ARM MTE Performance in Practice (Extended Version)

Taehyun Noh
UT Austin

Yingchen Wang
UC Berkeley

Tal Garfinkel
Google

Mahesh Madhav
Ampere Computing

Daniel Moghimi
Google

Mattan Erez
UT Austin

Shravan Narayan
UT Austin

Abstract

We present the first comprehensive analysis of ARM MTE hardware performance on four different microarchitectures: ARM Big (A7x), Little (A5x), and Performance (Cortex-X) cores on the Google Pixel 8 and Pixel 9, and on Ampere Computing's AmpereOne CPU core. We also include preliminary analysis of MTE on Apple's M5 chip. We investigate performance in MTE's primary application—probabilistic memory safety—on both SPEC CPU benchmarks and in server workloads such as RocksDB, Nginx, PostgreSQL, and Memcached. While MTE often exhibits modest overheads, we also see performance slowdowns up to $6.64\times$ on certain benchmarks. We identify the microarchitectural cause of these overheads and where they can be addressed in future processors. We then analyze MTE's performance for more specialized security applications such as memory tracing, time-of-check time-of-use prevention, sandboxing, and CFI. In some of these cases, MTE offers significant advantages today, while the benefits for other cases are negligible or will depend on future hardware. Finally, we explore where prior work characterizing MTE performance has either been incomplete or incorrect due to methodological or experimental errors.

1 Introduction

Memory safety vulnerabilities remain one of the biggest challenges in systems security. Consequently, ARM's memory tagging extension (MTE) [14]—that detects and probabilistically mitigates these bugs—has been highly anticipated [18, 30]. MTE promises several important properties including: *generality*—detecting both temporal and spatial safety bugs; *compatibility*—the ability to work with unmodified applications and only minimal changes to existing allocators and operating systems; and *low overhead*—allowing it to be on by default on production settings. The reality of this last promise, the actual cost of MTE in it different modes (SYNC, ASYNC, and ASYMM), can only be found in silicon. Understanding that reality is our goal in this paper.

Security features live and die by their performance. The choice to use any feature is a cost-benefit analysis; the higher the cost, the less likely a feature is to see production use. At present, there is a significant gap in our knowledge of MTE's performance costs in real hardware; what those costs are for different use cases; where those costs come from at the microarchitectural level; how those costs vary across different microarchitectures; and which costs are fundamental vs. fixable in the next generation of hardware.

To bridge this gap, we analyze the performance of MTE in five different microarchitectures, ranging from mobile (Pixel 8 and 9 "Performance" (Cortex-X), "Big" (A7x), and in-order "Little" (A5x)) cores [1] to laptop (Apple M5 [12]) and high-end servers (AmpereOne [10]). We evaluate these architectures using traditional and server benchmarks across a variety of scenarios. We analyze performance for both the primary use case of probabilistic memory safety and several advanced applications of MTE inspired by prior work [24, 48, 56, 71, 85, 90], including time-of-check time-of-use (TOCTOU) attack mitigations, memory tracing, sandboxing, and control-flow integrity. Finally, we carefully investigate performance regressions with different MTE modes and use targeted microbenchmarks and performance analysis tools to attribute those to microarchitecture decisions in different core types or software issues.

Probabilistic memory safety. We measure MTE's overhead in its primary use case—probabilistic memory safety—using SPEC CPU 2006 INT [50] (§4) and a collection of popular server workloads including RocksDB, Nginx, PostgreSQL, and Memcached [1, 3, 4, 33].

Our main takeaway is that while many benchmarks run with low performance overheads, each microarchitecture exhibits outlier performance regressions. On SPEC INT 2006, these can reach up to $6.64\times$ (Pixel Performance core with MTE SYNC) and be above $1.82\times$ even in the more relaxed security guarantees of MTE ASYNC (Pixel Big core). The AmpereOne core also exhibits a wide range of performance

---

[1] We technically measured 6 different Pixel cores—3 on the Pixel 8, and 3 on the Pixel 9. But both SoCs exhibit similar performance patterns.

overhead that is mostly in the single digits, but can reach $1.43\times$. A similar range of overheads is observed on the server benchmarks with AmpereOne.

Microarchitectural analysis. We investigate the cause of these performance cliffs in MTE's SYNC and ASYNC modes and found that they stem from some limitations in current microarchitectures:

Store serialization: The Pixel 8 and 9 Performance cores exhibit significant performance cliffs of up to $6.64\times$ on specific SPEC CPU benchmarks. We use targeted microbenchmark experiments to attribute these overheads to a design decision that serializes stores in MTE SYNC. This choice is limited to the Performance core.

Structural hazard on tag-checking of loads: The Pixel 8 and 9 Big core exhibits slowdowns of up to $1.82\times$ in both SYNC and, more importantly, ASYNC on certain SPEC CPU benchmarks. We used microbenchmarks to narrow down the cause of this overhead to bottlenecks in the structures that support in-flight tag-checks, which artificially restrict the number of allowed in-flight memory operations.

Store-to-load forwarding: AmpereOne had overheads on specific benchmarks up to $1.43\times$. We identified that this was because its store-to-load forwarding—an important optimization in out-of-order cores that forwards the result of older stores to younger loads without passing through the cache—appears to have inconsistent behavior.

On a positive note, when we shared our findings with Ampere, they confirmed the presence of a performance issue in the store-to-load forwarding mechanism when interacting with MTE tag checks. They also mentioned that this has been fixed in the next-generation silicon supporting MTE. This provides strong support for the idea that MTE's current limitations are surmountable and that an MTE implementation with consistently low overheads is possible and on the horizon.

Other sources of overheads. While overheads on server benchmarks were usually under $1.18\times$, some memcached workloads slowed down by up to $1.40\times$ (§4.3). Due to the Linux kernel's reliance on an ambiguous portion of ARM's specification, tagging overheads were imposed on all kernel memory operations—rather than just user space pages with tagging enabled—on Ampere chips. We shared our findings with Ampere and together we developed a fix, following suggestions from Linux kernel developers [98].

Beyond memory safety. Next, we turn our attention to MTE's performance for other use cases. We developed MTE-based tools for other important uses of tagged architectures including: time-of-check time-of-use prevention based on buffer revocation (§5.1.1), memory tracing (§5.1.2), optimizing sandboxing (§5.2.1), optimizing/enforcing CFI (§5.2.2).

For memory tracing, we found that MTE enables simple implementation and low overhead, with performance roughly $2-3\times$ faster than other approaches like using page permissions. For TOCTOU prevention, MTE offers performance

boosts of up to $1.087\times$ over alternate approaches like copying on most cores (although Pixel's Performance core alone shows a pessimization when using this technique). In contrast, we find using MTE to enforce sandboxing or CFI offers no clear performance benefits over existing software approaches [90, 100]; in particular for sandboxing, where its performance cliffs could induce high and hard to predict overheads.

Other contributions. We also explore how the performance overheads we observed are not captured by prior estimates of MTE's performance using analogs (§6.1), and sometimes due to experimental error (§6.2), highlighting the importance of accurate evaluations on real hardware to provide a solid foundation for other work to build on.

We offer a few observations (§7) on how ARM could improve future standards to enhance MTE's functionality, e.g., enhancing MTE's value as a tracing mechanism, and improving ASYNC mode to provide sufficient information to diagnose MTE traps.

Finally, we note that while the microarchitecture implementations of MTE differ in design and performance characteristics, all seem to have encountered similar challenges, evidenced by performance degradation and cliffs on overlapping benchmark sets. This is true even for the original SPARC ADI implementation which struggles on the same benchmarks [84]. Thus, future implementations—e.g., ARM based CPU's from cloud providers [9, 45] and RISC-V implementations [80]—should be aware of these issues.

2   Why MTE?

ARM's Memory Tagging Extension (MTE) [14] is unique among current memory safety mitigations. It promises to detect and mitigate (with the caveats below) a wide range of temporal and spatial memory-safety bugs with no application changes[2], low enough overhead to be deployed in production, and a modest enough cost in silicon to make adoption feasible.

The key challenge of enforcing memory safety is tracking metadata, e.g., array bounds for spatial safety. Precisely tracking this state in hardware can be very challenging, and existing solutions often compromise performance, complexity, or compatibility [72, 95]. MTE avoids many of these compromises using a simpler approach to tracking metadata, which sacrifices precision but retains these other useful properties.

MTE relies on 4-bit tags, which map all allocations into 16-congruence classes. Upon allocation, the allocator assigns a random tag to the address range of each allocation and returns a pointer with the top bits containing a matching tag. Thus, if that pointer is used to access an object with a different tag (e.g., an out-of-bounds memory access, or a use of a dangling pointer), there is a 15/16 chance it will trap.

---

[2]MTE only requires small changes to the OS and user space allocator.

Allocators also employ techniques to improve MTE's detection of memory safety errors by reserving a unique tag for allocator data-structures or employing guard regions after large allocations [74], however, these techniques do not change the fact that MTE offers only probabilistic defenses against memory safety bugs.

MTE's probabilistic nature and other limitations, such as its reliance on tag secrecy to prevent pointer foraging, its vulnerability to tag leaks via side channels, and its lack of intra-object safety, have led to the perception that MTE is a debugging rather than a security feature.

However, this is misleading. Although MTE does not provide sound memory safety, it is very likely to render a great majority of memory safety-based exploits unreliable and detectable. Both of these are powerful properties.

Unreliable exploits are far less valuable to attackers. Imagine if spyware that relies on zero-click exploits to deploy—such as NSO's Pegasus [27]—instead of silently compromising an application, caused it to repeatedly crash. Not only would this alert victims, but the subsequent crash report would also signal the bug's presence to both the application developer and platform vendor, likely significantly shortening the life of the zero-day exploit used.

Moreover, high-end vulnerabilities of this type are already worth millions of dollars [73]; businesses that sell offensive tools that rely on them are only viable because they can often amortize this cost across many uses. If most of the memory safety vulnerabilities on targeted devices were rendered unreliable and rapidly detected and remediated, it could significantly change the cost of carrying out such attacks [34].

As MTE can detect the immediate causes of intrusions, it could also potentially enable more reliable detection and faster remediation of server-side bugs. Thus, while MTE is not a substitute for sound mitigations for memory safety bugs, it could significantly shift the battlefield of the "eternal war in memory" [89] toward the defender's favor.

2.1 How MTE Works

MTE is a form of tagged memory [31, 37, 53, 95–97], i.e., it associates metadata (a tag) with each portion of the processor's address space. In MTE's case, this metadata is a 4-bit tag (color) for each 16-byte range (granule) of physical memory. The high-level operation of checking tags is shown in Figure 1. This design is based on the Application Data Integrity (ADI)—the memory tagging feature in the SPARC M7 [8] that used 64 byte granules. MTE's smaller granule size reduces the alignment constraint that ADI's larger granules impose on allocators, resulting in less wasted memory [84]. MTE supports three different modes of operation that control when and how it handles tag mismatches:

SYNC mode. In SYNC mode, the processor takes a precise exception on a tag mismatch. Thus, it provides *precise*
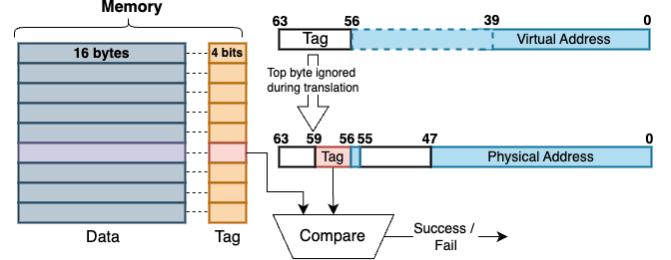

Figure 1: MTE overview. MTE allows assigning of 4-bit tags to each 16-byte memory granule. Subsequent reads from (or writes to) tagged memory must have the correct tag in bits[56:59] of the virtual address. An incorrect tag results in a fault in MTE SYNC, or a deferred fault in MTE ASYNC.

enforcement—enabling access control; and *precise reporting*—it identifies the faulting instruction, making it ideal for diagnosing the cause of a fault. While SYNC mode is the most useful, it is also challenging to implement efficiently. This is because MTE's tag check adds an extra operation to each memory instruction; for example, a load becomes: load-tag, check-tag, load-value. In SYNC mode, the first two operations must complete before the final operation (the actual load) can commit. This additional work and the ordering constraint require careful modifications to several parts of the pipeline to support efficiently; as we see in §4.4, implementations without careful optimization can add large overheads.

ASYNC mode. In ASYNC mode, the dependency between the load-tag/check-tag and the memory operation (load) is relaxed. The only guarantee the processor provides is that a flag register will be set on a tag mismatch that the operating system can poll. By convention, the operating system checks this flag at the next system call and it immediately generates a fault if set. This convention implies that a mismatch will still be caught before there are externally visible side effects (modulo side channels and shared memory). Even so, ASYNC's lack of precision makes triaging crashes much harder—millions of crashes already go undiagnosed in production systems [29]—thus, it is less useful for bug finding or intrusion detection.

ASYMM mode. ASYMM offers a trade-off between overhead and precision, using SYNC for loads and ASYNC for stores. However, for security, this is less than ideal, as memory corruption is generally caused by stores.

Probabilistic Use. MTE was designed to probabilistically detect memory safety bugs. To enable this, the system allocator assigns a random color (1 of 16) to the memory of each allocation (e.g., a call to malloc()), and returns a pointer whose top byte has the same color as the allocation it refers to.

On every memory access (e.g., a load), the processor checks if the pointer's tag still matches the tag on the accessed memory. If the tag no longer matches, the processor takes

an exception, either synchronously—to stop the access, or asynchronously—to detect it (§2.1). This exception implies that the pointer references the wrong object; either it is out of bounds (a spatial bug), or it is using an object that has been reallocated (a temporal bug). Thus, there is a 15/16 chance that a bug will be caught.

Threat Model. The strength of this randomized tag approach depends on the attacker's model. For example, if an attacker can determine the tag of a particular region of memory, e.g., through a read gadget or a side-channel attack (e.g., Spectre) [18, 55], and they have access to bugs that let them modify pointers, they can combine these elements to forge new pointers that can bypass MTE checks.

However, other applications of MTE (e.g., §5) do not necessarily depend on the secrecy of the tags. We consider analysis of side-channel attacks to be out-of-scope for this work.

3 Contrasting MTE implementations

While the ARM ISA specifies MTE's interface, it leaves several details up to the implementer. We briefly describe two different MTE implementations we analyzed in this paper.

3.1 Google Pixel MTE implementation

The Google Pixel 8 and 9 are based on the Tensor G3 and G4 processors, respectively. Each comes with three different types of cores: a low-power in-order "Little" core (A5x), a faster out-of-order "Big" core (A7x), and the fastest out-of-order "Performance" core (Cortex-X) [3]. All cores support the different modes of MTE: SYNC, ASYNC, and ASYMM. To the best of our knowledge, these cores implement MTE using the reference implementation ARM provides to chip vendors. Consequently, our results likely generalize to other devices that use ARM's reference cores. For example, the Samsung S23 relies on an almost identical configuration of Cortex-X, A7x and A5x cores as that of the Pixel 8.

Tag storage. ARM ISA allows MTE implementations to flexibly choose how they store the tags associated with memory granules. In ARM's reference design, tag storage works by reserving a portion of the physical RAM exclusively for this purpose. Since MTE must store 4-bit tags for each 16 bytes of physical memory, this means that roughly 3% of physical memory (4/128) is reserved upfront for tag storage.

