	2.22	2.00
subp	3.49	0.99	3.81	1.00	2.50	2.00
subps	2.88	0.99	3.80	1.00	2.50	2.00
stg	1.00	–	1.81	–	1.00	–
st2g	1.00	–	1.84	–	0.46	–
stzg	1.00	–	1.84	–	0.98	–
st2zg	0.34	–	1.79	–	0.45	–
stgp	1.00	–	1.69	–	0.98	–
ldg	2.92	–	1.91	–	0.93	–
	PAC					
pacdza	1.01	4.97	1.51	5.00	0.20	4.99
pacda	1.01	4.97	1.42	5.00	0.20	5.00
autdza	1.01	4.97	1.51	5.00	0.20	7.99
autda	1.01	4.97	1.43	5.00	0.20	7.99
xpacd	1.01	1.99	1.56	2.00	0.20	4.99

Table 1. MTE and PAC instruction throughput (instructions per cycle, higher is better) and latencies (cycles, lower is better). We only show PAC instructions using the Data A-key (da).

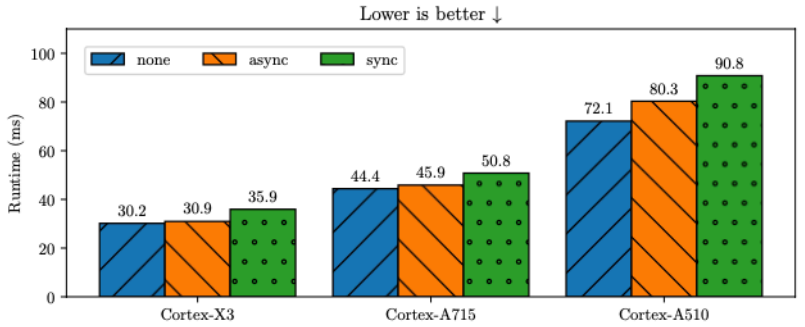


Figure 4. Performance overhead of MTE sync and async mode for writing 128 MiB of memory. See §7.1 for details on the experimental setup.

***memset* microbenchmark**

Synchronous: 19.1%, 14.4%, and 29.9% slower compared to baseline

Asynchronous: 2.6%, 3.3%, and 11.3% slower

StickyTags

Apple's Memory Integrity Enforcement (MIE) 1/2

- Pointer Authentication Codes (PAC) - 2018
- Secure Memory Allocators (Object Type, Zone randomization, etc)
  - IsoHeap and GigaCage (WebKit)
  - kalloc_type (Kernel-level allocator, iOS 15)
  - xzone malloc (User-level allocator, iOS 17)
- Enhanced Memory Tagging Extension (EMTE) - 2022
  - "Always On" synchronous MTE without the huge performance cost

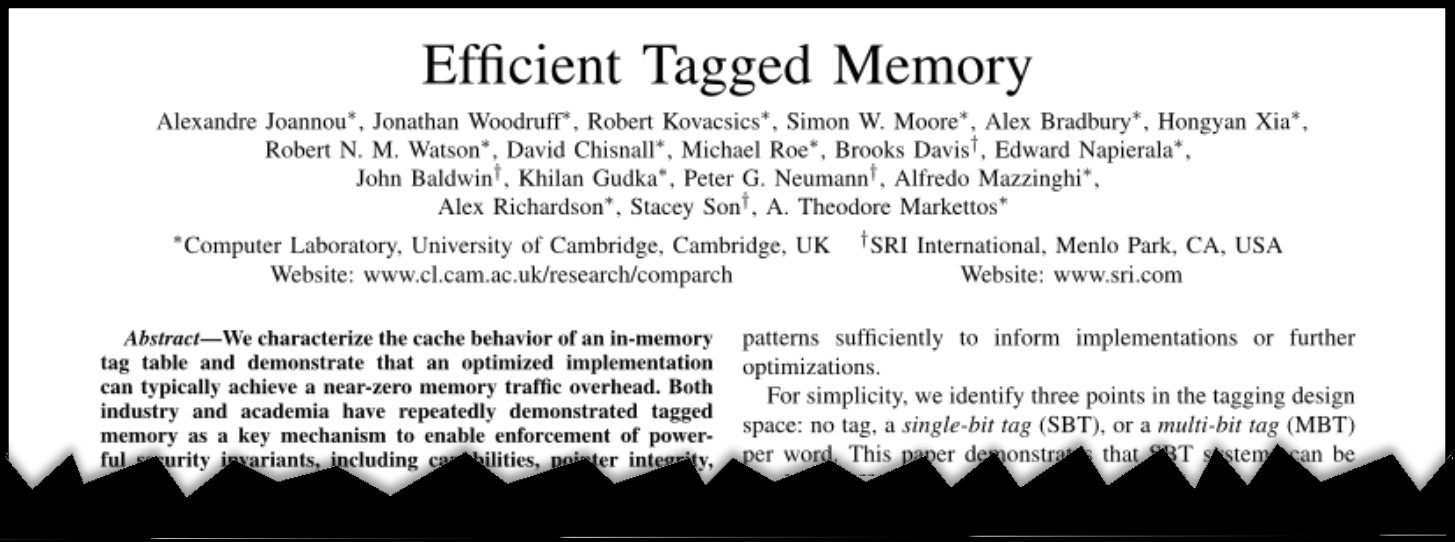
Apple's Memory Integrity Enforcement (MIE) 2/2

