MEMES: Memory Encryption-based Memory Safety on Commodity Hardware

David Schrammel[*], Salmin Sultana[†], Karanvir ("Ken") Grewal[†], Michael LeMay[†], David M. Durham[†], Martin Unterguggenberger[*], Pascal Nasahl[*] and Stefan Mangard[*]

[*]*Graz University of Technology*

[†]*Intel Labs*

[*]{*firstname.lastname*}*@iaik.tugraz.at*

[†]{*firstname.lastname*}*@intel.com*

Keywords: Fine-Granular Memory Encryption, Memory Safety, Exploit Mitigation, Intel® TME-MK

Abstract: Memory encryption is an effective security building block broadly available on commodity systems from Intel® and AMD. Schemes, such as Intel® TME-MK and AMD SEV, help provide data confidentiality and integrity, enabling cryptographic isolation of workloads on shared platforms. However, due to their coarse encryption granularity (*i.e.*, pages or entire virtual machines), these hardware-enabled primitives cannot unleash their full potential to provide protection for other security applications, such as memory safety. To this end, we present a novel approach to achieving sub-page-granular memory encryption without hardware modifications on off-the-shelf systems featuring Intel®'s TME-MK. We showcase how to utilize our fine-grained memory encryption approach for memory safety by introducing MEMES. MEMES is capable of mitigating both spatial and temporal heap memory vulnerabilities by encrypting individual memory objects with different encryption keys. Compared to other hardware-based memory safety schemes, our approach works on existing commodity hardware, which allows easier adoption. Our extensive analysis attests to the strong security benefits which are provided at a geometric mean runtime overhead of just 16–27%.

1 INTRODUCTION

In modern CPUs, memory encryption is a ubiquitous technology protecting either confidentiality alone or confidentiality and integrity of data in the external memory (DRAM). Major vendors, such as Intel® and AMD, offer a wide variety of different protection solutions. For example, AMD's secure encrypted virtualization (SEV) (AMD, 2020) technology and the secure nested paging (SEV-SNP) extension enable the CPU to encrypt data with different encryption keys to protect confidentiality (Kaplan et al., 2020). Intel®'s total memory encryption (TME) engine also enables the system to transparently encrypt data in DRAM using a single encryption key. The total memory encryption - multi-key extension (TME-MK) enhances TME with multiple keys (Intel®, 2022). Furthermore, with the introduction of Intel® TDX (Intel®, 2023), Intel®'s memory encryption engine also provides authenticated encryption to help enforce data integrity.

However, the use cases for memory encryption on commodity hardware are currently limited by the coarse granularity of the underlying memory encryption engine. More specifically, both Intel®'s and AMD's solutions are intended to encrypt memory with different encryption keys on the page granularity. At this granularity, current use cases focus on providing protection against physical adversaries, e.g., performing cold boot attacks (Halderman et al., 2008) or cryptographic isolation of virtual machines on shared platforms. Other applications, such as memory safety, require much finer granularity encryption (*i.e.*, object granular). Thus, we identify the following research questions:

R1: *Can off-the-shelf memory encryption be leveraged to enforce memory safety on commodity systems?*

R2: *Can we achieve sub-page encryption granularity to protect individual memory objects within the same page?*

In this paper, we introduce an approach based on page table aliasing, allowing us to achieve **fine-grain memory encryption** without hardware changes on CPUs supporting Intel®'s TME-MK. The basic idea of page table aliasing is to map the virtual memory to the same physical memory multiple times with different encryption keys. Using this technique, we show that the encryption granularity of TME-MK is only limited by the block size of the underlying encryption primitive, *i.e.*, 128-bit for AES.

Moreover, we introduce MEMES (**M**emory **E**ncryption-based **ME**mory **S**afety), a software scheme leveraging fine-granular memory encryption for full memory safety on commodity hardware. At its core, MEMES encrypts data objects on the heap with different encryption keys to comprehensively address memory safety vulnerabilities. On memory safety errors, depending on the underlying capabilities of the encryption engine, MEMES either triggers an error or probabilistically prevents the exploitation. Thus, MEMES can provide security similar to memory tagging but without the associated memory overhead of traditional tagging schemes. We integrate MEMES into a runtime library to automatically protect memory objects on the heap. Finally, we discuss the security benefits of our encryption-based memory safety scheme and evaluate its runtime overhead using the SPEC CPU® 2017 benchmarks.

Contributions. In short, our contributions are as follows:

- We introduce a mechanism that enables off-the-shelf CPUs featuring Intel®'s TME-MK to achieve much more fine-granular memory encryption.
- Based on this mechanism, we design MEMES to protect individual memory objects from memory safety vulnerabilities.
- We thoroughly analyze the security benefits and limitations of our design.
- We implement our design as a binary-compatible heap allocator and show its performance and memory characteristics.

Outline. The paper is structured as follows: Section 2 describes the required background on Intel®'s TME-MK, virtual memory, and memory safety. Section 3 and Section 4 provide the design and implementation of MEMES. Section 5 and Section 6 analyze MEMES in terms of security and performance overhead. Section 7 discusses related work, and Section 8 concludes this paper.

2 BACKGROUND

This section provides background on memory safety vulnerabilities, virtual memory, and TME-MK.

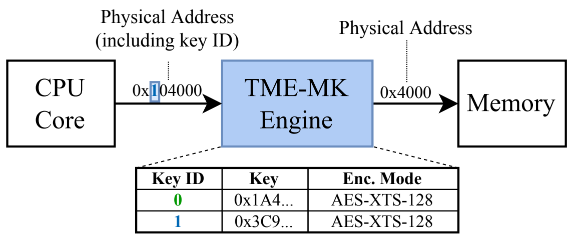
2.1 Memory Safety

Memory safety errors are a persistent problem of software written in unsafe programming languages, such as C and C++. Among all security bug fixes in Windows and Chrome, 70% of them are due to memory safety vulnerabilities (Microsoft, 2019; The Chromium Projects, 2020). These vulnerabilities enable an adversary to hijack the control flow and compromise the target system, e.g., return-oriented programming (ROP) (Shacham, 2007) and jump-oriented programming (JOP) (Bletsch et al., 2011). Furthermore, an adversary can corrupt data stored in memory (data-oriented programming (DOP) (Hu et al., 2016)) in order to attack the system. Besides control flow and data-only attacks, an adversary can also leak sensitive or secret data (Durumeric et al., 2014). For example, the Hearbleed security bug (Durumeric et al., 2014) enabled remote adversaries to leak data on web servers.

