ARM MTE Performance in Practice

Taehyun Noh, Yingchen Wang, Tal Garfinkel, Mahesh Madhav,
Mattan Erez, Daniel Moghimi, Shravan Narayan





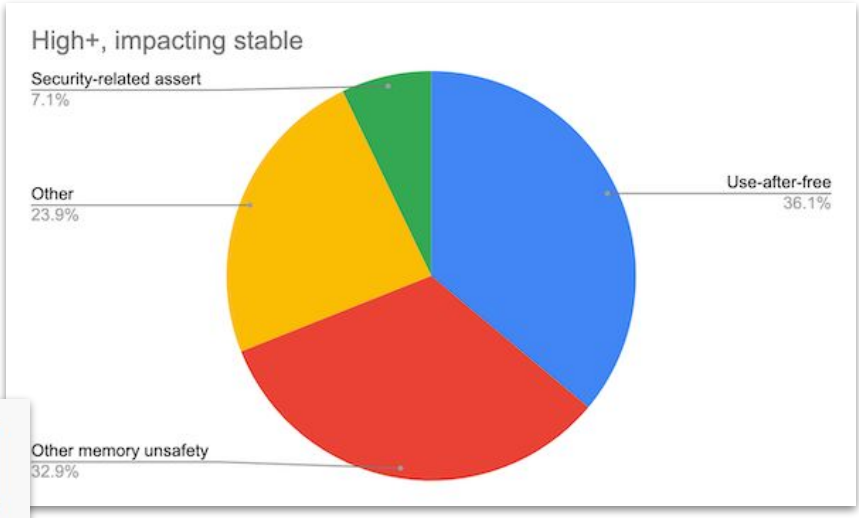


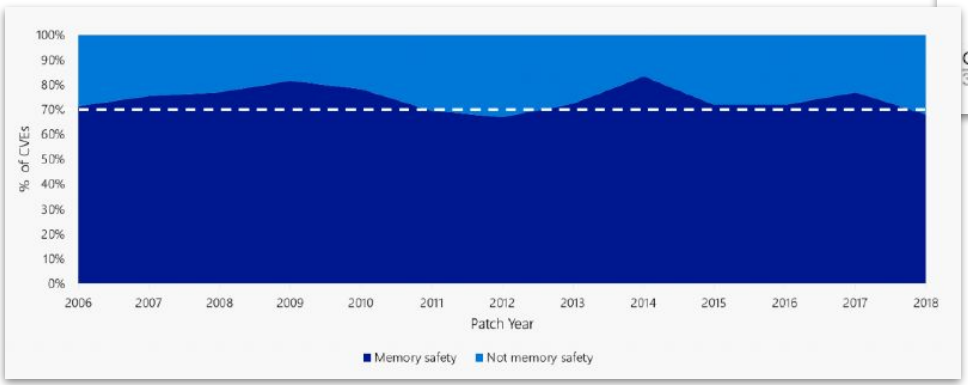
Memory safety bugs are everywhere!

Total CVE Count	Memory Unsafety Bugs	Percentage	Release
44	36	81.8%	10.14.6
45	40	88.9%	10.14.5
38	20	52.6%	10.14.4
23	22	95.7%	10.14.3
13	11	84.6%	10.14.2
71	40	56.3%	10.14.1
64	44	68.8%	10.14

Apple MacOS

The Chromium Project





Microsoft

Rust / Safer-C dialects help, but aren't perfect



Safe C++

Document #:	D3390
Date:	2024-09-11
Project:	Programming Language C++
Audience:	SG23
Reply-to:	Sean Baxter <seanbax.circle@gmail.com> Christian Mazakas <christian.mazakas@gmail.com>

Checked C: Making C Safe by Extension

Archibald Samuel Elliott, University of Washington
Andrew Ruef and Michael Hicks, University of Maryland
David Tarditi, Microsoft Research

"A large-scale rewrite of existing C++ code… will likely remain impractical."
Google Security Engineering Report on Memory Safety

"We're going to be living with software with memory safety issues for quite some time."
Eric Rescorla, Former Firefox CTO

Memory Tagging: hardware mitigation for memory safety

SPARC M7 processor - “Application Data Integrity” (2016)