- Tag Confidentiality Enforcement and Hardening
  - Hardware hides timing differences so that tag values can't influence speculative execution in any way
  - Allocators assign random tags, the random number in allocators are reseeded often
  - For Spectre variant 1 or V1 (branches) attacks, uses VUSec's TDI information leakage mitigations.

EMTE was added in A18 (iPhone 17) hardware.

How do we implement tagged memory?

Microarchitectural tag storage for off-the-shelf DRAM



Efficient Tagged Memory

Alexandre Joannou*, Jonathan Woodruff*, Robert Kovacsics*, Simon W. Moore*, Alex Bradbury*, Hongyan Xia*,
Robert N. M. Watson*, David Chisnall*, Michael Roe*, Brooks Davis†, Edward Napierala*,
John Baldwin†, Khilan Gudka*, Peter G. Neumann†, Alfredo Mazzinghi*,
Alex Richardson*, Stacey Son†, A. Theodore Markettos*

*Computer Laboratory, University of Cambridge, Cambridge, UK
Website: www.cl.cam.ac.uk/research/comparch

†SRI International, Menlo Park, CA, USA
Website: www.sri.com

***Abstract*—We characterize the cache behavior of an in-memory tag table and demonstrate that an optimized implementation can typically achieve a near-zero memory traffic overhead. Both industry and academia have repeatedly demonstrated tagged memory as a key mechanism to enable enforcement of powerful security invariants, including capabilities, pointer integrity,**

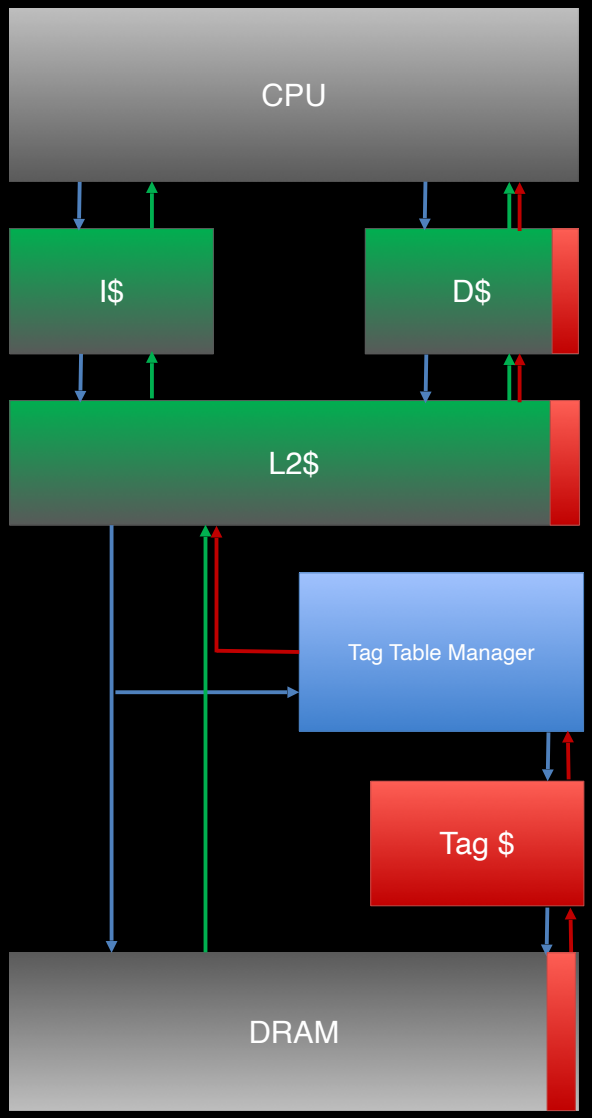
patterns sufficiently to inform implementations or further optimizations.