Generally, we distinguish between two types of memory safety: spatial and temporal memory safety (Szekeres et al., 2013). *Spatial* memory violations exceed the intended bounds of a memory object (e.g., buffer underflow or overflow into an adjacent memory object). Besides adjacent memory safety violations, out-of-bounds memory accesses (*i.e.*, arbitrary reads or writes) allow the adversary to tamper with data located on an arbitrary memory location. In contrast, *temporal* memory safety violations target the object's liveness. Use-after-free (UAF) is an error where a dangling pointer, *i.e.*, a pointer referring to an already freed memory object, exists. The adversary can misuse this dangling pointer to manipulate data located at the same memory location. Additionally, uninitialized memory access violations allow an adversary to leak secret data.

2.2 Virtual Memory

Modern operating systems rely on virtual memory, which allows them to isolate multiple processes in their own virtual address space. Applications are presented with a virtual view of the physical memory. Such virtual (linear) addresses are translated by the CPU to physical addresses. Specifically, the memory management unit (MMU) translates these addresses using page tables that are configured by the operating system (OS). In most systems, the virtual address space is divided into fixed-size, e.g., 4 kB, contiguous memory blocks, *i.e.*, pages. For each page, address translation information is stored in the page table entries (PTE) of a page table. This information comprises the mapping from virtual to physical ad-



Key ID	Key	Enc. Mode
0	0x1A4...	AES-XTS-128
1	0x3C9...	AES-XTS-128

Figure 1: The TME-MK memory encryption engine processes data transferred between the CPU core and the memory controller. The key identifier encoded in the upper bits of the physical address selects the key used for data encryption.

dresses as well as other metadata, such as permission bits. During a memory access, the system conducts a page table walk to resolve the physical address. As these page table walks are costly, address translation information of frequently accessed memory objects are cached in the CPU's translation lookaside buffers (TLBs).

2.3 Intel® TME-MK

Intel®'s *total memory encryption - multi-key* (TME-MK) (Intel®, 2022) is a hardware feature introduced with the Ice Lake platform. TME-MK enables the system to transparently encrypt all data that is stored in the DRAM with multiple different encryption keys.

As illustrated in Figure 1, the encryption engine is located between the core and the memory controller. Internally, the engine maintains a mapping from key identifiers to encryption keys and encryption modes. Depending on the used platform, a maximum of 2^{15} keys are available. TME-MK uses the key identifier embedded into the upper previously unused physical address bits to encrypt or decrypt each data object with the corresponding encryption key. The key identifier and the rest of the physical address are set in the page table entry. Hence, currently, the intended TME-MK encryption granularity is page granular.

For the encryption, TME-MK currently uses AES in the XTS mode with either 128-bit or 256-bit keys. It uses the physical address as an input such that same data in different locations is encrypted differently. In this mode of operation, TME-MK provides data confidentiality for the DRAM. The memory encryption engine used in Intel®'s TDX (Intel®, 2023) enhances TME-MK with a SHA-3-based MAC that is stored in the ECC memory to additionally help provide data integrity (Intel®, 2020). For simplicity, in the remainder of this document, we use *key* to denote a key identifier that then maps to the actual AES key material.

3 DESIGN

In this section, we first introduce our novel fine-grain memory encryption mechanism on unmodified hardware. We then introduce MEMES, which utilizes our fine-grain memory encryption approach to mitigate spatial and temporal memory vulnerabilities. As heap vulnerabilities are one of the most dominant memory safety issues (Kim et al., 2020), we focus, in our work, on protecting heap-allocated data. The proposed solution, however, can also be adapted for other types of data.

3.1 Fine-Grain Memory Encryption

Intel® TME-MK is only designed for encrypting entire pages (*i.e.*, 4 kB) of memory (cf. Section 2.3). For memory safety, however, where allocations are much smaller, this is not suitable. To cryptographically protect memory objects from each other, we would need to place them on separate pages to use different keys for each allocation. For allocations that can be as small as 16 B, this means that up to 256x more (physical) memory would be needed with that naïve approach.

In this work, we enhance TME-MK-based page-granular memory encryption to allow for a much finer granularity (e.g., cache line granularity), which in turn greatly improves memory usage efficiency. TME-MK is an inherently page-granular mechanism since the key is encoded in the physical page number that is stored within a page table entry (PTE). However, we can create multiple different PTEs (*i.e.*, aliases) that use the same underlying physical page but with a different encryption key. This provides us with two different virtual addresses for the same physical memory page but with a distinct view of the memory. Depending on the address used, the page is either en-/decrypted with one or the other key. We can scale this to the number of available keys on the platform, which is up to 2^{15}. Since we set the key identifiers in the page tables, which are indexed using virtual addresses, it effectively behaves like encoding a tag or encryption key into the virtual address. Thus, our approach is similar to memory tagging schemes like ARM® MTE (ARM Limited, 2019). However, instead of using dedicated tagged memory hardware, we achieve the same functionality with memory encryption, which already exists in today's hardware. Since we can now access the same page with different keys, we can encrypt smaller blocks of memory with different keys. This allows us to use existing hardware, intended for page-granular virtual machine isolation, to be used for fine-granular use cases like

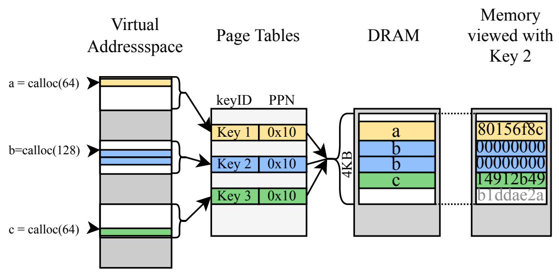


Figure 2: An overview of our scheme using 3 encryption keys. On the left side, the virtual address space of a program is shown. A memory page (shown in white) is mapped multiple times in the virtual address space but with different keys. Cache lines and PTEs are colored distinctly for each key. On the left, three consecutive allocations (a,b,c) are placed in different mappings since each of them is encrypted with a different key. The PTEs show that all three pages map to the same physical page number (0x10), however, the key identifier part of the physical address is distinct. On the right side, the memory page is shown when viewed with key 2 (*i.e.*, with the pointer from allocation *b*). Using that pointer, data adjacent to the allocation only contains garbled data or, in the case of authenticated encryption, an authentication error can be triggered.

memory safety. Figure 2 gives an overview of our approach.

