Memory Tagging Extension (MTE) in AArch64 Linux

Authors: Vincenzo Frascino <vincenzo.frascino@arm.com>
Catalin Marinas <catalin.marinas@arm.com>

Date: 2020-02-25

This document describes the provision of the Memory Tagging Extension functionality in AArch64 Linux.

Introduction

ARMv8.5 based processors introduce the Memory Tagging Extension (MTE) feature. MTE is built on top of the ARMv8.0 virtual address tagging TBI (Top Byte Ignore) feature and allows software to access a 4-bit allocation tag for each 16-byte granule in the physical address space. Such memory range must be mapped with the Normal-Tagged memory attribute. A logical tag is derived from bits 59-56 of the virtual address used for the memory access. A CPU with MTE enabled will compare the logical tag against the allocation tag and potentially raise an exception on mismatch, subject to system registers configuration.

Userspace Support

When `CONFIG_ARM64_MTE` is selected and Memory Tagging Extension is supported by the hardware, the kernel advertises the feature to userspace via `HWCAP2_MTE`.

PROT_MTE

To access the allocation tags, a user process must enable the Tagged memory attribute on an address range using a new `prot` flag for `mmap()` and `mprotect()`:

`PROT_MTE` - Pages allow access to the MTE allocation tags.

The allocation tag is set to 0 when such pages are first mapped in the user address space and preserved on copy-on-write. `MAP_SHARED` is supported and the allocation tags can be shared between processes.

Note: `PROT_MTE` is only supported on `MAP_ANONYMOUS` and RAM-based file mappings (`tmpfs`, `memfd`). Passing it to other types of mapping will result in `-EINVAL` returned by these system calls.

Note: The `PROT_MTE` flag (and corresponding memory type) cannot be cleared by `mprotect()`.

Note: `madvise()` memory ranges with `MADV_DONTNEED` and `MADV_FREE` may have the allocation tags cleared (set to 0) at any point after the system call.

Tag Check Faults

When `PROT_MTE` is enabled on an address range and a mismatch between the logical and allocation tags occurs on access, there are three configurable behaviours:

- *Ignore* - This is the default mode. The CPU (and kernel) ignores the tag check fault.
- *Synchronous* - The kernel raises a `SIGSEGV` synchronously, with `.si_code = SEGV_MTESERR` and `.si_addr = <fault-address>`. The memory access is not performed. If `SIGSEGV` is ignored or blocked by the offending thread, the containing process is terminated with a `coredump`.
- *Asynchronous* - The kernel raises a `SIGSEGV`, in the offending thread, asynchronously following one or multiple tag check faults, with `.si_code = SEGV_MTEAERR` and `.si_addr = 0` (the faulting address is unknown).
- *Asymmetric* - Reads are handled as for synchronous mode while writes are handled as for asynchronous mode.

The user can select the above modes, per thread, using the `prctl(PR_SET_TAGGED_ADDR_CTRL, flags, 0, 0, 0)` system call where `flags` contains any number of the following values in the `PR_MTE_TCF_MASK` bit-field:

- `PR_MTE_TCF_NONE` Â - *Ignore* tag check faults
  (ignored if combined with other options)
- `PR_MTE_TCF_SYNC` - *Synchronous* tag check fault mode
- `PR_MTE_TCF_ASYNC` - *Asynchronous* tag check fault mode

If no modes are specified, tag check faults are ignored. If a single mode is specified, the program will run in that mode. If multiple modes are specified, the mode is selected as described in the "Per-CPU preferred tag checking modes" section below.

The current tag check fault configuration can be read using the `prctl(PR_GET_TAGGED_ADDR_CTRL, 0, 0, 0, 0)` system call. If multiple modes were requested then all will be reported.

Tag checking can also be disabled for a user thread by setting the `PSTATE.TCO` bit with `MSR TCO, #1`.

Note: Signal handlers are always invoked with `PSTATE.TCO = 0`, irrespective of the interrupted context. `PSTATE.TCO` is restored on `sigreturn()`.

Note: There are no *match-all* logical tags available for user applications.

Note: Kernel accesses to the user address space (e.g. `read()` system call) are not checked if the user thread tag checking mode is `PR_MTE_TCF_NONE` or `PR_MTE_TCF_ASYNC`. If the tag checking mode is `PR_MTE_TCF_SYNC`, the kernel makes a best effort to check its user address accesses, however it cannot always guarantee it. Kernel accesses to user addresses are always performed with an effective `PSTATE.TCO` value of zero, regardless of the user configuration.