For simplicity, we identify three points in the tagging design space: no tag, a *single-bit tag* (SBT), or a *multi-bit tag* (MBT) per word. This paper demonstrates that SBT systems can be

- Efficient Tagged Memory
  - Published in the IEEE International Conference on Computer Design (ICCD'17)
- Shift from flat to hierarchal tag table to hold tags in DRAM
  - Exploit inconsistent density of tags in physical memory
  - Reduces DRAM access overhead for a variety of workloads

Simple Tag Cache Hierarchy

- Store tags with data in cache hierarchy
- Tag controller does tag table lookup on DRAM access
- Cache lines of tags from DRAM

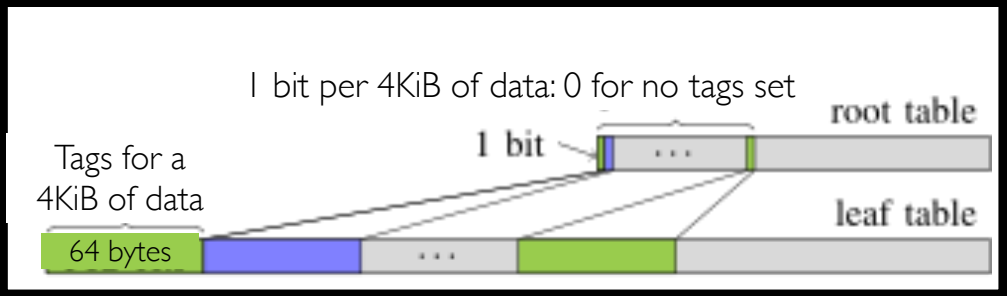


Hierarchical Tag Compression

- Size tag cache line length to 64-byte DDR4 burst transfer size
  ⇒ one line covers tags for 8KiB of memory (128-bit capabilities)

- Many lines don’t contain tags (code, large blocks of data, disk cache, etc.)
  - So handling tag sparseness is important
  - **Only want to pay for tagging when needed**