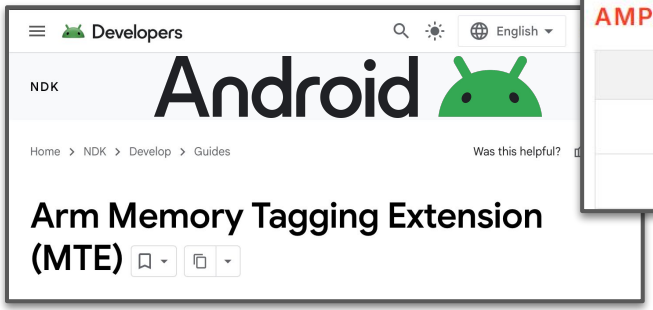


ARM “Memory Tagging Extension” since ARMv8.5 (2019)

arm

Late 2023: Google Pixel 8’s Tensor G3 supports MTE

Apple





AMPERE



MTE in a nutshell

Pointer Tag: Pointers carry a 4-bit tag (aka. color)

⇒ Tag values in bits[56:59]

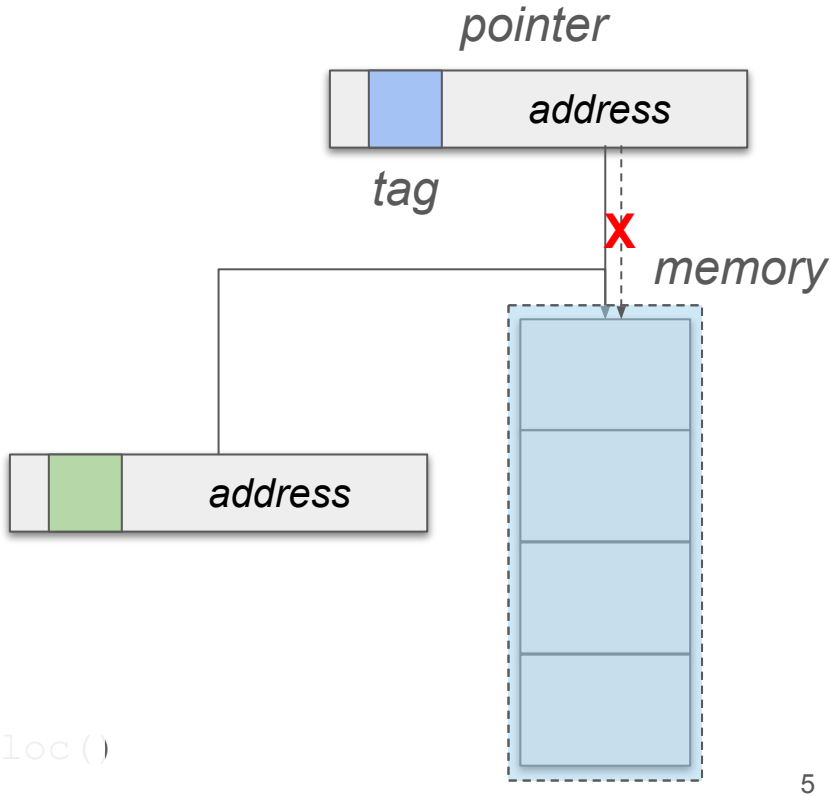
Mem. Tag: Each mem granule (16 bytes) has a tag

⇒ Tag values are store in a shadow memory

Every memop performs a **tag check**.

Mismatch ⇒ **Exception**

Secure **heap allocations:** ⇒ random tag on every `malloc()`



SYNC vs ASYNC MTE: Immediate vs Delayed exceptions

Mode	Behavior on tag mismatch	Use case
Synchronous MTE	Raise Immediate exception (Instruction-level Precision)	Sanitizers / Debugging tools
Asynchronous MTE	Delay exception until the next kernel entry (Good Performance)	Production environment
Asymmetric MTE	Load with immediate exception, Store with delayed exception	Production environment

4-bit tags, MTE is probabilistic

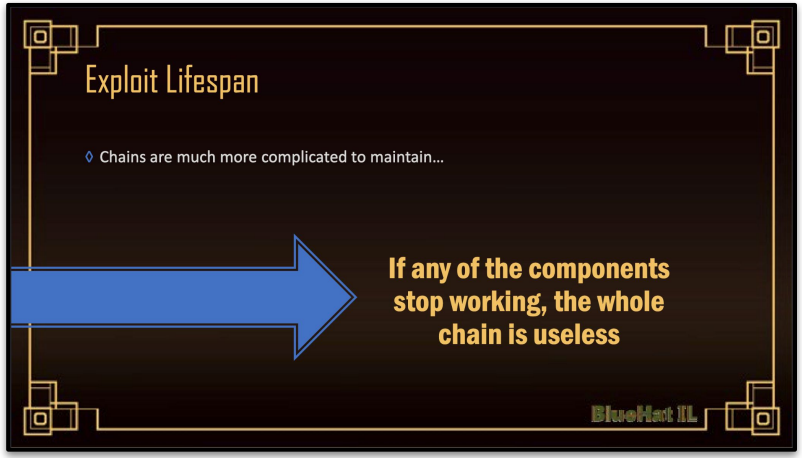
- Purist view ⇒ this is a weak probabilistic guarantee

— but they **break the economics of attacks**

Without MTE: The attacks are silent and successful.

