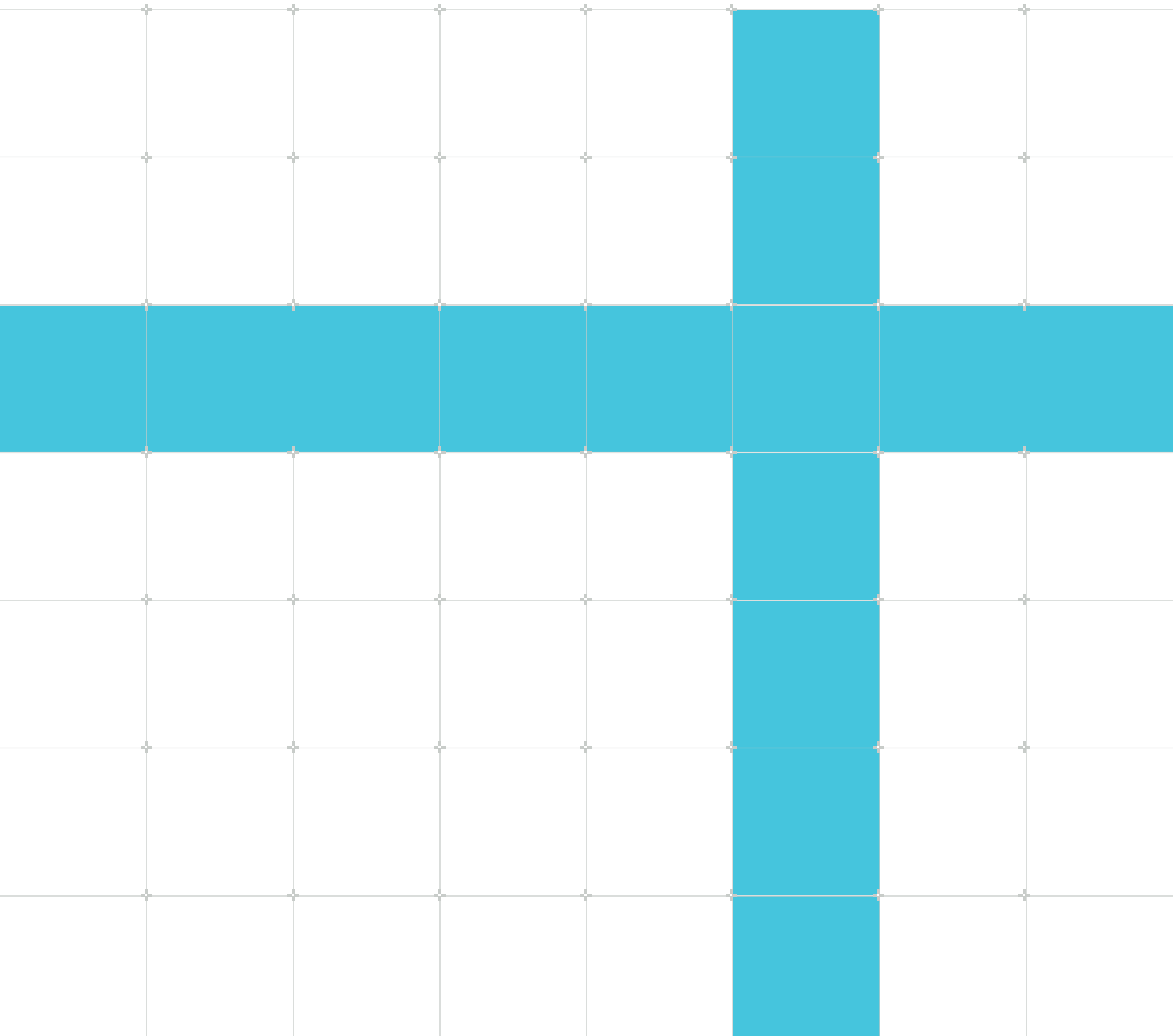
MTE User Guide for Android OS

Version 1.0

Issue 01
108035_0100_01_en



MTE User Guide for Android OS

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	25 May 2023	Non-Confidential	Initial release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction to the Memory Tagging Extension

Arm introduced the Memory Tagging Extension (MTE) as part of the Armv8.5 architecture. MTE is a significant enhancement to the Arm architecture. It improves the security of connected devices by detecting and mitigating memory-related vulnerabilities.

This guide introduces MTE. It shows developers how to use MTE to increase the robustness and security of their software.

This guide includes the following information:

- Memory safety bugs explains why MTE is needed to detect memory bugs.
- How does MTE work? describes how MTE protects against memory bugs.
- Enabling MTE in Android projects shows different ways of enabling MTE in your Android project.
- MTE bug reports explains how to obtain and navigate the bug report provided by MTE after a memory bug is detected.
- Debugging with Android Studio and MTE describes how to use Android Studio to debug code and locate memory bugs.
- Integrating MTE with memory management systems provides information for users who are implementing their own memory allocators.

Google are running an MTE beta device signup program. You can register your interest at the following link:

- MTE beta device signup

2. Memory safety bugs

Memory safety bugs are errors in handling memory by software. Memory safety bugs can occur in the following situations:

- Software accesses memory beyond its allocated size and memory addresses. This is called a spatial memory safety bug.
- Software accesses a memory location outside of the expected lifetime of the data, for example after memory has been freed and reallocated. This is called a temporal memory safety bug.

Google's Memory Safety Report states that memory safety bugs in native code continue to be a major source of end-user crashes, a negative contributor to quality and stability, and the biggest source of security vulnerabilities.

