# ARM Memory Tagging Extension - A Deep Dive

ARM MTE Benchmarking, Testing Legacy Exploits, and Code Analysis For a Google Pixel 8

Juan Manuel Treviño
Columbia University in the City of New York
New York City, New York
j.trevino@columbia.edu

Grange Nagy
Columbia University in the City of New York
New York City, New York
gn2339@columbia.edu

## 1 Introduction

Memory safety errors, such as temporal and spatial errors, are a large percentage of security vulnerabilities. Microsoft has found that, in severe-level security bugs, 70 percent were a result of underlying memory safety problems [3]. With memory safety prevention becoming a prevalent field within academia [23][24][27][29][30][31][32][35], industry [28][28][35][40][44], and governments [1], the line between the security guarantees being offered and their associated overheads has become blurred and/or obfuscated. The Memory Tagging Extension (MTE) is one memory safety protection mechanism introduced by ARM in ARMv8.5-A which seeks to enhance the memory safety of code written in unsafe languages without changing the source code or even the compiled binary [19]. While MTE has notable security and memory promises alongside industry praise [5][6], there is little third-party exploratory research into these claims. Thus, our research aimed to explore ***four*** sides of MTE:

1. **Claims of minimal or negligible overhead:** Google claims a RAM overhead of 3-5% and CPU overhead of "low-single-digit%" [8]
2. **Bug Detection Probabilities**: Confirming the proposed 15/16 odds of catching bugs [8]
3. **Tag Storage and Generation:** due to the closed-source nature of MTE, we aim to reverse-engineer where and how tags are stored.
4. **Conglomerate previous MTE research:** to demystify MTE, we would like to group previous findings to allow developers to decide whether MTE meets their use case.

The remainder of this paper is laid out as follows: we will first discuss relevant background information related to the threat model MTE implements followed by our testing environment and an introduction to MTE across Sections 2.1-2.3. We will then follow with a deep dive into the first three of our four-pronged MTE approach in Section 3. We will then discuss our benchmarking results in Section 4 followed by the aforementioned MTE research in Section 5, finishing with a discussion and conclusion in Section 6.

## 2 Background

ARM's MTE has seen industry adoption namely in Google's Pixel series but has gained interest in Samsung [10] and could see possible adoption from current ARM SoCs as seen in Qualcomm [40], Apple [41], Huawei[1] [42], cloud providers like AWS [10], Azure, Google Cloud, Oracle OCI, Alibaba Cloud Services [12], and even automobile makers like Tesla [11]. MTE has not experienced wide-spread adoption yet due to its relatively new nature, being offered only in post-2021 ARMv9 core designs. Thus, we will first lay out the threat model MTE assumes and then discuss our Google Pixel 8-based testing environment in this section.

### 2.1 Threat Model and Claimed Benefits

ARM claims MTE has five key reasons for adoption: lower cost, reduced time to market, more secure and safer user experience, flexible configurations, and high scalability [14]. MTE is designed to be utilized for cheaper and faster pre-development testing[2][18], crowdsourced bug detection through testing in production through actionable data deduplicated bug reports, and always on security mitigation with per-process knobs. Google, an early adopter of the technology, has expressed their belief that memory tagging will identify the most frequent types of memory safety bugs found in real-world situations, aiding vendors in detecting and resolving them, which in turn deters malicious actors from exploiting them. [43]

[1] With Huawei able to license Armv9 US regulation-free
[2] In comparison to ASAN

The MTE threat model defined by ARM, is that MTE is designed to enhance resistance against attacks trying to manipulate code processing malicious data provided by attackers. It doesn't address algorithmic vulnerabilities or malicious software. MTE is solely aimed at detecting memory safety violations and increasing resistance against attacks enabled by those violations. MTE is designed to be used alongside other protections such as Branch Target Identification (BTI) and Pointer Authentication Code (PAC) to provide comprehensive defenses against attacker control [19].