CPU caches are typically tagged with the physical address, which also contain the key identifier. Thus, from a security perspective, it does not matter if physical memory that is used with different keys maps to the same cache set/way or to different ones. At the cost of additional cache evictions, it allows encrypting memory even at a sub-cache line granularity, down to the block size of the encryption engine (*i.e.*, 128 bit). Note that this is the same granularity as ARM® MTE uses.

3.2 Threat Model

MEMES repurposes the TME-MK memory encryption engine to enforce fine-grained memory safety. In this context, we consider a threat model, similar to prior work (cf. AOS (Kim et al., 2020) and C^3 (LeMay et al., 2021)), consisting of an adversary that exploits one or several memory safety vulnerabilities to attack the system. We assume the adversary has access to an arbitrary read-and-write primitive or a stale pointer to mount a software attack. MEMES provides exploit mitigation for user applications. Thus, we assume that the operating system or hypervisor is trusted and free of exploitable bugs. Moreover, as most memory safety vulnerabilities target the heap (Microsoft, 2019), we focus only on protecting objects on the heap. Nevertheless, a similar

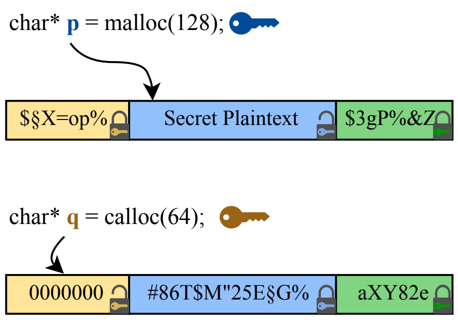


Figure 3: Lock and key access to the memory. Two memory objects p and q are allocated and the pointers map to two different PTEs that have a different key encoded. Accessing the memory objects with the corresponding pointers returns the correctly decrypted plaintext. Adjacent memory encrypted with different keys are inaccessible as garbled data is retrieved or an authentication error may be triggered.

approach can be used to achieve protection for stack and global data by adapting our design.

Furthermore, we expect common countermeasures, such as address space layout randomization (ASLR) and data execution prevention (DEP), to be enabled. This means that the attacker does not know the exact address layout and cannot inject and execute arbitrary code. Additionally, we consider side-channel and microarchitectural attacks out of scope for this work.

3.3 Memory Safety through Encryption

We design our memory safety scheme, MEMES, based on fine-grain memory encryption. At its core, it consists of three parts. First, the TME-MK-based fine-grain encryption mechanism introduced in Section 3.1. Second, a kernel extension that allows TME-MK to be used from userspace applications via syscalls. Finally, a heap allocation library called *TMEalloc*, which leverages the newly added syscalls to provide fine-grained protection of individual heap memory objects.

Heap allocator. A heap allocator is responsible for managing allocations at runtime (e.g., via `malloc`) as well as their associated metadata. It requests pages from the OS and hands them out in smaller chunks, usually 16 B, to the application. In this work, we design our scheme as part of a heap allocator which has some advantages. E.g., it allows our library to be retrofitted to existing pre-compiled binaries. Thus, we do not only achieve mere binary compatibility, but instead, we can provide transparent protection even for unaware libraries.

On a high-level, our allocator (TMEalloc) assigns each allocation a different encryption key. This gives us a lock-and-key access to the memory, as illustrated in Figure 3. Similar to memory tagging, here, the number of keys is also limited by the hardware. Hence not every allocation can have a distinct encryption key. Thus, we provide probabilistic security, similar to ARM® MTE, which can only distinguish 16 tags. However, the allocator can choose the keys such that adjacent allocations always use different encryption keys, which prevents linear buffer overflows deterministically.

To facilitate multiple encryption keys per page, each heap memory page is mapped in the virtual address space, once for each used key on that page. We use the available virtual address space to place these aliases at random locations which prevents guessing a pointer with a different encryption key.

Kernel extension. In order for the heap allocator to utilize TME-MK, we need to add a few syscalls for managing aliased pages with their respective encryption keys. We design our modification for the Linux kernel and use its existing key retention facility for creating and destroying TME-MK keys supported by our system (*i.e.*, AES-XTS-128). Furthermore, we add one new syscall `encrypt_mprotect`, such that the userspace application can apply specific keys for virtual memory ranges.

4 IMPLEMENTATION

We implemented our MEMES design on top of the high-performance heap allocator mimalloc (Microsoft, 2022) developed at Microsoft. In the following, we detail the most important aspects of our implementation.

Aliasing. We use the standard `mmap` syscall with the `MAP_SHARED` which allows creating shared memory. Note that this also allows mapping the same physical memory multiple times *within* the same process. We use this to create heap memory aliases with different encryption keys. For this purpose, we use the previously introduced `encrypt_mprotect` syscall, which allows our allocator to set a unique key for each aliased mapping.

Pointer Layout. Figure 4 shows the layout of a pointer returned from the allocator, which points to one of the aliases. Assuming a heap size of 16 GB, we use the pointer bits 34 through 47 for placing the aliases in the available address space. The upper bits (48 to 63) are set to 0 to conform to the architectural requirements of canonical addresses. The only other requirement here is that the aliased regions do not intersect each other. This is done by placing the respective aliases, in this case, at least 16 GB apart. For applications requiring more memory, this minimum distance can be increased accordingly.

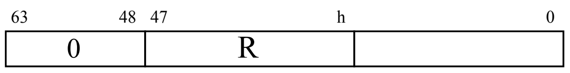


Figure 4: The format of a heap pointer. We use bits [h:47] for randomization of the placement for each key, where $h = log_2(heap, size)$. E.g., $h = 34$ for a heap size of 16 GB.