Modified memory accesses and caches. In ARM's implementation, fetching L1 and L2 data cache lines fetches the corresponding tags; i.e., each 64-byte cache line will also pull in 2 bytes of tags into the cache. If a miss occurs, the CPU performs two independent reads from memory to retrieve both the data and its associated tags. Data and tags are stored together in extended cache lines. The need to fetch additional

data (i.e., tags) on a cache miss implies that ARM's design consumes more memory bandwidth when MTE is enabled.

Impact of tag checks on the store pipeline. Tag checks requires an important change in the store pipeline. Typically, stores are buffered in the store buffer (the CPU can asynchronously push these into main memory) and do not require cache lines to be retrieved during execution. However, with MTE SYNC enabled, stores must retrieve and check their associated tags before executing. Since tag fetches are coupled with data fetches, stores operate similarly to loads—fetching both data and tags into the cache during execution. This introduces a potential bottleneck that, if not handled with care, can significantly hinder application's performance (§4.4).

3.2 Ampere's MTE implementation

Ampere's AmpereOne CPUs are server-class ARM-compatible CPUs that offer up to 192 Ampere-1a cores [10]. Ampere designs their own cores; thus, their performance profile is distinctly different from ARM's reference implementation. Ampere cores exclusively support SYNC mode.

Tag storage. Unlike ARM's reference design, Ampere has adopted co-located MTE tags with data across the storage hierarchy. Specifically, tags are stored in bits normally used for error-correction codes (ECC bits) [54] that are only accessible by the memory controller. This is particularly important in server environments where managing terabytes of physical memory is common—ARM's reference implementation, in contrast, would be harder to deploy in this environment, as reserving 3% of physical memory results in a large reduction of usable memory. Using ECC bits to store tags can potentially impose reductions on RAM reliability metrics. In its documentation [54], Ampere discusses an optimization they've used to minimize this reduction while supporting MTE.

Modified memory accesses and caches. Like the ARM reference implementation, Ampere CPUs also fetch data and tags together; these are stored in larger cache lines (extended from 64 bytes to 66 bytes), where the two extra bytes store tags. Unlike the ARM reference implementation, however, Ampere can retrieve data and associated tags with a single query to physical memory upon cache misses. Coherency also requires only a single transaction, compared to the reference implementation which requires two transactions. This means that cache misses do not impose any performance penalties or bandwidth reductions when MTE is enabled.

Impact of tag checks on the store pipeline. The impacts are similar to those in the ARM reference implementation.

3.3 Apple's MTE implementation

Apple added MTE support under the name Memory Integrity Enforcement (MIE) on the M5 chip (Apple Macbook Pro M5) and A19 chip (iPhone 17) [12]. Both chips come with

---

[3]Specifically, **Pixel8**: A510, A715, and X3. **Pixel 9**: A520, A720, and X4.

SPEC CPU2006	Pixel 8 Perf Core			Pixel 8 Big Core			Pixel 8 Little Core			Ampere
	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**

SPEC CPU2017	Ampere SYNC	Apple M5 SYNC
600.perlbench_s	1.02	1.01
602.gcc_s	1.03	1.03
605.mcf_s	1.08	1.00
620.omnetpp_s	1.03	1.12
623.xalancbmk_s	1.14	1.03
625.x264_s	1.00	1.02
631.deepsjeng_s	1.01	1.02
641.leela_s	1.01	1.00
657.xz_s	1.03	1.03
Geomean	**1.03**	**1.02**
502.gcc_r	1.12	1.09
502.gcc_r (input 5)	1.25	1.29

Figure 2: **Performance Cliffs Across MTE Impl. and Modes.** *MTE Overhead on SPEC CPU INT 2006 on Pixel 8 and AmpereOne and SPEC CPU INT 2017 on AmpereOne and Apple M5. Pixel SYNC mode incurs high overhead in some benchmarks. Certain ASYNC benchmarks also incurs high overheads, contradicting general expectations. Ampere and Apple's SYNC mode also shows some overhead spikes, albeit smaller.*

performance and efficiency cores, and like the Ampere implementation support only MTE's SYNC mode. Since these chips were announced shortly before this paper's submission, we only discuss our preliminary analysis of the M5's performance core, and do not analyze micro-architectural details.

4 MTE for Probabilistic Memory Safety

We start by exploring the overhead of MTE for probabilistic heap memory safety with SPEC (§4.2) and server workloads (§4.3). We then explore the microarchitectural sources of these overheads with targeted microbenchmarks (§4.4).

4.1 Experimental setup

Processor/Memory: Pixel 8 (Tensor-G3/12GB), Pixel (Pro) 9 (Tensor-G4/16GB), AmpereOne (A192-32X/512GB), MacBook Pro M5 (M5/32GB). **Operating systems:** Pixel 8-Android 14 (AP2A.240605.024), Pixel 9-Android 15 (AP4A.241205.013), AmpereOne-Fedora 42 / Linux kernel 6.14., MacBook Pro M5-MacOS Tahoe 26.2 (25C56) **Build:** For each benchmark we use the same binary across all devices except Apple M5. Binaries are compiled with Clang 18 and statically linked with glibc-2.36. MTE is enabled in GLIBC's heap allocator via tunables (glibc.mem.tagging) and run on debootstrap [7] environment on Pixel. Apple requires its own binaries, and we used a modified Mimalloc [58] allocator to tag every allocation[4].

Frequency Pinning: Every task is pinned to a specific core. Pixel 8 and 9—80% of maximum allowed frequency to keep down benchmarking times while still providing thermal stability. AmpereOne–we pin the frequency to 2.6GHz for all benchmarks. Apple M5 does not support core/frequency pinning. All MTE modes (SYNC, ASYNC, ASYMM) are evalu-

ated on Pixel, while we evaluate only SYNC on AmpereOne and Apple M5 as they don't support other modes(§3.2).

4.2 Probabilistic memory safety on SPEC CPU

Figure 2 shows the MTE overhead for heap memory safety on SPEC CPU 2006 [50] for the Pixel 8 and Ampere cores. Pixel 9's performance is similar to Pixel 8, and is available in Appendix E(Figure 16). We run SPEC CPU 2017 on AmpereOne and Apple M5 as only these devices have sufficient memory to run this benchmark. The coefficient of variation is typically under 3%. A few programs like libquantum or mcf can have variation up to 5%, so we increase the iterations for these programs to 10 iterations.

Interpretation. We observe substantial variation in performance degradation across benchmarks and cores. Pixel's in-order Little cores show the smallest variation across workloads, with modest slowdowns of 20% at most.

Pixel's out-of-order Big and Performance cores exhibit more variation. The Performance core in particular has up to a 6.64× slowdown on 456.hmmer and 2.37× on 464.h264ref. A similar 5× slowdown was observed on the SPARC M7 ADI's implementation of memory tagging [84], but was not analyzed/explained. Briefly, these slowdowns are attributable to store operations in the Performance core not being allowed to run speculatively. We explore this in detail in §4.4.1.

AmpereOne's SYNC implementation broadly shows low overheads on CPU 2006 with a geomean of 1.10×, with a notable jump for 456.hmmer of 1.43×. Briefly, we are able to attribute this overhead to an implementation gap in its store-to-load forwarding and find that the geomean of 1.10× would significantly reduce if this is fixed. We explore this in more detail in §4.4.2.

Despite MTE ASYNC being the suggested option for production use, it exhibits large overheads that are comparable to those of MTE SYNC when running 403.gcc, 471.omnetpp, and 483.xalanc on the Big core. Briefly, we are able to

---

[4]The default MTE allocator for MacOS is "Xzone Malloc" [11]. We do not use Xzone for our benchmarks as it employs opaque tagging policies and its source code was not publicly available until recently.
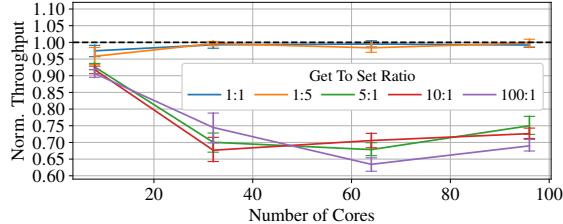
Figure 3: **MTE-enabled Memcached performance on the AmpereOne.** Memcached shows slowdowns of up to $1.40\times$; However, we found this is due to performance problems in the Linux kernel's MTE support, rather than hardware limitations.

attribute the overhead of MTE ASYNC in these three benchmarks to pipeline backend stalls—a behavior that is seen when the tag-checking mechanism is saturated by a mix of random and sequential loads. We explore this in detail in §4.4.3.

SPEC CPU 2017 single-core results (`refspeed`) on AmpereOne and Apple M5 show low overheads—geomean of $1.03\times$ and $1.02\times$ respectively. Multi-core benchmarks (`refrate`) are similar, and are thus omitted from Figure 2 with one notable outlier—when `502.gcc_r` is run on one of SPEC CPU 2017's multi-core inputs (input 5), it slows down by $1.25\times$ on AmpereOne and by $1.29\times$ on Apple M5. We attribute this to overheads of assigning tags in large allocations. We explore this in §4.4.4.
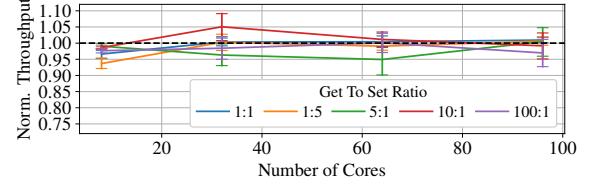
4.3 Server Workloads

We test MTE's probabilistic heap-memory safety performance with four popular server-side programs: RocksDB 1.7.0, Nginx 3.0.1, PostgreSQL 1.15.0, and Memcached 1.2.0 [1, 3, 4, 33] using benchmark configurations from Phoronix's `pts/server` benchmark suite version 10.8.4.[5]
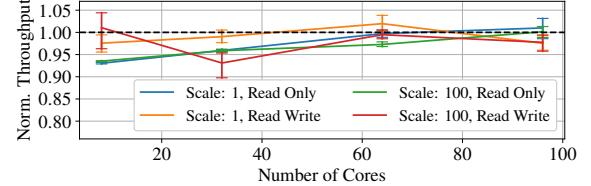
In addition to using `glibc`'s `malloc`, some of these benchmarks also directly allocate memory with `mmap` or use alternative allocators. To ensure all heap data is tagged, we use `strace` [87] to identify these cases, and modified the source to add the `PROT_MTE` flag to all allocators. Error bars in all server benchmark figures represent 95% confidence interval. Additional details on setup are included in Appendix B.

Initial Evaluation. We start by evaluating overheads of applications where servers and workload generators run on the same machine—analogous to deployments where scalable applications made of multiple services runs these services on shared physical hardware. This setup also eliminates network noise and isolates CPU and memory performance highlighting overheads of MTE hardware.
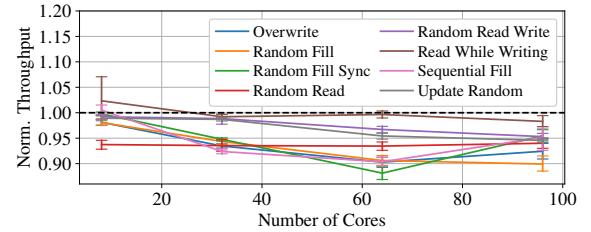
We observe that overheads of RocksDB, Nginx, and PostgreSQL are similar to the overheads in application bench-

---

[5]This version of Phoronix's benchmarks had an error with inconsistent PostgreSQL configuration in its scripts. We fix this error in our tests.
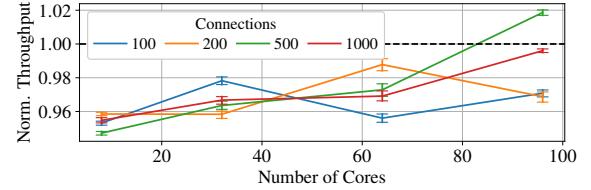

(a) Memcached: querying key-value pairs with diff. get-to-set ratios.

(b) PostgreSQL: SQL queries on databases of different size.

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

(d) Nginx: tests on serving static files.

Figure 4: **MTE server workload performance on AmpereOne after our kernel patch.** MTE overhead are generally under $1.13\times$. Most importantly, our patch successfully addresses the Memcached regression shown in Figure 3, improving the worst case $1.40\times$ to about $1.05\times$ overhead.

marks like SPEC CPU 2006 (§4.2)—typically under $1.18\times$, and negligible in some cases (Appendix Figure 18).

In contrast, Memcached showed a larger performance drops up to $1.40\times$ when running on many cores. In particular, these occur when the Set-to-Get ratio, i.e., the ratio of key-value writes to reads, skews heavily toward reads (Figure 3). Further analysis showed that Memcached's overheads persisted even when MTE was never used by the application, but simply enabled at the kernel-level for the benchmarked process.

Unnecessary Tag Checks in Linux. We narrowed the source of overhead to unnecessary tag checks in the Linux kernel that occur for two reasons. First, the kernel marks all memory in its address space as "taggable" to preserve tag metadata on any user pages mapped in the kernel address space. Second,

the kernel ensures that accesses to kernel pages do not check tags—done by configuring three registers: the Tag Check Fault 0 (`TCF0`) and Tag Check Fault register (`TCF`) registers which toggles tag checks on user and kernel mappings respectively, and the Tag Check Override register (`TCO`) which disables checks on both accesses.

The kernel code incorrectly assumes that any configuration that disables tag check faults via these registers also disables tag checks and thus avoid their overheads; however, ARM MTE's specification is ambiguous about this. While Pixels disable tag checks altogether avoiding overheads, AmpereOne suppresses the faults and thus continue to incur tag check overheads when accessing the untagged kernel memory.

To resolve this, we initially developed a fix that configures these registers differently to avoid overheads (which we rely on for the rest of this section). However, we note that discussions with kernel reviewers [98] yielded a simpler solution that relies on a different configuration register—`TCMA1`—that explicitly disables tag checks for kernel memory accesses avoiding overheads.

Evaluation with the kernel fix. We demonstrate the efficacy of our initial fix in Figure 4. The patch eliminates the performance regression in Memcached, reducing the relative overhead from nearly $1.40\times$ (as seen in Figure 3) to under $1.05\times$. Also, our patch improves the overall performance of other server apps, showing overheads under $1.13\times$. We have submitted this patch to the Linux Kernel upstream [98].

Multi-server setup. To estimate the impact of MTE in a cloud deployment scenario, we additionally conduct multi-server tests using a remote load generator. In this multi-server setup, we observe no statistically significant performance difference between runs with or without MTE for Memcached and Nginx. This result is consistent with numbers reported by Ampere's [54][6]. Applying our patch yields no changes to the overall throughput in the multi-server setup as well.

4.4 µ-architectural causes of MTE slowdowns

We investigate the worst-case slowdowns in Pixel's Performance and Big cores and the AmpereOne core with microbenchmarks and analyze microarchitectural causes.

4.4.1 MTE SYNC slowdown in Pixel's Perf. Core

The Performance cores in both Pixel 8 and Pixel 9 (we report for Pixel 8 X3 core below) exhibit the highest overall overheads when enforcing probabilistic memory safety for `456.hmmer` ($6.64\times$) and `464.h264ref` ($2.37\times$), as shown in Figure 2. Examining the assembly, we find that both benchmarks have a tight loop that includes store operations.