In a further deep-dive as explored by Microsoft [34] and briefly described in Section 4, MTE protection can be broadly placed into five categories both inside and outside of the scope of MTE:

Inside the Scope:

1. **Spatial Memory Access Vulnerabilities**: spatial memory errors occur when a program accesses memory outside its allocated region. MTE ensures adjacent memory allocations have different tags, thus causing access to unallocated or incorrectly allocated memory to fault. Accesses to non-adjacent locations are thus probabilistic as attackers would be required to guess or learn tags to bypass MTE.
2. **Temporal Memory Errors**: temporal memory errors are the result of incorrectly using memory, such as use-after-free or use-before-initialization, which are probabilistically mitigated.
3. **Implementation Limitations**: workflows that require briefly disabling MTE such as shared memory across a trust boundary or memory accesses that do not necessarily generate a tag check.

Outside the Scope:

4. **Side-Channel Attacks**: these are never mentioned as a potential attack and thus are out of the scope, but we will explore this significant assumption in Sections 4 and 5.
5. **Attacks such as the following**: arbitrary memory access (MTE checks the legitimacy of the tags and not the memory access itself), and intra-object memory access (MTE does not mitigate errors within the bounds of a single memory object as the entire object has the same tag).

## 2.2 Google Pixel 8

The Google Pixel 8's system architecture is laid out as follows[15]: a Tensor G3 (CPU) core with 1x ARM Cortex-X3 (Big Core), 4x ARM Cortex-A715 (Mid-Core), 4x ARM Cortex-A510 (Little Core), and a ARM Mali-G715 GPU. In Section 2.3, we will dive deeper into the tag storage and checking process.

### *2.2.1* Setting up the Pixel 8 and Development Environment

During testing, we utilized a rooted and unlocked Google Pixel 8 running custom C/C++ application(s) on Android Studio. To root the device, we followed steps similar to various guides online [47][48].

## 2.3 ARM MTE

MTE was first introduced in ARM's MTE white paper [19] in August 2019 as a subcomponent of the Armv8.5 ISA before being integrated into the Armv9 CPUs, specifically the Cortex-A510 Cortex-A710 and Cortex-X2 in May 2021. In August 2019, Google also announced the adoption of MTE into the Android ecosystem and released the Pixel 8 with MTE support in late 2023. In late 2022, Honor announced they would commit to making MTE enabled MagicOS6.x/7 mobile devices, with the Honor Magic 5 releasing in March 2023[20].

Memory tagging aims to utilize the unused upper bits in 64-bit pointers to store 4 byte tags to represent 16 random and unique 'colors' for each 16-bytes of memory. Upon loads and stores, MTE utilizes the tag stored in the pointer to allow only authenticated reads/writes. If the tag check fails, an exception is raised depending on the operation mode of MTE. In MTE synchronous mode, optimized for debugging over performance while also acting as a security mitigation strategy, tag mismatches immediately terminate the offending load/store and return a SIGSEGV alongside information about the memory access and the faulting address. Conversely, in asynchronous mode, optimized for performance over bug report accuracy, tag mismatches allow the process to continue execution until the nearest kernel entry such as system calls or timer interrupts happen where the process is then terminated again with SIGSEGV but do not return the faulting address or load/store. Per Android documentation, they strongly recommend SYNC mode (enabled globally for all processes using the environment variable or the build system) during bring-up/development/testing and strongly recommend using ASYNC mode in production. As of Android 12, Google uses MTE ASYNC [6] as a part of the following components to detect end-user crashes and act as an additional layer of defense-in-depth:

- Networking daemons and utilities (except netd)
- Bluetooth, SecureElement, NFC HALs, and system applications

- Statsd daemon
- System_server
- Zygote64 for enabling MTE in applications

MTE is currently disabled by default and only enabled in the developer settings of the Pixel 8 after a reset. MTE is enabled in the phone's bootloader[45]. According to Android 13 documentation, the bootloader reads *misc_memtag_message* within the *bootloader_message.h* misc partition. *Misc_memtag_message* is a 64-byte struct which holds the following: version number, magic number[3], memtag_mode, and reserved bits. A valid misc_memtag_message allows the bootloader to perform the following operations:

1. A bool calculates if MTE should be default, disabled, enabled, or enabled for one boot.
2. If the bool is true, the bootloader sets up MTE tag reservation, enables tag checks in lower privilege levels, and communicates the tag reserved region to the kernel via the device tree[4].
3. Another bool then calculates whether kernel memory tagging should be permanently enabled or enabled one time.