Granularity. Since cache lines are tagged with the physical address, and thus with the key, it would lead to significant performance degradation if multiple keys per cache line were used. Hence, we align and pad all memory allocation on the heap to the cache line granularity, *i.e.*, 64 B on our system. This trade-off incurs a slight memory overhead for applications relying on very small allocation sizes, which we analyze in Section 6. This also means that any in-line metadata (*i.e.*, heap library metadata that is typically located near the allocated data) must also be aligned.

Optimizations. Our fine-grain encryption design inevitably leads to an increase in required TLB entries. Hence, we opted to base our implementation around 2 MB-sized pages, to alleviate some of the TLB pressure.

Page Zeroing. When a userspace application requests new memory, the kernel wipes the pages by overwriting them with `0` bytes. This is done to prevent any sensitive data leakage across processes. Furthermore, the `mmap` syscall specifies that new memory returned to the application needs to be zero-initialized. Many applications rely on this for performance optimizations. E.g., when calling `calloc`, which also ensures that the returned heap memory is zero-initialized, a heap allocator can simply pass the memory to the user without setting it to zero, if it was a fresh page received from the kernel. However, when using different encryption keys for a single page, it is not obvious which key should be used for zeroing. Because, when viewed with a different key, a zeroed page is no longer zeroed. Hence, in our kernel implementation, we wipe the page as usual with the default key. In addition, we defer zeroing to our heap library, which then uses the correct key for zeroing (e.g., the key used for heap metadata).

5 SECURITY DISCUSSION

In this section, we discuss the security benefits provided by our design in detail. Thereby, we examine how our design facilitates both temporal and spatial memory safety for heap memory during runtime. MEMES provides different security benefits depending on the operation mode of the memory encryption engine, *i.e.*, encryption-only or authenticated encryption.

TME-MK Encryption-Only. By utilizing memory encryption, MEMES provides strong confidentiality for the protected data. On every memory safety violation, data would be accessed with the wrong decryption key, which ultimately leads to reading garbled data. Hence, an adversary cannot leak secret data or inject specific data controlled by the attacker. Thus, the MEMES memory encryption approach provides exploit mitigation for various types of software-based attacks. More specifically, as the attacker does not know the secret encryption key, deterministically writing data, e.g., a return address or a function pointer, with the wrong encryption key is not possible. This effectively mitigates powerful exploitation techniques, such as return-oriented programming (ROP) (Shacham, 2007) and jump-oriented programming (JOP) (Bletsch et al., 2011).

Although by continuously reading the victim data, an attacker could still see *when* the data was changed, but they do not know the old or new plaintext value. An attacker would only be able to roll back the value to a previous, possibly valid, value. However, in practice, this side-channel is hard to exploit and requires strong attacker capabilities, which we do not consider in our threat model. Nevertheless, TME-MK can also protect against this as shown in the following.

TME-MK Authenticated Encryption. TME-MK also allows for an authenticated encryption mode, which is used in TDX (Intel®, 2023). While our system configuration did not make it possible to add such keys to the encryption engine, using it would enhance MEMES with additional detection capabilities. In particular, authentication enables the detection of all classes of memory safety violations, *i.e.*, temporal and spatial memory safety violations, by possibly triggering a hardware exception where we can terminate the program. This means that arbitrary read or write memory violations (e.g., out-of-bounds errors) would result in authentication failures of the TME-MK memory decryption instead of returning garbled data. Similarly, dereferencing dangling pointers (e.g., by mounting a use-after-free (UAF) attack) is detected as well.

Spatial Memory Safety. For the spatial safety analysis, we differentiate between adjacent, non-adjacent, and intra-object memory access violations. The memory allocator can ensure that two consecutive allocations are encrypted using different keys. Thus, adjacent memory violations, *i.e.*, linear buffer overflows, deterministically lead to garbled data or authentication errors.

Non-adjacent memory violations are mitigated on a probabilistic basis (based on the number of utilized keys). Therefore, we distinguish between inter-page and intra-page access violations. The inter-page memory access violation can result in two cases. First, the memory access could lead to a page fault and immediate program abortion, which is similar to ASLR-based approaches. Second, the attacker accesses a mapped page and decrypts the memory object with the corresponding key identifier. The probability that this key matches the correct one used for the encryption of the memory object depends on the number of available keys, which differs depending on the CPU model and platform software. At maximum, up to 2^{15} keys are available. Similarly, the intra-page collision probability depends on the number of utilized keys. Since our allocation granularity is at least 64 B, using 64 different keys allows us to uniquely assign a key for each allocation on a page. Thus, when using at least 64 keys or larger allocations, there is no spatial intra-page collision possible.

Intra-object memory access violations are currently not protected by our scheme. This requires detailed knowledge of how the memory is used and would need to be implemented in the compiler itself. Hence, our heap allocator cannot protect against such violations. In summary, depending on the underlying hardware capabilities, *i.e.*, encryption-only or authenticated encryption, the spatial violation would lead to reading/writing garbled data, or directly triggering a hardware exception.

Temporal Memory Safety. For the temporal safety analysis, we consider uninitialized memory and use-after-free (UAF) attacks. UAF attacks misuse pointers referring to already freed memory, so-called *dangling pointers*. UAF is typically only security-relevant in use-after-reallocation (UAR) cases. These occur when a newly allocated object gets allocated on the same chunk of memory of a previously freed object. There, the attacker could alter the data of the new memory object using the previous, stale, pointer. MEMES prevents such attacks on a probabilistic basis by assigning different keys for the corresponding memory objects. On every memory reallocation, the allocator pseudo-randomly chooses a key and returns the corresponding pointer. The probability that the keys of two memory allocations coincide again depends on the number of available and utilized keys (up to 2^{-15}). Similarly, uninitialized memory accesses are mitigated since a memory read using the wrong key would result in garbled data or detection of the memory violation. The immediate detection of such violations depends on the utilized TME-MK mode (*i.e.*, encryption-only or authenticated encryption).

Multi-threading. Software-only exploit mitigations (e.g., verifying a pointer before dereferencing) usually suffer from time-of-use race conditions (Wei and Pu, 2005) that can be exploited in multi-threaded applications (Farkhani et al., 2021; Conti et al., 2015). In contrast, our approach is hardware-based and does not suffer from the same issue. MEMES benefits from TME-MK, which is deeply embedded into the memory sub-system of the CPU. TMEalloc does not flush cache lines (e.g., after freeing memory) in our experiments, however, it is noteworthy that the TME-MK specification recommends flushing caches when changing its associated key identifier (Intel®, 2022).