With MTE: the exploits are likely to crash the app (until it succeeds)

⇒ unreliable and detectable



Source: A slide from “Inside The Zeroday Market” by Mark Dowd.

The multi-million dollar exploit will lose its value rapidly

⇒ makes the exploit business model far less viable

Another pragmatic question: Performance?

MTE is expected to be fast!

Alternate viewpoint: it's a debugging feature ⇒ slow

2018 LLVM dev meeting

MTE is better than ASAN

- RAM overhead: 3%-5% (vs ~ 2x)
- CPU overhead: *expected* low-single-digit %

Usenix Security Article, 2019

- CPU overhead is expected to be very small, hopefully a small single-digit percentage, while AddressSanitizer typically has 2x–3x slowdown.

ARM Community Blog

Synchronous Mode

Suitable for Development

- Identify the precise instruction and address that caused failure
- Slight performance impact

"The Case for Memory Safe Roadmaps", CISA, NSA, etc.

Arm introduced another technology called the Memory Tagging Extension (MTE) to some of its CPU product lines to detect use after free and out-of-bounds (also called buffer overflow) type bugs.[34] When memory is allocated, the system assigns it a tag. All further access to that memory must be made with that tag. The CPU will raise an error if the tags do not match. Arm estimates that the overhead for MTE is between 1–2 percent. Mobile devices may see the first widespread deployments.[35] Intel also announced memory tagging capabilities in future chipsets. [36]

So, is it really as good as promised?

SPEC2006 with heap MTE: 7 Cores (Pixel 8/9, AmpereOne) x 3 modes

Pixels have three types of cores,
e.g., Pixel 8 Perf = Cortex-X3, Big = Cortex-A715, Little = Cortex-A510

	Pixel 8 Perf Core			Pixel 8 Big Core			Pixel 8 Little Core			Ampere
SPEC CPU2006	**ASYNC**	**SYNC**	**ASYMM**	**ASYNC**	**SYNC**	**ASYMM**	**ASYNC**	**SYNC**	**ASYMM**	**SYNC**
400.perlbench	1.13	1.63	1.16	1.22	1.24	1.23	1.16	1.18	1.17	1.12
401.bzip2	1.08	1.84	1.16	1.00	1.40	1.00	1.01	1.10	1.01	1.14
403.gcc	1.21	1.57	1.29	1.82	1.83	1.81	1.19	1.26	1.20	1.23
429.mcf	1.05	1.19	1.05	1.12	1.17	1.12	1.07	1.07	1.07	1.01
445.gobmk	1.01	1.04	1.01	1.02	1.04	1.03	1.02	1.02	1.01	1.01
456.hmmer	1.00	6.64	1.12	1.00	1.01	1.00	1.02	1.05	1.02	1.43
458.sjeng	1.03	1.06	1.03	1.02	1.02	1.00	1.22	1.24	1.22	1.01
462.libquantum	1.07	1.06	1.05	1.05	1.06	1.04	1.01	1.12	1.05	1.04
464.h264ref	1.00	2.37	1.01	1.00	1.08	1.00	1.03	1.14	1.03	1.07
471.omnetpp	1.16	1.38	1.18	1.54	1.60	1.51	1.10	1.20	1.11	1.13
473.astar	1.03	1.07	1.03	1.07	1.10	1.05	1.07	1.32	1.07	1.04
483.xalancbmk	1.15	1.32	1.16	1.54	1.57	1.55	1.14	1.18	1.16	1.09
Geomean	**1.08**	**1.56**	**1.10**	**1.18**	**1.23**	**1.17**	**1.08**	**1.14**	**1.08**	**1.10**

Anomaly 3

Anomaly 1

Anomaly 2

Anomaly 1: HMMER (6.7x on MTE SYNC, Pixel Perf)

Analysis using perf: one hot loop runs for > 90%

```
% time name
 95.19 P7Viterbi
  2.24 sre_random
  1.96 FChoose
```

Hot loop: a dynamic programming algorithm

⇒ Repeat use of the pattern `ic[k] = mpp[k] + tpmi[k]`

Our hypothesis: consecutive stores cause the overhead.