Now that MTE is enabled, the kernel and userspace heap allocator can augment each allocation with metadata. Tags are stored in a special metadata region corresponding to allocated heap memory [21]. In a separate ARM document, delineating MTE, they similarly state that "tags must be fetched from and stored to the memory system". A final ARM Documentation [22] states that "the way memory tags are stored is a hardware implementation detail". Thus, we can now break down a speculated summary of memory tag storage: we believe that, based on ARM's MTE documentation and the specifics of the Cortex-A710 integration, it's likely that the 4-bit memory tags are stored in the Cortex-A710 L1/2 data caches, DSU L3 cache, and CL-700 HN-F SLC used in the Pixel 8. Thus, most tags are most likely stored in the CPU cache with tags being evicted to DRAM if needed. To store tags into DDR memory controllers without native MTE support, the Memory Tag Slave Interface (MTSX) and Memory Tag Unit (MTU) support the separation of data and tag storage in systems without MTE support. The MTU in the MTSX divides the DRAM memory space into two regions: the data storage and tag storage sections. In external memory, tags are thus stored as an array with corresponding consecutive associated 16-byte regions as seen in the official ARM Documentation image below:

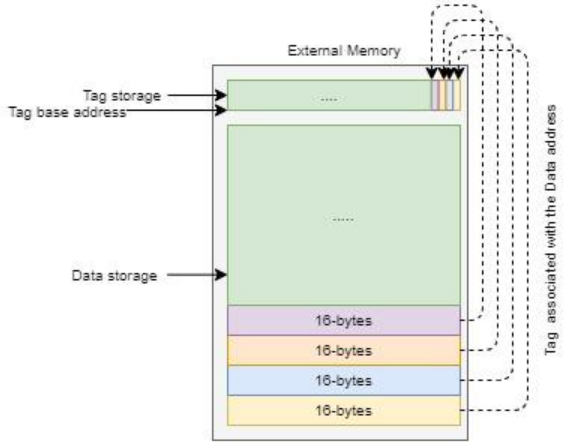


*Figure 1*

The MTU is also what is responsible for tag checking logic. Below is an image that shows the overall MTSX and MTU workflow for checking and storing tags:

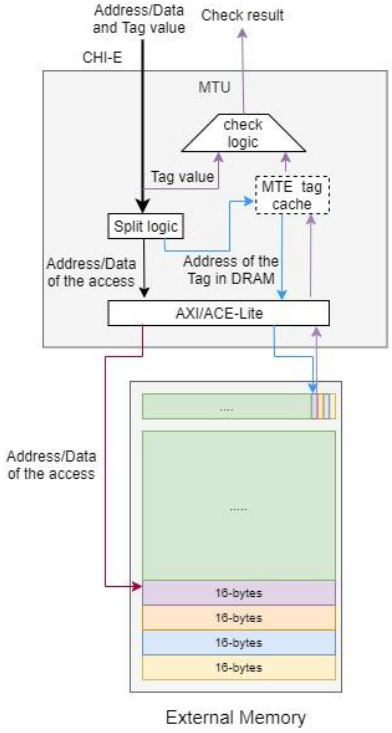


*Figure 2*

As we can see, tags are optionally stored in external memory and fetched using the MTU via the AXI/ACE-lite bus. From there, the MTU splits the tag/data, checks the tag validity, and only returns if successful. We will further explore this in Section 3.2. In summary, while we cannot undoubtedly confirm where and how tags are stored and checked,

[3] Treble Compatibility Check which is used for faster and easier updates to Android OS by separating hardware-specific drivers and firmware used by manufacturers from Android OS.

[4] We presume DT, as they refer to it, is Device Tree

we speculate that they are stored in the middle core cache and swapped out to DRAM through the big core in the event the pre-allocated space in the bootloader is overfilled.