Excluding Tags in the `IRG`, `ADDG` and `SUBG` instructions

The architecture allows excluding certain tags to be randomly generated via the `GCR_EL1.Exclude` register bit-field. By default, Linux excludes all tags other than 0. A user thread can enable specific tags in the randomly generated set using the `prctl(PR_SET_TAGGED_ADDR_CTRL, flags, 0, 0, 0)` system call where `flags` contains the tags bitmap in the `PR_MTE_TAG_MASK` bit-field.

Note: The hardware uses an exclude mask but the `prctl()` interface provides an include mask. An include mask of `0` (exclusion mask `0xffff`) results in the CPU always generating tag `0`.

Per-CPU preferred tag checking mode

On some CPUs the performance of MTE in stricter tag checking modes is similar to that of less strict tag checking modes. This makes it worthwhile to enable stricter checks on those CPUs when a less strict checking mode is requested, in order to gain the error detection benefits of the stricter checks without the performance downsides. To support this scenario, a privileged user may configure a stricter tag checking mode as the CPU's preferred tag checking mode.

The preferred tag checking mode for each CPU is controlled by `/sys/devices/system/cpu/cpu<N>/mte_tcf_preferred`, to which a privileged user may write the value `async`, `sync` or `asymm`. The default preferred mode for each CPU is `async`.

To allow a program to potentially run in the CPU's preferred tag checking mode, the user program may set multiple tag check fault mode bits in the `flags` argument to the `prctl(PR_SET_TAGGED_ADDR_CTRL, flags, 0, 0, 0)` system call. If both synchronous and asynchronous modes are requested then asymmetric mode may also be selected by the kernel. If the CPU's preferred tag checking mode is in the task's set of provided tag checking modes, that mode will be selected. Otherwise, one of the modes in the task's mode will be selected by the kernel from the task's mode set using the preference order:

1. Asynchronous
2. Asymmetric
3. Synchronous

Note that there is no way for userspace to request multiple modes and also disable asymmetric mode.

Initial process state

On `execve()`, the new process has the following configuration:

- `PR_TAGGED_ADDR_ENABLE` set to 0 (disabled)
- No tag checking modes are selected (tag check faults ignored)
- `PR_MTE_TAG_MASK` set to 0 (all tags excluded)
- `PSTATE.TCO` set to 0
- `PROT_MTE` not set on any of the initial memory maps

On `fork()`, the new process inherits the parent's configuration and memory map attributes with the exception of the `madvise()` ranges with `MADV_WIPEONFORK` which will have the data and tags cleared (set to 0).

The `ptrace()` interface

`PTRACE_PEEKMTETAGS` and `PTRACE_POKEMTETAGS` allow a tracer to read the tags from or set the tags to a tracee's address space. The `ptrace()` system call is invoked as `ptrace(request, pid, addr, data)` where:

- `request` - one of `PTRACE_PEEKMTETAGS` or `PTRACE_POKEMTETAGS`.
- `pid` - the tracee's PID.
- `addr` - address in the tracee's address space.
- `data` - pointer to a `struct iovec` where `iov_base` points to a buffer of `iov_len` length in the tracer's address space.

The tags in the tracer's `iov_base` buffer are represented as one 4-bit tag per byte and correspond to a 16-byte MTE tag granule in the tracee's address space.

Note: If `addr` is not aligned to a 16-byte granule, the kernel will use the corresponding aligned address.

`ptrace()` return value:

- 0 - tags were copied, the tracer's `iov_len` was updated to the number of tags transferred. This may be smaller than the requested `iov_len` if the requested address range in the tracee's or the tracer's space cannot be accessed or does not have valid tags.
- `-EPERM` - the specified process cannot be traced.
- `-EIO` - the tracee's address range cannot be accessed (e.g. invalid address) and no tags copied. `iov_len` not updated.

- `-EFAULT` - fault on accessing the tracer's memory (`struct iovec` or `iov_base` buffer) and no tags copied. `iov_len` not updated.
- `-EOPNOTSUPP` - the tracee's address does not have valid tags (never mapped with the `PROT_MTE` flag). `iov_len` not updated.

Note: There are no transient errors for the requests above, so user programs should not retry in case of a non-zero system call return.

`PTRACE_GETREGSET` and `PTRACE_SETREGSET` with `addr == ``NT_ARM_TAGGED_ADDR_CTRL` allow `ptrace()` access to the tagged address ABI control and MTE configuration of a process as per the `prctl()` options described in Documentation/arm64/tagged-address-abi.rst and above. The corresponding `regset` is 1 element of 8 bytes (`sizeof(long))`).

