Memory Tagging using Cryptographic Integrity on Commodity x86 CPUs

David Schrammel
Graz University of Technology
Graz, Austria
david.schrammel@iaik.tugraz.at

Martin Unterguggenberger
Graz University of Technology
Graz, Austria
martin.unterguggenberger@iaik.tugraz.at

Lukas Lamster
Graz University of Technology
Graz, Austria
lukas.lamster@iaik.tugraz.at

Salmin Sultana
Intel Labs
USA
salmin.sultana@intel.com

Karanvir Grewal
Intel Labs
USA
ken.grewal@intel.com

Michael LeMay
Intel Labs
USA
michael.lemay@intel.com

David M. Durham
Intel Labs
USA
david.durham@intel.com

Stefan Mangard
Graz University of Technology
Graz, Austria
stefan.mangard@iaik.tugraz.at

***Abstract*—Memory tagging allows to establish memory safety for software developed in unsafe languages like C/C++. Since it is an effective mechanism with low architectural complexity, ISA extensions, like ARM MTE or SPARC ADI, already integrate memory tagging on the architectural level for commodity computer systems. However, despite being in high demand, memory tagging features are currently absent in modern x86 processors.**

This work presents *IntegriTag*, a hardware-enforced memory tagging solution for existing commodity x86 CPUs. We leverage the Intel® Total Memory Encryption-Multi-Key (Intel® TME-MK) hardware feature that was initially envisioned for virtual machine isolation to instead provide memory tagging capabilities on off-the-shelf x86 processors. Unlike ARM MTE and SPARC ADI, this does not require the integration of a separate tagged memory architecture, which would increase the overall system complexity. Instead, our solution allows us to implicitly enforce the desired security policies by incorporating them into the existing memory encryption integrity checks. In addition, our design addresses security issues that affect tagged memory architectures with small tag spaces. Intel® TME-MK allows for a greater number of key identifier bits, thus offering significantly stronger security compared to the 4-bit tags of ARM MTE and SPARC ADI. We implement a holistic open-source software framework based on Intel® TME-MK, supporting several software-controlled and hardware-enforced memory safety policies. Moreover, we evaluate our design's performance overhead and security properties, underlining the practicability and efficacy of our approach. Our design is binary-compatible with existing software and provides both temporal and spatial memory safety while imposing an overhead of 32–41%, which is significantly lower than the overheads of memory safety schemes in software on commodity hardware that provide comparable security properties.

***Index Terms*—Memory Safety, Memory Sanitization, Intel® TME-MK, Memory Encryption, Memory Tagging**

1. Introduction

System-level software written in unsafe programming languages like C/C++ is susceptible to memory safety vulnerabilities. Memory safety issues occur frequently in large software products such as web browsers and operating systems. Recent studies conducted by Microsoft [38] highlight that approximately 70 % of security bugs in their products are categorized as memory safety errors, *i.e.*, temporal and spatial memory safety vulnerabilities [65]. Many large and complex software projects will continue to use unsafe languages for the foreseeable future and only incrementally adopt safety improvements (e.g., via partial rewrites). A recent Google whitepaper also notes that an evolution of C/C++ towards full memory safety is unlikely [49].

To mitigate memory safety vulnerabilities, various countermeasures have been proposed in recent years [24], [31], [74], [76], [80], which complement the slow transition toward memory-safe languages. Traditionally, bounds-checking mechanisms [11]–[13], [32] are proposed to protect against spatial memory violations, such as linear buffer overflows and (non-adjacent) out-of-bounds (OOB) errors. These additional bounds checks can be implemented in software [7], [41], [45] and hardware [10], [25], [40]. In general, countermeasures implemented in software [3], [5], [7], [24], [45], [59] tend to introduce high runtime overheads, while hardware-enforced mechanisms [47], [50], [51], [57], [60], [79] can be challenging to deploy on a large scale. Since system performance is a crucial aspect of software systems, there is a noticeable shift by CPU vendors toward adopting hardware-supported mechanisms for security.

Memory tagging can mitigate a wide variety of both spatial and temporal memory safety vulnerabilities, and

it can be enforced in hardware [55], [56]. With memory tagging, metadata in the form of a *memory tag* is associated with each memory location and compared with a tag value embedded in available pointer bits to enforce different security policies. Memory tagging is adopted in commercial products like the ARM memory tagging extension (MTE) [55] and SPARC application data integrity (ADI) [1], [56] hardware features. These instruction set architecture (ISA) extensions implement a so-called *lock-and-key* approach to ensure memory safety. During object allocation, a pseudorandom memory tag is generated and encoded into the pointer. Subsequently, the memory tag is associated with the corresponding memory location. Following memory accesses using pointers are only permitted when the memory tag of the pointer and the tag of the memory location match, *i.e.*, the key matches the lock. Tagged memory architectures, integrating this lock-and-key approach, can be used for both memory sanitizers and runtime protection.

Memory sanitizers [46], [54], [62] are commonly used for software testing to detect C/C++ memory safety issues preemptively. Notably, two memory sanitizers integrated into Clang/LLVM [30] based on memory tagging have gained substantial popularity: HWASan [66] and MemTagSanitizer [36]. HWASan employs software-based memory tagging based on ARM's top-byte ignore (TBI) [34] hardware feature in combination with shadow memory. However, HWASan suffers from relatively high performance and memory overheads. In contrast, MemTagSanitizer detects memory safety violations using hardware-enforced memory tagging based on ARM MTE, which is expected to perform significantly better.

While sanitization allows the detection of errors in pre-release software, memory tagging can also be utilized for runtime protection [55], [56]. Several memory allocators provide support for ARM MTE and SPARC ADI to protect against memory safety violations during program execution. For instance, the GNU C/C++ standard library offers support for ARM MTE-based heap memory safety. Furthermore, Google Chrome's memory allocator, PartitionAlloc [15], implements StarScan [16], an effective countermeasure against temporal safety violations using ARM MTE in combination with memory quarantining. Hardware-supported memory tagging is a highly demanded security and debugging feature of modern CPUs. However, memory tagging has not been implemented in modern x86 processors.

Contributions. In this paper, we present *IntegriTag*, a novel mechanism for memory tagging on commodity x86 CPUs using the Intel® Total Memory Encryption-Multi-Key (Intel® TME-MK) [20] hardware feature. This feature was originally designed to cryptographically isolate virtual machines (VMs) at runtime and defend against HW/SW attacks including physical attacks, cold-boot attacks, cross-VM data injection, and replay attacks. Instead, our approach allows leveraging it for the *detection of memory safety violations*. We use memory aliasing in combination with the MAC-based integrity protection of the memory encryption hardware feature to assign different keyIDs for distinct memory objects. This way, we can enforce security policies based on a lock-and-key approach similar to memory tagging. However, in contrast to conventional tagged memory architectures, such as ARM MTE and SPARC ADI, our approach offers several advantages:

First, our design introduces memory tagging capabilities for existing commodity x86 architectures without additional memory overhead. We provide one of the most demanded security features (*i.e.*, memory tagging) for memory sanitization and runtime safety based on authenticated memory encryption. We show that cryptographic memory integrity protection, designed to protect trusted execution environments (TEE) from physical attacks, offers properties associated with memory tagging and can be used in a similar manner to achieve memory safety. Note that while SPARC's ADI also uses the term "integrity", their design offers only logical integrity, which is much weaker than the cryptographic memory integrity protection of Intel® TME-MK, that addresses actual data corruption detection in DRAM.

Second, conventional tagged memory architectures require the memory controller to perform additional fetch requests to receive the tag metadata from DRAM. These double fetches increase the memory pressure substantially. Especially for data-centric workloads and multi-core environments (e.g., cloud computing), these additional overheads can cause a significant performance penalty. In contrast, the MAC used for the integrity checks of Intel® TME-MK resides in memory that is otherwise reserved for error correction codes (ECC) [8]. Thus, the metadata is read and written without additional DRAM pressure, *i.e.*, no DRAM double fetches. Furthermore, since the SHA-3-based MAC can be calculated much faster [67] and in parallel to the AES memory decryption, there should be no additional memory latency.