### *2.3.1* MTE Usage and Reporting Bugs

As mentioned in Section 2.1, MTE is intended to act as a bug detection mechanism rather than an explicit line of defense against the attacks mentioned in Section 2.1 [7].

#### Bug Reports

Bug reports contain the following: device logs (system services via dumpsys, error logs via dumpstate, and system message logs via logcat), stack traces (error and messages written from all applications with the *log* class. When running in SYNC mode, the Android allocator records stack traces for all allocations and deallocations for the bug report), backtrace (frame number, program counter value, name of mapped region, and symbols). Bug reports also include an explanation of memory errors such as use-after-free, or buffer-overflow, and the stack traces of relevant memory events. When a tag mismatch is detected, the processor aborts execution immediately, terminates the process, and logs the information about the memory access and the faulting address alongside the process ID, thread ID, and cause of the crash.

#### Tombstones

While the information mentioned in bug reports only comes from submitting a bug report, it is possible to view the information – with root access – by reading the tombstones generated during a crash. Tombstones are the basic crash that are generated under /data/tombstones[5]. Per documentation, the tombstone file contains the following: Stack traces for all the threads in the crashed process, including the thread that caught the signal, a full memory map, a list of all open file descriptors. Additional functionality has been added to support MTE in the tombstone, with a segment being added that records the tag data for any MTE tagged memory pages in the process.

## 3 Work Performed Deep Dive

We will now dive deeper into the work performed on the device by first discussing the benchmarking metrics, followed by a deep exploration of the external memory usage mentioned in Section 2.3, and finally concluding with an analysis of the tag distribution and its implications on the security guarantees discussed in Section 2.1.

[5] Again, only viewable through root access

### 3.1 System Benchmarks

We identified three areas of benchmarking we believe measure the overhead of MTE holistically: device performance through CPU utilization and memory throughput, memory overhead through per-process memory usage and cache behavior, and a stress test through high allocation rates and concurrent accesses and we delineate work done across Sections 3.1.1-3.1.3. The benchmarks were developed using Android Studio and can be replicated by creating a sample C/C++ project and loading the changes seen in the class GitHub Repository.

### *3.1.1* Device Performance

To benchmark CPU utilization and memory throughput, we created an Android Studio C++ project to simulate a memory-intensive operation that utilizes tagging in various modes. The code works by allocating 1,000 16 byte sized blocks where we iterate over each block and tag the block using the IRG instruction and use the STG instruction to set the tag of the allocated address. We then repeat this process 10,000 times and accuminate the total time it takes to perform each iteration and the total runtime of the program. We then repeated this process for when MTE is disabled, set to OFF, ASYNC, and SYNC modes. Between OFF, ASYNC and SYNC trials, only the MTE_MODE was altered. Between disabling MTE and enabling MTE, we removed any code that was necessary to run MTE such as the android:memtagMode="<mode>" and tools:replace="android:memtagMode" in AndroidManifest.xml and set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -march=armv8.5-a+memtag") and set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -march=armv8.5-a+memtag") in CMakeLists.txt. To test the CPU utilization, we can run the profile the program with complete data using Android Studio to obtain a CPU analysis across the device. We obtained the following average times across three trials as seen in table one. Most importantly, we found only a 25.82% increase in runtime when transitioning from a disabled MTE to asynchronous mode but found a 158.65% overhead when using synchronous mode.

| Mode | Total Time (seconds) | Time Per Iteration (seconds) |
|---|---|---|
| SYNC | 4.725101185s | 0.0005091236667s |
| ASYNC | 0.7062799483s | 0.00007058166667s |
| OFF | 0.5379132083s | 0.00003764123333s |

| | | |
|---|---|---|
| Disabled | 0.54478989s | 0.000054432s |

*Table 1*

The CPU measurements profiling gives measurements for, for example, Big/Little/Mid cores during the process runtime. While this means an arbitrary inclusion of background processes, we can obtain a rough estimate of the overhead the MTE mode has on the device components. Most notably, we noticed a 118.68%, 89%, and 36% increase in big/little/mid cores respectively from MTE disabled to synchronous mode. From MTE disabled to asynchronous modes, we notice a 12.25%, 25.41%, and 42.80% increase in big/little/mid core usage.

