# ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety

KOSTYA SEREBRYANY

Konstantin (Kostya) Serebryany is a Software Engineer at Google. His team develops and deploys dynamic testing tools, such as AddressSanitizer, MemorySanitizer, ThreadSanitizer, and libFuzzer. Prior to joining Google in 2007, Konstantin spent four years at Elbrus/MCST working for Sun compiler lab and then three years at Intel Compiler Lab. Konstantin holds a PhD from Moscow State University of Economics, Statistics, and Informatics and an MS from Moscow State University.
kcc@google.com

I discuss memory safety bugs typical to C and C++, current tools and approaches to finding such bugs or mitigating their risk, and a new hardware feature, ARM MTE, that promises to be the biggest improvement since the introduction of page protection.

## Memory (Un)safety

More than 30 years after the Internet Worm, we are still talking about memory safety bugs in C and C++ programs. Numerous improvements in the software development process are dwarfed by the exponential increase in the amount of software, its exposed attack surface, and the discovery of new attack techniques.

*Memory safety bug* is an umbrella term to represent program defects inherent in C and C++ but also present in other languages. The most common classes of bugs are buffer overflows, heap-use-after-free, and stack-use-after-return.

These bugs often make the code vulnerable to exploitation. Malicious actors can leverage memory-unsafe behavior to remotely execute code, leak sensitive information, escalate privileges, or escape VMs. A buffer overflow in OpenSSL, nicknamed Heartbleed, achieved notoriety for its ease of exploitation and high impact. It allowed attackers to steal a server's private memory, including cryptographic information such as keys and passwords, without being detected. But *named* bugs like Heartbleed and Stagefright, a family of remotely exploitable bugs in Android, are just the tip of the iceberg.

Thousands of memory safety bugs are filed as CVEs every year. Roughly two-thirds of all CVEs in the Android platform are memory safety bugs. A similar picture is seen across the industry, affecting browsers, operating systems, and server-side and IoT software [1, 2]. And even these bugs are still the tip of the iceberg. Many more bugs do not get CVEs assigned, and many others remain unknown to software vendors. Some are being silently exploited, others cause hard to detect data corruption, and some lie dormant waiting to strike.

### *Typical Bugs*

Before we dive deeper, let's take a closer look at two of our most beloved insects.

A **heap-buffer-overflow** happens when an object of a certain size is allocated on the heap, and then a pointer to this object is used to access memory outside of the object bounds. Typically, the object is an array of *n* elements, and the code accesses the i-th element where i < 0 or i >= n.

```
int *array = new int[n];  // heap allocation
array[n] = 42;  // buffer overflow
array[-1] = 42; // buffer overflow (underflow)
array[100500] = 42;  // buffer overflow, assuming n <= 100500
```

A **heap-use-after-free** happens when an object is allocated on the heap, and later deallocated, but a pointer to the object is preserved somewhere and is used to access the deallocated memory.

```
Object *obj = new Object;   // heap allocation, or "malloc"
delete obj;             // heap deallocation, or "free"
obj->member = 0;        // heap-use-after-free, or
                        // access via a dangling pointer
```

In both cases the buggy memory access touches someone else's memory. In the C and C++ standards this is considered undefined behavior. In real life it may cause a loud crash, a silent data corruption, or a convenient back door.

## Existing Tools and Practices

We haven't been exactly ignoring the problem for 30 years.

Coding practices and testing tools reduced the likelihood of introducing a memory bug. A test-driven development process together with dynamic testing tools like AddressSanitizer [3] or Valgrind will help avoid many bugs. Fuzzing (and, ideally, fuzz-driven development [4]) will pick up the next layer of bugs. Some memory bugs can be spotted by static analysis.

Software-based code-hardening techniques make it harder for attackers to exploit memory safety bugs that reach production. Stack cookies, non-executable memory, ASLR, control flow integrity (LLVM CFI, Microsoft CFG, Shadow Call Stack), and other techniques help prevent memory safety bugs from diverting program control flow, the end goal of many exploits. Hardened memory allocators, such as Scudo Hardened Allocator or Chrome's Partition Alloc, frustrate exploitation and may make it impossible in some cases.

Hardware-based solutions have begun to appear as well. ARM Pointer Authentication, already available in the most recent Apple hardware, cryptographically authenticates return addresses and discourages attackers from using return-oriented programming (ROP). Intel Control-flow Enforcement Technology is expected to appear soon to solve ROP in a different way, by keeping the return address on a separate stack with special permissions.

All these tools are making our software more stable and secure, but they are not enough. No amount of testing guarantees the absence of bugs, and existing exploit mitigations only prevent some attacks, while almost entirely ignoring others, e.g., data-oriented attacks.