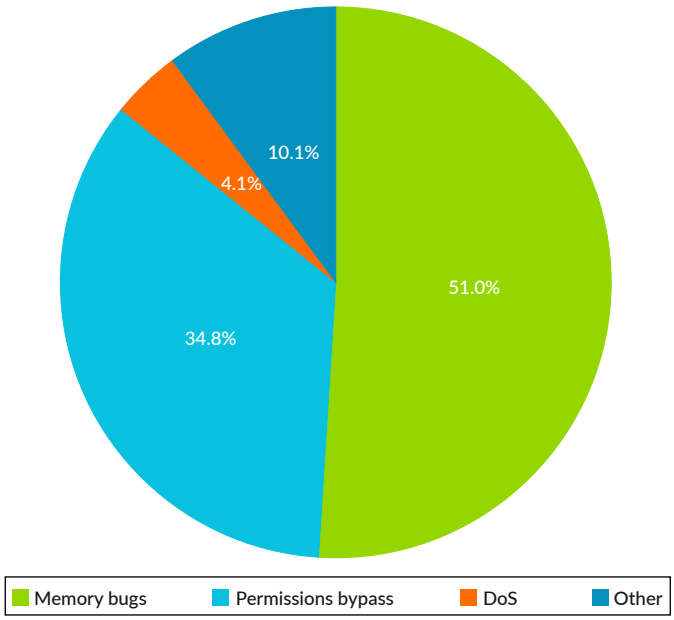
Memory safety bugs are the most common issue in the Android codebases. They account for over 70% of high severity security vulnerabilities and for millions of user-visible crashes.

Native code written in memory-unsafe languages like C, C++, and assembly, represents over 70% of the Android platform code and is present in approximately 50% of Play Store applications.

Memory bugs have a negative impact on quality and stability. They account for a significant share of the crashes observed on end-user devices. Therefore, a high density of memory safety bugs results in a poor user experience.

Memory safety bugs are consistently the top contributor to Android security vulnerabilities. The Google Project Zero team share their zero-day exploit tracking spreadsheet. This spreadsheet shows that memory corruption issues comprise the majority of bugs used as security vulnerabilities for attacks. The following diagram shows that memory safety bugs are the top contributor to Android security vulnerabilities, as reported in Android Documentation: Memory Safety:

Figure 2-1: Memory safety bugs contribution to Android vulnerabilities



Keeping devices up-to-date with security fixes costs the Android ecosystem millions of dollars annually. The high density of memory safety bugs in low-level vendor code significantly increases both fix and test costs. However, detecting these bugs early during the development cycle can lower these costs. Research shows that detecting bugs earlier can reduce costs by up to 6 times.

Unfortunately, it is often very difficult to detect and fix memory safety bugs. Code must first trigger the error before the memory bug can be detected. Memory bugs are often intermittent and difficult to reproduce, so testing and fixing these errors is often a costly and time-consuming process.

From Android 12, Google is introducing systemic changes to reduce memory safety bugs in Android codebases. As part of this effort Google is committed to the following:

- Extending the Android memory safety tools
- Introducing new requirements that encourage the Android ecosystem to address memory safety bugs

As part of these new requirements, the Android Compatibility Definition Document (CDD) strongly recommends the use of memory safety tools during development, integrated with Continuous Integration (CI) and testing processes.

Existing memory safety tools include the following:

- AddressSanitizer (ASan)
- HWAddress Sanitizer (HWASan)
- Kernel Address Sanitizer (KASan)

These tools insert compiler instrumentation for each memory operation to help detect a wide range of memory safety bugs. However, this adds significant overheads to performance, code size, and memory usage.

MTE provides developers with an effective and easy-to-use tool that detects memory bugs, with a low impact on both performance and memory footprint.

2.1 Common memory safety bugs

Memory safety issues fall into two categories: spatial and temporal.

Spatial safety includes buffer-overflow vulnerabilities. A buffer overflow, or buffer overrun, occurs when more data is put into a fixed-length buffer than the buffer can handle, causing data to overflow into adjacent storage. We see this type of issue in applications written in languages that allow pointer manipulation, such as C and C++. The programmer must ensure that pointers stay within the bounds of allocated objects.

The following example demonstrates a buffer overflow bug:

```
extern "C"
JNIEXPORT void JNICALL
Java_com_example_mte_1test_MainActivity_heapOutOfBoundsC(JNIEnv *env, jobject thiz)
 {
    char * volatile p = new char[16];
    p[16] = 42;  // Trying to access a non-allocated array element.
    delete[] p;
    p[0] = 42;  //  Use-after-free error
}
```

Temporal memory safety issues are related to memory locations containing different data at different times during program execution. When the application frees memory and then reallocates it, the application can not assume that the original data is still in memory. A common situation is when the program does the following:

1. Creates a pointer to some memory.
2. Frees the memory but retains the original pointer. This is called a dangling pointer.
3. Attempts to use the dangling pointer to access data.

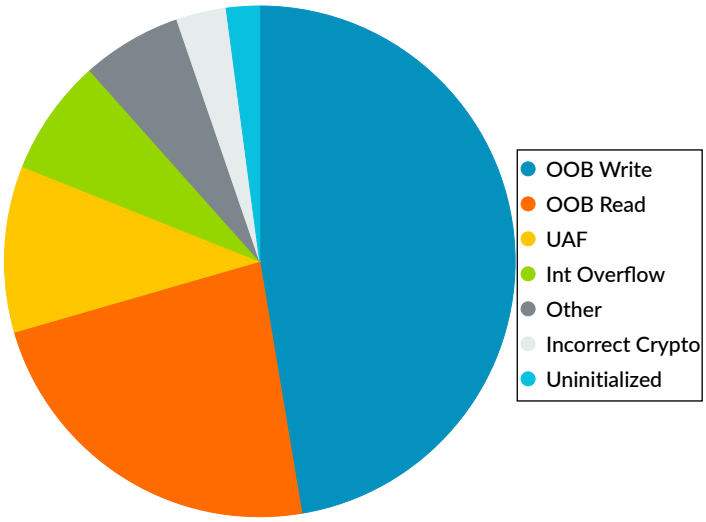
This situation is called a use-after-free (UAF) bug. This causes undefined behavior. Also, there is a risk of leaking information or for an attacker to take control of the application.

MTE detects the most common causes of memory bugs:

- Use-after-free
- Buffer overflow
- Double free

According to a Google security blog post, most Android vulnerabilities are caused by UAF and out-of-bounds (OOB) reads and writes, as the following diagram shows:

Figure 2-2: Memory vulnerabilities by cause



3. How does MTE work?

Arm developed MTE in collaboration with Google to detect memory safety bugs both in existing codebases and in new code as it is written. MTE increases the memory safety of the large existing ecosystem of code written in memory-unsafe languages such as C and C++.