Microbenchmark results. We construct a simple, tight-loop, store-heavy microbenchmark that performs a store to a fixed

address. This simple program exhibits a $7.4\times$ slowdown—similar to the slowdown of `456.hmmer`. We use this microbenchmark to explain the microarchitecture behavior.

Interpretation. We attribute the large slowdown to an implementation choice in the Performance core outlined in Section 4.15 of its optimization guide [13]—speculative stores are disallowed for MTE SYNC. If there are two consecutive stores on tagged memory, the second store instruction can only be executed after the first store's tag check has finished. The implication is that each store acts as a memory barrier (fence).

We confirm this by inserting a memory-store barrier (`DMB ST` instruction) after the store instructions within the loop and observing that: (1) the modified microbenchmark (with MTE disabled) slows down by $6.2\times$, and (2) the modified microbenchmark does not slow down more than the original with MTE SYNC. Hence, the MTE SYNC implementation in this core indeed acts as a store memory barrier.

We note that the SPEC CPU benchmarks and our microbenchmark do not show significant overheads on the Big core (A715), indicating that the limitations on MTE SYNC stores are not fundamental and can potentially be fixed with additional implementation effort.

4.4.2 MTE SYNC slowdown in the Ampere core

The AmpereOne CPU generally imposes lower overheads to enforce probabilistic heap safety; however, `456.hmmer` still incurs an overhead of $1.43\times$ in Figure 2.

To identify the root cause, we run this benchmark using `perf` with both MTE enabled and disabled. We found that enabling MTE resulted in significant decreases in `ld_from_st_fwd`, indicating that the performance drop is related to the implementation of store-to-load forwarding.

To test this theory, we disabled store-to-load forwarding on the AmpereOne using an option in Ampere's BIOS and re-ran the full CPU 2006 suite with and without MTE. This time, we observed negligible overheads (under 1%) for `456.hmmer` when enabling MTE; in fact, the geomean of the entire suite fell from $1.10\times$ to $1.05\times$.

As part of other experiments in this paper (from §5.2.1), we identified an additional smaller benchmark that replicated this pattern with more exaggerated effects: a simple mathematical benchmark, `jacobi-2d` from the PolyBench/C numerical benchmark suite [76], displayed overheads of $8.8\times$ when MTE was enabled[7]. It also showed a drop in `ld_from_st_fwd` in `perf`, and its overhead was also mitigated when store-to-load forwarding was disabled.

Interpretation. Our testing points to a limitation in the store-to-load forwarding logic in the AmpereOne when MTE is enabled. However, from what we know of MTE implementations (§3), there is no reason why this points to a fundamental

---

[6]Ampere was motivated to conduct and release these benchmark numbers after we shared a draft of our paper with them (See Section VIIB in [54]).

[7]When compiled natively as done here, `jacobi-2d` has higher overheads than when compiled with sandboxing compilers in §5.2.1

challenge. Logically, store-to-load forwarding circuits in a CPU can be made MTE-safe by modifying store buffers to track the MTE tag of the stores and checking that any store forwarded has the same tag as the load it is fulfilling.

Thus, it must be the case that AmpereOne either does not implement the logic to carry tags in the store buffer, or the optimization works inconsistently when MTE is enabled. Given that we still observe some successful store-to-load operations in `perf` when MTE is enabled, we believe it is the latter.

We reached out to Ampere to check our interpretation. Ampere confirmed that the AmpereOne CPU has an MTE-related performance issue where MTE tags sometimes interfere with store-to-load forwarding, which caused the overheads we observed in `456.hmmer` and `jacobi-2d`. More interestingly, they confirmed that this was already resolved in their next-generation implementation and will recover the lost performance, validating our hypothesis that this is not a fundamental issue for MTE implementations.

4.4.3 MTE-ASYNC/SYNC slowdown in Pixel's Big Core

Unlike on Pixel's Performance core, several benchmarks such as `403.gcc` (1.82×), `471.omnetpp` (1.54×), and `483.xalancbmk` (1.54×) show significant overhead under both *ASYNC and SYNC* modes on Pixel's Big Core (Figure 2). This is particularly troubling, as MTE ASYNC's goal is to minimize overheads for production use.

We re-ran these workloads using `perf` and found that the overhead stems from increased back-end stalls (consistent with prior observations from Gorter et al. [46]). Further analysis revealed that these stalls are caused by code patterns that involve a mix of randomized and sequential memory accesses[8]. To better understand these hazards and why they cause slow downs, we construct a microbenchmark with a parameterizable version of this pattern.

Microbenchmark construction. Our microbenchmark consists of a link-list of L nodes, where each node has a pointer to a byte-array of size A, resulting in a total of $L \times A$ bytes. The microbenchmark iterates through the linked-list from head to tail. For each visited node, it strides through the node's array and sums up every S-th element. The pseudocode for this microbenchmark is shown in the Appendix Algorithm 1.

Running this microbenchmark with different values of (L, A, S) allows us to test different aspects of the microarchitecture. For example, we can control the ratios of randomized (L) vs. sequential (A/S) loads—this controls metrics such as cache hits/misses, in-flight/pending memory access instructions, etc. Changing A and S also allows us to test the effects of microarchitectural components like the stride prefetcher.

Microbenchmark results. We run this benchmark 10 times after 10 warm-up iterations on the Pixel Big core with MTE ASYNC, and report overheads normalized to when MTE is

---

[8] `403.gcc`'s `reg_is_remote_constant_p` function exhibits this pattern.
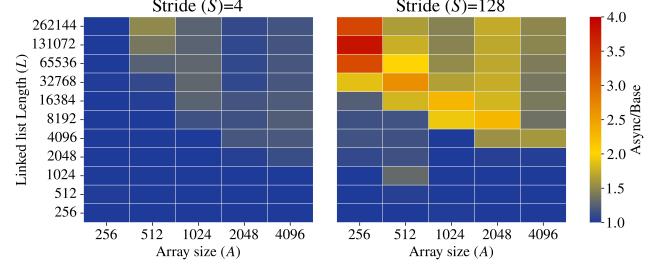
Figure 5: **Uncovering structural hazards due to MTE ASYNC.** *Using a stride access pattern within small arrays stored in a linked list generates cache misses which stress the tag-load/tag-check operations in the Pixel Big Cores data path. This reveals structural hazards that bottleneck loads, slowdowns can be seen in the heatmap—lighter is slower.*

disabled. We repeat the benchmark for a range of values of (L, A, S) and show an interesting sample of these results in Figure 5—namely for a range of values of L and A, when S is set to 4B and 128B.

We observe that: (1) the overhead of MTE ASYNC becomes noticeable when $A \times L \geq 8\text{MiB}$—the size of the Pixel's last level cache,[9] and (2) the maximum overhead is sensitive to the stride S, ranging from 1.5× at $S = 4$ to 3.8× at $S = 128$. As a sanity check, we used `perf` to ensure that the slowdown has the same root cause (increased back-end stalls) as the three slow SPEC CPU benchmarks.

Interpretation. We attribute the slowdowns to structural hazards—bottlenecks in the Pixel Big core's concurrent MTE tag-checking and memory access hardware—and find that the overheads of the three slow SPEC CPU benchmarks are approximated by a combination of overheads at different strides.

Such hazards are possible as MTE support fundamentally requires hardware to perform and track two steps in each memory operation—the tag check and the actual memory access. This means that when MTE is enabled, the ability to execute a memory instruction without stalling, is contingent on the CPU being able to issue a tag check operation (even in MTE's ASYNC mode).

While we cannot identify the exact microarchitectural structure that saturates under tag checking without white-box access to ARM's MTE design, our parameterizable microbenchmark sheds light on the phenomenon.

Concretely, when the size of the link-list exceeds the last-level-cache size ($A \times L \geq 8\text{MiB}$), both the random pointer dereference (one per array) and the subsequent A/S striding accesses to the array likely cause cache misses requiring both data and tag fetches from memory. The resulting accesses fill the microarchitectural structures that track in-flight accesses, causing stalls and degrading performance (in both SYNC and

---

[9] We reverse-engineered the size of system last level cache, which aligns with prior industry tech reports [91]

ASYNC modes). Specifically, we observe two scenarios:

When $S \leq 64$: We see that the maximum overhead is $\approx$ $1.5\times$ and appears only when A is sufficiently large ($\approx 512B$). Our interpretation is that when both A and S are small, the A/S strided accesses are largely covered by cache hits and next-line/stride prefetchers and thus the overhead vanishes.

When $S \geq 128$: We see the maximum overhead jumps to $3.0\times$, especially when A is small (256). Our hypothesis is that at this large stride, there is insufficient chance for prefetching and there are no cache hits. As a result, more pressure is placed on in-flight tracking hardware and prefetchers cannot compensate by working ahead of the demand miss stream.

We believe such structural hazards can be fixed for future implementations. As one data point, we observe that both SPEC and our microbenchmarks don't show significant overheads on Pixel's Performance core in the MTE ASYNC mode. This indicates that such overheads are not fundamental.

4.4.4 Tagging cost on AmpereOne and Apple M5

`502.gcc_r` in Figure 2 shows relative high overhead on AmpereOne and Apple M5. We observe this often allocates large memory regions, thus we hypothesized that this slowdown is due to the cost of instructions for tagging these regions (e.g., `stg`).

To validate our hypothesis, we modified the allocator to selectively tag only allocations smaller than 32KiB. Indeed, several production allocators employ similar policies [11, 63, 74]. For large allocations, these allocators instead rely on guard pages (unmapped pages after an allocation) to prevent some buffer overflows.

The performance difference is marginal for most workloads in SPEC, showing 0-2% improvement in the geomean across the tested cores. However, specific benchmarks show notable improvements for some micro-architectures. For `403.gcc` (SPEC 2006) overhead dropped from $1.23\times$ to $1.15\times$ on AmpereOne and from $1.26\times$ to $1.15\times$ on Pixel 8 Little Core. Similarly, SPEC 2017's `502.gcc_r` benchmark with input 5 (discussed in §4.2) decreases from $1.25\times$ to $1.17\times$ on AmpereOne and from $1.29\times$ to $1.08\times$ on Apple M5.

Interpretation. While the cost of instructions like `stg` are not typically bottlenecks, their overhead is noticeable in specific workloads and microarchitectures for large allocations.

5 Additional Applications of MTE

We analyze MTE's performance in a range of different applications beyond memory-safety to explore its flexibility. In some cases, it provides advantages (our positive results). In other cases, it is equivalent or inferior to existing hardware or software mechanisms (mixed to negative results). Unless noted otherwise, our experimental setup is similar to §4.

5.1 Positive Results: BufLock & Mem. Tracing

MTE can be used as a fast substitute for copying to prevent time-of-check time-of-use attacks. It also offers a memory tracing mechanism with better performance than page protections and better scalability than hardware watchpoints.

5.1.1 TOCTOU protections via revocation (BufLock)

Prior work has observed that MTE offers efficient time-of-check time-of-use (TOCTOU) mitigations by allowing revocation of buffers from untrusted code [24]. We demonstrate this with an example in the Firefox browser. Firefox runs its XML parser in an in-process (SFI) sandbox [69, 93]. Concretely, this means that the XML parser runs in a dedicated memory pool—the sandbox memory—and cannot access other memory even in the presence of memory-safety bugs.

For security, Firefox must copy any data produced by the XML parser out of the sandbox memory prior to use; this ensures a compromised XML parser cannot unexpectedly modify parsed data as a means to attack Firefox, i.e., a TOCTOU attack[10]. However, despite requiring data copies to preserve security, copying all data generated by the XML parser results in a 10% slowdown—something too slow to deploy [21].

The TOCTOU vulnerability here is caused by code reentrancy (rather than threading). Concretely, Firefox repeatedly makes calls to the sandboxed XML parsing library to parse chunks of an XML file as it comes in over the network, storing the parsed data in sandbox memory. Firefox checks the parsed data and assumes that the already parsed XML data will not change during subsequent calls to the XML parsing library; however, a compromised or malicious XML parser can violate this expectation. This would result in a TOCTOU bug in Firefox when the data is next used.

BufLock. MTE offers an alternative to slow data copies [24, 99]. If we assume the XML parser can't use any tagged pointers, we can prevent it from conducting TOCTOU attacks just by tagging these data structures with any tag (BufLock). This is, in effect, a poor man's capability revocation mechanism.

The remaining question: how do we ensure the sandboxed XML parser doesn't use tagged pointers (i.e., pointers with non-zero tag bits)? Fortunately, SFI mechanisms such as WebAssembly [49] used by Firefox [69] guarantee this automatically. SFI works by employing a custom compiler that inserts runtime checks to ensure all dereferenced pointers have a known prefix [93]. This also guarantees that the tag bits of pointers are zero. Additionally, SFI compilers are, by default, configured to disallow any instructions that attackers may find useful—including instructions like `stg` which can change tags on memory, or system calls that disable MTE.

Experimental setup. While our experimental setup is broadly similar to §4, Firefox is compiled as a native Android appli-

---

[10]This is analogous to OS kernels copying system call parameters from user memory to kernel memory prior to using the data.

Cores	Insecure (sec)	Data copies	BufLock
Pixel8 (Little)	1.19	1.28 (1.079×)	1.20 (1.01×)
Pixel8 (Big)	0.31	0.35 (1.118×)	0.32 (1.031×)
Pixel8 (Perf)	0.17	0.19 (1.132×)	0.24 (1.415×)
Pixel9 (Little)	0.86	0.92 (1.071×)	0.89 (1.037×)
Pixel9 (Big)	0.23	0.25 (1.104×)	0.25 (1.085×)
Pixel9 (Perf)	0.13	0.14 (1.103×)	0.18 (1.381×)
AmpereOne	0.23	0.25 (1.097×)	0.24 (1.051×)

Figure 6: **Replacing Copying with MTE for Fast TOCTOU Protection.** *Replacing memcpy with re-tagging (BufLock) can often reduce the cost of TOCTOU protection when sharing data across isolation boundaries. Cases where BufLock reduces these overheads for protected buffers from an XML parser (sandboxed libexpat) in Firefox are shown in green.*

cation using the native `bionic` libc (rather than a Debian application that is run on `debootstrap` with `glibc`). Additionally, to avoid noise, we modified Firefox to pin only its XML parsing thread to an isolated core with a pinned frequency; all other processes/threads run on different cores.

Evaluation. We used BufLock to "lock" the data structures produced by Firefox's sandboxed XML parser and thus prevent TOCTOU attacks. We use the same benchmark used by Firefox engineers to test the performance of the XML parser [21]: measuring the time taken to parse a large SVG image (represented in XML) from Google Docs, tiled 10 times vertically. After ignoring 10 runs, we report the median time to render the SVG image 100 times. Figure 6 shows the performance of three Firefox builds—an insecure version that neither copies data nor uses BufLock to prevent TOCTOU, one that copies data, and one that uses BufLock.

Interpretation. Copying memory and managing extra allocations have visible CPU performance overheads (between 1.037× and 1.415× with the coefficient of variation under 3%); this is consistent with the 1.10× overhead reported by Mozilla engineers [21].

BufLock significantly reduces these overheads on the AmpereOne core as well as the Pixel Big and Little cores, which makes shipping protections for this vulnerability much more practical. However, overhead on the Pixel's Performance core goes up significantly, caused by the limitation of Pixel's Performance core identified in §4.4.1. However, if this bug in the Performance core is fixed, BufLock would offer a meaningfully faster way to secure this operation.