6 EVALUATION

In this section, we evaluate the performance and memory overhead of our implementation.

6.1 Performance Evaluation

For evaluating the performance of MEMES, we use all C and C++ SPEC CPU® 2017 (Standard Performance Evaluation Corporation, 2023) benchmarks and Linux 5.15 running on an Intel® Xeon® Platinum 8480+ CPU, which has support for TME-MK. On that system, the DRAM is always encrypted using TME, which means that a single key, chosen at boot-time, is used to encrypt the entire DRAM. Thus, for our baseline, where we run SPEC CPU® 2017 without protecting heap allocations, full DRAM encryption is still active. Note that authenticated encryption is currently not supported on our evaluation platform. Hence, our evaluation results are based on the encryption-only mode (AES-XTS-128). The authenticated encryption variant used in TDX stores its MAC in the ECC memory (Intel®, 2020). Thus, while we cannot evaluate the overhead of that mode, we do expect negligible differences because no extra memory fetches are necessary.

We compiled all SPEC CPU® 2017 C and C++ benchmarks with our memory allocator utilizing page table aliasing and encryption using different keys for fine-grained memory safety. Thereby, all allocations are aligned to 64 B. We evaluated our prototype for different configurations: Namely "Padding only" where we only add padding and alignment to conform to the cache line sizes, and "1 bit" to "6 bit" where we additionally encrypt each allocation with 2^1 to 2^6 different keys, respectively.

Figure 5 illustrates our experimental performance overheads of MEMES for different configurations and benchmarks. In addition, the figure shows the correlating relative number of cache misses, TLB misses, and page faults for each benchmark. Compared to the baseline, encrypting all allocations but not their metadata with the same encryption key, *i.e.*, configuration "1 bit", induces a geometric mean runtime overhead of 16 %. Note that for "1 bit" there are still two different keys used in total. One for the heap allocations, and one for the remaining memory like the heap metadata, code, and non-heap program data. Thus, even this configuration can double the number of required TLB entries compared to no heap protection. When adding more keys, and thus aliases, we measured a geometric mean runtime overhead between 16 % and 27 %.

This performance overhead is introduced by *(i)* the memory alignment since all memory objects need to be aligned and padded to 64 bytes. Depending on the allocation sizes of the dedicated benchmarks, the alignment is responsible for a large proportion of the incurred overhead (cf. Figure 6). For example, `mcf` almost exclusively allocates very small memory objects. Note that for `mcf`, most of the performance overhead is due to padding as visible from the "Padding only" bar. This alignment overhead does not only increase the memory consumption of the program, it also increases the runtime overhead due to caching effects. While it would be possible to remove any alignment and padding requirements, this would drastically decrease performance due to the increased number of cache evictions if multiple keys were used for a cache line that is accessed frequently.

On the other hand, our page table aliasing approach *(ii)* increases the TLB pressure leading to a performance decrease depending on the memory access pattern and memory access frequency. This is, because each key that is used within a page requires one TLB entry.

Finally, the *(iii)* management of the heap library introduces some overhead. The memory allocator needs to prepare the setup of the page table aliasing and manage the received pages from the operating system.

The total performance overhead is the result of a complex interplay of the following factors: Memory allocation sizes (Figure 6), total memory usage (Figure 7), memory usage patterns, and number of used keys per page (Figure 5). From the given figures, it is clear that all these factors have a performance impact with varying degrees. In the following subsection, we detail the added memory overhead to the memory alignment.

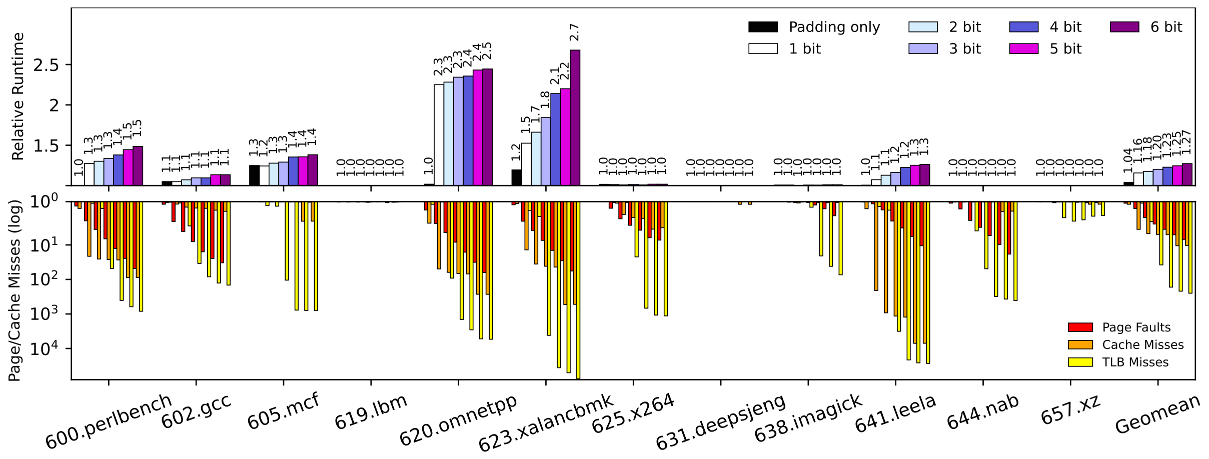


Figure 5: The performance overhead incurred by the MEMES design. We evaluate our prototype implementation using different numbers of key ids. The number of key ids derives the amount of page table mappings leading to increased TLB pressure responsible for the majority of the runtime overhead.

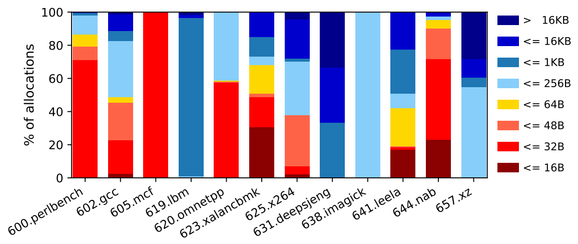


Figure 6: The distribution of the allocation sizes of the SPEC CPU® 2017 benchmarks.