Among the hardware-based solutions two stand out, SPARC ADI and ARM MTE, both implementations of a concept known as **memory tagging** or memory coloring. SPARC ADI has been available in mass-produced hardware since 2016; we covered this feature in an earlier paper [5]. This article focuses on ARM MTE.

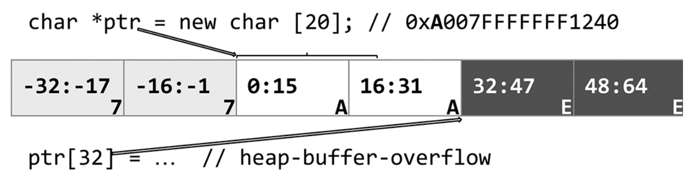


**Figure 1:** Heap-buffer-overflow is detected by MTE because the pointer's address tag 0xA does not match the memory tag 0xE.

## ARM MTE

On September 2018 ARM announced the **Memory Tagging Extension**, or MTE [6], a part of the ARM v8.5 architecture. It does not yet exist in real hardware, but everything else about this extension is very promising.

The extension introduces a notion of two types of tags: address tags and memory tags.

An **address tag** is a 4-bit value stored at the top of every pointer in the process. MTE utilizes *top-byte-ignore*, an existing AArch64 feature that instructs the hardware to ignore the topmost byte of addresses, allowing this byte to be used as user-controlled metadata. Therefore MTE is applicable only to 64-bit software.

A **memory tag** is a 4-bit value associated with every aligned 16-byte region of application memory (*memory granule*). The way memory tags are stored is a hardware implementation detail. Logically, every 16 bytes of memory now contain an extra 4 bits of metadata in addition to 128 bits of data.

Every time a heap region is allocated, the software chooses a random 4-bit tag and marks both the address *and* all the newly allocated memory granules with this tag. The load and store instructions verify that the address tag matches the memory tag, causing a hardware exception on tag mismatch. MTE introduces new instructions to manipulate the tags.

Let's look at the example in Figure 1. When the user code requests 20 bytes of heap to be allocated, operator `new()` rounds up the size to the 16-byte boundary (i.e., to 32), allocates a 32-byte chunk of memory (i.e., two 16-byte memory granules), chooses a random 4-bit tag (in this case, 0xA), puts this tag into the top-byte of the address, and updates the tags for the two newly allocated memory granules (the white-colored regions in the diagram). The adjacent memory regions have different memory tags (light gray granules have the tag 0x7, dark gray granules have the tag 0xE), so when the code tries to access memory at offset 32 from the pointer, MTE raises an exception because the tag of the pointer does not match the tag of the memory granule being accessed.

Figure 2 demonstrates an example of how heap-use-after-free is detected. On deallocation, operator `delete()` changes the tag of all three deallocated granules of memory from 0xD to 0x4,

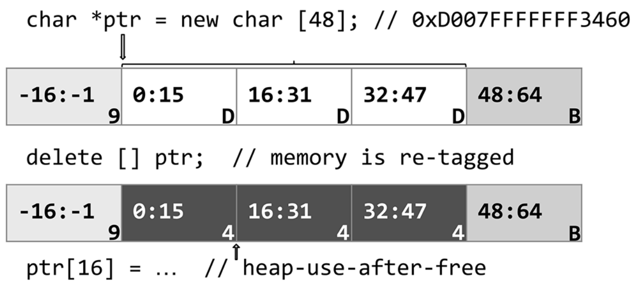


**Figure 2:** Heap-use-after-free is detected by MTE because the pointer's address tag 0xD does not match the memory tag 0x4.

so that any access to this memory via an old (*dangling*) pointer causes an exception because the pointer still has the old tag 0xD. The adjacent memory regions (tagged with 0x9 and 0xB) are not affected by retagging of this region.

You may have noticed that bug detection with MTE is probabilistic. Indeed, there are only 16 possible values of a 4-bit tag. One random tag will be different from another random tag with a probability of 15/16 or ~93%. It is up to the software to decide whether to increase this probability with other tricks. For example, in order to detect contiguous buffer overflows with perfect accuracy, the allocator may enforce that tags for adjacent chunks are never equal.

With MTE, the heap memory is tagged inside `malloc()` and `free()`, and the tag checking is performed by the hardware. It means that recompilation will not be required for detecting heap-related bugs. MTE can also identify stack-use-after-return and buffer overflows on the stack or in global variables, but it will require recompilation with extra compiler options.

### *Comparison with AddressSanitizer*

AddressSanitizer is a widely used tool for detecting memory safety issues. It uses compiler instrumentation to observe all loads and stores. Its specialized malloc "poisons" *red zones* around heap objects to detect buffer overflows and keeps freed memory in *quarantine* to detect use-after-free. The red zones and the quarantine are the major causes of AddressSanitizer's high memory overhead.

MTE is conceptually similar to AddressSanitizer: both detect bugs at runtime, both require special functionality in malloc and free, and both require some amount of compiler support.