MTE helps you find the following:

- Potential vulnerabilities before deployment, by increasing the effectiveness of testing and fuzzing
- Vulnerabilities at scale after deployment

In Android 12 the kernel and user-space heap memory allocator can augment each allocation with metadata. This metadata enables MTE to detect UAF and heap bugs. These are the most common source of memory safety bugs in Android codebases. This usage model, called heap tagging, means that only codepaths which need to allocate or free memory need to know about MTE. Heap tagging can therefore be deployed in a backwards compatible way, with no source code modifications required for most applications.

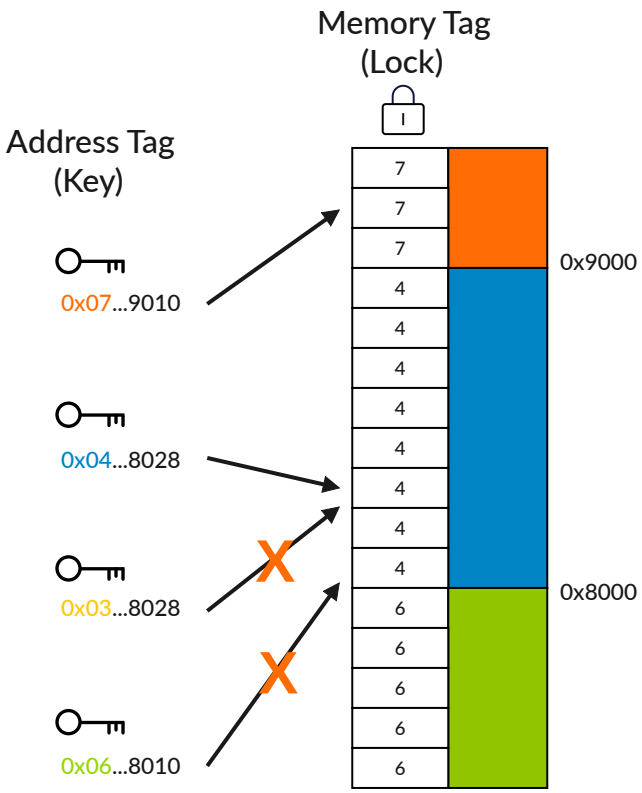
MTE also provides a usage model called stack tagging, where memory allocated on a run-time stack is also tagged. This is a more invasive technique, however. It requires re-compilation, and is not backwards compatible with older devices.

MTE uses an underlying lock and key model to access memory. At runtime, when memory is allocated or freed, that region of memory is assigned a 4-bit memory tag, or lock, as part of the memory address. Memory accesses using pointers to that address must use the same tag, or key, as part of the requested address:

- If the lock and key match, then memory access is granted.
- If the lock and key do not match, then a memory access violation has occurred. A mismatch between the lock tag in memory and the key tag in the address results in a tag check fault, and an error is raised.

The following diagram shows how MTE works for several example memory accesses:

Figure 3-1: MTE lock and key model



The memory accesses shown in this diagram proceed as follows:

- `0x07...9010`

  The key `0x07` matches the lock `0x07`. Memory access succeeds.
- `0x04...8028`

  The key `0x04` matches the lock `0x04`. Memory access succeeds.
- `0x03...8028`

  The key `0x03` does not match the lock `0x04`. Tag check fault.
- `0x06...8010`

  The key `0x06` does not match the lock `0x04`. Tag check fault.

The tag check mechanism means that infrequent, transient, or hard-to-test memory safety errors are detected easily. You can configure tag check faults either to cause a synchronous exception, or to be asynchronously reported.

MTE causes some performance overhead, because 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.

There are several ways that you can minimize this overhead:

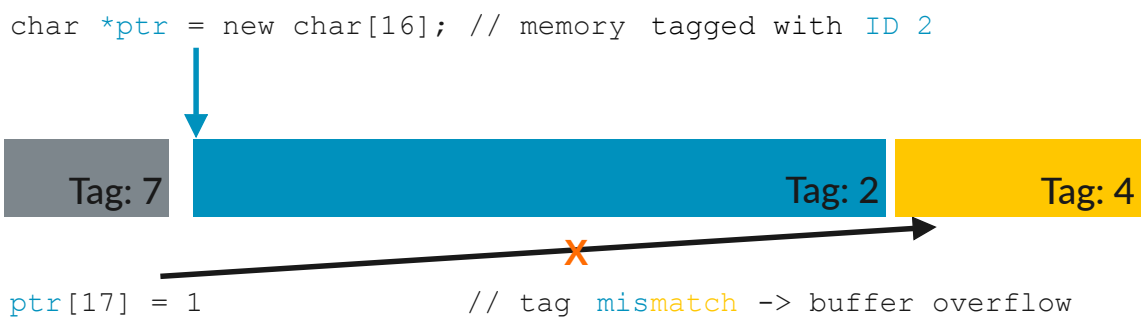
- Write tags and initialize memory concurrently when implementing allocators.
- Avoid over-allocating address space that never has data written to it.
- Avoid excessive de-allocation and re-allocation.
- Avoid large fixed-size allocations on the stack.

For more information about minimizing the MTE overhead, see the Armv8.5-A Memory Tagging Extension White Paper.

3.1 Example: buffer overflow

The following example shows how MTE detects a buffer overflow bug:

Figure 3-2: MTE detecting a buffer overflow bug



The process is as follows:

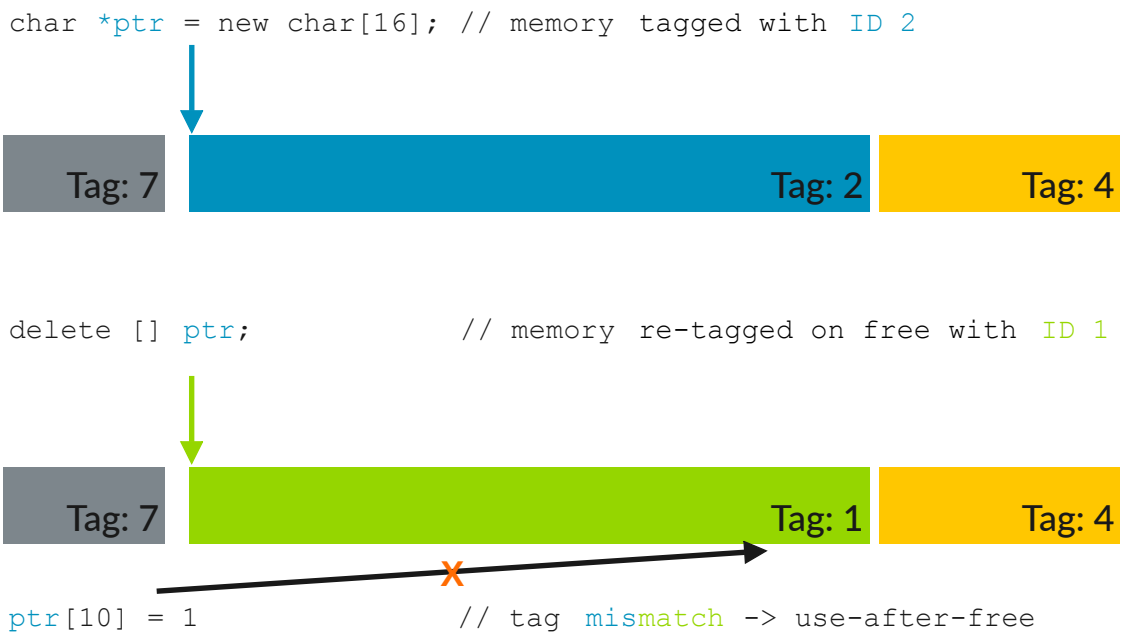
1. Creating the new array pointer `ptr` allocates a region of memory for that array. This memory is tagged with the ID 2, while the memory regions either side are tagged 7 and 4.
2. When a memory is accessed using the array pointer `ptr`, the pointer includes the tag associated with that memory region: 2.
3. MTE checks that the tag used in the memory access matches the tag for the memory region.

   In this example, the memory access overflows, attempting to access the 17th element in a 16-element array. A memory access with tag 2 is attempting to access a region of memory with tag 4, which does not match.

3.2 Example: use-after-free

The following example shows how MTE detects a use-after-free bug:

Figure 3-3: MTE detecting a use-after-free bug



The process is as follows:

1. Creating the new array pointer `ptr` allocates a region of memory for that array. This memory is tagged with the ID 2, while the memory regions either side are tagged 7 and 4.
2. On deleting the array pointer `ptr`, that region of memory is freed and re-tagged with ID 1.
3. When a memory access is made using the array pointer `ptr`, the pointer includes the tag associated with the deleted pointer: 2.
4. MTE checks that the tag used in the memory access matches the tag for the memory region.

   In this example, the memory access occurs after it has been freed. A memory access with tag 2 is attempting to access a region of memory with tag 1, which does not match.

3.3 MTE modes

You can use MTE in the following modes:

- Synchronous mode (SYNC)

In synchronous mode, a mismatch between the tag in the address and the tag in memory causes a synchronous exception. This identifies the precise instruction and address that caused the failure, at the cost of a slight performance impact.

SYNC mode prioritizes accuracy of bug detection over performance. It is most useful during development, or as part of a continuous integration system. In these situations, the precise bug detection capability is more important than the performance overhead.

- Asynchronous mode (ASYNC)

  In asynchronous mode, when a tag mismatch occurs the processor continues execution until the next kernel entry, such as a syscall or timer interrupt. At this point, it terminates the process with SIGSEGV using code SEGV_MTEAERR. The processor does not record the faulting address or memory access. ASYNC mode has a smaller impact on performance than SYNC mode.

  ASYNC mode is optimized for performance over accuracy of bug reports. The information about where the bug occurred is less precise than SYNC mode. ASYNC mode provides a low-overhead detection mechanism for memory safety bugs, and is useful for production systems when performance is more important than detailed bug information.

- Asymmetric mode (ASYMM)

  Newer devices support a mode called asymmetric (ASYMM) that combines the benefits of SYNC and ASYNC modes. In asymmetric mode, read memory accesses are processed as SYNC, while write memory accesses are processed as ASYNC. The performance of asymmetric mode is typically very close to that of ASYNC mode. To identify whether your device supports ASYMM mode, look for the presence of the string `mte mte3` in `/proc/cpuinfo`.

  Android OS does not give programmers a choice for using ASYMM mode. On compatible devices, ASYNC mode is redefined to ASYMM. The implication for programmers is that ASYNC mode may generate synchronous faults, that is SIGSEGV with code SEGV_MTESERR, instead of asynchronous faults.

The main difference between modes is the impact on performance and the accuracy of the error detection. Choosing between modes is a compromise between performance and information.

3.4 When to use SYNC and ASYNC MTE modes

If you use MTE only during development and testing, you might not cover all possible usage scenarios. This means that some bugs may not be discovered. Therefore you should consider using ASYNC mode in production for most critical processes.

The performance overhead of the ASYNC mode, when evaluated across tested workloads and benchmarks, is in the region of 1-2 percent. This means that ASYNC mode is generally acceptable even on production systems. ASYNC mode is recommended in production on well-tested codebases where the density of memory safety bugs is known to be low. ASYNC mode also defends against previously unknown, rarely occurring bugs and 0-day exploits.

SYNC mode is recommended during development and testing, to help identify and fix memory bugs. SYNC mode can also be useful in production systems when the target process represents a vulnerable attack surface. For example, in critical system processes where security takes precedence over runtime performance.

Also, systems can run in ASYNC mode until a bug is detected, then use runtime APIs to switch execution to SYNC mode and obtain an accurate bug report. See MTE bug reports for more information.

4. Enabling MTE in Android projects

MTE is disabled by default. There are several different ways you can enable MTE for system processes and applications. Enabling MTE is a process-wide property: enabling MTE applies to all native heap allocations in a process.

4.1 Enabling MTE using the Android build system

As a process-wide property, MTE is controlled by the build-time setting of the main executable. The following options change this setting for individual executables, or for entire subdirectories in the source tree. The setting is ignored for libraries, and any target that is neither executable nor a test.