6.2 Memory Utilization

As discussed in Section 3, we optimize the performance by aligning all memory allocations on the heap to the cache line granularity, *i.e.*, 64 B on our system. The memory overhead incurred by the alignment to 64 bytes largely depends on the allocation sizes of the specific applications.

To analyze the impact on memory utilization, we first analyzed the size of each allocation for the SPEC CPU® 2017 benchmarks. Mimalloc already provides debug information about allocation sizes. Thus, for this analysis, we run the benchmarks using the unmodified library in debug mode. Figure 6 shows the occurrence of different allocation sizes. We group them together into buckets for easier visualization. Smaller buckets are shown in red to highlight that these require alignment and padding. Especially large allocations (*i.e.*, larger than 1 kB) benefit from our approach by introducing a negligible performance and memory overhead.

Furthermore, we evaluated the additional memory required due to the above memory layout constraints. For this, we measured the physically used memory of each of the SPEC CPU® 2017 benchmarks during their entire execution. We place the benchmark processes in their own `cgroup` and measure the `memory.current` field to capture the total memory usage of the benchmark. This is illustrated in Figure 7. Expectedly, benchmarks like `lbm`, `x264`, `deepsjeng`, `imagick`, and `xz` which predominantly use larger allocation sizes, also have a negligible memory overhead of 1 % – 11 %. These benchmarks also incur the smallest performance overhead, as shown in Figure 5. On the other hand, `perlbench`, `gcc`, `mcf`, `omnetpp`, and `xalancbmk` use much smaller allocations that require more padding, which in turn

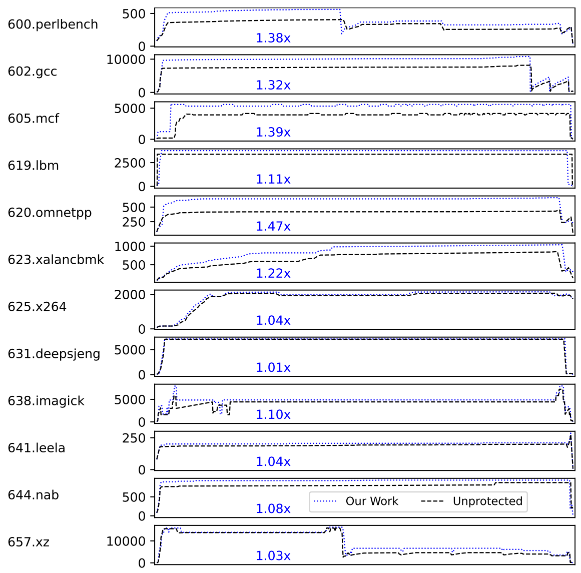


Figure 7: The memory utilization of the SPEC CPU® 2017 benchmarks. This figure compares the memory usage of the unmodified allocator with our TMEalloc that aligns every allocation to 64 bytes. The numbers shown in blue is the factor by which TMEalloc uses more memory.

Table 1: Comparison of memory safety countermeasures. Legend: ● Mitigated, ♦ Cryptographic mitigation with possible detection, ◐ Probabilistic, ○ None.

Mechanism	OOB	UAF	Uninit.	Commodity HW
SoftBound+CETS	●	●	○	✔
C^3	♦	♦	♦	✘
CrypTag	♦	♦	♦	✘
ARM® MTE	◐	◐	○[‡]	✔[*]
SPARC ADI	◐	◐	○[‡]	✔
MEMES	♦	♦	♦	✔

[‡] MTE/ADI add instructions to zero the memory when setting the tag, which helps mitigating uninitialized memory use/leaks.

[*] Note that CPUs with MTE are not yet widely available.

leads to an increased memory overhead of 22 % – 47 %.

The allocation sizes have a direct impact on the memory usage, which in turn has an impact on the performance. This is because the CPU caches are used less efficiently if the cached data includes unused padding bytes. That means that more cache lines need to be used to store the same amount of data, and more memory requests are made to fetch the data from the DRAM.

7 RELATED WORK

In this section, we discuss the related work of memory safety and compare common approaches with MEMES. Memory safety countermeasures proposed by academia and industry present different trade-offs regarding security, performance, flexibility, and memory usage. Table 1 provides an overview and comparison of related memory safety countermeasures.

Bounds-checking. Several countermeasures propose the mitigation of spatial memory safety violations by introducing bounds checks (Akritidis et al., 2009; Ziad et al., 2021; Duck et al., 2017; Duck and Yap, 2016; Simpson and Barua, 2013) for every memory operation. These bound-checking mechanisms are implemented either in software, e.g., CCured (Necula et al., 2002) and Cyclone (Jim et al., 2002), or hardware, e.g., Hardbound (Devietti et al., 2008), Watchdog (Nagarakatte et al., 2012; Nagarakatte et al., 2014), and Intel® MPX (Oleksenko et al., 2018). Typically, software-based approaches tend towards high runtime overheads (cf. 116 % runtime overhead for SoftBound (Nagarakatte et al., 2009) and CETS (Nagarakatte et al., 2009)) while hardware-enforced mechanisms typically require intrusive hardware changes, which can be hard to deploy on a large scale. Additionally, hardware bounds-checking mechanisms, like CHERI (Woodruff et al., 2014; Watson et al., 2015), often use so-called *fat pointers*, which co-locate the bounds information with the pointer, thus, requiring application binary interface (ABI) changes. In contrast, MEMES enables a lightweight and encryption-based solution for memory safety that provides more flexibility while maintaining binary compatibility.

Use-after-free Protection. Temporal memory safety is commonly achieved by meta-information (Farkhani et al., 2021; Nagarakatte et al., 2010; Burow et al., 2018; Lee et al., 2015), which represents the liveness of memory objects. For example, CETS (Nagarakatte et al., 2010) associates unique identifiers managed in a disjoint data structure that is checked on every memory access. In contrast, mechanisms like MarkUs (Ainsworth and Jones, 2020) provide temporal memory safety by managing a quarantine list for freed objects. Memory is only reallocated if no dangling pointers referring to the quarantined memory exist. Furthermore, xTag (Bernhard et al., 2022) utilizes pointer tagging in combination with page aliasing to achieve temporal safety. Similarly, MEMES utilizes heap aliasing. However, our approach combines page aliasing with memory encryption, enforcing full memory safety (*i.e.*, temporal and spatial safety). Arbitrary read and write violations are a serious threat to software systems, and countermeasures that only target a subset of memory safety (e.g., only temporal safety) often lack widespread adoption.