However, the use of address tags makes MTE sufficiently different: it does not require red zones or quarantine to detect bugs. This allows MTE to consume less memory. Moreover, MTE performs checking in hardware, thus eliminating the overhead of compiler instrumentation for every load and store.

Compared to AddressSanitizer, MTE brings the following benefits:

- MTE checking can be turned on and off at runtime.
- CPU overhead is expected to be very small, hopefully a small single-digit percentage, while AddressSanitizer typically has 2x–3x slowdown.
- MTE can find heap-related bugs without recompilation.
- Due to the small overhead, the same binary can be used for testing and for production.
- MTE's memory overhead is 3%–5%, compared to 2x–3x for AddressSanitizer.
- Memory accesses that happen far from the object bounds or long after the object lifetime are more likely to be spotted by MTE than AddressSanitizer, which makes MTE a better exploit mitigation.

The only downside of MTE is that it may fail to detect buffer overflows that happen within the 16-byte granule:

```
char *array = new char [13];  // allocates one 16-byte granule
array[14] = 0;  // access within the same 16-byte granule
```

Various software strategies are possible to improve bug detection for such cases with additional cost or complexity.

### *Uses of MTE*

We envision several different usage modes for MTE.

First, MTE is going to be a much nicer version of AddressSanitizer for testing and fuzzing. It will find more bugs at a fraction of the cost. In many cases it will allow testing using the same binary as shipped to production.

Second, MTE could be used as a mechanism for testing in production (e.g., crowdsourced bug detection), always-on or enabled randomly. For client software, such as web browsers, it means that when a bug happens on a user device it will be detected, and, with user consent, an actionable bug report will be sent to the vendor. For server-side software it means that even the rarest bugs will be detected immediately once they get triggered.

Finally, MTE can be seen as a strong security mitigation. It is true that it prevents exploitation with less than 100% probability, but the probability is still very high, and the first failed exploitation attempt will warn the user and the software vendor. We believe that memory tagging will detect the most common classes of memory safety bugs in the wild, helping vendors identify and fix them and discouraging malicious actors from exploiting them.

Other clever ways to use MTE will likely be discovered. MTE may allow building debuggers with infinite hardware watchpoints, efficient race detectors, or faster garbage collectors.

### *HWASAN*

The full potential of memory tagging will only be available with future hardware, several years from now. But you can reap some of the benefits now, like significantly reduced memory consumption, by using a software implementation of memory tagging: HWASAN (hardware-assisted AddressSanitizer) [7]. HWASAN is similar in spirit to AddressSanitizer, but its smaller memory footprint makes it a better choice on memory-restricted devices, such as mobile phones. Today, the tool only supports 64-bit ARM CPUs, since it requires the top-byte-ignore feature and a small modification in the kernel to allow passing tagged addresses to system calls.

### *Compatibility*

MTE and HWASAN offer a high level of compatibility with existing code bases. We build the Android platform and the Chromium browser with HWASAN with few source code changes.

However, we have observed several cases of incompatibility. In one such case, pointers to a particular type had application-specific metadata stored in the top 16 address bits. In another case, a pointer was cast to double and then back, losing the lower address bits. In one more case, the code computed difference between the addresses of local variables from different stack frames as a way to measure recursion depth. All these cases were easy to fix.

## Related Work

With this article I hope to increase the awareness of the concept of memory tagging, as well as ARM's fantastic Memory Tagging Extension, so that other CPU vendors adopt it sooner rather than later. Unlike most other existing hardware security extensions, ARM MTE directly addresses the memory safety bugs, that is, the root cause of many vulnerabilities, not just how attackers happen to exploit their consequences today. Beyond its effectiveness as a mitigation, MTE also serves as an effective bug detection tool that can be deployed in the wild. But even MTE is not a panacea for all classes of memory safety bugs.

### *Intra-Object-Buffer-Overflow*

There are other classes of C/C++ bugs waiting to be dealt with. One such bug class is called *intra-object-buffer-overflow*.

```
struct S {
  int array[5];
  int another_field;
};

int GetInt(int *p, size_t idx) {
  return p[idx];
}
int Foo(S *s) {
  return GetInt(s->array, 5);
}
```

Here, by accessing an array out of bounds we end up reading another field in the same struct. In this case, AddressSanitizer, HWASAN, or MTE will not find the bug because the access happens within the same heap- (or stack-) allocated object. The Undefined Behavior Sanitizer (UBSan) can detect some simper cases, but not the more complex ones like this one because the function `GetInt()` that accesses the memory has lost the static bound information available in `Foo()`. There were multiple attempts to solve this problem (including at least one hardware extension, Intel MPX), but none were practical enough to be widely used.