4.15 Memory Tagging - Synchronous Mode

In synchronous tag checking mode, stores cannot be performed speculatively. Each store must complete a tag check before the next store can be executed non-speculatively. Thus, performance of stores in synchronous tag checking mode will be diminished.

It is recommended to use asynchronous or asymmetric mode for better performance.

Microbenchmark: a loop of one store + an optional DMB ST barrier

Reproduces the ~7× slowdown; adding the fence changes nothing

⇒ tag checks already serialize every store on this core

Source: ARM Cortex-X3 Core Software Optimization Guide

Anomaly 2: Why is ASYNC so slow?

50~80% overhead on ASYNC mode (big core)

perf: load-dominant functions grow the most

- e.g., search in any container data structure
- Xalancbmk with MTE enabled
  - `ValueStore::contains` 19% → 39% of runtime

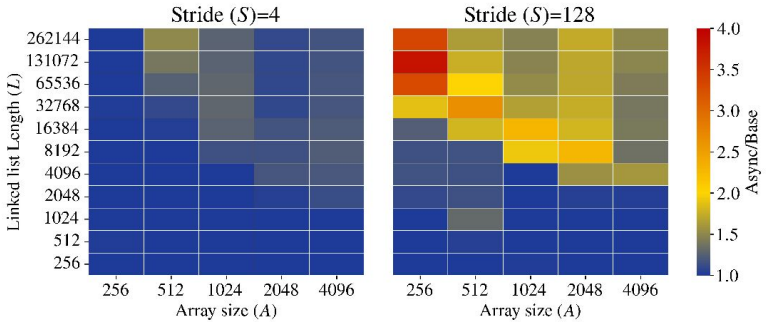
Microbenchmark: walk a linked list of arrays

- list length → random accesses
- array size / stride → sequential accesses

	Pixel 8 Big Core		
SPEC CPU2006	**ASYNC**	**SYNC**	**ASYMM**
400.perlbench	1.22	1.24	1.23
401.bzip2	1.00	1.40	1.00
403.gcc	1.82	1.83	1.81
429.mcf	1.12	1.17	1.12
445.gobmk	1.02	1.04	1.03
456.hmmer	1.00	1.01	1.00
458.sjeng	1.02	1.02	1.00
462.libquantum	1.05	1.06	1.04
464.h264ref	1.00	1.08	1.00
471.omnetpp	1.54	1.60	1.51
473.astar	1.07	1.10	1.05
483.xalancbmk	1.54	1.57	1.55
Geomean	**1.18**	**1.23**	**1.17**

Microbenchmark: Testing ASYNC memory loads (Big core)

ASYNC MTE / no MTE (Big core)



Microbenchmark Results

- **Heatmap:** Overheads for many link-list and record sizes
- We see hotspots of ~4x
- perf shows similar increases in "*backend stalling*"

Why are we seeing this "*backend stalling*"?

- MTE load = memory load + tag check
- More cache misses ⇒ more in-flight loads, tag checks
- **Hypothesis:** low capacity of in-flight tag checking
  ⇒ tag check throughput < memop throughput

Anomaly 3: HMMER on AmpereOne

perf: significant decreases on `ld_from_st_fwd`

We disable store to load forwarding globally.

Result: removes MTE overhead from 43% to negligible 1%!

- Ampere confirmed this implementation bug
- They have fixed the implementation

	Ampere
SPEC CPU2006	**SYNC**
400.perlbench	1.12
401.bzip2	1.14
403.gcc	1.23
429.mcf	1.01
445.gobmk	1.01
456.hmmer	1.43
458.sjeng	1.01
462.libquantum	1.04
464.h264ref	1.07
471.omnetpp	1.13
473.astar	1.04
483.xalancbmk	1.09
Geomean	**1.10**

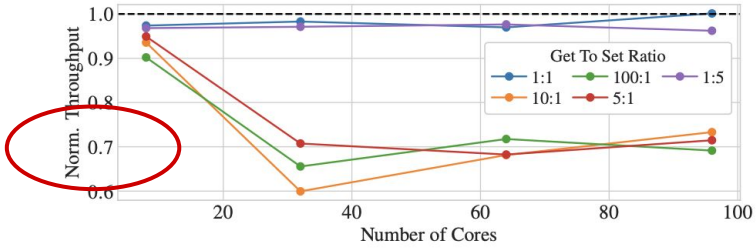
Server apps on AmpereOne

RocksDB, Nginx, PostgreSQL, Memcached — mostly under 18% loss, consistent with SPEC

(a) RocksDB: tests on different I/O patterns in database operations.