To offer additional insight into why BufLock's tagging approach is faster than copying, we implemented a microbenchmark on these operations in Appendix C.2. We observe that using the optimal mix of tag instructions (`dc gva` to tag entire cache lines, `st2g` and `stg` to tag two and one granules), results in better performance than copying data.

Multi-threading. While BufLock works well in this single-threaded use case, its costs in a multi-threaded program could be more expensive given MTE's current design (§7).

5.1.2 Memory Tracing

Efficiently tracing data access can support interactive debugging [28], data profiling for cache optimization [17], dynamic program analysis [47, 48], and intrusion detection [88]. Unfortunately, software binary-translation approaches such as PIN [64] and DynamoRio [20] are complex and often expensive; hardware supports only a limited number of watchpoints (usually 4 to 8); and using page permissions for tracing is expensive due to the high costs of changing page permissions and the problem of spurious traps due to false sharing.

MTE offers the potential to enable an unlimited number of efficient hardware data watchpoints, and other work has already put this to good use for efficient online data race detection [85]. To explore this capability, we developed two MTE-powered tracing tools: "MTE-tracer", which leverages MTE to trace accesses to data structures from user space, and "MTE-kernel-tracer", which is an additional optimization that pairs the MTE-tracer approach with a custom kernel module for better performance. We also built "Page-tracer", a tracer leveraging page permissions; we use Page-tracer and DynamoRIO [20] as baselines in our evaluation.

Implementing MTE-tracer and Page-tracer. MTE-tracer's basic approach is to simply tag any memory/data structures we want to trace with a chosen tag. Thus, any accesses to these data structures will result in a segmentation fault with an extra flag indicating an MTE tag mismatch.

When a fault occurs, the tracer executes a routine we call `log-step-and-resume`; it logs the access, temporarily untags the data, re-executes the faulting instruction, retags the data, and then resumes normal execution. While one can leverage single-stepping like a standard debugger to perform these steps, the overhead of redundant signals/exceptions and following context switches is prohibitive. To avoid this, MTE-tracer dynamically generates a code snippet for the `log-step-and-resume` sequence and executes that code at every fault instead of every single step.

We also implement Page-tracer, which uses page permissions rather than MTE to provide another baseline (in addition to DynamoRIO). Page-tracer uses essentially the same approach; however, it must (1) use expensive `mprotect` system calls in place of retagging data and (2) check that the segmentation fault was not due to false sharing (i.e., the bytes on the page being accessed are not those being traced).

Implementing MTE-kernel-tracer. The control-flow path of MTE-tracer is unnecessarily complex and expensive as using signal handling for event delivery requires four user-kernel context switches for every event. MTE-kernel-tracer adds a kernel module to optimize these transitions away by directly logging events in memory in the kernel, which user space can periodically drain. The module works by using `kprobes` to bypass the Linux kernel's normal MTE fault handling and runs `log-step-and-resume` instead. After this, execution of the process is resumed. This approach only requires two

(a) RSA Sign   (b) RSA Verify
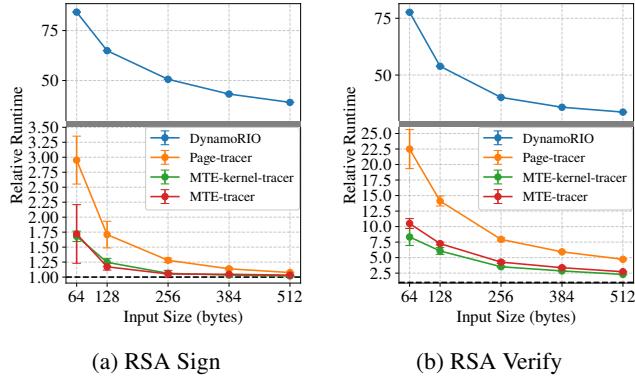
Figure 7: **MTE-based and Page-based data tracing overheads on Pixel 8 Performance Core:** *We compare the performance of tracing key material through OpenSSL using different tracers; the time taken is normalized to the baseline performance with no tracing enabled. We see that the MTE-based tracer is orders of magnitude faster than DynamoRIO and is 2 to 3 times faster than the Page-tracer for small-sized inputs, and the gap decreases as the buffer size increases.*

user-kernel context switches for every event.

Setting up DynamoRIO. We configure DynamoRIO to instrument every memory access to check if the access is in an address range that is being traced. If so, the instrumentation logs the access and continues execution.

Benchmark setup. We evaluate the overheads of memory tracing on OpenSSL using two of its supported performance ("OpenSSL-speed") benchmarks: performance of RSA key signing and RSA key verification. We measure the slowdown due to tracing memory accesses to a variable-sized input buffer—from 64 bytes up to 1024 bytes—containing a digital signature passed to the sign and verify operations. Since the benchmark accesses the buffer linearly, the size of the buffer is directly correlated to the number of traced memory accesses in this program. We add 10 warm-up runs prior to the benchmark to eliminate any one-time costs in benchmark setup.

OpenSSL normally runs these benchmarks for 10 seconds using a timer that issues a `SIGALRM` signal to terminate the benchmark. However, as DynamoRIO support for `SIGALRM` is buggy [19], we modify these benchmarks to run for a fixed amount of iterations instead—1000 for RSA Signing, 10000 for RSA Verification excluding warm-up runs.

Minimizing unrelated costs in baselines. We use DynamoRIO and PageTracer as baselines for evaluation. DynamoRIO's dynamic instrumentation is normally applied on the first run of the program, which could increase overheads on the first run; however, any costs associated with this are excluded because we use warm-up runs. PageTracer would normally encounter a large number of spurious traps due to false sharing; however, we place the input buffer being traced in its own dedicated page so we only consider the overheads

due to the use of `mprotect` calls in the PageTracer.

Evaluation. Figure 7 shows the performance of different tracing approaches on the Pixel 8 performance core, along with 95% confidence intervals. Page-tracer incurs throughput slowdowns of 2.9× and 22.5× for RSA sign and RSA verify for small buffers. In contrast, MTE-tracer is much faster incurring slowdowns of 1.7× and 11×, while MTE-kernel-tracer is faster still and incurs only 1.7× and 8× slowdowns. Performance of the other cores are similar, with minor variations such as the Ampere core being a bit faster; more data is available in Appendix in Figure 19.

5.2 Mixed Results: Sandboxing, CFI

We investigate if ARM MTE can optimize sandbox enforcement [93] and control-flow integrity [6]—properties typically enforced using compiler-inserted runtime checks. Broadly, we see that MTE is not ideal for sandboxing due to high and variable overheads, but may offer reasonable performance depending on the workload and hardware. MTE does not clearly provide meaningful performance benefits for CFI but can offer additional options for applications using JIT compilers.

5.2.1 Enforcing/Optimizing Sandboxing schemes

Prior work [24, 38, 70, 71] has proposed using MTE for enforcing sandboxing/SFI-style isolation. The idea behind these approaches is that a program component may be sandboxed if all of its memory accesses unconditionally use a fixed MTE tag. This guarantees that the isolated component can only access memory tagged with that fixed tag, isolating it from other pages. We use ColorGuard [70, 71]'s implementation of this idea, as the other implementations mentioned above [24, 38] require tagging an application's entire private memory (rather than just the sandbox memory pages), meaning these incur much higher overheads due to MTE (See §6.2).

ColorGuard is an optimization that lowers the virtual memory footprint of existing SFI tools like WebAssembly (Wasm) [49] and Native Client [81]. These SFI tools enforce isolation by inserting runtime bitmasking instructions or bounds-checking instructions to ensure all memory addresses accessed by code remain in a specified memory region—*the sandbox memory* [57, 93]. To optimize these runtime checks, SFI tools also place large regions of unmapped memory called guard regions (4 GiB in size in Wasm and 80 GiB in size in Native Client) after the sandbox memory. This allows SFI tools to elide runtime checks for any memory accesses that can statically be shown to remain within the guard regions.

ColorGuard observes that SFI tools can eliminate guard regions between adjacent sandboxes but continue to leverage the above guard-region optimizations if they ensure that adjacent sandboxes each use a different Memory Colors/Tag. In this case, adjacent sandboxes act like guard regions for each other. The original paper implemented Memory Colors

leveraging the efficient page-coloring of Intel MPK [51]; they also suggested ARM MTE could also be used for ColorGuard but stopped short of testing this claim.

Evaluation. We implemented ColorGuard-MTE in wasm2c [43] Wasm compiler. In our implementation, as with ColorGuard-MPK, each sandbox is tagged with a single color that does not change during the course of its lifetime. The per-sandbox tag is stored as part of the "sandbox memory" address used for runtime checks; this guarantees that the sandbox is restricted to the specified tag.

We evaluated ColorGuard-MTE with PolyBench/C, a numerically heavy benchmark suite [76] that is frequently used to benchmark Wasm SFI compilers [49,67,75]. Each program in PolyBench/C runs 10 iterations, and we report the median of them with normalized minimum and maximum as error bars. Note that, in contrast to probabilistic memory safety, there is no runtime overhead from retagging; all overheads are due to the costs of tag checking.

Interpretation. The results of our evaluation are in Appendix E. While the performance on Pixel's Little and Big core and the Ampere core is competitive for many workloads, the worst-case overheads of the Pixel Little, Big, Performance, and Ampere cores are $1.31\times$, $1.26\times$, $3.65\times$, and $5.6\times$ respectively (Figures 20a and 20b). This is noticeably higher than the typical single-digit overheads we expect from modern ARM SFI tools [100]. This overhead stems from the bottlenecks discussed in Section 4.4; however, such slowdown is exacerbated because Wasm moves stack arrays onto the heap by design (similar to SafeStack [56]).

Consequently, this result is mixed. In general, the runtime overheads imposed by MTE make it a poor replacement for ColorGuard's usual page-level tagging mechanisms [71] that impose no runtime overhead. However, since no current ARM hardware ships with page-level tagging (POE [15]), ColorGuard-MTE may still be a reasonable option.

5.2.2 Enforcing/Optimizing Control-Flow Integrity

Forward and backward edge control flow integrity (CFI [6]) are popular techniques for exploit mitigation with both software [90] and hardware support [52,60,77,79]. Prior work has suggested that tagging could be used to optimize CFI [62,86].

We evaluate if ARM MTE can be used to optimize CFI. We focus on two techniques: first, we optimize software-CFI by using tag checks to eliminate CFI's runtime checks; second, we tag code pointers to protect their integrity [56]. In general, we find that tagging with current MTE hardware does not offer benefits over existing software/hardware approaches.

Backward-edge CFI. We developed two forms of backward-edge CFI. First, we introduce an enhanced shadow stack that isolates the shadow stack using MTE. This is compelling, as current software CFI such as Clang's ShadowCallStack [5] relies only on randomization to protect the shadowstack, leaving it potentially vulnerable to attack. Prior work [22, 92]

leverages MPK to protect the shadow stack on x86 platform, and MTE may be a drop-in replacement on the ARM platform. Second, we tag return addresses on the stack to protects the integrity of return addresses in place. This is attractive as the return address does not have to be duplicated or moved from the stack frame; thus, longjmp in C, or bail-outs in JIT compilers [41] are easier to support.

Evaluation. Our implementation of both schemes in LLVM is provided in Appendix D and evaluated on SPEC CPU 2006.

We observe that while using MTE to isolate shadow stacks is fast, incurring only $1.023\times$ to $1.028\times$ across all cores (Appendix D, Figure 12a), this is comparable performance to existing approaches like PACStack [60]—a technique for backward-edge CFI using code pointer signing and MACs.

We observe that using MTE to preserve the integrity of return pointers on the stack incurs prohibitively expensive overhead on the Pixel big and performance cores of $1.164\times$ and $1.335\times$ respectively (even if they are reasonable on other cores) due to the cost of frequent retagging (Figure 12b).

Forward edge CFI. For forward-edge CFI, we explored replacing software CFI checks (e.g., on virtual tables and indirect jump tables) with MTE tag checks in the hopes of improving performance. Tagging also provides an alternative to placing landing-pad instructions for forward-edge CFI [6] in code (e.g., inserting ARM64's bti instruction at valid jump targets); applications can use tagging to store valid targets as vtables protected from modification by tags.

This is compelling, as ARM64 JIT compilers often place large program constants at small offsets from functions, i.e., in code pages, to enable efficient (PC-relative) addressing. Since attackers can often control these constants, this also opens the door to forging new bti instructions[11].

Evaluation. Our implementation of both schemes in LLVM is provided in Appendix D and evaluated on SPEC CPU 2006. Unfortunately, we observe that MTE's performance penalties cancel out any speedups tagging offers to software-CFI, and the end result is pretty similar performance (Appendix Figure 11). However, tagging-based CFI may still offer some value in JIT environments where its competitive performance with out-of-band landing pads offers benefits.

6 Analyzing prior work on MTE performance

A variety of prior work has attempted to characterize MTE performance. Some [25, 26, 32, 42, 59, 61, 66, 82, 83, 94] used performance analogs—instruction snippets chosen to approximate the costs of MTE. While others [38, 46, 101] used MTE in the Google Pixel 8. Unfortunately, the performance analysis in these papers is often either incomplete or incorrect due to methodological limitations or implementation errors.

---

[11]This is a known problem in V8 [44], and RISC-V's recently introduced branch target instruction (lpad), even supports an optional 20-bit immediate with a hash of the function signature to mitigate such attacks.

6.1 Accuracy of MTE performance analogs

In the absence of real hardware or cycle-accurate simulators with MTE support, emulating MTE's cost with analogs was the only option for a number of years. However, now that hardware is available, we can test the accuracy of these analogs and the conclusions reached using them.

We focus on two analogs for MTE's ASYNC mode introduced by HAKC [66] and SFITag [82], which were subsequently used in most other papers. The HAKC analog uses regular memory store instructions to simulate the costs of tagging instructions like `stg` and assumes tag checks during memory operations impose no additional costs. The SFITag analog also uses regular memory stores to simulate costs of `stg` but additionally augments all memory operations with an extra dummy load to simulate the additional memory bandwidth imposed by fetching tags.

We implement the HAKC analog by modifying `glibc`'s allocation function to use the analog in place of `stg` instructions. We implement the SFITag analog by additionally building an assembly rewriter tool (leveraging tooling infrastructure from LFI [100]) that inserts dummy loads after every memory operation in a given binary.

Evaluation. We test these analogs using the SPEC CPU 2006 benchmarks from §4.2 and compare the overheads from the analogs to actual hardware. The results (and associated error bars) are shown in Appendix Figures 14 and 16.

We find that neither analog accurately tracks real overheads and cannot usefully predict which benchmarks are likely to have higher overheads across microarchitectures. The HAKC analog seems to predict near-zero overheads for all workloads, while the SFITag analog predicts arbitrarily high overheads even when overheads are negligible (e.g., SFITag predicts over a $3.5\times$ higher runtime for `456_hmmer` on Pixel 8's Big core with MTE ASYNC, while the reality is that the overhead on real hardware is negligible).

6.2 Leveraging MTE for fast isolation

Prior work [38] has also investigated the performance of MTE SYNC to speed up WebAssembly sandboxing by measuring slowdowns of PolyBench/C on the three Google Pixel 8 Cores—the same as our measurement setup in §5.2.1. However, due to a bug in their implementation, they incorrectly report extremely good performance with MTE-based isolation in Wasm, reporting speedups of $1.037\times$, $1.051\times$, and $1.339\times$ on Wasm—rather than the large spikes in performance of up to $3.65\times$ that we see in our measurements.

