Armv8.5-A Memory Tagging Extension

arm

White Paper

Abstract — The Internet worm of 1988 took offline one tenth of the fledgling network, and severely slowed down the remainder [1]. Over 30 years later, two of the most important classes of security vulnerability in code written in C-like languages are still violations of memory safety. According to a 2019 BlueHat presentation, 70% of all security issues addressed in Microsoft products are caused by violations of memory safety [2]. Similar figures have been reported by Google for Android, where over 75% of vulnerabilities are violations of memory safety [3]. While many of these violations would be impossible in newer languages, the base of in-use code written in C and C++ is vast. Debian Linux alone contains over half a billion lines [4].

But TYCHE is written in Rust so it's safe?

This paper introduces the Armv8.5-A Memory Tagging Extension (MTE). MTE aims to increase the memory safety of code written in unsafe languages without requiring source changes, and in some cases, without requiring recompilation. Easily deployable detections of and mitigations against memory safety violations may prevent a large class of security vulnerabilities from being exploitable.

Introduction

Violations of memory safety fall in to two main categories; spatial safety and temporal safety. Exploitable violations form the first stage in attacks designed to deliver malicious payloads or be chained with other types of vulnerabilities to gain control of a system or leak privileged information.

Spatial safety is violated when an object is accessed outside of its true bounds. For example, when a buffer on the stack is overflowed. This may be exploited to overwrite the function's return address, which can form the basis of several types of attack.

Temporal safety is violated when a reference to an object is used out of scope, typically after the memory backing the object has been reallocated. For example, when a type containing a function pointer of some kind is overwritten with malicious data, which can also form the basis of several types of attack.

MTE provides a mechanism to detect both main categories of memory safety violation. MTE assists the detection of potential vulnerabilities before deployment by increasing the effectiveness of testing and fuzzing. MTE also assists detection of vulnerabilities at scale after deployment.

With careful software design, sequential safety violations where memory is accessed immediately before or after the true bounds can always be detected. 'Wild' violations to arbitrary locations in the address space can be detected probabilistically.

Locating and fixing vulnerabilities before deployment reduces the attack surface of deployed code. Detecting vulnerabilities at scale after deployment supports reactively fixing vulnerabilities before they are widely exploited. Research into the economics of cybercrime [5] shows it to be sensitive to scale. Prompt detection and reactive patching may be highly effective at disrupting cybercrime at scale.

Threat Model

MTE is designed to provide robustness against attacks attempting to subvert code processing malicious, attacker-provided, data. It does not address algorithmic vulnerabilities or malicious software.

MTE is designed to detect memory safety violations and to increase robustness against attacks that the violations enable. In dynamically linked systems, legacy code benefits from MTE for heap allocations without recompilation.

Application of MTE to the stack requires recompilation. The MTE architecture is designed with the assumption that the stack pointer is trustworthy. When deploying MTE for stack allocations it is therefore important to combine the use of MTE with other features such as Branch Target Identification (BTI) and Pointer Authentication Code (PAC) to reduce the probability that a gadget exists that would allow an attacker to take control of the stack pointer.

Memory safety with MTE

The Arm Memory Tagging Extension implements lock and key access to memory. Locks can be set on memory and keys provided during memory access. If the key matches the lock, the access is permitted. If it does not match, an error is reported.

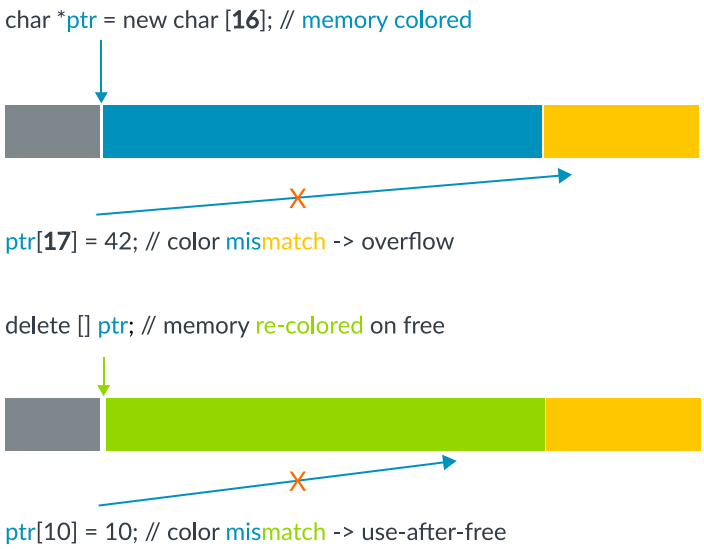
Memory locations are tagged by adding four bits of metadata to each 16 bytes of physical memory. This is the Tag Granule. Tagging memory implements the lock.

Applied on Virtual Memory

Pointers, and therefore virtual addresses, are modified to contain the key.

In order to implement the key bits without requiring larger pointers MTE uses the Top Byte Ignore (TBI) feature of the Armv8-A Architecture. When TBI is enabled, the top byte of a virtual address is ignored when using it as an input for address translation. This allows the top byte to store metadata. In MTE four bits of the top byte are used to provide the key.