### *3.1.2* Memory Overhead

Testing memory overhead is difficult as we do not know *exactly* what to monitor. Thus, we attempted to run the same program as in 3.1.2 with an infinitely large loop and, in a root adb shell, execute the *dumpsys meminfo <pid>* command to obtain information about system services running with MTE disabled versus in synchronous mode. While *dumpsys meminfo <pid>* prints the memory usage for the process uptime, thus obtaining two different measurements arbitrary to when you run the command, we noticed something interesting: there is a section labeled Unknown, described as "Any RAM pages that the system couldn't classify into one of the other more specific items" that experiences a rather large uptick when MTE synchronous mode is enabled even when it has ran for less time than MTE off. In our comparison, for an uptime of ~ 3009.387 MB, we see an *Unknown* value of ~ 900 KB resident set size total. In MTE synchronous mode, for 107.772 MB, we strangely see 2344 KB usage. Scaled proportionally, we would see an equivalent 70,320 KB usage of Unknown RAM pages when MTE is in synchronous mode. While this could be due to a confounding factor, we speculate this could be attributed to tags being stored in RAM rather than in the CPU for the Middle core. Combined with the CPU utilization seen in the previous section, we can further speculate that the Big core is what does heavy movement from external memory between the program and the tags stored and checked in RAM.[6]

### *3.1.3* Memory Stress Test

For a program that allocates too much memory, Scudo (Android's default allocator) eventually reaches an internal map failure leading to malloc failures which in turn lead to segmentation faults caught by MTE and finally stack smashing. Unfortunately, while we believe the stack smashing is either an intentional error (as it is caught by MTE and PAC) or a bug within Android (less likely), we were unable to find any exploitable bypasses as MTE caught the segmentation fault before the process terminates.

## 3.2 Swap Space

When a page with MTE tags is selected for swap, the kernel needs to ensure that the tags associated with the page are properly saved so they can be restored when the page is brought back into memory later. This save is hooked into the swap function itself. If the page to be swapped is MTE tagged, the kernel allocates memory to store the tags in ***tag_storage*** which is stored inside of the ***mte_pages*** XArray associated with the swap entry of the page. If MTE tags need to be invalidated while they are outside of memory in the swap, the kernel can reach out to these XArrays and modify them. When the page is brought back into memory, the tags are restored from the XArray. An interesting note is that even if the page is MTE tagged, if the kernel fails to restore the tags when the data is brought back from the swap space, no error, warning, or log is created.

To test if tags are moved to swap space when memory pressure is applied, we wrote a program that generated a large buffer and then continued to infinitely fine-grain generate and set tags infinitely, allowing us to monitor the swap space through a trivial *cat /proc/meminfo* command. What we noticed was the swap free space eventually decreases to 0 KB and a background daemon refreshes it back to its maximum of ~ 3,767,060 KB. While we were unable to test this theory, we hypothesize that, similar to a CLKSCREW attack, we could feed the swap space with pre-written tags that could lead to an MTE bypass[7].

## 3.3 Tag Distribution

To test the tag distribution, we are able to set a pointer to 0x00000000 and then run the IRG instruction on the pointer and view the tag generated for the pointer. Per the ARM white paper, the IRG instruction is responsible for the following: "In order for the statistical basis of MTE to be valid, a source

[6] We also wonder if we could test the theory if tags are stored in DRAM using a rowhammer attack for the specific DRAM area MTE uses and see if tag checks fail!

[7] Assuming there is no tag value authentication or disabling writing to swap during the time the daemon is cleaning the swap space

of random tags is required. IRG is defined to provide this in hardware and insert such a tag into a register for use by other instructions.”. Theoretically, if we save the output of each IRG instruction to a 0-15 hashmap, we should see an even distribution of randomly generated tags for ARMs statistical basis claims. Through trials ranging from 100,000 to 100,000,000 iterations across synchronous and asynchronous modes, we found a very skewed distribution which experienced a bias towards tags 3, 5, 6, 9, 10, 12, and 15 as seen in figure 1 below.