To understand the discrepancy, we investigated their open artifact [39] and discovered two implementation bugs.

Bug 1. The MTE-aware Wasm compiler they implemented (a fork of Wasmtime [23]) did not apply the `PROT_MTE` flag to any pages requested after the start of the application—rather, the flag was only applied to memory allocated at startup.

Bug 2. Several of the initial memory pages of the application contained the applications static data (corresponding to an ELF file's `.data` section); Wasmtime loads this into memory as copy-on-write file-backed memory mappings for efficiency [36]. Unfortunately, such file-backed mappings do not support the `PROT_MTE` flag[12].

As a result of these two bugs, very few memory pages of the sandboxed application actually enforced MTE checks. For instance, their toolchain only tagged 16 of the total 784 memory pages in PolyBenchC's cholesky benchmark correctly with the `PROT_MTE` flag. This would account for the much better performance reported in their paper.

Additional sources of overhead. Even if the above bugs are fixed, this system [38] as well as some of the other systems that leverage MTE for sandboxing [24] are susceptible to under-reporting overheads. To see why, suppose one of these schemes is being relied on to sandbox some part of an application (e.g., to isolate a library). Because these systems exclusively rely on MTE tags to enforce isolation, all pages in a process (and not just in the sandbox) must be mapped with `PROT_MTE` and incur the cost of MTE. Thus, in our example, the cost of sandboxing is not just the MTE overheads imposed on the sandboxed library, but on the entire application as well.

Importantly, this distinction does not apply to the ColorGuard [70, 71] use case evaluated in §5.2.1, which assumes only sandbox memory has the `PROT_MTE` flag and incurs overheads. The reason being, with ColorGuard, the finer-grained MTE isolation is layered on top of coarse-grained SFI isolation, which isolates the rest of the process from the sandboxes.

6.3 Improving performance by reusing tags

Despite some gains from selective tagging for large-allocations (§4.4.4), substantial overhead persists on Pixel 8 Big and Perf core. Prior work from Gorter et.al [46] indicated that one way to recover performance was to reuse tags across re-allocations. Since they tested this approach on MTE ASYNC on the performance core of the Pixel, we check if their approach avoids the overheads of MTE ASYNC we saw on the Big core (§4.4.3). We expect that if our conclusions in §4.4.3 are correct, this would not change overheads much.

We repeated the `gcc` benchmark from Figure 2, but used a modified allocator that tags all memory pages used by the benchmark only once during program initialization. We ran this benchmark on the Pixel's Big core and found that this was able to account for only $1.14\times$ of the $1.82\times$ overheads we see on this benchmark.

Interpretation. Reusing tags offers a concrete performance improvement, consistent with the claims of prior work [46]. However, in the context of the large overheads we see in ASYNC performance on Pixel's Big core, we conclude that

---

[12]Their implementation specifies this flag, but the system call fails as tags on file-backed memory is not supported.

re-tagging is not the dominant source of overhead in these extreme cases such as `403.gcc`.

7 An MTE Wishlist

More precise MTE ASYNC reporting. As discussed in §2.1, ASYNC MTE currently sacrifices precise enforcement and reporting in exchange for relaxed handling of the data dependency between a tag-load/tag-check and a load/store instruction. Given Ampere's results with SYNC, it's not clear that this relaxation is even required for performance. However, in the cases where implementers do opt to support ASYNC, providing precise reporting (i.e., saving the PC of the faulting instruction in a register) would greatly increase the odds that detected bugs will be rapidly diagnosed and fixed, while still providing greater freedom for implementers than SYNC.

Another relatively cheap way to simplify diagnosing the cause of ASYNC traps would be delivering an interrupt (imprecise trap) when a tag mismatch occurs. Even though these could potentially be thousands of cycles later given the size of modern instruction windows, the trap handler would still have a much fresher view of the system state than if the fault is not reported until the next system call, potentially making it easier to identify the faulty loop, function, etc., where the bug is, and offering the possibility of a fresher core dump.

Logging tag mismatches. Our fast memory tracer could be faster and more versatile if tag mismatches were cheaper to record. At present, MTE SYNC traps on every mismatch. A trap per-event is very expensive—including running an interrupt handler and a full pipeline flush, which is increasingly costly on modern out-of-order processors. A new MTE mode where the precise PC and operands of the current instruction could be written to a buffer on a tag mismatch (similar to Intel PT [51]) would significantly accelerate tracing. Per our earlier point, since this only requires precise reporting, it could also be supported with MTE ASYNC.

Revocation in multi-threaded programs. While BufLock offers a sound solution for TOCTOU in re-entrant environments, it would not be safe in multi-threaded programs unless we also incorporate expensive `DMB` fences after tagging. Adding such synchronization barriers obviates any performance benefits vs. copying. This is unfortunate, because revocation is a general idea that is useful in a variety of contexts beyond just BufLock, including in the enforcement of memory safety [99]. Including dedicated instructions for synchronized tag updates in the MTE standard may allow hardware vendors to provide more optimized instructions for this purpose.

8 Conclusion

A deeper understanding of MTE's performance in real hardware is necessary to provide a solid foundation for both future hardware implementations and the software that uses it. As we have seen, there are a variety of implementation pitfalls that can hamstring MTE's performance and limit its deployment. However, this should not be viewed as a recommendation against MTE; rather our analysis offers guidance on avoiding such pitfalls. MTE also offers a variety of new opportunities for performance optimization in software systems. We look forward to the next generation of memory tagging systems that incorporate the lessons of prior generations and hope our analysis contributes to this effort.

Acknowledgments

Ethical Considerations

Our work focuses on understanding and improving the performance of hardware support for a security feature, namely ARM MTE. We performed all experiments on local systems and did not use any systems with personal data. Concretely, we evaluate ARM MTE performance on the Google Pixel 8 and 9, and Ampere's *AmpereOne* chip. They are commercially available [68, 78] devices open for anyone to use.

While ARM MTE is focused on enforcing security, our analysis was limited to analyzing its performance, and did not focus on finding (or find) any security-related bugs that required reporting. Our work didn't involve human subjects.

We reported our performance findings to Ampere and Google, both of whom acknowledged our report, viewed our findings positively, and passed on the report to internal teams. In Ampere's case, we also received a note about this being fixed in the next generation of CPUs which we have included in the paper. Both Google and Ampere are aware that all of our findings will be published at academic conferences and stated they don't have any concerns about this. The following lists entities that are possible stakeholders.

▶ This paper's authors and author affiliated institutions.

▶ All ARM CPU vendors, specifically Google and Ampere.

▶ Any customers of ARM chips interested in MTE, especially from Google or Ampere.

▶ Maintainers of MTE software support in the Linux kernel and libc.

▶ Developers of memory tagging standards on other platforms like RISC-V [80] and x86 [35].

▶ Authors of the most related prior work, whose work we have cited, reproduced, or extended. In particular, authors of the papers.

1. "Sticky tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags" [46]

2. "Cage: Hardware-accelerated safe WebAssembly" [38]

3. "Memory tagging and how it improves C/C++ memory safety" [84]

4. "Segue&Colorguard: Optimizing SFI performance and scalability on modern architectures" [71]

5. "Preventing kernel hacks with HAKC" [66]

6. "Sfitag: Efficient software fault isolation with memory tagging for ARM kernel extensions" [82]

Potential negatives. Since we show that MTE has overheads in some implementations and workloads, our results could be (mis)-interpreted in several scenarios

▶ Hardware vendors may decide not to deploy this security extension in products in the near term. This would mean that an important security extension is not available to customers, reducing the effective security of consumer devices.

▶ Software vendors may delay enabling MTE on their products, which can result in software being insecure and attackable for longer. This may be exacerbated if they incorrectly assume that the worse performance overheads apply to them without actually measuring their software.

▶ Software vendors may also inadvertently assume the opposite—that they have low MTE overheads as shown in some of our workloads and choose to deploy MTE without careful analysis. This would be an issue if their application falls into a case that has high overheads as shown in the paper or is a usecase we have not measured.

▶ Users may chose to disable MTE on their devices at the system level as they worry about overheads.

The net effect would nevertheless mean that an important security extension is not available to customers, reducing the effective security of consumer devices.

Mitigations. While the above scenarios are possible, we have taken great care to specify which workloads are affected as well as the context in which these results apply to minimize probability of misinterpretation. We have also noted this directly in our conclusion. Our research, especially our observations on how to fix performance cliffs, methodology on how to evaluate performance for the next generation of MTE chips, provides a valuable resource as it outlines a path to address performance issues. The positive outcome of this research would be hardware and software vendors working together to increase adoptability of this important security technology.

We believe our work clarifies the state of MTE, as well as the path forward to improve performance (and thus adoption and security of end user devices), by offering a clearer understanding of the costs and benefits. On balance, we believe the net benefit to hardware vendors and software applications outweigh the potential but low-probability risks of misinterpretation or slowing down adoption in the short term.

Open Science

As part of our commitment to transparency and reproducibility of our work, we have released all tools developed, source code for the tools, build scripts, benchmark scripts and results under an open-source license. All artifacts are available at `https://github.com/UT-Security/mte-root` and 10.5281/zenodo.17953065.

Our artifact is divided into the following directories:

▶ `mte-playground`: Contains the scripts, data, and config files required for the SPEC benchmarks (Sections 4 and 6; Figures 2, 14, 15, 16 and 17).

▶ `mte-root`: Contains data from BufLock, MemTrace, server benchmarks, and all microbenchmarks discussed in the paper.

▶ `mte-server`: Includes the custom Phoronix test suite and profiles used for MTE server benchmarks (Section 4.3; Figures 3, 4 and 18).

▶ `mte-bench-clean`, `mte-performance`, `mte_benchmarks`: These directories contain the MTE ASYNC microbenchmark (Section 4.4.3; Figure 5) and additional evaluative microbenchmarks (Appendix C; Figures 8, 9 and 10), respectively.

▶ `mte-polybench`: Contains the ColorGuard-MTE implementation alongside the Polybench/C benchmark (Section 5.2.1, Figure 20).

▶ `firefox-mte`: Contains BufLock Implementation, the TOCTOU protections for Firefox (Section 5.1.1; Figure 6).

▶ `mtetrap`: Contains MemTrace Implementation, the fast data tracing tool with MTE (Section 5.1.2; Figures 7 and 19).

▶ `llvm-mte`, `mte-glibc`: These contain the TagCFI implementation and the custom glibc. The glibc is used for analogs as well (Sections 5.2.2 and 6 and appendix D; Figures 11, 12 and 13).

References

[1] memcached. `https://memcached.org/`.

[2] memtier_benchmark: A high-throughput benchmarking tool for redis & memcached. `https://redis.io/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/`.

[3] nginx. `https://nginx.org/index.html`.

[4] PostgreSQL. `https://www.postgresql.org/`.

[5] ShadowCallStack — Clang documentation. `https://clang.llvm.org/docs/ShadowCallStack.html`.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM TISSEC*, 2009.

[7] J. Agnel. adeb: A debian-based shell environment designed for android and adb. https://github.com/joelagnel/adeb, Feb. 2023.

[8] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, et al. M7: Oracle's next-generation sparc processor. *IEEE Micro*, 2015.

[9] Amazon Web Services. AWS graviton processor family. https://en.wikipedia.org/wiki/AWS_Graviton, 2025.

[10] Ampere Computing. AmpereOne® Product Brief. https://amperecomputing.com/briefs/ampereone-family-product-brief, 2024.

[11] Apple. Xzone malloc. https://github.com/apple-oss-distributions/libmalloc/blob/libmalloc-792.41.1/doc/xzone_malloc.md.

[12] Apple SEAR. Memory Integrity Enforcement: A complete vision for memory safety in Apple devices. https://security.apple.com/blog/memory-integrity-enforcement/.

[13] ARM. Arm Cortex-X3 Core Software Optimization Guide. https://developer.arm.com/documentation/pjdoc466751330-590747/latest/.

[14] ARM. Armv8.5-a memory tagging extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[15] ARM. Permission indirection and permission overlay extensions. https://developer.arm.com/documentation/102376/latest/Permission-indirection-and-permission-overlay-extensions.

[16] Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/, 2025.

[17] H. Brais, R. Kalayappan, and P. R. Panda. A survey of cache simulators. *ACM CSUR*, 2020.

[18] M. Brand. MTE As Implemented, Part 1: Implementation Testing. https://googleprojectzero.blogspot.com/2023/08/mte-as-implemented-part-1.html.

[19] D. Bruening. CRASH: SIGSEGV on sigreturn under QEMU for AArch64-on-x86. https://github.com/DynamoRIO/dynamorio/issues/7371, Mar. 2025.

[20] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*. IEEE, 2003.

[21] Sandbox libexpat using rlbox. https://bugzilla.mozilla.org/show_bug.cgi?id=1688452#c37, 2021.

[22] N. Burow, X. Zhang, and M. Payer. Sok: Shining light on shadow stacks. In *IEEE S&P*, 2019.

[23] Bytecode Alliance. Wasmtime: A standalone runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime, 2021.

[24] X. Chen, Z. Li, T. Jain, V. Narayanan, and A. Burtsev. Limitations and opportunities of modern hardware isolation mechanisms. In *USENIX ATC*, 2024.

[25] X. Chen, Y. Shi, Z. Jiang, Y. Li, R. Wang, H. Duan, H. Wang, and C. Zhang. MTSan: A Feasible and Practical Memory Sanitizer for Fuzzing COTS Binaries. In *USENIX Security*, 2023.

[26] Y.-C. Chen and S.-W. Li. Hemate: Enhancing heap security through isolating primitive types with arm memory tagging extension. In *ARES*, 2024.

[27] Citizen Lab. NSO Group / Q Cyber Technologies Over One Hundred New Abuse Cases. https://citizenlab.ca/2019/10/nso-q-cyber-technologies-100-new-abuse-cases/, 2019.

[28] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun. REPT: Reverse debugging of failures in deployed software. In *OSDI*, 2018.

[29] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *ICSE*, 2016.

[30] Cybersecurity, US and others. The case for memory safe roadmaps: Why both c-suite executives and technical experts need to take memory safe coding seriously, 2023.

[31] A. A. De Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *IEEE S&P*, 2015.

[32] K. Dinh Duy, K. Cho, T. Noh, and H. Lee. Capacity: Cryptographically-enforced in-process capabilities for modern arm architectures. In *CCS*, 2023.

[33] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage*, 2021.

[34] M. Dowd. Inside the zero day market. https://no complexity.com/wp-content/uploads/2024/06/ bluehat2023-mdowd-final.pdf, 2023.

[35] A. Evangelista. ChkTag: x86 Memory Safety. https://community.intel.com/t5/Blogs/Tech-Innovation/open-intel/ChkTag-x86-Memory-Safety/post/1721490.

[36] C. Fallin. Wasmtime 1.0: A look at performance. https://bytecodealliance.org/articles/wasmtime-10-performance, 2022.

[37] E. A. Feustel. On the advantages of tagged architecture. *IEEE Transactions on Computers*, 1973.

[38] M. Fink, D. Stavrakakis, D. Sprokholt, S. Chakraborty, J.-E. Ekberg, and P. Bhatotia. Cage: Hardware-accelerated safe webassembly. In *CGO*, 2025.