1. To enable MTE in the Android blueprint file `Android.bp` for a particular project, use the following settings.

   To enable ASYNC mode:

   ```
   sanitize: {
       memtag_heap: true,
   }
   ```

   To enable SYNC mode:

   ```
   sanitize: {
       memtag_heap: true,
      diag: {
          memtag_heap: true,
      },
   }
   ```

   To enable MTE in the `Android.mk` file for a particular project:

MTE mode	Setting
ASYNC	`LOCAL_SANITIZE := memtag_heap`

SYNC	`LOCAL_SANITIZE := memtag_heap`
	
	`LOCAL_SANITIZE_DIAG := memtag_heap`

2. To enable MTE on a subdirectory in the source tree, use the following settings.

   To enable MTE on a subdirectory in the source tree using the PRODUCT_MEMTAG_* variables:

MTE mode	Setting
ASYNC	`PRODUCT_MEMTAG_HEAP_ASYNC_INCLUDE_PATHS`

MTE mode	Setting
SYNC	`PRODUCT_MEMTAG_HEAP_SYNC_INCLUDE_PATHS`

To enable MTE on a subdirectory in the source tree using the MEMTAG_HEAP_* variables:

MTE mode	Setting
ASYNC	`MEMTAG_HEAP_ASYNC_INCLUDE_PATHS`
SYNC	`MEMTAG_HEAP_SYNC_INCLUDE_PATHS`

To enable MTE on a subdirectory in the source tree by specifying the exclude path of an executable:

MTE mode	Setting

ASYNC	`PRODUCT_MEMTAG_HEAP_EXCLUDE_PATHS`
	or
	`MEMTAG_HEAP_EXCLUDE_PATHS`

SYNC	`PRODUCT_MEMTAG_HEAP_EXCLUDE_PATHS`
	or
	`MEMTAG_HEAP_EXCLUDE_PATHS`

4.2 Enabling MTE using system properties

To override the build settings listed in Enabling MTE using the Android build system, set the following system property:

```
arm64.memtag.process.<basename> = (off|sync|async)
```

Where `basename` stands for the base name of the executable.

For example, if your executable is `/system/bin/myapp` or `/data/local/tmp/myapp`, use `arm64.memtag.process.myapp`.

This property is only read once, at process startup. Arm recommends using this mechanism for rapid prototyping and experimentation with different MTE modes. This property does not apply to Java applications. The MTE setting for Java applications is based on the `AndroidManifest.xml` configuration. For Java applications, see Enabling MTE for applications in the Android manifest. This describes how to use the compatibility framework `compat` feature to change settings using either the developer options or ADB commands.

4.3 Enabling MTE using an environment variable

You can specify the MTE mode by defining the following environment variable:

```
MEMTAG_OPTIONS=(off|sync|async)
```

If both the `MEMTAG_OPTIONS` environment variable and the `arm64.memtag.process.<basename>` system property are defined, the environment variable takes precedence.


Note

Using the `MEMTAG_OPTIONS` environment variable to specify the MTE mode is only supported for command-line applications.

4.4 Enabling MTE for applications in the Android manifest

To configure an application to use MTE, set the `android:memtagMode` attribute under the `<application>` or `<process>` tag in the `AndroidManifest.xml`. If the `android:memtagMode` attribute is not specified, then MTE is disabled.

```
android:memtagMode=(off|default|sync|async)
```

For example:

```
    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MTE_test"
        tools:targetApi="31"
        android:memtagMode="sync"
        tools:ignore="MissingPrefix">
    </application>
```

When set on the `<application>` tag, the attribute affects all processes used by the application. The attribute can be overridden for individual processes by setting the `<process>` tag.

For experimentation, compatibility changes can be used to set the default value of the `memtagMode` attribute for applications that do not specify a value in the manifest, or specify `default`. These compatibility changes are available in the global setting menu under **System** > **Advanced** > **Developer options** > **App Compatibility Changes**. Setting `NATIVE_MEMTAG_ASYNC` or

`NATIVE_MEMTAG_SYNC` enables MTE for a particular application. Alternatively, you can use the `am compat` command as follows:

```
$ adb shell am compat enable NATIVE_MEMTAG_[A]SYNC my.app.name
```

5. MTE bug reports

When MTE detects a memory bug, the application exits. After the application exits, you can access a bug report. This bug report contains device logs, stack traces, and other diagnostic information to help you find and fix memory bugs in your application.

You can access the bug report in the following ways:

- Using **Developer Options** on your Android device.
- Using Android Debug Bridge, `adb`, on your development machine.

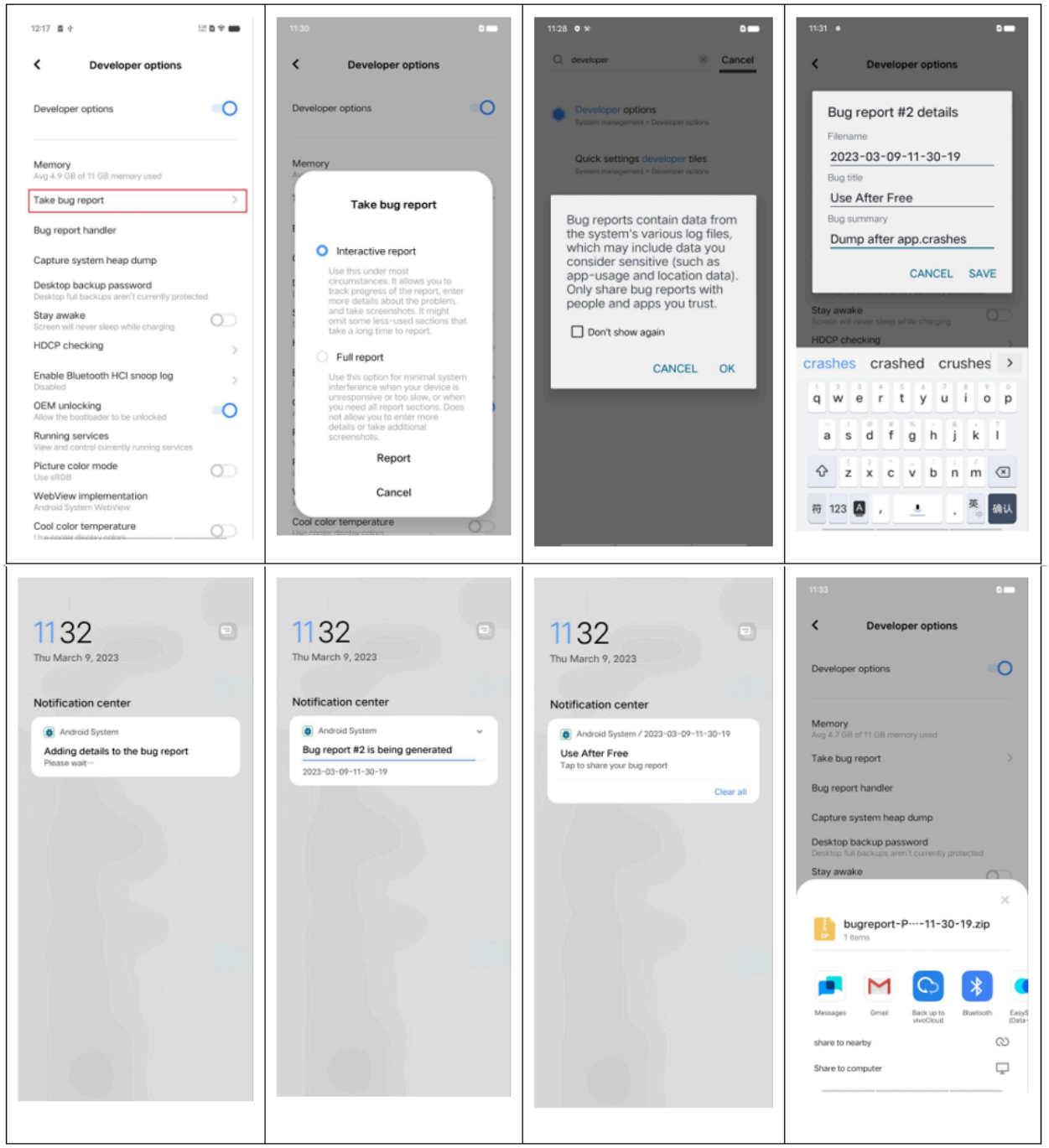
5.1 Capture a bug report using Developer Options on your Android device