Core dump support

The allocation tags for user memory mapped with `PROT_MTE` are dumped in the core file as additional `PT_ARM_MEMTAG_MTE` segments. The program header for such segment is defined as:

p_type:	`PT_ARM_MEMTAG_MTE`
p_flags:	0
p_offset:	segment file offset
p_vaddr:	segment virtual address, same as the corresponding `PT_LOAD` segment
p_paddr:	0
p_filesz:	segment size in file, calculated as `p_mem_sz / 32` (two 4-bit tags cover 32 bytes of memory)
p_memsz:	segment size in memory, same as the corresponding `PT_LOAD` segment
p_align:	0

The tags are stored in the core file at `p_offset` as two 4-bit tags in a byte. With the tag granule of 16 bytes, a 4K page requires 128 bytes in the core file.

Example of correct usage

MTE Example code

```
/*
 * To be compiled with -march=armv8.5-a+memtag
 */
#include <errno.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/auxv.h>
#include <sys/mman.h>
#include <sys/prctl.h>

/*
 * From arch/arm64/include/uapi/asm/hwcap.h
 */
#define HWCAP2_MTE              (1 << 18)

/*
 * From arch/arm64/include/uapi/asm/mman.h
 */
#define PROT_MTE                 0x20

/*
 * From include/uapi/linux/prctl.h
 */
#define PR_SET_TAGGED_ADDR_CTRL 55
#define PR_GET_TAGGED_ADDR_CTRL 56
# define PR_TAGGED_ADDR_ENABLE  (1UL << 0)
# define PR_MTE_TCF_SHIFT       1
# define PR_MTE_TCF_NONE        (0UL << PR_MTE_TCF_SHIFT)
# define PR_MTE_TCF_SYNC        (1UL << PR_MTE_TCF_SHIFT)
# define PR_MTE_TCF_ASYNC       (2UL << PR_MTE_TCF_SHIFT)
# define PR_MTE_TCF_MASK        (3UL << PR_MTE_TCF_SHIFT)
# define PR_MTE_TAG_SHIFT       3
# define PR_MTE_TAG_MASK        (0xffffUL << PR_MTE_TAG_SHIFT)

/*
 * Insert a random logical tag into the given pointer.
 */
#define insert_random_tag(ptr) ({                       \
        uint64_t __val;                                 \
        asm("irg %0, %1" : "=r" (__val) : "r" (ptr));   \
        __val;                                          \
})
```

```c
/*
 * Set the allocation tag on the destination address.
 */
#define set_tag(tagged_addr) do {                                      \
        asm volatile("stg %0, [%0]" : : "r" (tagged_addr) : "memory"); \
} while (0)

int main()
{
        unsigned char *a;
        unsigned long page_sz = sysconf(_SC_PAGESIZE);
        unsigned long hwcap2 = getauxval(AT_HWCAP2);

        /* check if MTE is present */
        if (!(hwcap2 & HWCAP2_MTE))
                return EXIT_FAILURE;

        /*
         * Enable the tagged address ABI, synchronous or asynchronous MTE
         * tag check faults (based on per-CPU preference) and allow all
         * non-zero tags in the randomly generated set.
         */
        if (prctl(PR_SET_TAGGED_ADDR_CTRL,
                  PR_TAGGED_ADDR_ENABLE | PR_MTE_TCF_SYNC | PR_MTE_TCF_ASYNC |
                  (0xfffe << PR_MTE_TAG_SHIFT),
                  0, 0, 0)) {
                perror("prctl() failed");
                return EXIT_FAILURE;
        }

        a = mmap(0, page_sz, PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (a == MAP_FAILED) {
                perror("mmap() failed");
                return EXIT_FAILURE;
        }

        /*
         * Enable MTE on the above anonymous mmap. The flag could be passed
         * directly to mmap() and skip this step.
         */
        if (mprotect(a, page_sz, PROT_READ | PROT_WRITE | PROT_MTE)) {
                perror("mprotect() failed");
                return EXIT_FAILURE;
        }

        /* access with the default tag (0) */
        a[0] = 1;
        a[1] = 2;

        printf("a[0] = %hhu a[1] = %hhu\n", a[0], a[1]);

        /* set the logical and allocation tags */
        a = (unsigned char *)insert_random_tag(a);
        set_tag(a);

        printf("%p\n", a);

        /* non-zero tag access */
        a[0] = 3;
        printf("a[0] = %hhu a[1] = %hhu\n", a[0], a[1]);

        /*
         * If MTE is enabled correctly the next instruction will generate an
         * exception.
         */
        printf("Expecting SIGSEGV...\n");
        a[16] = 0xdd;

        /* this should not be printed in the PR_MTE_TCF_SYNC mode */
        printf("...haven't got one\n");

        return EXIT_FAILURE;
}
```