[39] M. Fink, D. Stavrakakis, D. Sprokholt, S. Chakraborty, J.-E. Ekberg, and P. Bhatotia. Cage: Hardware-accelerated safe webassembly - artifact. https://doi.org/10.5281/zenodo.13772996, 2025.

[40] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *ACSAC*, 2014.

[41] George Neis, et al. Deoptimization and Backward-Edge CFI. https://docs.google.com/document/d/1N-D3Ljk8KdiLiDa9JTMiMAOzH_GfT1OG1UuKFlXoUO8/edit?tab=t.0#heading=h.vs15z08v62wy, 2021.

[42] Y. Giannaris. *Securing Operating Systems using Hardware-Enforced Compartmentalization*. PhD thesis, MIT, 2021.

[43] Github. wasm2c. https://github.com/WebAssembly/wabt/tree/main/wasm2c.

[44] Google. V8 CFI Plans. https://docs.google.com/document/d/1O2jwK4dxI3nRcOJuPYkonhTkNQfbmwdvxQMyXgeaRHo, 2022.

[45] Google Cloud. Introducing Google's new Arm-based CPU: Axion. https://cloud.google.com/blog/products/compute/introducing-googles-new-arm-based-cpu, 2024.

[46] F. Gorter, T. Kroes, H. Bos, and C. Giuffrida. Sticky tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags. In *IEEE S&P*, 2024.

[47] A. Gosain and G. Sharma. A survey of dynamic program analysis techniques and tools. In *FICTA*, 2015.

[48] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin. A case for unlimited watchpoints. *ACM SIGPLAN Notices*, 2012.

[49] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *PLDI*, 2017.

[50] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[51] Intel® 64 and IA-32 architectures software developer's manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2023.

[52] intel.com. A technical look at Intel's Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[53] S. Jero, N. Burow, B. Ward, R. Skowyra, R. Khazan, H. Shrobe, and H. Okhravi. Tag: Tagged architecture guide. *ACM Computing Surveys*, 2022.

[54] S. Kaushik, M. Madhav, N. Aboulenein, J. Bessette, S. Brahmadathan, B. Chaffin, M. Erler, S. Jourdan, T. Maciukenas, R. Masti, J. Perry, M. Sutera, S. Tetrick, B. Toll, D. Turley, C. Worth, and A. Bajwa. Optimized Memory Tagging on AmpereOne Processors. *arXiv:2511.17773*, 2025.

[55] J. Kim, J. Park, S. Roh, J. Chung, Y. Lee, T. Kim, and B. Lee. TikTag: Breaking ARM's memory tagging extension with speculative execution. *arXiv:2406.08719*, 2024.

[56] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer integrity. In *OSDI*, 2014.

[57] H. Lefeuvre, N. Dautenhahn, D. Chisnall, and P. Olivier. Sok: Software compartmentalization. In *IEEE S&P*, 2025.

[58] D. Leijen. Mimalloc. https://github.com/microsoft/mimalloc, 2020.

[59] H. Liljestrand, C. Chinea, R. Denis-Courmont, J.-E. Ekberg, and N. Asokan. Color my world: Deterministic tagging for memory safety. *arXiv:2204.03781*, 2022.

[60] H. Liljestrand, T. Nyman, J.-E. Ekberg, and N. Asokan. Authenticated call stack. In *DAC*, 2019.

[61] S. Y. Lim, T. Prasad, X. Han, and T. Pasquier. Safebpf: Hardware-assisted defense-in-depth for ebpf kernel extensions. In *CCSW*, 2024.

[62] Z. Liu, Y. Rong, C. Li, W. Tan, Y. Li, X. Han, S. Yang, and C. Zhang. Cctag: Configurable and combinable tagged architecture. In *NDSS*, 2025.

[63] llvm.org. Scudo hardened allocator. https://llvm.org/docs/ScudoHardenedAllocator.html.

[64] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN notices*, 2005.

[65] lwn.net. Shadow stacks for 64-bit arm systems. https://lwn.net/Articles/940403/.

[66] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow. Preventing kernel hacks with hakcs. In *NDSS*, 2022.

[67] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni. Twine: An embedded trusted runtime for WebAssembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021.

[68] T. P. Morgan. "Polaris" AmpereOne M Arm CPUs Sighted In Oracle A4 Instances. https://www.nextplatform.com/2025/10/20/polaris-ampereone-m-arm-cpus-sighted-in-oracle-a4-instances/, Oct 2025.

[69] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the firefox renderer. In *USENIX Security*, 2020.

[70] S. Narayan, T. Garfinkel, E. Johnson, D. Thien, J. Rudek, M. LeMay, A. Vahldiek-Oberwagner, D. Tullsen, , and D. Stefan. Segue & ColorGuard: Optimizing SFI Performance and Scalability on Modern x86. In *PLAS*, 2022.

[71] S. Narayan, T. Garfinkel, E. Johnson, Z. Yedidia, Y. Wang, A. Brown, A. Vahldiek-Oberwagner, M. LeMay, W. Huang, X. Wang, et al. Segue & colorguard: Optimizing sfi performance and scalability on modern architectures. In *ASPLOS*, 2025.

[72] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack. *POMACS*, 2018.

[73] Operation Zero. Zero-day exploit prices. https://opzero.ru/en/prices/, 2025.

[74] A. Partap and D. Boneh. Memory tagging: A memory efficient design. *arXiv:2209.00307*, 2022.

[75] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova. ewasm: Practical software fault isolation for reliable embedded devices. *IEEE TCAD*, 2020.

[76] PolyBench/C: the Polyhedral Benchmark suite. https://www.cs.colostate.edu/~pouchet/software/polybench/.

[77] Qualcomm. Pointer authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[78] B. Rakowski. Meet Pixel 8 and Pixel 8 Pro, our newest phones. https://blog.google/products/pixel/google-pixel-8-pro/, Oct 2023.

[79] RISC-V Foundation. RISC-V Control-Flow Integrity (CFI) Specification. https://github.com/riscv/riscv-isa-manual/blob/main/src/unpriv-cfi.adoc, 2025.

[80] RISC-V Foundation. RISC-V Memory Tagging. https://github.com/riscv/riscv-memory-tagging, 2025.

[81] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security*, 2010.

[82] J. Seo, J. You, Y. Cho, Y. Cho, D. Kwon, and Y. Paek. Sfitag: Efficient software fault isolation with memory tagging for ARM kernel extensions. In *CCS*, 2023.

[83] J. Seo, J. You, D. Kwon, Y. Cho, and Y. Paek. Zometag: Zone-based memory tagging for fast, deterministic detection of spatial memory violations on arm. *IEEE TIFS*, 2023.

[84] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov. Memory tagging and how it improves C/C++ memory safety. *arXiv:1802.09517*, 2018.

[85] J. Shastri, X. Wang, B. A. Shivakumar, F. Verbeek, and B. Ravindran. Hmtrace: Hardware-assisted memory-tagging based dynamic data race detection. *arXiv:2404.19139*, 2024.

[86] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *IEEE S&P*, 2016.

[87] strace: Linux syscall tracer. https://strace.io/.

[88] S. Suneja, C. Isci, E. De Lara, and V. Bala. Exploring vm introspection: Techniques and trade-offs. *ACM SIGPLAN Notices*, 2015.

[89] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE S&P*, 2013.

[90] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In *USENIX Security*, 2014.

[91] R. Triggs. Google Tensor G3: Everything you need to know about the Pixel 8 processor. `https://www.androidauthority.com/google-tensor-g3-explained-3324692`, May 2024.

[92] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert. You shall not (by)pass! practical, secure, and fast pku-based sandboxing. In *EuroSys*, 2022.

[93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[94] Y. Wang, A. Li, J. Wang, S. Baruah, and N. Zhang. Opportunistic data flow integrity for real-time cyber-physical systems using worst case execution time reservation. In *USENIX Security*, 2024.

[95] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, et al. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE S&P*, 2015.

[96] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to RISC-V. In *NDSS*, 2019.

[97] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *ASPLOS*, 2002.

[98] C. Worth. arm64: mte: Unify kernel MTE policy and manipulation of TCO. `https://lore.kernel.org/linux-arm-kernel/20251030-mte-tighten-tco-v1-0-88c92e7529d9@os.amperecomputing.com/`, 2025.

[99] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. Watson, et al. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *MICRO*, 2019.

[100] Z. Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In *ASPLOS*, 2024.

[101] J. You, J. Seo, K. Lee, Y. Cho, and Y. Paek. BASTAG: Byte-level access control on shared memory using arm memory tagging extension. In *CCS*, 2025.

Appendix

A Detailed algorithm for identifying structural hazards

In §4.4.3, we analyzed the root cause of the performance drop of MTE ASYNC in the Big core. The algorithm for the microbenchmark used is presented in Algorithm 1.

Algorithm 1: Algorithm for microbenchmark to detect hazards used in §4.4.3.

input : Array-Length, LL-Length, Stride

1   L = LL-Length
2   A = Array-Length
3   S = Stride
4   Construct a Linked-List ll of size L
5   **for** *Element el $\in$ ll* **do**
6   el.array = Malloc(A)
7   fill-with-random-values(el.array)
8   Evict ll from cache
9   Head = ll.head
10   Time the while loop below after warming it up
11   **while** *Head $\neq \perp$* **do**
12   Sum = 0
13   **for** $i = 1$, $i < A$, $i+S$ **do**
14   Sum = Sum + Head.array[i]
15   Head = Head.next

B Server benchmark setup

In §4.3, we evaluate performance of server workloads: Nginx, RocksDB, PostgreSQL, and Memcached. We evaluate performance when MTE is enabled, normalized against when MTE is disabled. These benchmarks are repeated 4 times, and is allowed to use a different number of cores each time: 8, 32, 64, and 96 cores. Following Phoronix, each of benchmark is iterated a minimum of 3 times, and is repeated until stable up to a maximum of 15 times.

Nginx. Performance of the Nginx web server serving static files is measured for different number of incoming connections, where Nginx is allowed to automatically choose the number of worker threads.

RocksDB. Performance of the RocksDB database is measured by the throughput seven different database access patterns (e.g., random read, sequential fill etc.)

Instruction	X3	A715	A510	X4	A720	A520	AmpereOne
Clock speed (GHz)	2.35	1.94	1.41	2.49	2.12	1.68	2.59
Loads							
ldr	2.98	2.64	1.01	2.99	2.38	2.00	2.00
ldr*	2.98	2.64	1.03	2.99	2.38	1.99	2.00
ldr +async	3.00	2.63	0.92	2.99	2.38	1.99	2.00
ldr*+async	3.00	2.64	1.02	2.99	2.38	2.00	2.00
ldr +sync	2.98	2.63	0.88	2.99	2.38	2.00	2.00
ldr*+sync	2.98	2.63	0.94	2.99	2.39	1.99	2.00
Stores							
str	1.98	1.83	1.00	1.99	1.74	1.00	2.00
str*	1.98	1.85	1.00	1.98	1.74	1.00	2.00
str +async	1.99	1.81	1.00	1.99	1.73	1.00	2.00
str*+async	1.99	1.81	1.00	1.99	1.72	1.00	2.00
str +sync	0.14	1.57	0.45	0.14	1.60	0.65	2.00
str*+sync	1.97	1.84	1.00	1.98	1.75	1.00	2.00
Tag Ops							
ldg	2.91	1.90	0.94	3.00	1.93	0.84	1.95
stg	1.00	1.82	1.00	1.00	1.76	1.00	1.00
st2g	1.00	1.83	0.46	1.00	1.77	0.50	0.50
stzg	1.00	1.84	0.98	1.00	1.76	1.00	1.00
stz2g	0.33	1.84	0.45	0.33	1.75	0.50	0.50
stgp	1.00	1.68	0.98	1.00	1.55	1.00	1.00
dcgva	0.13	0.14	0.19	0.14	0.14	0.03	1.00

* Instruction accesses a memory page allocated without `PROT_MTE`

Figure 8: Instruction throughput measurements across cores.

PostgreSQL. Performance of the PostgreSQL database is measured with `pgbench`, which can test the database for different sizes of the database, different number of clients, worker threads, for both read-only and read-write SQL workloads. Additionally, the maximum number of connections is limited to 6000. Shared memory buffer size is set to 8GB.
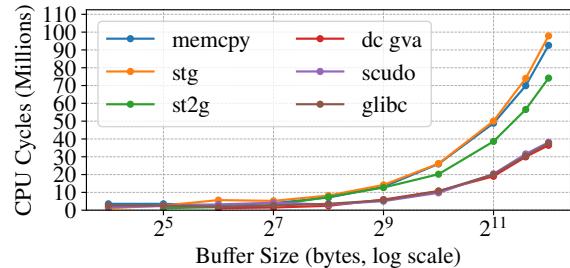
Memcached. Performance of the Memcached in-memory key-value store is measured by evaluating performance on a workload generated by Memtier [2]. Performance is measured for different values of Memcached threads, Memtier threads (where each thread has 50 clients making requests), for different ratios of set and get operations. Additionally, the maximum number of connections is limited to 4096.

C Additional microbenchmarks
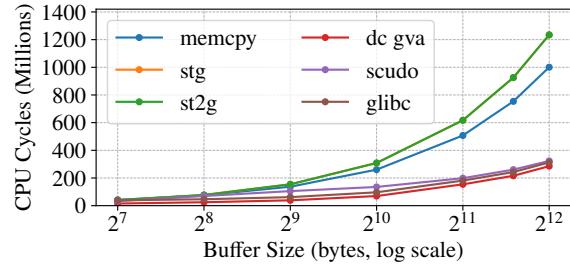
We executed a number of microbenchmarks to understand the performance of MTE hardware. While these were not used to directly explain the overheads we saw in different benchmarks, they were informative to our understanding of the implementations.

C.1 Instruction cycles

We measured the performance of individual instructions in Figure 8 to help us identify the sources of different overheads.

(a) Pixel 8 Little Core

(b) AmpereOne

Figure 9: The average number of CPU cycles to tag a given buffer with different tagging primitives on the Little core of Pixel 8 (performance is very similar on other Pixel cores) and AmpereOne. We see that simply using tagging primitives in a loop is slow (as a reference, it is slower than a `memcpy`). However, glibc's and scudo algorithms that use a mix of these operations in unrolled loops significantly improve performance.

C.2 Overheads of tagging operations

For the use cases we consider such as BufLock (§5.1.1), we need to evaluate the performance of tagging primitives (like `stg` which tags 16 bytes of memory) vs. the performance of `memcpy` (since BufLock achieves its speedups by substituting tagging for data copies). Fink et. al [38] also ran a similar test to check the performance of different tagging primitives and compared this to the performance of `memset`, however, this doesn't quite help us understand BufLock.

Evaluation. We run a microbenchmark that allocates a buffer and measures the time taken to tag the buffer with each of these primitives 1000 times; we also flush the data cache on each iteration for consistent results. We evaluate the MTE tagging primitives for different sizes: `stg` (16 bytes), `st2g` (32 bytes), `dcgva` (cache-line, i.e., 64-bytes)[13]. We compare this to the tagging approaches used in glibc and Android's Scudo libc; both libraries use a complex combination of `stg`, `st2g`, `dcgva` to efficiently tag buffers of different sizes. Finally, we also compare the performance of raw `memcpy` of data, rather than tagging it.