Figure 1 shows an example of lock and key access to memory.



MTE relies on the lock and the key being different to detect memory safety violations. As there are a limited number of tag bits available, it cannot be guaranteed that two memory allocations will have different tags for any specific execution. However, a memory allocator can ensure that the tags of sequential allocations are always different and thus ensure that the most common types of safety violations are always detected.

More generally, MTE supports random tag generation and pseudo-random tag generation based on a seed. With a sufficient number of executions of a program, the probability that at least one of them will detect the violation tends towards 100%.

Architectural Details

MTE adds a new memory type, Normal Tagged Memory, to the Arm Architecture. With some exceptions, where it is possible to statically determine the safety of the access, loads and stores to this new memory type perform an access where the tag present in the top byte of the address register is compared with the tag stored in memory.

A mismatch between the tag in the address and the tag in memory can be configured to cause a synchronous exception or to be asynchronously reported.

How does it exactly work?
Which tags are active?
Physical vs Virtual memory

When mismatches are configured to be asynchronously reported, details are accumulated in a system register. A control is provided to ensure that this register is updated on entry to software running at a higher exception level. This enables the operating system kernel to isolate the mismatch to a particular thread of execution and make decisions based on this information.

A synchronous exception is precise in that it is possible to exactly determine which load or store instruction caused the tag mismatch. Conversely, asynchronous reporting is imprecise as it is only possible to isolate the mismatch to a particular thread of execution.

MTE adds instructions to the Armv8-A Architecture that are outlined below and grouped into three different categories [6]:

Where do we have access which tags are active?

Instructions for tag manipulation applicable to stack and heap tagging.

are these instructions privileged.

IRG

In order for the statistical basis of MTE to be valid, a source 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.

GMI

This instruction is for manipulating the excluded set of tags for use with the IRG instruction. This is intended for cases where software uses specific tag values for special purposes while retaining random tag behavior for normal allocations.

we can, in a way change the tags

LDG, STG, and STZG

These instructions allow getting or setting tags in memory. They are intended for changing tags in memory either without modifying the data or zeroing the data.

ST2G and STZ2G

These are denser alternatives to STG and STZG which operate on two granules of memory when allocation size allows them to be used.

STGP

This instruction stores both tag and data to memory.

Instructions Intended for pointer arithmetic and stack tagging:

ADDG and SUBG

These are variants of the ADD and SUB instructions, intended for arithmetic on addresses. They allow both the tag and address to be separately modified by an immediate value. These instructions are intended for creating the addresses of objects on the stack.

SUBP(S)

This instruction provides a 56-bit subtract with optional flag setting which is required for pointer arithmetic that ignores the tag in the top byte.

Instructions intended for system use:

LDGM, STGM, and STZGM

These are bulk tag manipulation instructions which are UNDEFINED at EL0. These are intended for system software to manipulate tags for the purposes of initialization and serialization. For example, they can be used to implement swapping of tagged memory to a medium which is not tag-aware. The zeroing form can be used for efficient initialization of memory.

In addition, MTE provides a set of cache maintenance operations designed for use with tags. These provide efficient mechanisms operating on entire cache lines.

Deploying MTE at Scale

Arm anticipates that MTE will be deployed in different configurations at various stages of product development and deployment.

Precise checks are intended to give the most information about the location of a failure.

Imprecise checks are intended to enable higher performance.

An OS kernel can choose whether to terminate a process that causes an exception due to a tag mismatch or record the occurrence and allow the process to carry on.

Testing a product with MTE enabled can find many of its latent issues. In this phase it is appropriate to detect and record as much information about as many issues as possible. The system does not need to be protected against an attacker. It might be appropriate to configure the system to:

- Perform precise checks.
- Accumulate data on tag mismatches rather than terminating processes.

This configuration allows the most information to be collected to support finding the maximum number of defects via directed testing and fuzzing.

After releasing a product, it might be appropriate to configure MTE to:

- Perform imprecise checks.
- Terminate processes on tag mismatches.

This configuration provides a balance between performance and detecting memory safety violations that might start exploits against the software.

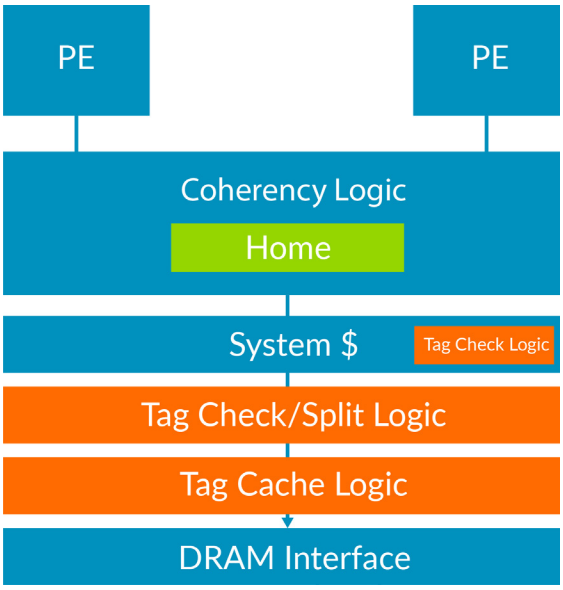
After release, it might be appropriate to configure processes with high value to hackers, such as cryptographic key stores, to perform precise checks so that accurate information about the location of a check failure can be relayed back to its developers by bug reporting and telemetry systems.