Third, ARM MTE and SPARC ADI suffer from poor detection capabilities due to their limited tag space. Their 4-bit tag yields a relatively high collision probability of 1/16 that two memory objects receive the same memory tag. Hence, their detection probability of memory safety issues is only 93.75 %. In contrast, our work accommodates up to 15-bit keys, substantially increasing the detection probability to up to 99.997%.

We develop a holistic open-source software framework using a custom heap allocator and a Linux kernel patch to support a variety of memory tagging policies for both temporal and spatial memory safety. These policies are implemented and enforced through our lock-and-key approach, leveraging Intel® TME-MK on the hardware level. Our instrumented memory allocator, built upon PartitionAlloc, establishes the tagging policies enforced in hardware while maintaining *binary compatibility*. Moreover, we conduct an extensive security analysis, including an empirical security evaluation based on the NIST Juliet C/C++ test suite [6], underlining our design's efficacy and strong detection capabilities. Our performance evaluation, across all C/C++ SPEC CPU2017 benchmarks, highlights competitive results of a 32–41% overhead for both temporal and spatial memory safety.

In short, our contributions are as follows:

- **We enable memory tagging on x86 CPUs.** We are the first to show that memory tagging can be implemented on top of the integrity-enabled memory encryption provided by commodity x86 CPUs.
- **We present *IntegriTag*.** We introduce *IntegriTag*, a countermeasure leveraging Intel® TME-MK, enforcing various security policies for temporal and spatial heap memory safety while maintaining binary compatibility.
- **We systematically analyze *IntegriTag*'s security.** We systematically analyze the security properties of *IntegriTag* and empirically evaluate the efficacy using the NIST Juliet C/C++ test suite.
- **We comprehensively evaluate *IntegriTag*.** We evaluate the performance impact using the C/C++ SPEC CPU2017 benchmark suite and the memory overhead of our design in various configurations.
- **We open-source our software framework.** To facilitate future research, we open-source our proof-of-concept prototype implementation [1].

Outline. The paper is structured as follows. Section 2 provides the required background for this work. Section 3 defines our threat model. Section 4 and Section 5 describe the design and implementation of our countermeasure. Section 6 analyzes the security of our design, and Section 7 evaluates the performance impact. Section 8 discusses related work, and Section 9 concludes this work.

2. Background

In this section, we give background on virtual memory, memory safety, memory tagging, and Intel® TME-MK.

2.1. Virtual Memory

Operating systems use virtual memory to isolate multiple processes within their own virtual address spaces. Thereby, each application has its own virtual representation of the physical memory. Modern CPUs typically support 39-, 48-, or 57-bit virtual address spaces used for process isolation.

For virtual-to-physical address translation, virtual memory is mapped to pages that are typically 4 KiB in size. These pages are managed through page table entries (PTEs), which are stored in a hierarchical structure called page tables. A PTE contains a physical page number (PPN) with its associated access permissions. The CPU translates virtual addresses into physical addresses through the memory management unit (MMU), utilizing page tables. When the system performs a page table walk to determine the corresponding physical address, the translations are cached in the translation look-aside buffer (TLB) to improve performance.

Through the virtualization of physical memory, it is possible that multiple virtual memory addresses refer to the same physical address. This is called aliasing and is often used for shared data and code pages across different applications.

1. github.com/IntelLabs/TME-MK-Fine-Grained-Encryption-Integrity

2.2. Memory Safety

C/C++ memory safety vulnerabilities are commonly categorized into temporal and spatial vulnerabilities [65]. For spatial safety, we can further distinguish between adjacent and non-adjacent memory safety violations. Such vulnerabilities can be misused to gain unauthorized access (read and write) to resources stored in memory. Adjacent memory safety violations, e.g., linear buffer overflows, are typically easier to detect using tripwires or red zones. Non-adjacent memory safety violations, e.g., arbitrary read and write primitives, are more complex to mitigate and require advanced protection mechanisms such as bounds-checking. Furthermore, intra-object violations occur by violating the boundaries between the fields of the same object.

When discussing temporal safety, we distinguish between use-after-free (UAF), double-free, and uninitialized memory errors. Typically, UAF errors arise from so-called *dangling* or *stale pointers*, *i.e.*, pointers that refer to a previously freed memory object. These dangling pointers allow an adversary to potentially manipulate data of objects located on the same chunk of reallocated memory. Similarly, an adversary can craft dangling pointers using double-free violations by tampering with the allocator's free list. Additionally, uninitialized memory access violations enable an adversary to leak potentially sensitive data from previously freed memory locations.

2.3. Memory Tagging

Memory tagging is a promising building block for mitigating temporal and spatial memory safety issues [1], [55], [56], [69]. From a high-level point of view, memory tagging means associating memory in DRAM with additional metadata (*i.e.*, *memory tags*). Depending on the tag size and tag granularity, different security policies can be implemented [23], [53], [61], [71], [73], [74], [78].

For instance, the ARM memory tagging extension (MTE) [55] and SPARC application data integrity (ADI) [1], [56] are ISA extensions that implement memory tagging in hardware. They use 4-bit tags on 16 B and 64 B granularity, respectively. Both extensions allow the enforcement of fine-grain memory safety policies by using a *lock-and-key* approach, where the pointer itself embeds a key, and the tag metadata co-located with the memory location acts as the lock. Thereby, ARM MTE encodes the 4-bit tag in the uppermost bits of a pointer (enabled by ARM's top-byte ignore (TBI) [34] feature), which would otherwise be unused. During memory allocation, the corresponding memory location is tagged (*i.e.*, locked) with the corresponding tag (*i.e.*, key). Subsequent memory accesses need to provide the correct tag (*i.e.*, key) to unlock access to the memory location. ARM MTE is effectively used for memory sanitization (e.g., Clang/LLVM's MemTagSanitizer [36]) and runtime safety [55] on recent ARM platforms.

Traditionally, the associated tag metadata causes memory and performance overheads due to storage requirements and the tag propagation by the tagged memory architecture (*i.e.*, additional DRAM fetches). However, existing work shows how these overheads can be mitigated in the case of ECC memory [21], [29], [64].

2.4. Intel® TME-MK

Memory encryption is a widespread technology in modern CPU architectures, such as Intel® and AMD, which can help provide confidentiality (and integrity) of data stored in DRAM. Specifically, Intel® uses Total Memory Encryption (TME) [20] for transparent encryption of DRAM data utilizing a single key. With the 3rd generation Intel® Xeon Scalable processors, the Total Memory Encryption-Multi-Key (Intel® TME-MK) extension was introduced, which enhances TME by supporting multiple keys. Currently, this feature is used to help protect against physical adversaries (*i.e.*, cold boot attacks [18]) or for the cryptographic isolation of virtual machines.

Figure 1 illustrates the memory encryption and decryption procedure. Intel® TME-MK maps so-called *key identifiers* (keyIDs) to corresponding encryption keys and encryption modes. It allows for up to 2^{15} distinct keys, depending on the underlying platform. When accessing data in DRAM, the keyID is used to decide which cryptographic key to use for the memory access. The identifier is embedded into the upper physical address bits in the PTE, which were previously unused by the system. For every memory operation, the transferred data is encrypted or decrypted transparently, enabling page-granular memory encryption.

The encryption engine uses AES in XTS mode with either 128-bit or 256-bit keys as its underlying cryptographic primitive. Thus, memory is encrypted in 128-bit blocks. The physical address serves as an additional input for the encryption procedure, ensuring that identical data in distinct locations is encrypted differently.