---

[13] MTE also provides `stgm` (variable size tagging), but this cannot be used by userspace applications, so we don't benchmark this

Interpretation. We find that naively using one of the tagging primitives in a loop does not give optimal performance to tag a buffer of a given size; indeed it is slower than a simple `memcpy` of the same buffer. This implies that BufLock §5.1.1 would be a pessimization if we tagged data with one of these primitives in a loop—something we tested and verified. Instead, for performant tagging, we should use the more careful combination of tagging instructions from glibc or Scudo, which are able to outperform `memcpy` on the buffer. Detailed data is available in the Appendix Figure 9.

C.3 Performance of different memory-access patterns

We also constructed three microbenchmarks that simulate common memory access patterns: (1) read after read (RAR), (2) write after write (WAW), and (3) read after write (RAW).

Setup. All benchmarks were statically compiled using aarch64-linux-gnu-g++ 13.3.0 (GCC). We enabled MTE on all mmap-allocated memory regions by specifying the `PROT_MTE` flag. We report performance results as the relative slowdown of MTE SYNC and MTE ASYNC modes compared to the baseline execution without MTE tagging enabled. Our hardware platform and setup selection is the same as that of Section 4.1.

Microbenchmark. Our microbenchmark first uses `mmap` to allocate a 16 MiB `uint64_t` main buffer, which has element count equals to $len = 16 * 1024 * 1024/8$. The buffer is mapped under three configurations: without MTE, with MTE ASYNC, and with MTE SYNC. We initialize the buffer with a pointer-chasing pattern, where each `uint64_t` element stores the address of another element within the buffer. This construction ensures that starting from the head element, it is possible to traverse the entire buffer by repeatedly following the stored addresses. We then allocate an additional `uint64_t` index buffer (without MTE) and initialize each element with a randomly selected location in the main buffer. By iterating over buffer[index[i]] with i from 0 to `len`, the benchmark accesses all elements of the main buffer in a randomized order.

- ▶ RAR: In this microbenchmark, we begin from the head element of the main buffer and iteratively dereference each visited `uint64_t` element. The benchmark features the pointer-chasing memory access pattern and issues a sequence of dependent memory reads, where each access relies on the address obtained from the previous one.

- ▶ WAW: In this microbenchmark, we iterate over buffer[index[i]] with i from 0 to `len` and perform write to each buffer[index[i]]. The benchmark issues a sequence of independent memory writes, with each write targeting a random location in the buffer.

- ▶ RAW: In this microbenchmark, we iterate over buffer[index[i]] with i from 0 to `len` and, for each

	RAR	WAW	RAW
Pixel 9 Little (ASYNC)	1.73×	1.00×	4.00×
Pixel 9 Big (ASYNC)	1.17×	1.00×	1.00×
Pixel 9 Performance (ASYNC)	1.25×	1.20×	1.25×
Pixel 8 Little (ASYNC)	1.16×	1.01×	4.90×
Pixel 8 Big (ASYNC)	1.12×	1.16×	1.02×
Pixel 8 Performance (ASYNC)	1.19×	1.00×	1.00×
Pixel 9 Little (SYNC)	1.76×	15.5×	14.0×
Pixel 9 Big (SYNC)	1.16×	1.30×	1.50×
Pixel 9 Performance (SYNC)	1.25×	2.80×	1.75×
Pixel 8 Little (SYNC)	1.20×	4.61×	5.18×
Pixel 8 Big (SYNC)	1.13×	1.67×	1.65×
Pixel 8 Performance (SYNC)	1.19×	2.17×	1.15×
Ampere (SYNC)	1.02×	1.80×	1.50×

Figure 10: Relative performance overhead of Microbenchmarks

step, load a value from buffer[index[i]] and store it into buffer[index[$i + 1$]]. This access pattern implements store-to-load forwarding, where each memory write directly depends on the result of the prior memory read, and all memory accesses in buffer are to random locations.

Results. For each microbenchmark running each testing device, we compute and report the ratio of MTE SYNC over baseline and MTE-ASYNC over baseline in Figure 10. Across all devices, RAR incurs less than 2× overhead. However, WAW incurs significant overhead across almost all devices when running in SYNC mode, and RAW incurs significant overhead on Pixel Little core.
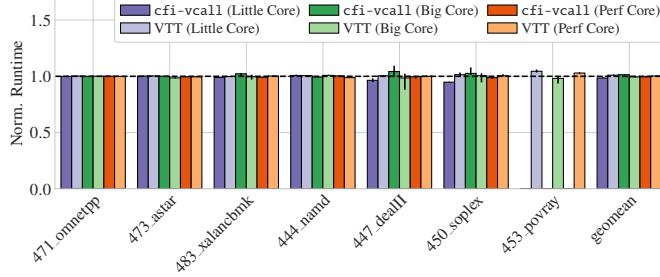
D Using MTE to speed up CFI enforcement

In §5.2.2, we evaluated the performance benefits of using MTE to speed up CFI. Our conclusion was that the performance of MTE-optimized CFI did not offer significant benefits over software CFI. We will provide more details on our MTE-enabled CFI schemes and their associated overheads.
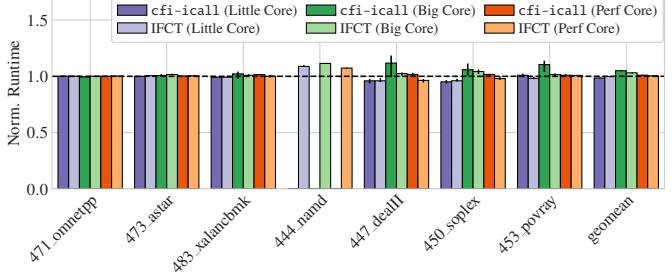
D.1 Forward-Edge CFI (VTT and IFCT)

Control-flow Integrity (CFI) restricts control flow to legitimate targets of indirect branches determined at compile time to mitigate code reuse attacks. As software CFI can be costly, processors have introduced branch target instructions for compilers to insert at valid indirect branch targets, such as x86-64's (endbr64) [52] and ARM64's (`bti`) [16].

For forward-edge CFI, our approach called Tag-CFI, uses *virtual table tagging* (VTT), to protect C++ virtual functions, and *indirect function call tagging* (IFCT), to protect indirect function calls.

(a) Protections for virtual table calls (VTT)

(b) Protections for indirect function calls (IFCT)

Figure 11: MTE-based and Clang's Software-based Forward-edge CFI schemes overheads: We compare the performance of TagCFI overheads on all C++ programs among SPEC CPU 2006 benchmark. TagCFI Performance numbers are normalized to the non-LTO baseline performance without any instrumentation and Clang CFI results are normalized to the LTO baseline performance without any instrumentation.

Virtual Table Tagging (VTT). Virtual tables (VTables) are a common target of memory safety exploits in C++ programs [40]. Since attackers cannot modify VTables directly (they are stored in read-only memory), they instead exploit bugs to corrupt pointers to VTables in C++ objects.

To prevent these attacks, our scheme starts by: (1) reserving a tag exclusively for VTable pointers; (2) tagging all VTable memory with this tag; (3) masking the appropriate tag into legitimate pointers to VTables prior to access. Thus, pointers are restricted to accessing legitimate VTables.

To add finer-grained CFI to this, we: (1) store function equivalence class labels [6]—a unique integer for each function signature—in the VTables next to the function pointers; (2) emit instrumentation before invoking virtual functions.

Finally, to prevent attacks that rely on misaligning the VTable pointer (e.g., incrementing it by 1) to read an incorrect FEC label as code, we force VTable pointers to be 16-byte aligned prior to label checks.

Indirect Function Call Tagging (IFCT). IFCT offers similar protections for indirect function calls (calls through function pointers) with the following steps: (1) IFCT modifies code generation to create a "jump table"—with the same function addresses and interleaved 4-byte labels format as VTT—populated with the address of any functions in the binary that have their address taken at any point in the code. (2) Any function pointer is replaced with an indexing address within this jump table, and any function pointer invocation uses this index to first retrieve the function address—with label and alignment checks similar to VTT—from the table.

Evaluation. We evaluated VTT and IFCT using C++ benchmarks in SPEC CPU 2006 [50]. We found that VTT shows that the Little core performance is slightly worse than the others, showing overhead of $1.05\times$ and $1.013\times$ for the worst performance and geomean, respectively. The results from the Big and Performance core stay within a geomean of $1.006\times$ as shown in Figure 11a. For IFCT in Figure 11b, its geomean

overhead is under $1.04\times$ on all cores; the worst-case overheads are $1.08\times$ –$1.12\times$ on povray. Additionally, we compare our scheme to Clang's CFI implementation where programs are compiled with `-fsanitnize=cfi-icall,cfi-vcall` options. Clang's CFI unfortunately introduces crashes in some programs in SPEC CPU 2006, so we exclude their performance from the geomeans for accurate comparison. We see that Tag-CFI is on par with or outperforms Clang's software-enforced CFI by a small margin, mostly on the Big core and the Performance core.
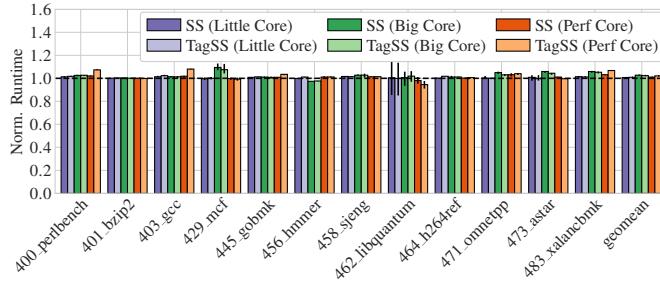
D.2 Backward-Edge CFI (TagSS and RAL)

Backward-edge CFI ensures that functions return to their original call site. Historically, backward-edge protections were offered through a shadow stack—a region of memory that stores copies of return addresses after call instructions. These copies are either directly used as return addresses (as they cannot be corrupted due to memory safety errors in the program) or checked against the return address on the stack (to ensure this has not been corrupted).
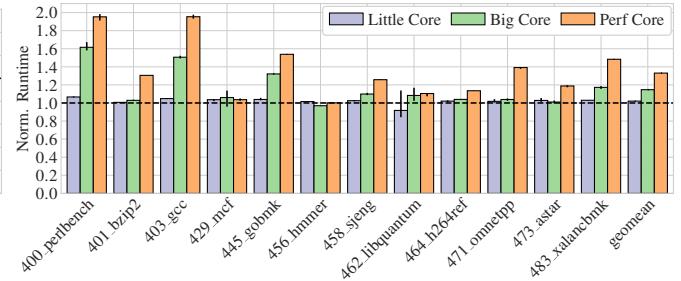
Unfortunately, software implementations of shadow stacks must rely on SFI/software sandboxing to isolate the shadow stack—an approach that can impose a penalty around 7% on the entire program [93, 100]. Thus, production implementations instead rely on randomization to hide the shadow-stack—a best-effort approach that is not secure.

ARM hardware also supports pointer authentication codes (PAC)—a mechanism which signs code pointers to protect them from tampering—an alternate way to protect return addresses by signing pointers. Unfortunately, PAC does not prevent return address-reuse; an attacker can simply read a return address in some prior frame and overwrite the current return address on the stack with the prior address.

Tagged Shadow Stack (TagSS). TagSS tags the shadow stack with a dedicated MTE tag that is not used by any other part of the program; this ensures that the shadow stack can only be

(a) Protection for a shadow stack (TagSS)

(b) Protection for return address slots (RAL)

Figure 12: Performance overhead of backward-edge TagCFI schemes. RAL Performance Overhead normalized to the baseline without any instrumentation. The Performance core shows high overhead due to serializing stores, and all workload with severe slowdown is stack-intensive workloads, such as `perlbench` and `gcc`. TagSS Performance Overhead compared to Clang ShadowCallStack (`-fsanitize=shadow-call-stack`). Its geomean is around 1% and the worst performance stays within $1.13\times$ across all cores.

accessed by the software shadow stack instructions. The program uses the shadow stack with a pinned register that holds a shadow stack pointer, `x18`. We use Clang/LLVM's ShadowCallStack instrumentation pass [5] that modifies function prologues to push the prior return address to the shadowstack along and pop the return address from the shadow stacks at the function epilogues. Our TagSS runtime prepares a tagged shadow stack and set `x18` with a tagged pointer to the top of the shadow stack.

Evaluation. In Figure 12a, we observe that TagSS imposes a geomean penalty of $1.023\times$, $1.028\times$, $1.026\times$ on the Little, Big, Performance core in order on the SPEC CPU benchmarks compared to Clang's ShadowCallStack [5]. TagSS achieves deterministically secure shadow stack with only small overheads. Unfortunately, when comparing security with other hardware such as Intel CET [52] or ARM GCS [65], TagSS does not offer a clear way to reserve a tag for its use. Hardware support allowing this could make this scheme more practical in the future.

An alternate approach (RAL). Memory tagging offers another way to support backward edge CFI—return address locking (RAL). The intuition behind RAL is that a dedicated tag is used to tag the stack the frame-pointer and return-address stack-slots in each stack-frame. This ensures that the values can not be corrupted by other memory operations.

Figure 12b shows the performance overheads of RAL, and it imposes an overhead of $1.034\times$, $1.164\times$, $1.335\times$ for the Little, Big, and Performance core respectively. Although the performance core is slow (for reasons discussed in §4.4.1), RAL has reasonable overheads for the Little and the Big cores. Overall, RAL represents a trade-off: better code compatibility for more permissive backward-edge behavior.

Finally, we show the performance of backward-edge protections on AmpereOne in Figure 13, to demonstrate how overheads change on more efficient MTE implementations.
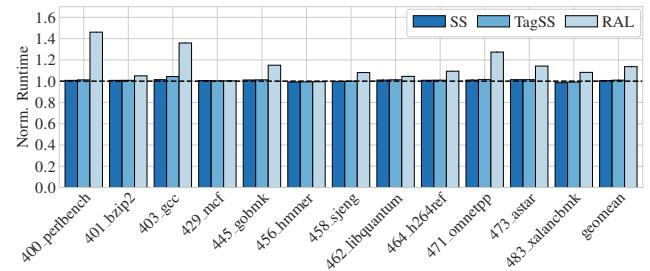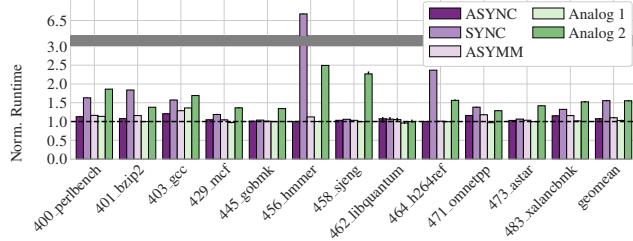

Figure 13: Performance overhead of backward-edge TagCFI schemes tested on AmpereOne. RAL Performance Overhead normalized to the baseline without any instrumentation. TagSS Performance Overhead compared to Clang ShadowCallStack (`-fsanitize=shadow-call-stack`).

E   Extra figures
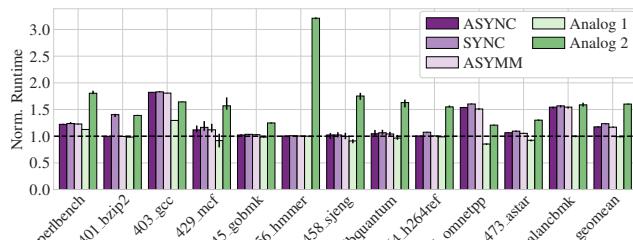
This appendix section contains extra figures that complement our findings presented in the earlier sections. The following list maps each figure to its relevant section.