It might also be appropriate for a system to adaptively change its MTE configuration. For example, if a process running with imprecise checks is terminated because of a tag-check failure, the next time that process is started it could begin with precise checks enabled to collect better diagnostic information for its developers. This deployment model melds the performance benefits of imprecise checks with the benefits of precise checks to provide better quality feedback.

Deploying MTE in Hardware

In order to support future Arm products that implement MTE, a new version of the AMBA 5 Coherent Hub Interface (CHI) specification is being developed which supports the transport and coherency requirements of MTE [7].

Figure 2 shows an example MTE-based system



Deploying MTE in Software

MTE is designed to support several levels of deployment.

Heap Tagging

In a dynamically linked system, it is possible to deploy a tagged heap without changing existing binaries. Only OS kernel and C library code need to be altered.

Arm prototyped MTE by adding support to the Linux kernel. The following areas needed to be altered:

- Ability to remove tags from user space pointers when they are used for address space management.

- Making the clear_page and copy_page functions in the virtual memory system aware of tags.
- Adding support for handling faults caused by mismatched tags, resembling the way translation faults are handled as a SIGSEGV.
- Converting memory mappings which might be exposed to user-space processes to use Normal Tagged memory.
- Adding detection of the extension and system register configuration to enable the extension.

Arm is in the process of contributing Linux Kernel support upstream.

In the C library, Arm modified these memory-related functions:

- malloc
- free
- calloc
- realloc

In addition, memory copy and string related functions were modified to prevent them overreading source buffers.

Stack Tagging

Tagging memory allocated on a run-time stack requires compiler support and kernel support. Binaries must be recompiled. Many different strategies for stack tagging are possible.

Our partners have prototyped a strategy of choosing a random tag, using the IRG instruction, during function entry when a new stack frame is allocated. The compiler then uses the ADDG and SUBG instructions to create tagged addresses for each stack slot within the function, where the tag is offset from the initial random tag. The stack allocation might be bulk-initialized using an appropriate tag store instruction but a compiler need not initialize any slot that will be provably initialized before use by the function's body code.

This strategy ensures that the statistical properties of MTE are valid for each call to a function and ensures that adjacent objects on the stack have different tags and thus sequential overflow and underflow will result in detections.

Protecting adjacent objects on the stack requires increasing the alignment of those objects to the Tag Granule, which is to 16 bytes. In some programs, MTE causes an increase stack usage because of this effect. Our benchmarking suggests that the increase is usually small.

To increase performance, memory accesses which use the immediate-offset from the stack pointer addressing mode are unchecked under MTE. This is because a compiler can statically prove them correct or issue a diagnostic at compile time.

Optimizing for MTE

MTE has been designed so that it requires no source code modifications to correct code. However, MTE necessarily causes overhead, as tags must be fetched from and stored to the memory system. This overhead is related to the size and lifetime of memory allocations and whether tags and data are manipulated together or separately. Overhead can be minimized in the following ways:

Write tags and initialize memory concurrently.

In many cases, memory must be initialized to zero and the tags set. For example, clearing pages in an OS kernel before handing them to user space. Arm's Linux-based prototype used the STZGM instruction for this purpose.

Avoid over-allocating address space that never has data written to it.

In some cases, software allocates far more address space than it needs and only touches a fraction of it before de-allocating. Using MTE, this is more expensive because even if data is never written to the memory, tags might need to be.

Avoid excessive de-allocation and re-allocation.

Avoiding excessive de-allocation and re-allocation is good practice generally, regardless of whether MTE is deployed. However, because the fixed cost of allocation and de-allocation rises using MTE, existing performance issues might be amplified.

Avoid large fixed-size allocations on the stack.

Large, fixed-size allocations on the stack tend to be under used, for example, buffers of fixed sizes such as PATH_MAX often contain relatively short strings. Avoiding such allocations reduces the overhead of protecting the stack by reducing the amount of unused memory tags must be written to.

Works Cited

[1] F. B. I. [Online]. Available: https://www.fbi.gov/news/stories/morris worm-30-years-since-first-major-attack-on-internet-110218

[2] M. Miller, "Bluehat Abstracts," [Online]. Available: https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf

[3] "Google Queue Hardening," [Online]. Available: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html

[4] Debian, "Stretch Statistics," [Online]. Available: https://sources.debian.org/stats/stretch

[5] "ACM," [Online]. Available: https://dl.acm.org/citation.cfm?id=2654847

[6] "AArch64 Instructions," [Online]. Available: https://developer.arm.com/docs/ddi0596/latest/base-instructions-alphabetic-order

[7] "Architecture Reference Manual," [Online]. Available: https://developer.arm.com/docs/ddi0487/latest