The introduction of Intel® trust domain extensions (TDX) [9] in 2023 incorporates additional support for authenticated encryption. Specifically, Intel® TME-MK is enhanced with a message authentication code (MAC) based on SHA-3 that is stored in ECC memory, providing data integrity [21], [22]. These MACs exist per cache line (64 B). This helps keep sensitive data secure not just from most software-based attacks but also from many hardware-based, attacks such as active attacks on the DRAM interface (modifying, relocating, splicing). However, it works without an integrity tree and, thus, offers no replay protection through physical attacks. When installing keys, one can specify the mode (*i.e.*, key size and whether integrity is required) and optionally the key itself, which is then bound to the corresponding keyID.

3. Threat Model

The threat model of this work is consistent with existing memory safety mechanisms [25], [31]. We assume a powerful adversary that can perform spatial memory safety violations, *i.e.*, has access to arbitrary read and write primitives. Moreover, we assume the attacker can exploit temporal safety violations, such as UAF errors, to tamper with data stored in memory. Similar to related work, we consider intra-object memory safety violations out-of-scope [31].

This work focuses on heap memory safety since most memory safety vulnerabilities affect the heap [38] and to maintain binary compatibility. Thus, we assume the attacker performs unauthorized memory accesses (*i.e.*, temporal and spatial violations) targeting the heap. In addition, we assume that the adversary cannot bypass our heap allocator due to logical programming errors.

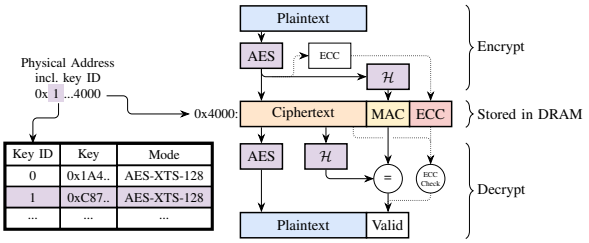

Figure 1. Simplified overview of Intel® TME-MK's memory encryption with integrity. There, key identifiers (keyIDs) are associated with encryption keys and modes. The keyID is integrated into the previously unused upper bits of the physical address stored in the page table entry (PTE). The data in the caches is available in plaintext, however, when interacting with the DRAM, the keyID determines the cryptographic key used to encrypt or decrypt the data, and to calculate the MAC. For writes, the MAC is stored within the ECC-bits in DRAM, and during reads, it is recalculated and compared with the stored value.

Furthermore, we assume that the operating system is benign and free of exploitable bugs and that common security features such as Write-XOR-Execute and ASLR are enabled by default. We consider side-channel [48], [77], microarchitectural [27], [35], and fault attacks [26], [39] out-of-scope for this work.

4. Design

In this section, we introduce *IntegriTag*, a novel technique that detects temporal and spatial memory safety vulnerabilities using cryptographic integrity checks. Existing work [52] that relied on memory encryption could only provide limited data confidentiality and a decreased likelihood of successful attacks through garbling data. We show that *data encryption alone is not sufficient to detect memory safety vulnerabilities* and elaborate on how *IntegriTag* overcomes this limitation, allowing us to reliably *detect* tag mismatches in memory accesses. Our solution supports memory tagging-like capabilities by leveraging the cryptographic integrity checks of Intel® TME-MK.

By designing *IntegriTag* as a purely software-based solution, we facilitate the adoption on commodity hardware. In the following, we elaborate on the design principles of *IntegriTag*. Furthermore, we introduce six memory tagging policies that are tailored towards the underlying memory encryption hardware.

At its core, our design uses memory aliasing to achieve sub-page granular memory encryption. Aliasing allows for fine-grain protection of individual memory allocations. Combining the integrity checks of existing memory encryption hardware with aliasing allows *IntegriTag* to implement memory tagging on commodity x86 CPUs.

4.1. Fine-grain Memory Encryption

Intel® TME-MK, while designed for page-granular encryption, can operate on a smaller granularity when using aliasing (Figure 2). When memory is retrieved, the

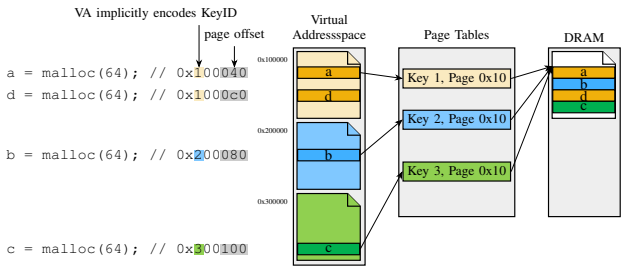

Figure 2. Memory aliasing for fine-grain memory encryption. Each allocation uses a different virtual-to-physical mapping and key. During memory safety violations, e.g., out-of-bounds accesses, the data would be read with a different key compared to the one with which the data was initially written. This results in either garbled data or an exception depending on the system configuration.

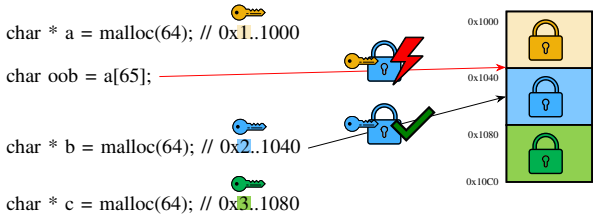

Figure 3. Memory tagging using Intel® TME-MK *with integrity* calculates a message authentication code (MAC) for every cache line. This MAC depends on the used key and is computed over the encrypted data, enabling cryptographic integrity for data. In our case, the key is encoded in the pointer itself. When accessing a cache line with the wrong key (e.g., during an out-of-bounds access), it leads to a MAC mismatch, which is detected.

page table entry (PTE) maps the virtual address to a physical address. This physical address also contains the keyID, which specifies how the physical page should be encrypted or decrypted. By introducing multiple virtual-to-physical mappings with distinct keyIDs, multiple keys can be employed for a single physical page. Consequently, multiple virtual addresses can refer to the same physical memory yet use different keyIDs when accessing the memory. This approach behaves similarly to the pointer tagging methods employed by ARM MTE [55] and SPARC ADI [1], [56], where different memory tags are associated with distinct virtual addresses.

Figure 2 visually represents how aliasing allows for fine-grain memory encryption. There, each of the three allocations uses a different virtual-to-physical mapping. Thus, each mapping uses a distinct key. Using a pointer that does not belong to the allocation leads to a load or store with a wrong keyID. Only the correct alias will yield the correct data when accessing an allocation, as decryption with a wrong key will always produce a garbled output. Viewing the keyIDs as memory tags allows us to consider each alias to be a pointer with a distinct associated tag. Thus, we can use the keyIDs to encode different tags in virtual addresses and use the underlying hardware feature to garble the data for out-of-bounds memory accesses.

However, encryption alone only protects the confidentiality of data stored in memory. Memory safety violations cannot be detected. Thus, silent data corruption is still possible leading to program execution with garbled data (*i.e.*, undefined behavior). Critically, security checks that compare for non-zero values (e.g., `is_admin != 0`) could be bypassed trivially by garbling the underlying data. Furthermore, when computing with garbled data, an attacker can learn when data was changed (*i.e.*, information leakage) and roll back the data to a previous state. Hence, an encryption-only approach [52] provides significantly weaker security properties compared to traditional tagged memory architectures. In the following, we show how to overcome these limitations to match and exceed the security provided by existing tagged architectures.

4.2. Detecting Memory Safety Violations

When using Intel® TME-MK with integrity, each cache line (64 B) is accompanied by a (28-bit) message authentication code (MAC). Figure 3 illustrates the memory tagging enabled by our design using Intel® TME-MK. A MAC is computed over the encrypted data and acts as a fingerprint used to detect data alterations. On each write access, the encryption engine computes and updates the MAC value associated with the encrypted data of the cache line. On subsequent reads, the MAC is recomputed with the key used during access and compared to the stored MAC. If the stored data was corrupted or a wrong key was used for the access, the MAC values do not match, and an exception is raised. Thus, tampering attempts are detected. We use this hardware primitive to implement memory tagging by treating the keyID as the memory tag associated with an allocation. The keyIDs are stored in the upper bits of the physical address in the PTE. Thus, only one tag per PTE is possible. We overcome this limitation through aliasing. Creating multiple virtual-to-physical mappings that point to the same physical memory allows for more than a single tag per page. Through aliasing, *IntegriTag* supports allocations that are smaller than a complete page. This is an essential improvement, as padding each allocation to page size would cause high performance and memory overheads.