Tagged Memory. Memory tagging (Serebryany, 2019; Jero et al., 2023; Joannou et al., 2017), often also referred to as *memory coloring* or *lock-and-key*, provides additional meta-information associated with every memory location. Tagged memory, implemented in software (Akritidis et al., 2008) or hardware, allows the enforcement of fine-grained access policies. For example, HWASan (Serebryany et al., 2018) uses 8-bit memory tags encoded into the pointers (enabled by hardware address masking) and implements the required tag checks in software. Furthermore, the ARM® memory tagging extension (MTE) (ARM Limited, 2019) and SPARC application data integrity (ADI) (Aingaran et al., 2015) are ISA extensions that implement tagged memory utilizing a 4-bit tag at a granularity of 16 and 64 bytes, respectively. The memory tags are used to enforce probabilistic memory safety. Similarly, MEMES provides probabilistic memory safety, however, the memory encryption key is not directly encoded into the pointer. Thus, it becomes harder for attackers to leak and forge pointers with their corresponding key values. Additionally, the memory encryption engine on our used system allows the usage of up to 64 different keys, increasing the probabilistic detection capabilities compared to the 16 distinct tag values of ARM® MTE and SPARC ADI. Note that TME-MK can support up to a maximum of 2^{15} keys. Normal pages can only hold a maximum of 64 allocations. Hence, per page, only 64 or less keys would be used. Thus, we assume that the performance overhead for using more than 64 keys would remain largely unchanged as it has no effect on the number of TLB entries needed. In contrast, memory tagging schemes like MTE scale much worse for each additional "key" bit. E.g., scaling MTE to 15 or 16 bits would already increase the tagged memory overhead by a factor of 4.

Encryption-based Mechanisms. Some research proposals utilize memory encryption for memory protection. CrypTag (Nasahl et al., 2021) introduces a scheme that, similarly to memory tagging, encodes a memory tag into the upper bits of the pointer. However, instead of storing the tags in memory, CrypTag uses the pointer tags to tweak the memory encryption procedure providing object-granular memory safety. Moreover, C^3 (LeMay et al., 2021) utilizes pointer and memory encryption for memory safety. The encrypted part of the pointer's address is used to decrypt the corresponding data in memory. There, memory safety violations, e.g., due to corrupted or stale pointers, are highly likely to lead to either a page fault or garbled data. Currently, memory encryption is primarily used for confidential computing, such as Intel® TDX (Intel®, 2023) and AMD SEV (AMD, 2020), which isolate virtual machines. Furthermore, SERVAS (Steinegger et al., 2021) proposes an enclave design that tweaks the memory encryption using the enclave metadata. In comparison, MEMES utilizes the TME-MK memory encryption engine (available on commodity hardware) in combination with heap aliasing in order to provide fine-grained memory safety. Moreover, EC-CFI (Nasahl et al., 2023) combines Intel®'s virtualization technology with memory encryption to provide cryptographic control flow integrity against fault attacks.

8 FUTURE WORK & CONCLUSION

In this section, we give a brief outlook on possible future work to improve the performance of MEMES further and finally give a conclusion of our work.

8.1 Future Work

Our concept implementation proves the efficacy of our design. In our implementation, we only implemented optimizations that do not negatively impact the security (e.g., cache line padding). However, there are more possible optimizations for specific workloads introducing different trade-offs.

Multiple small allocations (e.g., 4x16 B) would fit on a cache line if there is no padding. There are two possible ways to allow for this. First, we could use the same keys for such allocations, which would improve both the performance and memory usage significantly but it results in a loss in detection for intra-cache line overflows. Second, we can optimize memory usage without impacting security by using different keys for each allocation within a cache line. However, the downside to this is increased cache evictions which lead to performance degradation.

Alternatively, it is possible to only protect a subset of an application, while the majority of data would be unencrypted. An example for this would be secure keystorage used in cryptographic libraries like OpenSSL (The OpenSSL Project, 2003).

8.2 Conclusion

In this paper, we presented a method for **fine-grain memory encryption** using Intel®'s TME-MK without any hardware changes. By using multiple identical mappings from virtual to physical memory pages with different TME-MK key identifiers, we are able to encrypt memory objects with different keys at the smallest possible encryption granularity, *i.e.*, 128 bit. Based on page table aliasing, we then introduced **MEMES**. MEMES mitigates memory safety vulnerabilities on the heap by encrypting memory objects with different key identifiers. Using this approach, we showcased that MEMES, depending on the underlying capabilities of the encryption engine, can either detect or prevent the exploitation of memory corruptions. Our performance analysis showed that MEMES induces a geometric mean runtime overhead of just 16–27% for SPEC CPU® 2017.

ACKNOWLEDGEMENTS

This project has received funding from the Austrian Research Promotion Agency (FFG) via the AWARE project (FFG grant number 891092). Additional funding was provided by a generous gift from Intel.

REFERENCES

Aingaran, K., Jairath, S., Konstadinidis, G. K., Leung, S., Loewenstein, P., McAllister, C., Phillips, S., Radovic, Z., Sivaramakrishnan, R., Smentek, D., and Wicki, T. (2015). M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro*, 35:36–45.

Ainsworth, S. and Jones, T. M. (2020). MarkUs: Drop-in use-after-free prevention for low-level languages. In *S&P*, pages 578–591.

Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M. (2008). Preventing Memory Error Exploits with WIT. In *S&P*, pages 263–277.

Akritidis, P., Costa, M., Castro, M., and Hand, S. (2009). Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security Symposium*, pages 51–66.

AMD (2020). Amd sev-snp: Strengthening vm isolation with integrity protection and more. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Accessed 2023-01-05.

ARM Limited (2019). Arm architecture reference manual for a-profile architecture, v8.5a. https://developer.arm.com/documentation/ddi0487/ea. Accessed 2023-01-05.

Bernhard, L., Rodler, M., Holz, T., and Davi, L. (2022). xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In *EURO S&P*, pages 502–519.

Bletsch, T. K., Jiang, X., Freeh, V. W., and Liang, Z. (2011). Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS*, pages 30–40.