(b) Nginx: tests on serving static files.

Anomaly 4: memcached with 30~40%



(d) Memcached: querying key-value pairs with diff. get-to-set ratios.

Specification ambiguity: Tag Check Fault field

The kernel was tag checking silently, but why?

TCF	Meaning
0b00	Tag Check Faults have no effect on the PE.
0b01	Tag Check Faults cause a synchronous exception.
0b10	Tag Check Faults are asynchronously accumulated.
0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.

1. Avoid comparing tags



ARM



2. Compare Tags, but ignore results

AMPERE

Source: ARM Architecture Reference Manual

Solution: TCMA1, yet another control register

Initially, we submitted a patch that toggles tag checks at various places.

- Performance impact of toggling could be noticeable in some corner cases.

We landed a simpler solution: **setting TCMA1 on boot**

- If TCMA1 == 1, An access is unchecked if a kernel pointer has a default tag (0b1111).
- Once the kernel is **NOT** using MTE, tags of kernel pointers will never change.

Memcached overhead is gone!

And our patch has been deployed at Linux 7.0+

Small runtime misconfiguration can distort our view on performance

High-level observations on MTE Perf

	Pixel 8 Perf Core			Pixel 8 Big Core			Pixel 8 Little Core			Ampere
SPEC CPU2006	ASYNC	SYNC	ASYMM	ASYNC	SYNC	ASYMM	ASYNC	SYNC	ASYMM	SYNC
400.perlbench	1.13	1.63	1.16	1.22	1.24	1.23	1.16	1.18	1.17	1.12
401.bzip2	1.08	1.84	1.16	1.00	1.40	1.00	1.01	1.10	1.01	1.14
403.gcc	1.21	1.57	1.29	1.82	1.83	1.81	1.19	1.26	1.20	1.23
429.mcf	1.05	1.19	1.05	1.12	1.17	1.12	1.07	1.07	1.07	1.01
445.gobmk	1.01	1.04	1.01	1.02	1.04	1.03	1.02	1.02	1.01	1.01
456.hmmer	1.00	6.64	1.12	1.00	1.01	1.00	1.02	1.05	1.02	1.43
458.sjeng	1.03	1.06	1.03	1.02	1.02	1.00	1.22	1.24	1.22	1.01
462.libquantum	1.07	1.06	1.05	1.05	1.06	1.04	1.01	1.12	1.05	1.04
464.h264ref	1.00	2.37	1.01	1.00	1.08	1.00	1.03	1.14	1.03	1.07
471.omnetpp	1.16	1.38	1.18	1.54	1.60	1.51	1.10	1.20	1.11	1.13
473.astar	1.03	1.07	1.03	1.07	1.10	1.05	1.07	1.32	1.07	1.04
483.xalancbmk	1.15	1.32	1.16	1.54	1.57	1.55	1.14	1.18	1.16	1.09
Geomean	**1.08**	**1.56**	**1.10**	**1.18**	**1.23**	**1.17**	**1.08**	**1.14**	**1.08**	**1.10**

Obs 1: Most programs perform well (~40% progs under 5% overhead, 67% under 15%)

But what about the perf cliffs?

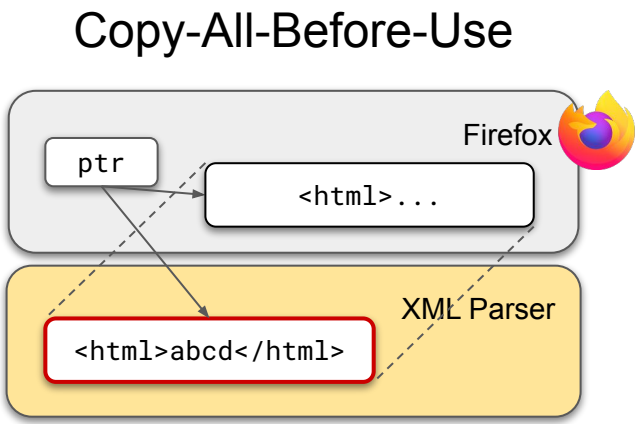
Obs 2: Every cliff is implementation-specific — none looks fundamental

- SYNC on Pixel Perf core: store serialization — absent on other cores
- ASYNC on Pixel Big core: tag-check throughput — balanced on other cores
- SYNC on AmpereOne (hmmer 1.43×): store-to-load forwarding bug — confirmed & fixed by Ampere
- Memcached on AmpereOne: kernel TCF misconfiguration — our patch landed in Linux 7.0

Heap memory safety is only one use-case for tagging!