Our design leverages the MAC computation and comparison to detect tag mismatches. As the tag is equal to the keyID, using a different tag will lead to a different MAC. If the computed MAC does not match the stored MAC, this is equivalent to a tag mismatch in regular tagged memory architectures. By design, the MAC is only checked when reading from memory. Thus, we can only detect MAC mismatches during read operations. This limitation has implications when *IntegriTag* is used for debugging. Since write operations that use a wrong tag do not trigger a mismatch, the exact origin of a memory safety violation may not be pinpointed. However, reading from the affected memory will detect the violation. From a security perspective, this is not an issue. If the corrupted memory is never read again, the corrupted data is never consumed and has, thus, no effect on the program.

Tagged memory architectures, such as ARM MTE and SPARC ADI, face limited detection capabilities due to their 4-bit tag size, resulting in a detection probability of just 93.75 %. In contrast, our work using Intel® TME-MK supports up to 15-bit keys, significantly enhancing the detection probability to up to 99.997%. Due to the underlying primitive, possible MAC collisions affect the provided security compared to memory tagging. We dis-

cuss this subtle difference in detail in Section 6.

Comparison of KeyIDs and Memory Tags. Traditionally, *memory tags* are additional metadata encoded within pointers and associated with the corresponding memory location. For every memory operation, the provided and the stored tags are compared. In our case, there are several layers of indirection. We create multiple virtual aliases with different keyIDs for a given physical page. These keyIDs are mapped to actual AES keys with which the memory is encrypted and decrypted. Besides encryption and decryption, the key influences the result of the MAC computation.

We detect memory safety violations by comparing the MACs instead of the tags. The aliases themselves are a form of pointer tags, while the MACs can be seen as tags stored in memory. Using a wrong alias (*i.e.*, tag) for an access leads to using a wrong keyID. This wrong keyID, in turn, leads to a wrong encryption key, and the computed MAC will differ from the one that was previously stored in memory. Thus, reading from memory through the wrong alias will be detected through the MAC mismatch.

Note that in theory, the OS could also reuse keyIDs with different underlying encryption keys. However, in our case, we globally allocate all available keyIDs and generate the associated encryption keys only once so that each application can use all keyIDs. Thus, in our case, the term "memory tag" is equivalent to "keyID" and "key". We use the term "memory tags" throughout this paper to refer to keyIDs, keys, and MACs when possible.

4.3. Memory Tagging Policies for Spatial and Temporal Memory Safety

Implementing memory tagging on top of Intel® TME-MK presents unique challenges. Depending on the number of keys used per page, multiple memory aliases and corresponding page table entries are required, which lead to additional TLB entries and, thus, TLB pressure. Using different key IDs within the same page may reduce the effectiveness of prefetching within that page. Furthermore, Intel® TME-MK with integrity limits us to the granularity over which the MAC is computed, which is 64 B. In contrast, heap allocators typically use a minimum granularity of just 16 B. In the following, we detail how *IntegriTag* overcomes these problems with tagging policies specifically designed with these underlying hardware characteristics in mind.

Related work on memory tagging, such as HWASan [66] and MemTagSanitizer [36], assign pseudorandom memory tags for each object. However, random tags cannot provide deterministic protection against both temporal and spatial memory safety vulnerabilities. Furthermore, given the above observations, we can improve the performance by systematically choosing the memory tags. In the following, we detail different tagging policies with different trade-offs by optimizing for deterministic security and performance. Each policy is visually represented in Figure 4. Since we use a bucket allocator, each allocation slot within a bucket (or page) has the same size. In Figure 4, we always show two pages with three or four memory slots each for illustration purposes.

Random Policy. The most straightforward policy is to tag each memory slot with a pseudorandom tag similar to related work [36], [52], [66]. Assume that the tag that is assigned to an allocation is chosen without any constraints by randomly selecting one of the available tags. This means that, due to the randomness, temporally or spatially adjacent memory allocations may have the same tag with a probability of $2^{-keybits}$. Thus, memory safety violations cannot be detected deterministically. However, it can still provide probabilistic protection against most linear and non-linear overflows and temporal memory safety vulnerabilities.

Temporal Policy. Full temporal memory safety is achieved by only changing the tag of a given slot when it has been freed or reallocated. The slot is quarantined once all possible tag values (*i.e.*, keyIDs) have been used, e.g., by incrementing the tag value at every free. Quarantined slots are no longer used for allocations. When all slots on a page are quarantined, the page can be unmapped and returned to the OS. While the physical page can be reused freely, the virtual page is no longer used, thus upholding the temporal memory safety protection. As an optimization, we do not provide spatial memory safety here to reduce the number of keys used per page. This improves the performance in the case that there are not many reallocations. First, a lower number of keys means a lower number of TLB entries. Second, the prefetcher has a higher chance of fetching correctly decrypted data if adjacent cache lines use the same keys.

While aliasing and quarantining reduce the available virtual address space, we deem it unlikely to ever run out of it on modern systems with at least 57-bit addressing. However, orthogonal techniques like memory scanning [16] can be applied here if necessary.

Spatial Policy. As an alternative, we can, e.g., also provide only (reduced) spatial memory protection. For this, we select two tag values that we use for a given page and assign them alternatingly for each adjacent slot on this page. We chose two different tag values for new pages since that does not increase the performance overhead in any way. We still only require two aliases and two TLB entries, irrespective of the tag values themselves. By selecting a different pair of tag values for new pages, we provide spatial protection not only on the same page but across pages as well.

Spatial + Temporal Policy. Both "*Temporal*" and "*Spatial*" can be combined to provide both temporal and spatial memory safety with the optimizations of each of the policies. Initially, we assign different tag values to adjacent slots. E.g., all even slots get an even tag value while all odd slots get a different odd tag value. Then, during free/reallocation, each tag value is incremented by two. This tagging policy always upholds the temporal and spatial protection since (spatially or temporally) adjacent slots always use different tags. Similarly to "*Spatial*", if reallocations and free operations are rare, this policy keeps the number of keys per page low.

Tripwires Policy. A different way of achieving spatial memory protection is to place tripwires between each memory slot. The tripwires are tagged with a unique tag that is never used by any regular allocation. Each memory slot can use the same tag given that only linear under- or overflows occur. By using only one tag per memory

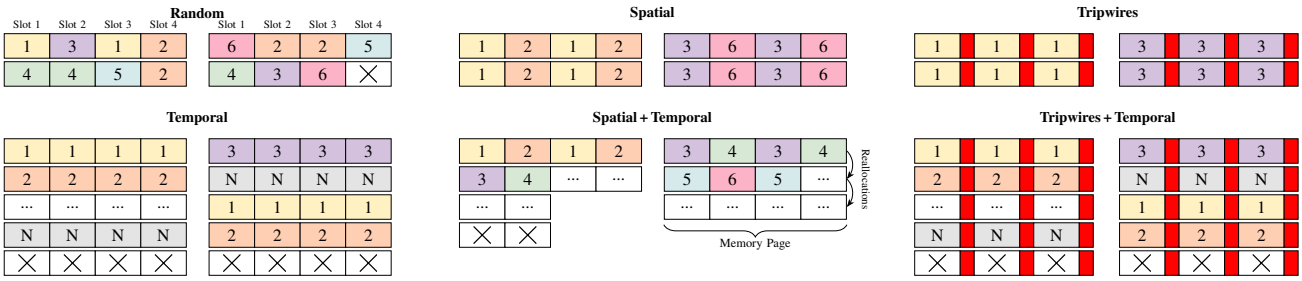

Figure 4. Visualization of our six tagging policies. Each policy shows two pages with three or four allocation slots each. Reallocations are shown in the y-axis, *i.e.*, reallocations are below the previous allocation. Quarantined slots (*i.e.*, slots that have used all possible tag values) are crossed-out. N denotes the number of available keyIDs, while the colors and numbers of each slot signify which keyID is used for that slot.

slot, we minimize the number of required TLB entries, thus improving the performance. At the same time, the chance of a successful prefetch is maximized. This is because the prefetcher uses the address of a previous memory access to fetch adjacent cache lines. Since, for valid memory accesses, the encoded keyID in the virtual address is always the same, the prefetcher will, therefore, fetch and decrypt data with the correct key. However, the trade-off here is that each tripwire must also be the size of a cache line, thus imposing additional memory overhead. While it is possible to store additional metadata for each allocation slot in the tripwire, our proof-of-concept does not implement this optimization. Thus, we treat tripwires as unusable memory in our analysis.