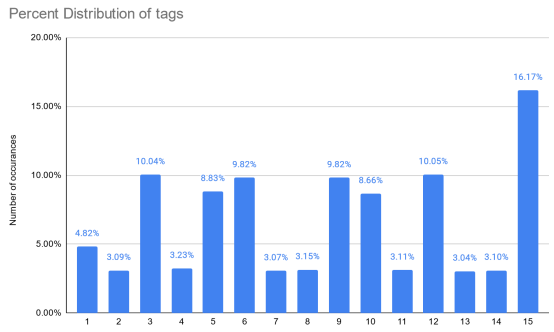


*Figure 1*

Thus, if ARM assumed that the tag entropy was sufficiently random but in reality favors certain tags, it is possible a skilled attacker could exploit this to either poison/game the tag distribution in the RAM, or (uneducated or educated) guess tags. For fun, the longest streak ChatGPT was able to write a sample guessing code for was 6 guessed tags in a row! We will also briefly elaborate on how non-uniform tag distribution rather than statistically-based

## 4 Previous MTE Research

As memory tagging has not been widely explored, we have included brief descriptions on relevant prior presentations we researched throughout our deep dive into MTE:

- **Color My World: Deterministic Tagging for Memory Safety [32]:** this paper modifies MTE to provide deterministic protection against sophisticated attacks (arbitrary read/writes). The researchers use compile-time static analysis to extend MTE to categorize memory safety into classes to prevent memory bug data manipulation.
- **Security Analysis of MTE Through Examples at Bluehat 2022 [33]:** this presentation explores MTE and its practical examples and efficacies in mitigating various memory safety issues. Through various examples, the presentation finds that, even though MTE provides substantial protection, it does not provide protection against sophisticated attacks that bypass or exploit the tagging mechanism itself.
- **Security analysis of memory tagging by Microsoft [34]:** This paper explores the effectiveness of memory tagging as a detection and mitigation strategy. The paper suggests memory tagging could deterministically prevent ~ 13% of memory safety related CVEs related to adjacent heap overruns. For non-adjacent heap overruns and use-after-frees, they describe a probabilistic mitigation with a ~ 6% attack success and summarizes with the stance that while memory tagging significantly enhances security against certain exploits, it does not fully eliminate all risks and needs to become a part of a broader security posture.
- **Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging [35]:** This paper uses object and page-level tagging and extends MTE by integrating multi-granular tags for increased detection capabilities. Using object-specific alongside page-specific tags (as seen in the Bluehat presentation) reduces tag collision likelihood.
- **PACMAN: Attacking ARM Pointer Authentication with Speculative Execution [23]:** This paper, while focusing on ARM’s Pointer Authentication Codes (PAC), showcases a cache side channel attack which leaks the result of a PAC check to bypass the pointer authentication codes used in ARM PAC.
- **Spectre Attacks: Exploiting Speculative Execution [36]:** This, alongside the PACMAN paper, can be used to theorize the contents of Jinbum Park’s presentation at Zer0Con [46] which discusses bypassing ARM MTE with Speculative Execution. While speculative execution is outside of the threat model, it is an undeniable fact that if tags are leaked through microarchitectural side channels, MTE’s capabilities as a defense mechanism are severely questionable.
- **Memory Tagging: A Memory Efficient Design [24]:** This paper explores how MTE is used in modern software systems, develops an architectural improvement to MTE that improves its memory efficiency (organizing process address space into regions bordered by guard pages with dynamic tagging, tags are first assigned a

random tag upon allocation and tags are incremented upon deallocation[8]), and discusses various ways to improve MTE against various memory safety attacks.