BufLock: Memory-efficient TOCTOU mitigation

Firefox isolates buggy libraries (e.g., its XML parser) in WebAssembly*.



1. Tag the buffer to Green
2. Ensure XML parser cannot use the Green tag

⇒ Free in WebAssembly

** Narayan et al., "Retrofitting fine grain isolation in the Firefox renderer," USENIX Security 2020*

Evaluation: `memcpy` vs BufLock

Addresses a real problem!

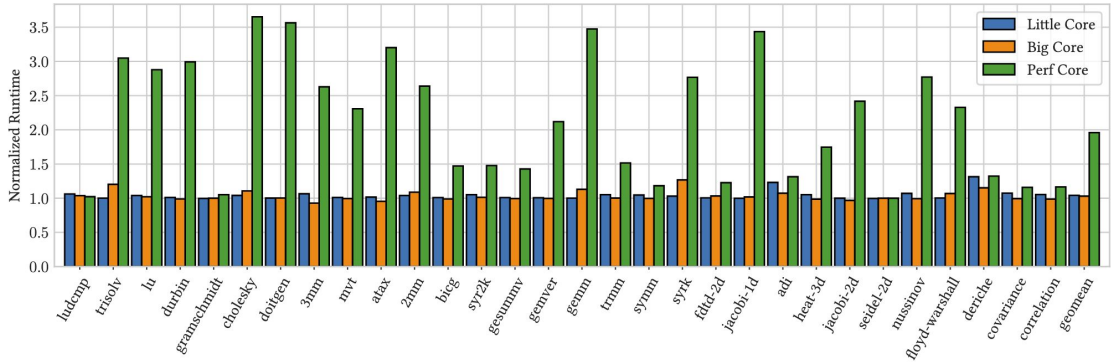
- **Today:** Firefox uses the insecure version due to cost of copying (~10%)
- BufLock overhead is in the “deployable” range.

Cores	Insecure (sec)	Data copies	BufLock
Pixel8 (Little)	1.19	1.28(+7.9%)	1.20(+1.0%)
Pixel8 (Big)	0.31	0.35(+11.8%)	0.32(+3.1%)
Pixel8 (Perf)	0.17	0.19(+13.2%)	0.24(+41.5%)
Pixel9 (Little)	0.86	0.92(+7.1%)	0.89(+3.7%)
Pixel9 (Big)	0.23	0.25(+10.4%)	0.25(+8.5%)
Pixel9 (Perf)	0.13	0.14(+10.3%)	0.18(+38.1%)
AmpereOne	0.23	0.25(+9.7%)	0.24(+5.1%)

BufLock: an efficient revocation mechanism to avoid TOCTOUs

Not all MTE uses are viable today, e.g., MTE sandboxes

Idea: A program that unconditionally uses a fixed MTE tag
⇒ cannot access memory with different tag.



We see overheads up to 3.5x!

Software sandboxing toolchains are much faster
⇒ LFI reports the worst case 17%, compared to native *

** Yedidia, Zachary. "Lightweight fault isolation: Practical, efficient, and secure software sandboxing."*

More in the paper

1. **Two more tagging schemes; one positive, one negative**
   MTE-based unlimited watchpoint and MTE-based CFI

2. **Evaluation pitfalls in prior MTE work**
   Software analogs show zero correlation with real hardware

3. **Preliminary look at Apple's MTE implementation**
   First numbers from Apple's implementation

4. **MTE wishlist for the next generation**
   Precise ASYNC reporting · mismatch logging instead of faulting · cross-core tag synchronization

Conclusion: The Path Forward for Memory Safety

- MTE: Provides probabilistic solutions to the memory safety problem.
  - It is going to impact the economics of attacks significantly.
- First gen MTE implementations have some issues, but they are addressable.
- Implementation is EVERYTHING: individual cost-benefit analysis

Questions?


taehyun@utexas.edu


@taehyun@infosec.exchange