A potential solution would combine dynamic bounds checking, static analysis (proving that either the code is correct or that dynamic checks are effective), and the banning of certain language constructs (like passing sub-objects without their bound information to unknown functions). For modern C++ code, perhaps the best solution is to replace arrays inside structs or classes with std::array and rely on the runtime for bounds checking.

### *Type-Confusion*

Another bug class not directly addressed by MTE is *type-confusion*.

```
struct Image {
  int pixels[100];
};

struct Secret {
  int sensitive_data[200];
};

Secret *secret = new Secret;
...
DrawOnScreen((Image*) secret);
```

This code performs a cast between incompatible types; the following memory accesses in `DrawOnScreen()` will mistakenly access sensitive data without violating object bounds or lifetimes.

A potential solution is to use a stricter subset of C++ that disallows some invalid casts statically (via compile-time errors) and some other invalid casts dynamically (using a mechanism such as implemented in LLVM CFI).

### *Uninitialized Memory*

A side effect of MTE is that whenever a memory allocation is tagged, it can also be initialized at no extra cost. The new ARM instructions can store memory tags and initialize the memory itself at the same time. Therefore, enabling MTE for an application's heap and stack will mitigate most vulnerabilities from another class, *uses of uninitialized memory*.

However, we do not have to wait for MTE to eradicate this class of bugs. For example, Clang/LLVM 9.0 will have an option [8] to automatically initialize all stack variables.

### *Safer Languages*

No discussion of memory safety in C and C++ can ignore the existence of "safe languages." Java, Go, Swift, and Rust, among others, are indeed much *safer*, and in many cases they are a better choice for developing new software.

But none of them are really *safe*. Go and Swift have data races, Java's huge runtime is itself written in C++, and only Rust comes close to being safe, at a cost of a (subjectively) steeper learning curve.

All of these languages, of course, have the "unsafe" escape hatch. Whenever the unsafe section is used, it turns the language into C, but just slightly worse, because fewer tools, practices, and habits are available for that language to avoid memory safety bugs. Here, again, Rust is probably the best with its support for AddressSanitizer and fuzzing. MTE will be useful for Rust and any other memory-safe language with "unsafe" code.

Besides, the billions of lines of C and C++ code are not going away any time soon.

### *GWP-ASan*

GWP-ASan [9] is another bug detection tool that finds heap-use-after-free and heap-buffer-overflows. It relies on protected guard pages, the old trick used in the Electric Fence Malloc and similar tools. But there is a twist: guarded allocations are sampled. This means that the overhead, and the bug detection probability, can be scaled to be arbitrarily small. The small probability of bug detection can be improved by deploying the tool at large scale in production. We are beginning to detect bugs this way in the Google Chrome browser and other software.

GWP-ASan is not a replacement for AddressSanitizer or HWASAN since it handles a smaller subset of bugs and has very low detection probability, but it finds bugs that evade testing and only manifest in production. In the most performance-critical applications, where even 1% overhead is prohibitively expensive, we will be able to use MTE to implement sampled bug detection similar to GWP-ASan, but with a much lower cost and hence higher sampling and detection rate.

## Conclusion

Once available in hardware, the ARM Memory Tagging Extension will reduce C and C++ memory unsafety from disastrous to tolerable. Hopefully, other hardware vendors will implement their variants of memory tagging. Before that happens, don't forget to test your software with all available testing tools (e.g., AddressSanitizer or HWASAN) and fuzzers (e.g., libFuzzer), and harden your binaries in production.

## Acknowledgments

I want to thank my colleagues Vlad Tsyrklevich, Dmitry Vyukov, Alexander Potapenko, and Evgeniy Stepanov for helping me prepare this article.

### *References*

[1] K. Serebryany, "Hardware Memory Tagging to Make C/C++ Memory Safe(r)," iSecCon'18: https://github.com/google/sanitizers/blob/master/hwaddress-sanitizer/MTE-iSecCon-2018.pdf.

[2] M. Miller, "Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape," BlueHat 2019: https://www.youtube.com/watch?v=PjbGojjnBZQ.

[3] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," 2012 USENIX Advanced Technical Conference (USENIX ATC '12): https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf.

[4] K. Serebryany, "OSS-Fuzz—Google's Continuous Fuzzing Service for Open Source Software," 26th USENIX Security Symposium (USENIX Security '17): https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany.

[5] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, D. Vyukov, "Memory Tagging and How It Improves C/C++ Memory Safety": https://arxiv.org/pdf/1802.09517.pdf.

[6] Arm A-Profile Architecture Developments 2018: Armv8.5-A: https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a.

[7] HWASAN documentation: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[8] J. F. Bastien, "Automatic Variable Initialization": https://reviews.llvm.org/D54604.

[9] GWP-ASan for Chromium documentation: https://chromium.googlesource.com/chromium/src/+/lkgr/docs/gwp_asan.md.