▶ **Figure** 15 shows SPEC 2006 overheads without the AmpereOne's store-to-load forwarding (which has MTE-related performance bugs) as discussed in §4.4.2.

▶ **Figure** 14, 16 and 17: These three figures include every data point in Figure 2, and additionally have results from software analogues, mentioned in §6.1.

▶ **Figure** 18 shows MTE overheads from other three server application before applying our patch, mentioned in §4.3.

▶ **Figure** 19 includes additional results of MemTrace on other cores; Pixel 8 Little, Big Core and AmpereOne mentioned in §5.1.2.
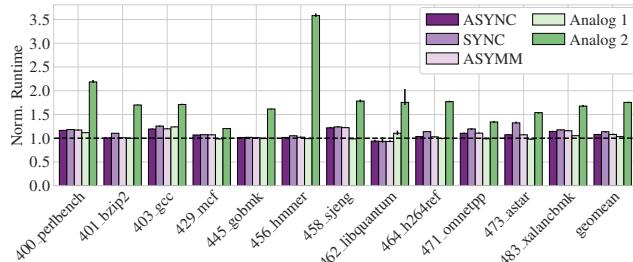
▶ **Figure** 20 shows the performance polybench/C benchmark suite when compiled with wasm2c-MTE—a WebAssembly compiler leveraging MTE discussed in §5.2.1.


(a) Pixel 8 Performance Core

(b) Pixel 8 Big Core

(c) Pixel 8 Little Core

Figure 14: Performance overhead across Pixel 8 cores for two prior MTE analogs, HAKC [66] ("Analog 1") and SFI-Tag [82] ("Analog 2"), compared with three hardware MTE modes (SYNC, ASYNC, ASYMM).
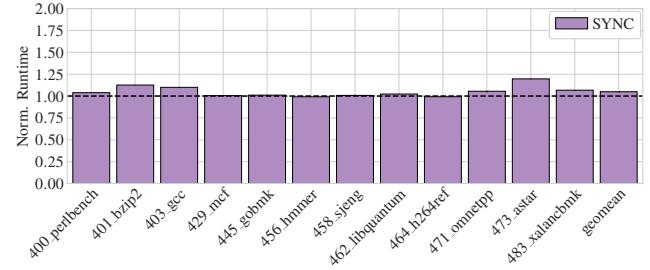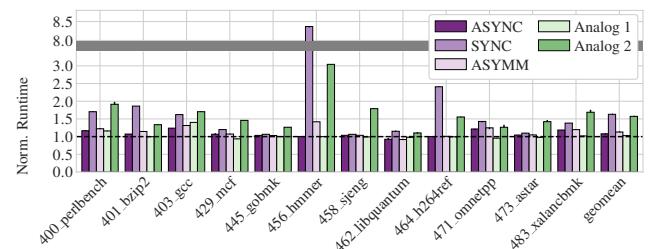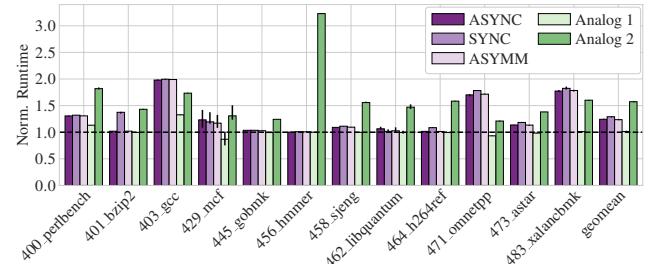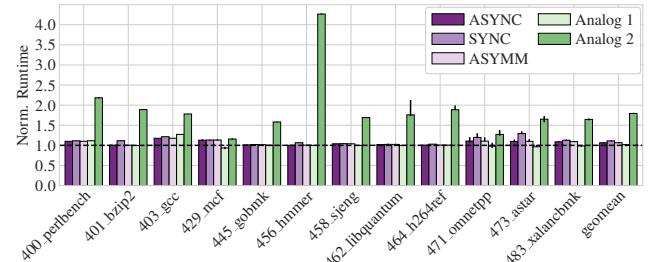

Figure 15: Relative overhead of MTE-hardened allocator with **Store-to-Load forwarding disabled** on CPU2006 INT benchmark. Overhead on `456.hmmer` is reduced from $1.43\times$ to negligible overhead, however a new bottleneck emerged on `473.astar` which increased from $1.10\times$ to $1.20\times$. Overall geomean of overhead dropped from $1.10\times$ to $1.05\times$ when removing store-to-load forwarding variability.

(a) Pixel 9 Performance Core

(b) Pixel 9 Big Core

(c) Pixel 9 Little Core

Figure 16: Performance overhead across Pixel 9 cores for two prior MTE analogs, HAKC [66] ("Analog 1") and SFI-Tag [82] ("Analog 2"), compared with three hardware MTE modes (SYNC, ASYNC, ASYMM).
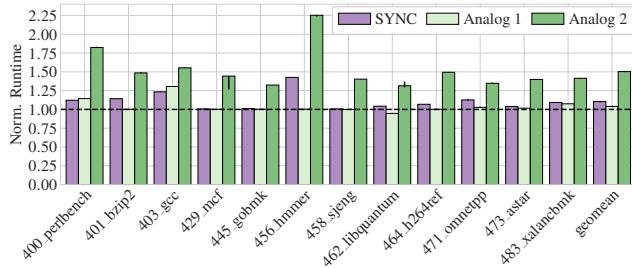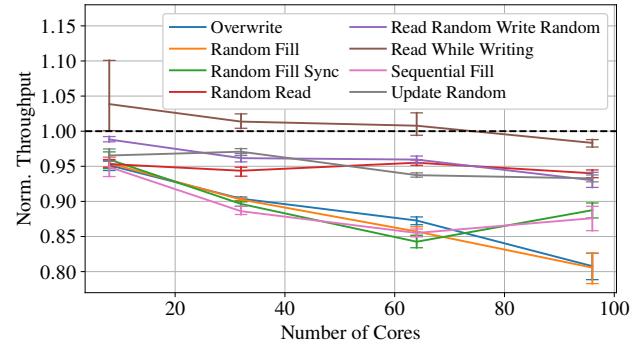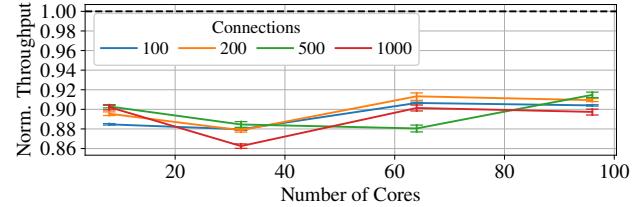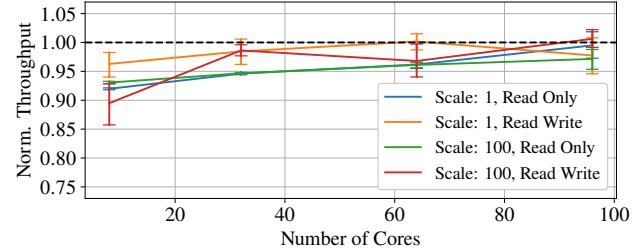
Figure 17: Performance overhead on AmpereOne CPU for two prior MTE analogs, HAKC [66] ("Analog 1") and SFI-Tag [82] ("Analog 2"), compared with its SYNC mode.


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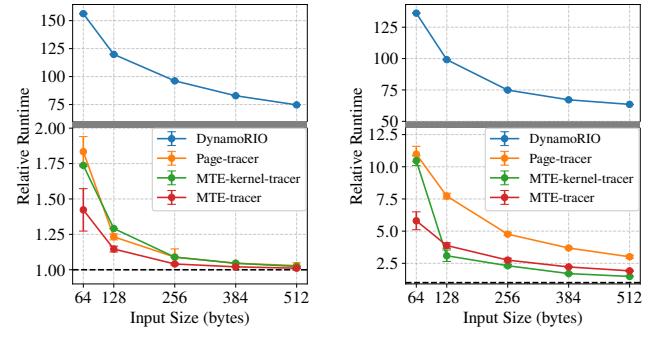


(b) Nginx: tests on serving static files.
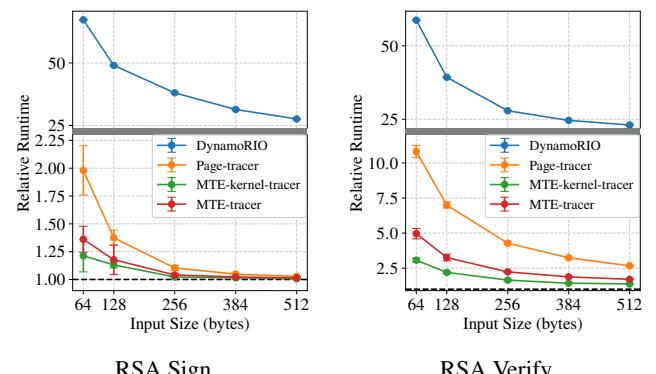
(c) PostgreSQL: SQL queries on databases of different size.
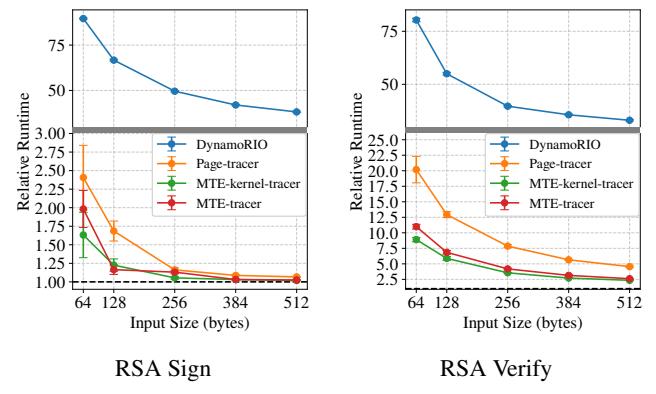
Figure 18: MTE server workload performance on the AmpereOne **without the kernel patch** introduced in §4.3. MTE slowdowns for RocksDB, Nginx, PostgreSQL remain under $1.18\times$.

(a) AmpereOne
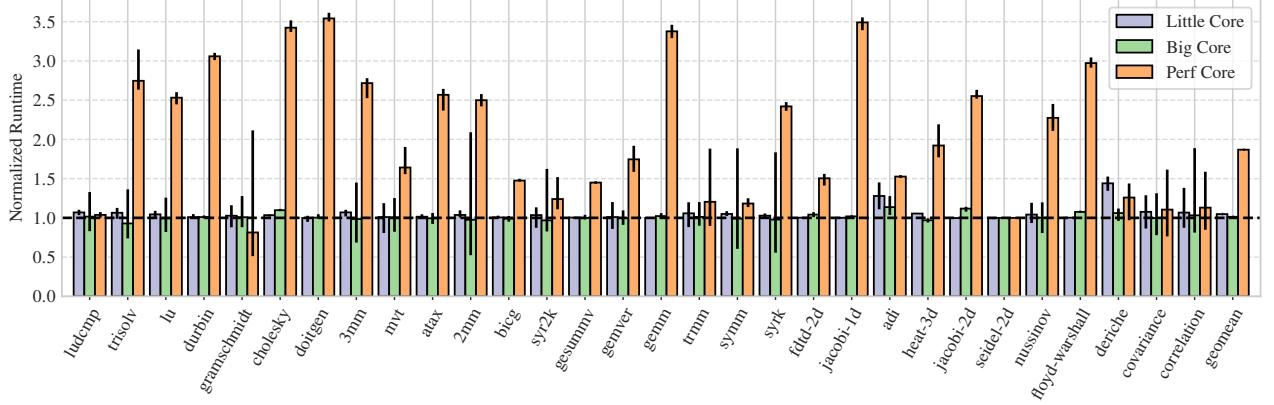
(b) Pixel 8 Little Core
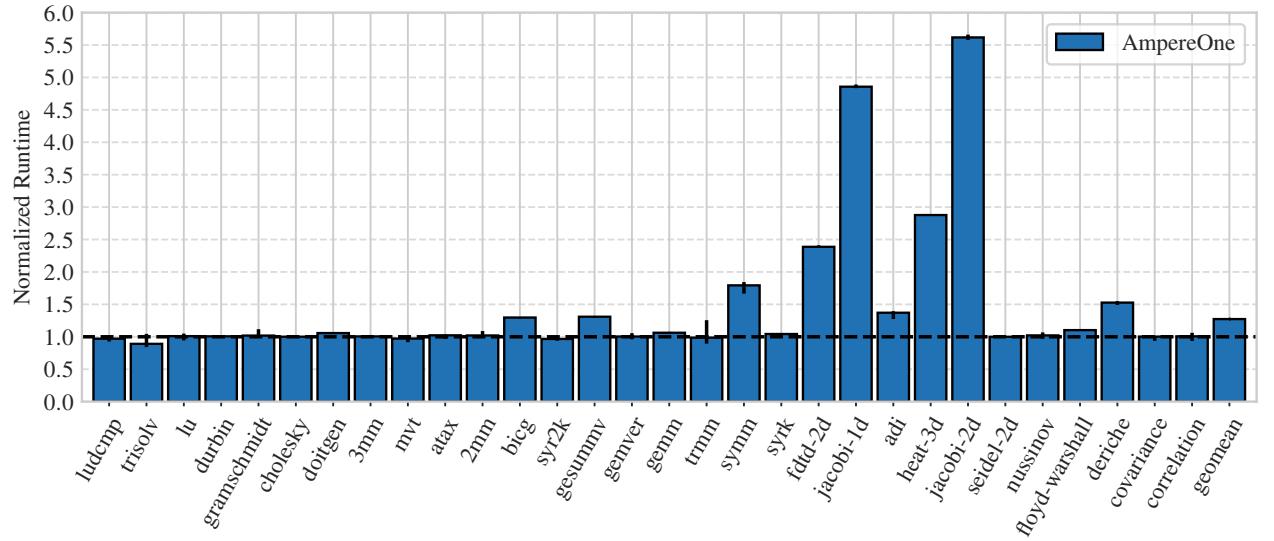
(c) Pixel 8 Big Core

Figure 19: MTE-base and Page-based data tracing overheads on AmpereOne, Pixel 8 Little Core and Pixel 8 Big Core: *We compare the performance of tracing key material through OpenSSL using different tracers; the time taken is normalized to the baseline performance with no tracing enabled. We see that MTE-based tracer is orders of magnitude faster than DynamoRio, and is 2 to 3 times faster than Page-tracer for small-sized inputs and the gap deceases as the buffer size increases.*

(a) Pixel 8 cores.

(b) AmpereOne

Figure 20: The overheads of using ColorGuard with MTE SYNC on the PolyBench/C benchmark suite across platforms. Colorguard eliminates the need for guard pages in SFI toolchains, but must tag all memory with a single tag for this. This tagging generally results in low overheads for the Little and Big cores with geomeans of $1.04\times$ and $1.03\times$ respectively, but has a high geomean overhead of $1.95\times$ in the performance core. This means while ColorGuard can be useful in this domain, the amount of performance variation does not make it the ideal approach.