- **xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64 [37]:** this paper uses a software–based approach to memory tagging for x86-64 based systems which lack hardware-accelerated memory tagging. Though, if we find up to 200% overhead in MTE, we predict much higher overhead as xTag mimics the functionality of MTE to embed metadata into pointers.
- **Evolving Operating Systems Towards Secure Kernel-Driver Interfaces [38]:** This paper aims to improve the security of kernel-driver interactions within operating systems by implementing secure isolation boundaries. The paper is inspired by MTE as a fine-grained hardware-backed memory isolation and attempts to replicate MTE at the OS level.
- **Gaining kernel code execution on an MTE-enabled Pixel 8 [39]:** this GitHub security post uses CVE-2023-6241 (a vulnerability in the ARM Mali GPU) to bypass MTE as the exploit manipulates GPU memory management to achieve kernel code execution and never relies on pointer dereferencing that MTE protects against.

## 5 Discussion and Conclusion

As seen in Section 4, MTE has been explored as a memory safety mechanism, attacked and bypassed, and improved upon by the research community. While MTE provides a great avenue to detect and mitigate memory safety errors, it comes with some crucial fine-print that needs to be further investigated before full community adoption. If MTE is truly susceptible to microarchitectural side channel attacks akin to PACMAN [23], MTE would instead act as a nuisance rather than a mitigation to a skilled and motivated attacker. If MTE synchronous mode uses almost 4x more power than disabling MTE (and even 1.15x versus asynchronous mode), it is an imperative tradeoff an average consumer would need to be aware of before mindlessly enabling MTE. Also, while this would require further testing across multiple devices, if MTE is built upon a truly uneven tag distribution, all guarantees assumed protected against probabilistic mitigation are rendered questionable. While MTE is a great resource for finding memory safety bugs in development and beta environments, it should not be relied upon to prevent and catch all memory safety errors such as those with highly sensitive data or those whose errors significantly impact the system such as in autonomous vehicles [11].

[8] The increment upon deallocation helps in detecting and preventing use-after-free errors

# Works Cited


[1] NSA Rust Paper
[2] Secure by Design: Google's Perspective on Memory Safety
[3] MSFT 70% of all sec bugs are memory safety issues
[4] Google: MTE - The promising path forward for memory safety
[5] ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety
[6] Android MTE Documentation Deep Dive
[7] MTE User-Guide
[8] Google presentation on MTE
[9] Samsung Exynos
[10] An AWS SoC (gravitron3) in production
[11] Tesla SoC chip
[12] Various cloud providers that use ARM SoCs
[13] Microsoft designing its own ARM SoC
[14] Memory Safety: How Arm Memory Tagging Extension Addresses this Industry-wide Security Challenge
[15] Google Tensor G3
[16] MTE Architecture Described
[17] Arm interconnect
[18] Crowdsourced bug detection in production: GWP-ASan and beyond
[19] ARM MTE White Paper
[20] Honor Magic 5 Phone
[21] Understanding MTE reports
[22] Community Question of "Where is the MTE Tag stored and checked?"
[23] PACMAN Paper
[24] Memory Tagging: A Memory Efficient Design
[25] ARM PAC
[26] REST Paper
[27] Hardware-Software Co-design for Practical Memory Safety
[28] Intel MPX
[29] CHERI Paper
[30] Califorms Paper
[31] Hardbound Paper
[32] Color My World Paper
[33] Security Analysis of MTE Through Examples
[34] Security Analysis of Memory Tagging
[35] Multi-Tag Paper
[36] Spectre Paper

[37] xTag Paper
[38] Evolving Operating System Kernels Towards Secure Kernel-Driver Interfaces
[39] Gaining kernel code execution on an MTE-enabled Pixel 8
[40] ARM Qualcomm SoC
[41] Apple ARM SoC (M1)
[42] Huawei free to license latest ARM architecture: report
[43] Memory Safety: How Arm Memory Tagging Extension Addresses this Industry-wide Security Challenge
[44] Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC
[45] MTE Bootloader Support
[46] Zer0Con Homepage
[47] How to Root Pixel 8/Pro via Magisk Patched Init boot
[48] How to Root Pixel 8 and 8 Pro Using Magisk Patched Boot Image

Appendices

*Appendix A*

Please find the GitHib Repository here. The repository contains information to replicate this project, project notes, and the weekly presentations! https://github.com/coms6424-s24/ARM-MTE-Study