To capture a bug report directly from your Android device, do the following:

1. Enable **Developer Options** on your Android device. For more information about how to do this, see Android OS Documentation: Configure on-device developer options.
2. In **Developer options**, tap **Take bug report**.
3. Select the type of bug report you want and tap **Report**.
4. After a short period of time, a notification appears informing you that the bug report is ready.
5. To share the bug report, tap the notification.

The following image shows screen captures of this process taken from a device running an MTE-enabled version of Android. The exact sequence of actions of this process varies depending on your device.

Figure 5-1: Accessing the bug report on a device



5.2 Capture a bug report using adb on your development machine

To capture a bug report using Android Debug Bridge, `adb`, on your development machine, do the following:

1. Run the following `adb` command from the console:

   ```
   adb bugreport
   ```

   The bug report is saved in the console prompt path as a zip file with a file name containing the build ID and date, for example:

   ```
   bugreport-PD2183-TP1A.220624.014-2023-03-09-11-48-36.zip
   ```

2. Unzip the bug report zip file.

   The zip file contains several files. The bug report is the text file with the same name as the folder:

   ```
   bugreport-BUILD_ID-DATE.txt
   ```

5.3 Interpreting the bug report

The bug report contains diagnostic output for system services (`dumpsys`), error logs (`dumpstate`), and system message logs (`logcat`). The system messages include stack traces when the device throws an 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 and uses them to produce the bug report.

The bug report includes an explanation of each memory error, such as use-after-free, or buffer-overflow, and the stack traces of the relevant memory events. These reports provide more contextual information and make bugs easier to trace and fix. On encountering a tag mismatch, the processor aborts execution immediately and terminates the process with `SIGSEGV`, using code `SEGV_MTESERR`, logging full information about the memory access and the faulting address. In addition, the crash report shows the process ID, the thread ID, and the cause of the crash.

To find this information in the bug report, search for the text `SEGV_MTESERR`. This search takes you to the initial block of the segmentation fault that describes the content of the CPU registers x0-x29 at the moment the SIGSEGV was received.

The following is an example of the information you can find in the bug report:

```
Softversion: PD2183C_A_13.1.6.4.W10.V000L1
Time: 2023-03-09 13:56:30
```

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'vivo/PD2183/PD2183:13/TP1A.220624.014/compiler03081056:user/
release-keys'
Revision: '0'
ABI: 'arm64'
Timestamp: 2023-03-09 13:56:30.251141624+0000
Process uptime: 6s
Cmdline: com.example.mte_test
pid: 9477, tid: 9477, name: xample.mte_test  >>> com.example.mte_test <<<
uid: 10237
tagged_addr_ctrl: 000000000007fff3 (PR_TAGGED_ADDR_ENABLE, PR_MTE_TCF_SYNC,
 mask 0xfffe)
pac_enabled_keys: 000000000000000f (PR_PAC_APIAKEY, PR_PAC_APIBKEY, PR_PAC_APDAKEY,
 PR_PAC_APDBKEY)
signal 11 (SIGSEGV), code 9 (SEGV_MTESERR), fault addr 0x08000072b1c2a500
x0  08000072b1c2a4f0  x1  0000007ff60668b0  x2  fffffffffffffff0  x3
 0000007ff6066ac0
x4  0000007ff6066b30  x5  0000000000000004  x6  0000000000000000  x7
 000000729a5e9c02
x8  08000072b1c2a4f0  x9  000000000000002a  x10 0000000000010000  x11
 0000000000000001
x12 00000000b1c2a500  x13 00000075d317c518  x14 ffffffffffffffff  x15
 00000000ebad6a89
x16 00000075d2f13dc0  x17 00000075d2e9cc60  x18 00000075d79a4000  x19
 0200007411a05380
x20 0000000000000000  x21 0000000000000000  x22 0000007285f7c322  x23
 000000000000106e
x24 000000729f800880  x25 0000007ff6066dd0  x26 0000000010380011  x27
 0000000000000004
x28 0000007ff6066cd0  x29 0000007ff6066cb0
lr  000000728644f428  sp  0000007ff6066c80  pc  000000728644f434  pst
 0000000060001000
```

In this example, we see the following line:

```
signal 11 (SIGSEGV), code 9 (SEGV_MTESERR), fault addr 0x08000072b1c2a500
```

This tells us that a SIGABRT signal was received, with code SEGV_MTESERR, caused by an access to memory address `0x08000072b1c2a500`.

Following this block, we see a backtrace that shows where in the code we were at the time of crash, for example:

```
backtrace:
#00 pc 000000000000d384  /data/app/~~8QwzBMNBT6U1-xi0qH9OdQ==/com.example.mte_test-
AE8qJx9ASOlRNa8HmD3iJA==/lib/arm64/libmte_test.so
#01 pc 000000000021a354  /apex/com.android.art/lib64/libart.so
 (art_quick_generic_jni_trampoline+148) (BuildId: 2d8a73ff5c99d5a227b31111c86db3e6)
#02 pc 000000000020a2b0  /apex/com.android.art/lib64/libart.so (nterp_helper+4016)
 (BuildId: 2d8a73ff5c99d5a227b31111c86db3e6)
#03 pc 00000000006fa40a  /data/app/~~8QwzBMNBT6U1-xi0qH9OdQ==/com.example.mte_test-
AE8qJx9ASOlRNa8HmD3iJA==/oat/arm64/base.vdex
And after a number of lines, we see a message pointing directly to the cause of the
 crash: use-after-free and the lines produced by the unwinder.
Note: multiple potential causes for this crash were detected, listing them in
 decreasing order of likelihood.
Cause: [MTE]: Use After Free, 42 bytes into a 128-byte allocation at 0x782d277bf0
deallocated by thread 10124:
      #00 pc 00000000000493a8  /apex/com.android.runtime/lib64/bionic/libc.so
 (scudo::Allocator<scudo::AndroidConfig,
      #01 pc 00000000000442b4  /apex/com.android.runtime/
lib64/bionic/libc.so (scudo::Allocator<scudo::AndroidConfig,
 &(scudo_malloc_postinit)>::deallocate(void*,
```

```
      #02 pc 000000000000d340  /data/app/~~8QwzBMNBT6U1-xi0qH9OdQ==/
com.example.mte_test-AE8qJx9ASOlRNa8HmD3iJA==/lib/arm64/libmte_test.so
      #03 pc 000000000021a354  /apex/com.android.art/lib64/libart.so
 (art_quick_generic_jni_trampoline+148) (BuildId: 2d8a73ff5c99d5a227b31111c86db3e6)
```

The backtrace provides information in columns, as follows:

1. Frame number.
2. PC value. PC values are relative to the location of the shared library rather than absolute addresses.
3. Name of the mapped region. This is usually a shared library or executable, but might also be JIT-compiled code
4. If symbols are available, the unwinder shows the symbol that the PC value corresponds to, along with the offset into that symbol in bytes.

Several lines later, there is a message that describes the cause of the crash, `Use After Free`, and shows the output from the unwinder:

```
Note: multiple potential causes for this crash were detected, listing them in
 decreasing order of likelihood.
Cause: [MTE]: Use After Free, 42 bytes into a 128-byte allocation at 0x782d277bf0
deallocated by thread 10124:
      #00 pc 00000000000493a8  /apex/com.android.runtime/lib64/bionic/libc.so
 (scudo::Allocator<scudo::AndroidConfig,
      #01 pc 00000000000442b4  /apex/com.android.runtime/
lib64/bionic/libc.so (scudo::Allocator<scudo::AndroidConfig,
 &(scudo_malloc_postinit)>::deallocate(void*,
      #02 pc 000000000000d340  /data/app/~~8QwzBMNBT6U1-xi0qH9OdQ==/
com.example.mte_test-AE8qJx9ASOlRNa8HmD3iJA==/lib/arm64/libmte_test.so
      #03 pc 000000000021a354  /apex/com.android.art/lib64/libart.so
 (art_quick_generic_jni_trampoline+148) (BuildId: 2d8a73ff5c99d5a227b31111c86db3e6)
```

For more information about how to interpret bug reports, see the following Android documentation resources:

- Android OS Documentation: Diagnosing Native Crashes
- Android OS Documentation: Debugging Native Android Platform Code

5.4 Tombstones

When your application crashes, a basic crash dump is written to the **Logcat** window in Android Studio. More detailed information is written to a tombstone file, located in the `/data/tombstones/` directory. This tombstone file contains detailed data about the crashed process, including 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

At the crash event, the **Logcat** window in Android Studio shows information about the location of the tombstone file. For example:

```
Tombstone written to: /data/tombstones/tombstone_07
```

To retrieve the tombstone file, use `adb bugreport` to capture the bug report as described in Capture a bug report using Developer Options on your Android device. After unzipping the bug report file, the tombstone files are in the folder `/FS/data/tombstones/`.

For more information about tombstone files, see Android OS Documentation: Crash dumps and tombstones.

6. Debugging with Android Studio and MTE

Android Studio lets you debug with MTE enabled to detect memory bugs in your applications. When MTE detects a memory bug and exits the application, Android Studio signals the exact line of code responsible for the error.

To use Android Studio with MTE enabled, do the following:

1. Enable MTE for your Android project. See Enabling MTE in Android projects for information about the various ways you can do this.
2. Enable **Developer Options** on your Android device. For more information about how to do this, see Android OS Documentation: Configure on-device developer options.
3. Enable either USB debugging or wireless debugging, depending how you connect your device to Android Studio. See Android OS Documentation: Enable USB debugging on your device or Android OS Documentation: Connect to a device over Wi-Fi for more information.
4. Connect your device and wait until the toolbar shows the device.
5. Click the **Debug** button to launch your application.

   When the application encounters a memory bug, MTE causes the application to exit. Android Studio debugger indicates the line of code responsible for the memory violation, as the following figure shows:

Figure 6-1: Memory bug detected by MTE in Android Studio debugger

To test this feature, you can write a simple piece of code that implements a memory bug and link the code to a button. For example:

```
extern "C"
JNIEXPORT void JNICALL
Java_com_example_mte_1test_MainActivity_useAfterFreeC(JNIEnv *env, jobject thiz) {
    // TODO: implement useAfterFreeC()
    char * volatile p = new char[10];
    delete[] p;
    p[5] = 42; //Trying to access an array element that no longer exists!
}
```

When you click the button that executes this code, the application exits and the debugger stops at the following line of code:

```
delete[] p;
```

For more information about debugging applications with Android Studio on MTE devices, watch the Introduction to Memory Tagging Extension video.

7. Integrating MTE with memory management systems

Unity has created its own memory management system to optimize memory usage and minimize the impact of memory handling on performance. This section describes Unity's experience in implementing MTE into their memory allocators.

The Unity memory management system implements a centralized memory manager that is used for all user-space allocations. The memory manager uses several underlying types of allocators, which are selected by category when allocating memory. When created at run-time initialization, these allocators define a low-level backing allocator. The low-level backing allocator is usually backed by paged virtual memory, but can be of any type such as standard C library malloc calls.

7.1 Adding memory tagging to existing memory management code

The approach used by Unity consists of two parts:

1. MTE utility implementation is a namespace of utility functions. A low-level utility implementation that performs memory tagging and related functionality. This can be used outside the scope of memory allocation, for example when a developer wants to check particular pieces of code in specific scenarios.
2. MTE allocator wrapper is an allocator wrapper that is used by the memory manager as a top-level allocator. It inherits existing underlying allocators and uses the MTE utility implementation to perform memory tagging related code before and after the corresponding base method is called. Therefore each allocator it inherits is unaware of the memory tagging and no allocator-specific code is needed in the existing allocators.

7.1.1 The MTE utility implementation

The MTE utility implementation is a collection of namespaced static methods which perform the following functions:

- System initialization
  - Initial setup, called once at startup
- Tagged pointers
  - Filtering with and without tagged bits
  - Extracting tag ID from pointer, which is useful for retagging pointers
- Memory addresses
  - Checking whether an input address is tagged or a real address
  - Examining the tag granule

- Tags
  - Tagging and untagging memory
- Memory copying
  - Copying data to and from tagged memory when alignment is off.

All the utility methods are low-level static functions that have no memory allocator awareness.

Each platform can conditionally include a specific implementation. If MTE is disabled, all functions are no-ops except for memory copying, which falls back to a standard memory copy.

Developers can use the utility functions to temporarily perform tests where a specific area of memory can be tagged and tested. However, the main purpose of the MTE utility implementation is to serve the allocator wrapper, so that it does not have to be aware of the specific MTE implementation.

7.1.2 The MTE allocator wrapper class

At run-time initialization of the memory manager, an allocator checks whether it is MTE-compliant. It does this by checking that it is using virtual paged memory for backing.

If an MTE target build is enabled and is compliant, the allocator is wrapped using the allocator wrapper class. This is the least intrusive implementation because, with a few minor exceptions, it does not require modifications to either the high (manager) or low (allocator) code.

The wrapper also checks if a particular allocation meets the requirements of tagged memory. That is, the allocation must be aligned and have the size granularity stated by the utility class.

When the size is not aligned, an exception occurs. However, the base allocator is designed to over-allocate enough memory for the requirement to be met, so the memory can be tagged.

7.2 Functional tests

The allocator wrapper is tested like any other allocator, but with extra tests for granularity requirements. Functional testing is performed using a fake allocator fixture to return specific results, and ensuring the allocator behaves as expected.

The utility implementation is very low-level and performs basic tests to ensure its implementation is working as expected. Additionally, a test using a signal handler to trap `SEGV_MTE*ERR` errors is implemented to ensure the system is active and working.

7.3 Performance tests

The utility implementation has performance tests which include the following:

- Tagging and untagging memory regions, both small 64 byte blocks and larger 4k blocks. This checks that there is no particular overhead except the actual tagging loop. These test the `STG` and `ST2G` instructions.
- Reading address information. This tests the `LDG` instruction.
- Reading and writing to tagged memory. The results are then compared to non-MTE builds on the same device.

7.4 Considerations and tips when implementing MTE

This section contains advice for anyone implementing MTE.

7.4.1 Overallocating vs untagged allocations

As described in The MTE allocator wrapper class above, Unity does not tag allocations that do not meet the MTE tagging requirements. This means that there are allocations that will not be tagged if the base allocator allocates on a size granularity less than the MTE granularity. This, with the exception of the base allocator itself, performs over allocation, and the MTE tagging size fits within the actual allocated memory as opposed to the requested size.

Another way of dealing with this would be to increase the size to meet the granulation size requirement. However, this changes the memory footprint and layout of the original code where there is no tagging. There are both advantages and disadvantages with each solution. Unity decided to, at least initially, use the first approach and keep as close to the original memory footprint and layout as possible.

7.4.2 MTE should be considered a complement, not a replacement of existing tools

As stated in the previous section, there are situations where MTE does not catch faults.

MTE might not tag a specific allocation due to size restrictions, or if the second approach is chosen, the memory footprint and layout can mask the issue.

However, MTE is extremely powerful to start with. Its primary benefit is being able to stop at the instruction where the actual fault happens, as opposed to post-verifying when deallocating memory for example.

7.4.3 Use ST2G when possible

Over 2.5x performance was measured when using `ST2G`.


Note

Even though `ST2G` has a granule of 32 bytes, it needs only to align to 16 bytes, which makes it easier to use than first anticipated.

7.4.4 Understand PROT_MTE

Ensure that virtual pages are protected using `PROT_MTE` and that they stay protected in your system as expected. See documentation on how this works with regards to mprotect/advice so that you are aware of conditions when protection is disabled. This may seem obvious but there are loopholes relating to, for example, madvice that are easy to miss.

7.5 Alternative approaches for MTE implementation

The following approaches were also considered when implementing MTE, but fell short compared to the chosen approach.

7.5.1 Memory manager implementation, high-level

This approach initially seemed to be the bets option because it would be the least intrusive approach code-wise to the existing system. Current allocators need not be aware of memory tagging, the actual MTE code would take place before and after any calls to the current allocators.

However, it quickly became complicated because the memory manager itself is quite extensive in functionality and there was not any optimal way of differentiating between system-tagged memory and user-space memory. Although this approach would have been possible, we started to look at other alternatives.

7.5.2 Per allocator type implementation, low-level

This approach is best for performance because each allocator handles memory differently, which is their reason for existing. Examples of optimizations are:

- Partial tagging on reallocation
- Disregarding untagging on deallocation under certain circumstances

However, this is the most intrusive approach and each allocator has to implement MTE on most methods. In many cases tests would need to be modified. There is also a major maintenance cost associated with any new allocators requiring code changes.

8. Related information

Here are some resources related to material in this guide:

- Delivering enhanced security through Memory Tagging Extension
- Memory Tagging Extension White Paper
- Android OS Documentation: Arm Memory Tagging Extension
- Android OS Documentation: Diagnosing Native Crashes
- Android OS Documentation: Debugging Native Android Platform Code