Tripwires + Temporal Policy. On top of "*Tripwires*", we can also provide temporal memory safety through the same approach described for the Spatial + Temporal policy. When freeing or reallocating memory, a new tag is assigned to the slot. Thus, it is impossible to access memory through a dangling pointer. Adding temporal safety increases security, but it comes with the cost of an increased performance impact. The number of used keys and, thus, the TLB pressure increases with each additional free or reallocation. Furthermore, the prefetch efficiency also degrades with an increasing number of distinct keys per page. As we need a unique tag for the tripwires, there is one fewer key that can be assigned.

5. Implementation

In this section, we describe the implementation details of all the components of our design. This includes standard library modifications, the heap allocator, and the Linux kernel patch.

5.1. Standard Library Modifications

Standard libraries such as the GNU C/C++ standard library (glibc) often provide highly optimized functions that fetch more memory than what actually belongs to the current memory object. String-handling functions like `strlen`, for example, potentially read out of the bounds of an allocation. For systems with tagged memory, such superfluous read operations can trigger exceptions that terminate the application. Since these spurious memory safety violations are a well-known problem, we use C^3's patched glibc [28], which ensures that no out-of-bounds accesses happen due to overfetching. Note that similar library modifications were also needed for MTE.

We compile glibc as a dynamic library and link it to the desired target application using `LD_PRELOAD` or the `patchelf` utility.

5.2. Heap Memory Allocator

We chose to implement our design on top of Google's PartitionAlloc [15], which is a bucketing heap allocator that uses no inline metadata between allocations. PartitionAlloc already has a memory tagging implementation tailored to ARM MTE. However, it does not store the tag values of each allocation slot in allocator metadata. Instead, it makes heavy use of the read-tag instruction to load tags from memory when needed.

Memory Tagging Without Reading Tags. Compared to ARM MTE, we cannot easily read tags for a given memory location when using Intel® TME-MK. All access checks are based on cryptographic MACs, thus making it impossible to extract the tag from the stored MAC. By analyzing the existing code base of the allocator, however, we found that reading tags is not needed at all. For allocated memory slots, the owner already has a correctly tagged pointer, which is then passed back to the allocator when that particular memory slot needs to be modified. For untagged slots, there are two possible cases. First, a slot can be an uncommitted range that has not been initialized. Such slots do not have a tag yet, so we do not need to remember the value of the tags. Second, a slot can be committed/initialized but not allocated. Such slots are stored in a singly-linked list called the *freelist*. We modify this list so that it uses tagged pointers instead of pointers that carry no tag. Thus, the tag of each list element is contained in the pointer that points to the element. For convenience, we added two new types to distinguish between untagged slot pointers and tagged slot pointers. Apart from the freelist, no other parts of the code need to know the underlying tags of slots. Thus, it is sufficient to extend the freelist as described above. This modification allows us to use PartitionAlloc without an instruction that reads tags from memory without incurring additional memory overhead.

Metadata Protection. Apart from the freelist, all metadata used by the allocator uses a different encryption key that is never used for normal allocations. Thus, normal heap pointers cannot be used to corrupt our metadata.

PartitionAlloc uses separate metadata for a collection of multiple slots called *SlotSpan*. This metadata contains the pointer to the (first entry of the) freelist. Each entry in the freelist points to the next free memory slot, and each memory slot again contains such a pointer. During free, we assign the slot a new tag. Thus, dangling pointers cannot be used to corrupt the freelist.

Memory Quarantining. Memory slots that have been reallocated multiple times and used up all of the available tags will be quarantined. These slots act like they are allocated (*i.e.*, they are not part of the freelist). To allow memory to be reclaimed, e.g., if all slots on a given memory page are quarantined, we count the number of quarantined slots per so-called *SlotSpan*, which is a collection of multiple slots. PartitionAlloc already has a metadata struct for each such SlotSpan, which has enough free bits to store this information. Thus, no additional metadata storage is required.

Note that we also support disabling quarantining, which allows for more efficient usage of memory at the cost of not being 100 % temporally secure. When a slot has used up all the available tags, it will start reusing them in a round-robin fashion. Thus, after a specific memory slot is reallocated more times than the number of available tags, we can no longer guarantee complete temporal protection in this mode.

Choosing Correct Tags. Some of our tagging policies, such as *Spatial*, use two alternating tags for a given memory page. Since not all slots are initialized at once, we need to remember the two distinct tag values for that page. In PartitionAlloc, so-called *super pages* (*i.e.*, 2 MiB blocks) have their own metadata, which also has enough free (padding) space in the struct to store the two tags used for the page. For *Spatial*, when committing memory (*i.e.*, when initializing slots with their tags), we use the two stored tag values per super page and assign them in an alternating pattern to each even and odd slot. For temporal policies, we also use these two values to keep track of the initial tag value of a slot. Thus, we can detect when we have used up all of the available tags. The tag values are incremented on each free or realloc. Tag values that exceed the number of possible tags wrap around and start again at the first possible tag value. Once the initial tag value is reached, all possible tags have been used up for the respective slot. It is possible to substitute the simple increment operation with e.g., a linear-feedback shift register (LFSR). This yields more unpredictable tag values and helps in case an attacker targets reallocations [69].

5.3. Linux Kernel Modifications

We use the Linux kernel v5.15 and extend the Linux key management facility. We base our work on the initial public Intel® TME-MK patch [19] and expand it to work with newer kernels and our required features. Our modifications allow userspace applications to use encryption keys with integrity support. For this, either the `keyctl` utility or the `add_key` syscall can be used. Similar to [52], we also allow encrypted pages to be aliased such that multiple encryption keys can be used for a single physical page.

Note that our specific prototype implementation only demonstrates that our concept works. However, there may exist many complex interactions with other kernel components and features, such as swapping or copy-on-write, which we have not implemented, as they are not part of our contribution. Similarly, sharing encrypted memory with other processes works as long as both processes map the memory using the same keyID. However, our prototype does not enforce this since our focus is on heap memory.

Memory Initialization. When using integrity protection, it is essential that memory pages used by the OS and by userspace applications are cleanly initialized (e.g., using the keyID 0) such that no unwanted exceptions are triggered in case encrypted memory pages are reused. Thus, we augment the `clear_page` and `copy_page` functions to use the same instruction sequence as the TDX module [9] when initializing new pages. Specifically, the `movdir64b` instruction writes whole cache lines (64 B) at a time, which is the same granularity as used for the MAC computation in Intel® TME-MK. Thus, when reading the same cache line again with the same keyID, the MAC can be verified successfully. Our allocator uses the same instruction when tagging memory areas.