Burow, N., McKee, D. P., Carr, S. A., and Payer, M. (2018). CUP: Comprehensive User-Space Protection for C/C++. In *AsiaCCS*, pages 381–392.

Conti, M., Crane, S., Davi, L., Franz, M., Larsen, P., Negro, M., Liebchen, C., Qunaibit, M., and Sadeghi, A. (2015). Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *CCS*, pages 952–963.

Devietti, J., Blundell, C., Martin, M. M. K., and Zdancewic, S. (2008). Hardbound: architectural support for spatial safety of the C programming language. In *ASPLOS*, pages 103–114.

Duck, G. J. and Yap, R. H. C. (2016). Heap bounds protection with low fat pointers. In *CC*, pages 132–142.

Duck, G. J., Yap, R. H. C., and Cavallaro, L. (2017). Stack Bounds Protection with Low Fat Pointers. In *NDSS*.

Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., and Paxson, V. (2014). The Matter of Heartbleed. In *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, pages 475–488.

Farkhani, R. M., Ahmadi, M., and Lu, L. (2021). PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *USENIX Security Symposium*, pages 1037–1054.

Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. (2008). Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium*, pages 45–60.

Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. (2016). Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *S&P*, pages 969–986.

Intel®(2020). Architecture Specification: Intel® Trust Domain Extensions (Intel® TDX) Module. https://www.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-module-1eas.pdf. Accessed: 2023-01-30.

Intel®(2022). Intel® Architecture Memory Encryption Technologies. https://www.intel.com/content/www/us/en/content-details/679154/intel-architecture-memory-encryption-technologies-specification.html. Revision 1.4, Accessed: 2023-01-31.

Intel®(2023). Intel® Trust Domain Extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf. Accessed: 2023-01-30.

Jero, S., Burow, N., Ward, B. C., Skowyra, R., Khazan, R., Shrobe, H. E., and Okhravi, H. (2023). TAG: Tagged Architecture Guide. *ACM Comput. Surv.*, 55:124:1–124:34.

Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. (2002). Cyclone: A Safe Dialect of C. In *USENIX ATC*, pages 275–288.

Joannou, A., Woodruff, J., Kovacsics, R., Moore, S. W., Bradbury, A., Xia, H., Watson, R. N. M., Chisnall, D., Roe, M., Davis, B., Napierala, E., Baldwin, J., Gudka, K., Neumann, P. G., Mazzinghi, A., Richardson, A., Son, S. D., and Markettos, A. T. (2017). Efficient Tagged Memory. In *ICCD*, pages 641–648.

Kaplan, D., Powell, J., and Woller, T. (2020). Amd sev-snp: Strengthening vm isolationwith integrity protection and more. Technical report, Technical Report. Advanced Micro Devices Inc.

Kim, Y., Lee, J., and Kim, H. (2020). Hardware-based Always-On Heap Memory Safety. In *MICRO*, pages 1153–1166.

Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., and Lee, W. (2015). Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS*.

LeMay, M., Rakshit, J., Deutsch, S., Durham, D. M., Ghosh, S., Nori, A., Gaur, J., Weiler, A., Sultana, S., Grewal, K., and Subramoney, S. (2021). Cryptographic Capability Computing. In *MICRO*, pages 253–267.

Microsoft (2019). Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf. Accessed 2023-01-05.

Microsoft (2022). mimalloc. hhttps://github.com/microsoft/mimalloc. Accessed 2023-01-05.

Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. (2012). Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ISCA*, pages 189–200.

Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. (2014). WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *CGO*, page 175.

Nagarakatte, S., Zhao, J., Martin, M. M. K., and Zdancewic, S. (2009). SoftBound: highly compatible and complete spatial memory safety for c. In *PLDI*, pages 245–258.

Nagarakatte, S., Zhao, J., Martin, M. M. K., and Zdancewic, S. (2010). CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, pages 31–40.

Nasahl, P., Schilling, R., Werner, M., Hoogerbrugge, J., Medwed, M., and Mangard, S. (2021). CrypTag: Thwarting Physical and Logical Memory Vulnerabilities using Cryptographically Colored Memory. In *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, pages 200–212.

Nasahl, P., Sultana, S., Liljestrand, H., Grewal, K., LeMay, M., Durham, D. M., Schrammel, D., and Mangard, S. (2023). EC-CFI: Control-Flow Integrity via Code Encryption Counteracting Fault Attacks. *CoRR*, abs/2301.13760.

Necula, G. C., McPeak, S., and Weimer, W. (2002). CCured: type-safe retrofitting of legacy code. In *POPL*, pages 128–139.

Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., and Fetzer, C. (2018). Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2:28:1–28:30.

Serebryany, K. (2019). ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. *login Usenix Mag.*, 44.

Serebryany, K., Stepanov, E., Shlyapnikov, A., Tsyrklevich, V., and Vyukov, D. (2018). Memory Tagging and how it improves C/C++ memory safety. *CoRR*, abs/1802.09517.

Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, pages 552–561.

Simpson, M. S. and Barua, R. (2013). MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Softw. Pract. Exp.*, 43:93–128.

Standard Performance Evaluation Corporation (2023). SPEC CPU® 2017. https://www.spec.org/cpu2017/. Accessed: 2023-01-31.

Steinegger, S., Schrammel, D., Weiser, S., Nasahl, P., and Mangard, S. (2021). SERVAS! Secure Enclaves via RISC-V Authenticryption Shield. In *ESORICS*, volume 12973 of *LNCS*, pages 370–391.

Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). SoK: Eternal War in Memory. In *S&P*, pages 48–62.

The Chromium Projects (2020). Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/. Accessed 2023-01-14.

The OpenSSL Project (2003). OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org.

Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N. H., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R. M., Roe, M., Son, S. D., and Vadera, M. (2015). CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P*, pages 20–37.

Wei, J. and Pu, C. (2005). TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. In *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*.

Woodruff, J., Watson, R. N. M., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R. M., and Roe, M. (2014). The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA*, pages 457–468.

Ziad, M. T. I., Arroyo, M. A., Manzhosov, E., Piersma, R., and Sethumadhavan, S. (2021). No-FAT: Architectural Support for Low Overhead Memory Safety Checks. In *ISCA*, pages 916–929.