Tag Compression

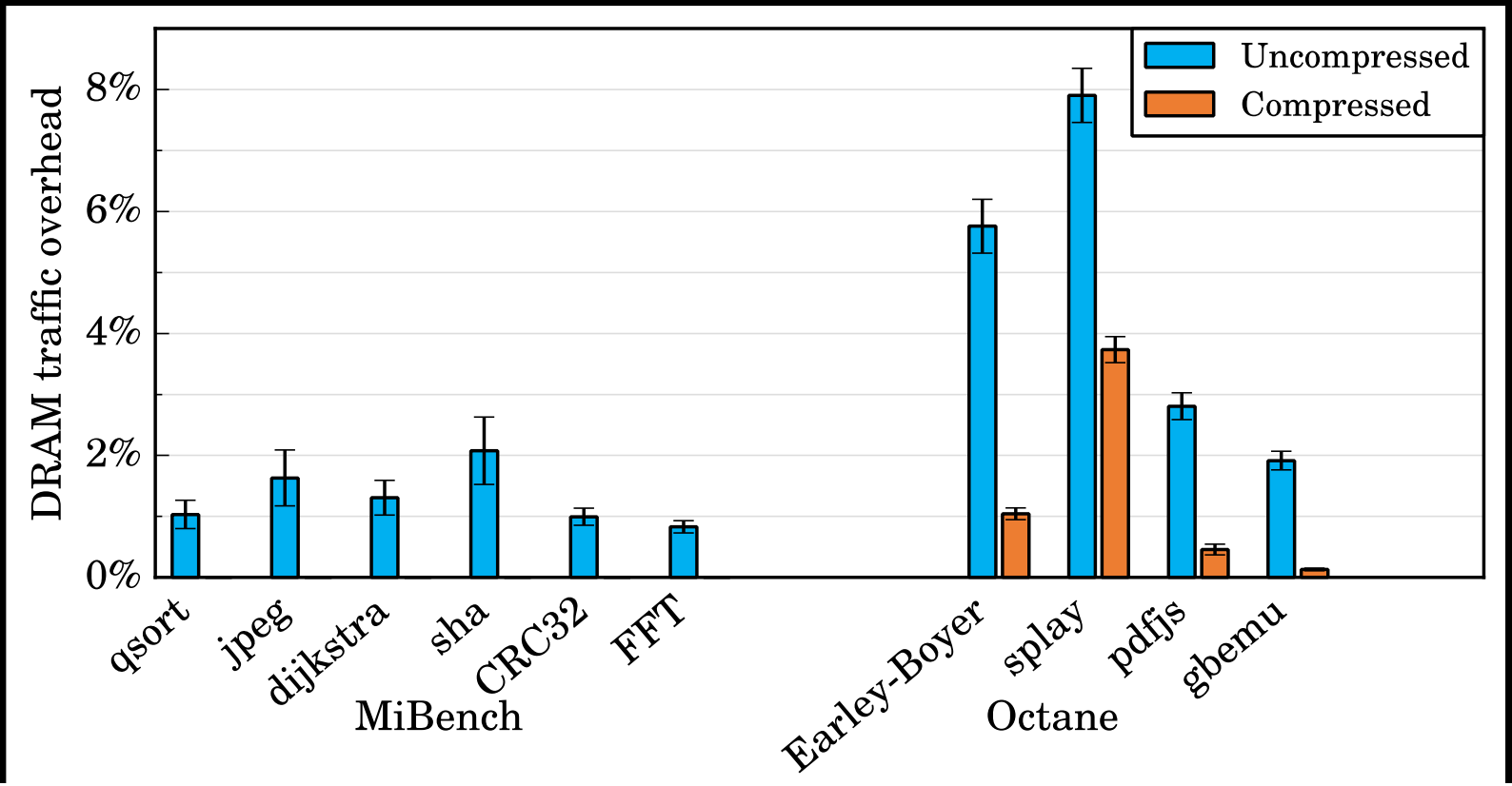


- 2-level tag table
- Each bit in the root level indicates all zeros in a leaf group
- Reduces tag cache footprint
- Amplifies cache capacity

Benchmarks in Hardware

DRAM Traffic Overhead in FPGA Implementation

Note: MiBench overheads with compression are approximately zero



Memory Tagging: Memory Overheads

- MTE uses 4 tag bits for every 16 bytes of memory: 3.125% overhead
  - iPhone 17 Air has 12GB of RAM - about 0.4 GB is needed for tags
- CHERI uses 1 tag bit for every 16 bytes of memory: 0.781% overhead
  - e.g, 12GB RAM - about 0.1 GB is needed for tags
- CHERIoT uses 1 tag bit for every 8 bytes of memory: 1.562% overhead
  - e.g, 12GB RAM - about 0.2 GB is needed for tags (Of course, a 32-bit CPU can address 12GB of RAM so this example is just for comparison.)

Is ARM PAC and (E)MTE worth it for something that seems kind of weak?

- Yes
  - Only have to implement a handful of instructions for each mechanism
  - Performance overhead is low
  - “Always on” MTE and pointer integrity provides lots of telemetry data for fixing bugs. Stack traces of applications with memory safety bugs are sent to Apple from millions of devices. The software bugs then are *usually* fixed.

CHERI Lecture Preview