Exception Handling. In case that the wrong keyID is used when accessing a given cache line, the MAC verification fails, which triggers a memory access violation similar to an uncorrectable ECC error. We treat such exceptions similarly to ECC errors and generate a signal such that the userspace application can handle the violation as it sees fit. In the case that no signal handler is registered, the application that triggered the integrity violation is terminated.

Copying Encrypted Pages. Since the same memory gets encrypted differently depending on the physical location in DRAM, pages cannot be copied without knowing the correct keyID with which the data on that page was written. Thus, our prototype currently does not implement support for features like swapping or forking since that requires copying encrypted pages, which only works if the correct keyID is used for each copied cache line in a page. Future work could attempt to emulate a read-tag instruction by testing all 2^{15} keyIDs for each cache line in a page and handling the exceptions accordingly. However, a naive implementation could cause a significant slowdown. A more attractive alternative is to include additional metadata in the allocator. This metadata can store the tags for each slot, which the kernel can then use to copy pages. However, this engineering effort is not part of our proof-of-concept implementation. Furthermore, implementing this feature without creating new exploitable side-channels (e.g., timing or power) is challenging.

6. Security Analysis

In this section, we analyze the security of *IntegriTag* in terms of temporal and spatial heap memory safety. Given that binary compatibility is a crucial property for memory safety countermeasures, and the vast majority of memory safety errors occur within heap memory [38], we exclude vulnerabilities for stack and global memory. Additionally, we exclude intra-object overflows since the heap memory allocator has no type information regarding

the object's internal structure. We assume a software-based attacker as defined in our threat model Section 3. We denote allocations that are placed adjacent to each other in memory as *spatially adjacent*. Allocations that use the same memory location at different points in time are denoted as *temporally adjacent*.

6.1. Systematic Analysis

In the following, we detail how *IntegriTag* protects against temporal and spatial memory safety errors.

Cryptographic Lock-and-Key Mechanism. The underlying hardware primitive that our scheme is based on is Intel® TME-MK with integrity. We configure the relevant MSRs [9], [20] such that encryption with integrity is enabled. The hardware encrypts 64 B cache lines with a secret key that is selected by the provided keyID (encoded within the pointer and the PTE). A 28-bit MAC is stored in ECC memory along the cache line [8]. The MAC authenticates the cache line during memory read operations and is used to detect integrity violations [21]. We use this hardware primitive for the *cryptographic integrity* checks of *IntegriTag*, *i.e.*, apply a cryptographic lock-and-key mechanism. To be precise, a fresh MAC is calculated and compared with the stored version when reading data. Consequently, if the read access uses a wrong key (selected by the keyID), the MACs do not match, and an exception is triggered. This exception is then handled by the OS (or the userspace application itself). Thus, accessing heap objects with the wrong keyID will be detected due to the integrity violation, and the OS will terminate the application.

Moreover, for comparability with other tagged memory architectures, we assume that an attacker knows where our aliases are mapped. Thus, an attacker only needs to correctly guess the keyID encoded in the virtual address, but not where exactly this alias is mapped. Related work [52] described that ASLR can be used to hide aliases to provide additional protection. However, assuming an attacker can observe the addresses of different allocations (*i.e.*, pointer harvesting), ASLR can relatively easily be circumvented. Thus, no additional security is gained from hiding aliases.

In the following, we provide an exemplary analysis of the *Random* policy in terms of temporal and spatial memory safety. In general, the security of *Random* depends on the number of available keyIDs, which can be seen as *memory tags* to use the nomenclature of memory tagging-based lock-and-key mechanisms (cf. ARM MTE and SPARC ADI) (introduced in Section 4.2). The probability that a tag mismatch stays undetected, *i.e.*, a tag collision occurs, is $2^{-keybits}$. For *Random*, this is the probability that a spatially or temporally adjacent slot receives the same tag. Intel® TME-MK is currently specified for up to 15-bit keyIDs ($keybits = 15$). However, the available keyIDs may differ between CPU models.

MAC Verification. Intel® TME-MK verifies the MACs only when reading data from memory (cf. Chapter 14.2 [21]). Writing a (partial) cache line with a wrong key (selected by the keyID) leads to a corrupted MAC. The corruption is not detected immediately but on a subsequent memory read. This behavior is not ideal for memory sanitization or debugging since the exact memory operation that causes the memory safety violation is not immediately detected. However, memory safety violations, where data is only written but never read, do not maliciously affect the program's execution. Thus, this imprecision does not affect the security provided by *IntegriTag*.

MAC Collisions. All security schemes that rely on cryptographic MACs ([33], [37], [43], [63], [68]) have the potential for MAC collisions. In our case, the probability of a MAC collision (second-preimage resistance) for a specific cache line due to a wrong key is 2^{-28}, as the MAC provided by Intel® TME-MK is 28 bits wide. In the case of such a MAC collision, it is possible to access a given cache line with a wrong keyID. However, an attacker would never learn the correct unencrypted data as a wrong keyID leads to decryption with a wrong key. Thus, data confidentiality is preserved, and the attacker only gains access to garbled data. Hence, the only consequence is that we do not immediately detect a memory safety violation if there is a MAC collision. However, as soon as the content of the cache line changes, the MAC of the cache line also changes. Thus, the violation will be detected on the next access with nearly 100 % certainty, *i.e.*, the probability of two consecutive MAC collisions ($1-2^{-56}$). For simplicity, in the remainder of this section, when we state whether a specific memory safety violation can be detected, we assume that no MAC collisions occur.

Multithreading. The underlying authenticated memory encryption hardware verifies memory accesses during read operations. If a pointer with a valid keyID is used, the MAC verification succeeds, and the data is cached. Accesses to cached data also succeed if the keyID matches. If another thread invalidates an allocation (e.g., by calling `free`) while the data is cached, there are two possible outcomes. First, the allocator may flush the affected cache lines with the old keyID. In this case, subsequent accesses with the old keyID will trigger a new MAC verification and cause an exception. If the data is not flushed and evicted from the cache, accessing the cached data with the old keyID will still succeed. However, once the data is written back to memory, it will corrupt the MAC. Thus, subsequent memory fetches will trigger an exception. An exception to this rule happens when the data is also cached with the new keyID and is written back after the cache line with the old keyID has been written to memory. Our experiments showed that, in this case, the cache line stays corrupted. Thus, the violation is still detected even after data with the new (correct) keyID has been written. In summary, our approach leveraging Intel® TME-MK is thread-safe, *i.e.*, an attacker cannot use time-of-check-to-time-of-use (TOCTTOU) [70] attacks to exploit the program. This is a significant advantage over software-based schemes that are challenging to protect against TOCTTOU attacks [75] in practice.

Temporal Memory Safety. Temporal memory safety vulnerabilities can be categorized into the use of uninitialized memory, double-free errors, and use-after-free (UAF) errors. *IntegriTag* leverages our cryptographic lock-and-key mechanism to detect integrity violations of mismatched keyIDs, which can be seen as memory tags. First, we address uninitialized memory use vulnerabilities by ensuring that memory is zeroed during the cache line initialization with the appropriate key upon object allocation. Second, double-free errors are directly mitigated by our design. To accomplish this, the allocator reads the data

(*i.e.*, performing a cryptographic integrity check) before deallocating the corresponding object, which triggers an exception if a dangling pointer is provided. Third, UAF errors, *i.e.*, memory accesses using a dangling pointer with an old keyID, are also detected by the MAC authentication. In particular, we further distinguish between the following three cases of dangling pointers that lead to UAF integrity violations.

The free'd memory is accessed, and no other allocation has (re)used the memory slot yet. During `free`, we re-initialize the slot with a new and different tag. Hence, accessing an old allocation with a dangling pointer triggers an exception due to the MAC mismatch.

The free'd memory is accessed, but another allocation has already (re)used the memory slot. As above, during `free`, the data is assigned a new tag for subsequent usage. Additionally, if memory quarantining is enabled, the tag of the dangling pointer can never match the new tag of the slot. A memory slot that has used up all the available tags will be quarantined and tagged with a special tag not used for regular allocations. Thus, the invalid access is always detected. Suppose memory quarantining is disabled, and the number of reallocations (of that memory slot) is at least the number of available tags. In that case, there is a chance that the tag of the dangling pointer collides with the new tag. This violation is not guaranteed to be detected. Assuming that the number of reallocations is unknown or that reallocations use unpredictable tags, the probability that the current tag matches the tag of the dangling pointer is $2^{-keybits}$.

All slots on the page are quarantined. If all slots on the page have already been quarantined, the page will be unmapped, and we will no longer reuse that virtual page. Accessing such a page through a dangling pointer will result in a page fault, leading to program abortion.

Spatial Memory Safety. Spatial memory safety vulnerabilities can be further categorized into *adjacent* (e.g., linear overflows) and *non-adjacent* (e.g., arbitrary out-of-bound (OOB) error) memory access violations. Similar to temporal safety, *IntegriTag* leverages our cryptographic lock-and-key mechanism, detecting spatial violations by mismatching memory tags (*i.e.*, keyIDs). Memory tagging schemes are bound to a certain *tag granularity*, determining the granularity at which memory violations can be detected. Therefore, we align and pad each allocation to the tag granularity (*i.e.*, Intel® TME-MK with integrity encrypts 64 B cache lines) and assume that accesses within a granule are not considered memory safety issues. While such accesses (*i.e.*, accesses larger than the requested size but still within the tag granule) would indicate a programming error, they do not threaten memory safety since this padding data is never read by the program.

With *Tripwires* (or *Tripwires + Temporal*), all linear overflows and underflows are detected since the directly adjacent memory granules will always have the tag of the tripwire. The tripwire uses a tag that is not used for regular allocations. Thus, it is impossible that the tag of the allocation matches the one of the tripwire by chance. For *Spatial* (or *Spatial + Temporal*), all linear overflows and underflows are detected as well. Since the allocator ensures that no adjacent memory slots use the same tag, it is impossible to access an adjacent slot.

For non-adjacent OOB accesses, detection depends on the tagging policy and whether the access is on the same page. In general, allocations on different pages use different randomly selected tags. For cross-page accesses, the probability of guessing the correct tag is $2^{-keybits}$. Thus, we can detect such violations with a likelihood of up to 99.997% (*i.e.*, 15-bit keyIDs). Whether we detect overflows within the same page depends on the tagging policy. For *Tripwires*, all allocations on the same page use the same tag. Thus, only overflows that touch one of the tripwires will be detected. If *Tripwires + Temporal* is used, the probability of detecting an OOB access depends on whether reallocations happened. As a reallocation causes a new tag to be generated, we can detect violations with a probability of $1 - 2^{-keybits}$. Similarly, for *Spatial* and *Spatial + Temporal*, detection depends on whether the memory access points to an even or odd memory slot. If the OOB access originates from an even slot and targets an odd slot, the tag will not match.

TABLE 1. EMPIRICAL SECURITY ANALYSIS OF *IntegriTag* USING THE NIST JULIET C/C++ TEST SUITE.

CWE	Description	Number of Test Cases	Passed
CWE-122	Heap buffer overrun	2985 Test Cases	100 %
CWE-415	Double-free	818 Test Cases	100 %
CWE-416	Use-after-free	459 Test Cases	100 %

6.2. Empirical Security Analysis

To evaluate the efficacy of our design, we evaluate different types of memory safety vulnerabilities that our allocator can protect against. We use the NIST Juliet C/C++ 1.3 test suite for our analysis [6]. The test suite consists of multiple different variants per common weakness enumeration (CWE), which are further subdivided into different test cases for each variant. Precisely, we use heap buffer overruns (CWE-122), UAF errors (CWE-416), and double-free errors (CWE-415) for our empirical security analysis. While Juliet is designed as a test suite for static analysis tools, it can still be used dynamically. We use it to empirically showcase how our design provides runtime security for applications that contain memory safety vulnerabilities. Note that intra-object memory violations are outside our threat model since a heap-based allocator cannot protect against them without e.g., additional compiler-based instrumentation. Hence, we exclude test cases for intra-object memory safety as well as test cases that already fail without our protection.

We use the Address Sanitizer (ASan) [54] to select the test cases that trigger heap-based memory safety violations. For double-free errors, which ASan does not detect, we consider all test cases that are not explicitly labeled as being "good". A test case is considered to be passed if our design detects the memory safety violation and triggers an exception. Due to our 64 B alignment requirement, we adapt some test cases to use allocations with the appropriate size, such that out-of-bound accesses trigger a violation. Intel® TME-MK only detects violations after reading the (corrupted) data back. Hence, similar to related work [31], we consider any test cases where *IntegriTag* would have caught the violation if a read operation happened afterward as successful. For this, we analyzed each

of those individual test cases and verified this by adding read operations at the appropriate places. Table 1 shows that our proof-of-concept implementation passes all of the test cases. Our results highlight the efficacy of our design and show that it protects against both temporal (CWE-415, CWE-416) and spatial (CWE-122) memory violations.

7. Evaluation

In this section, we evaluate the performance and memory overhead of *IntegriTag* for different configurations.

7.1. Performance Evaluation

Methodology. We evaluate our design for the tagging policies introduced in Section 4.3. The SPEC CPU2017 C/C++ benchmark suite serves as the workload for our evaluation. We evaluate all temporal tagging policies with and without memory quarantining. Furthermore, for better comparability with other schemes that have, e.g., only have 4-bit tags, we evaluate each policy with a range of keyIDs. The number of keyIDs ranges from 2-bit keys up to the maximum supported value by our specific CPU. While Intel® TME-MK is specified for up to 15-bit keyIDs, our system supports 6-bit wide keyIDs. Thus, $2^6 = 64$ different keyIDs are available. We use Linux v5.15 with our kernel extension on an Intel® Sapphire Rapids CPU [2] with 64 GB of RAM. We execute all benchmarks 10 times and plot the median result for each benchmark and the geomean across all benchmarks. We omit plotting standard deviations for clarity and because the runtime of the benchmarks does not follow a normal distribution. All results are normalized to the baseline, which uses the unmodified allocation library. Note that PartitionAlloc currently only allows up to 16 GB of heap memory.

Performance Overhead. Figure 5 shows the geomean of the relative runtimes for our tagging policies when quarantining is disabled. The first bar, at 3 %, overhead, shows the runtime when we only enforce our padding requirement. This value acts as a reference and provides no spatial or temporal protection. We can see that the performance impact increases with an increasing number of keyIDs. There are two main reasons for this behavior. First, more keyIDs mean that more aliases are used per page. This results in more TLB entries and, thus, more TLB pressure. Second, prefetching performance declines because memory that is (pre)fetched from outside the current memory object uses a different encryption key.

2. Intel technologies may require enabled hardware, software or service activation. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. The software development and performance tests were done on an internally available experimental pre-release Sapphire Rapids (SPR) system, customized to enable Intel® Total Memory Encrption – Multi Key (Intel® TME-MK) with cryptographic-integrity for system software. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. No product or component can be absolutely secure. Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade. No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

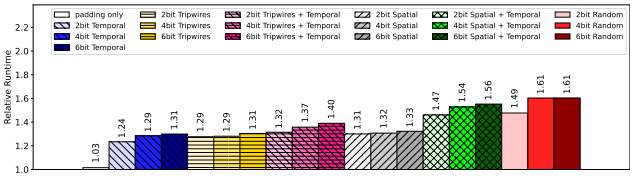

Figure 5. Geomean runtimes of the individual SPEC CPU2017 benchmarks with our different tagging policies *without* quarantining.

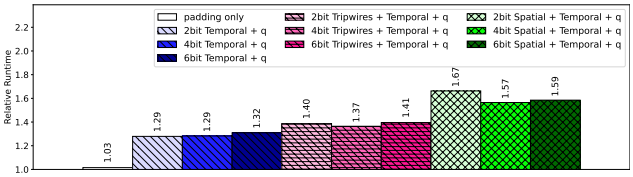

Figure 6. Geomean runtimes of the individual SPEC CPU2017 benchmarks with our different tagging policies *with* quarantining. Policies that do not allow quarantining are omitted.

Our *Temporal* policy provides temporal safety and shows overheads of 24–31 %, depending on the number of keyIDs. Next, the *Tripwires* policy that only offers limited spatial memory safety causes a 29–31 % slowdown. Contrary to the first policy, we notice that more keyID bits do not significantly affect the performance penalty. The main source of additional overhead here is our prototype implementation that unnecessarily creates alias mappings for all keyIDs, even if only one or two keys are used for a given memory mapping. Some of the overhead is, however, caused by the additional tripwires necessary for each allocation, which take up 64 B each. The *Tripwires + Temporal* policy is a combination of the previous two policies, as indicated by the hatching pattern. For this policy, we measure an overhead of 32–40 %. This policy offers both temporal and spatial memory safety. While its intra-page spatial protection is limited to linear overflows, it still provides a good trade-off between security and performance. The *Spatial* policy uses different tags for adjacent allocations, thus providing spatial safety. As expected, since the number of keys per page is kept low, the overheads are also kept at just 32–33 %. The *Spatial* policy adds temporal protection, which increases the performance overhead to 47–56 %. Finally, for comparison, we evaluate the *Random* policy, which randomly assigns keys for each allocation. There, the number of keys per page is usually higher than with other policies. This results in a high performance overhead of 49–61 % while still only providing probabilistic security. Hence, we do not recommend using this policy with Intel® TME-MK.

In summary, our evaluation shows that, compared to other tagged architectures like ARM MTE, how we select the tag values for different memory objects has a strong impact on performance. Across all policies, *Tripwires + Temporal* offers the best trade-off between performance overheads and the provided level of security.

Figure 6 illustrates how the performance impacts of the different policies change when quarantining is enabled. Again, the *Tripwires + Temporal* policy is the recommended policy and shows only slightly higher overheads than the variant without quarantining at 40–41 %. When using quarantining, more keyIDs do not necessarily entail

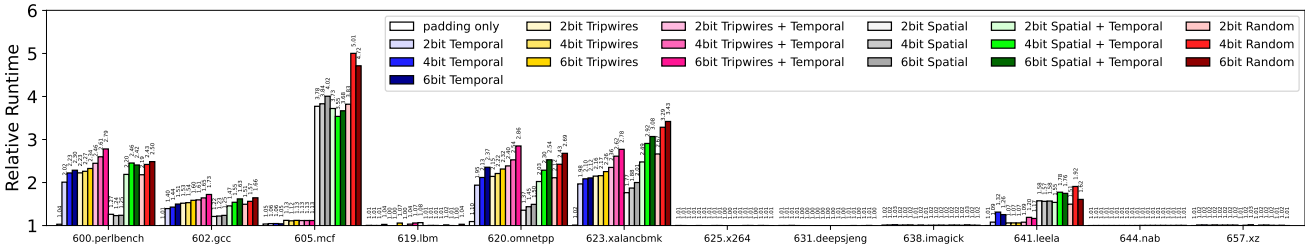

Figure 7. Relative runtime of the individual SPEC CPU2017 benchmarks *without* quarantining.

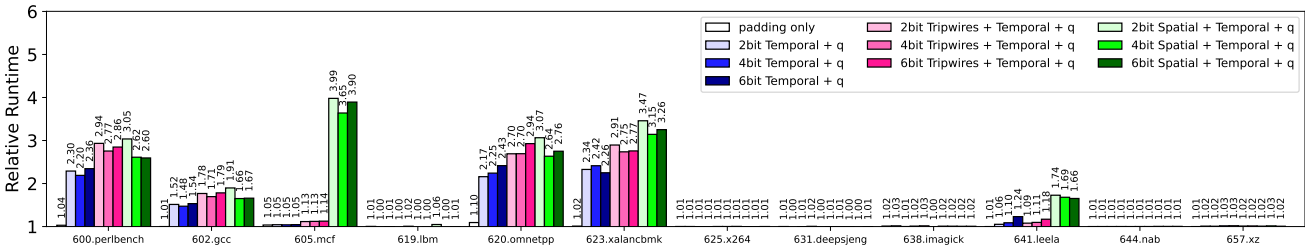

Figure 8. Relative runtime of the individual SPEC CPU2017 benchmarks *with* quarantining.

a stronger performance impact. This is due to the fact that a higher number of available keyIDs means that a memory slot can be reused more often before it is quarantined. This offsets some of the overheads that are inherent to using more aliases.

For completeness, Figure 7 and Figure 8 show the results of each individual benchmark which are summarized in Figure 5 and Figure 6. Since the benchmarks exhibit vastly different memory access patterns, different (re)allocation behavior, and use different allocation sizes [52], the overheads, depending on the policy, also differ.

A 4 KiB page can only have up to $4096/64 = 64$ different allocations. Hence, at any point in time, a page is only ever used by a maximum of 64 different keyIDs. Thus, we assume that if we had access to more than 6-bit (=64) keyIDs, the performance impact would be minimal.

7.2. Memory Usage

Since Intel® TME-MK works on a cache line granularity, we pad and align all allocations to that granularity. This added padding means that we use more memory than baseline using an unmodified allocator. Furthermore, quarantining increases the memory usage since quarantined slots cannot be reused. They can only be reclaimed when all memory slots on a given page have been quarantined.

TABLE 2. MAXIMUM MEMORY USAGE OF THE SPEC BENCHMARKS WITH 6-BIT KEYIDS COMPARED TO THE UNMODIFIED BASELINE.

Tripwires:	**without tripwires**		**with tripwires**	
Quarantining:	✗	✓	✗	✓
600.perlbench	1.00x	1.04x	1.08x	1.20x
602.gcc	1.01x	1.22x	1.22x	1.37x
605.mcf	1.00x	1.00x	1.00x	1.00x
619.lbm	1.00x	1.00x	1.00x	1.00x
620.omnetpp	1.18x	1.43x	1.59x	1.93x
623.xalancbmk	1.07x	1.27x	1.25x	1.54x
625.x264	1.00x	1.00x	1.01x	1.01x
631.deepsjeng	1.00x	1.00x	1.00x	1.00x
638.imagick	1.00x	1.00x	1.00x	1.00x
641.leela	1.00x	1.00x	1.01x	1.00x
644.nab	1.00x	1.00x	1.00x	1.00x
657.xz	1.00x	1.00x	1.00x	1.00x
Geomean	**1.02x**	**1.07x**	**1.09x**	**1.14x**

We measure the physical memory usage of the individual benchmarks by creating a new Linux control group for the benchmark process. Table 2 shows the maximum memory usage measured compared to the baseline. We have an additional need for memory due to our 64 B alignment requirement for both the allocations and tripwires, as well as due to quarantining. We use *Tripwires + Temporal* and *Temporal* with and without quarantining for this evaluation and find that the geomean memory overhead is between 2 % and 14 %. Compared to other memory safety schemes that provide temporal memory safety, these numbers are low. We reach such low memory overhead as the memory encryption engine does not need to store the tags separately in DRAM. Instead, a MAC is calculated and stored in the reserved ECC portion of the DRAM. Thus, storing the tags introduces zero additional memory overhead on systems that already use ECC memory.

Note that there is also a slight increase in used kernel memory due to the additional required page tables for the aliases. However, we were not able to measure this overhead directly. Instead, we estimate the overhead as follows. Each 4 KiB memory page requires an 8 B PTE for each translation layer. Thus, each mapping/alias requires roughly 0.2 % of additional memory for PTEs. We estimate that other metadata, which usually exists per mapped memory range instead of per page, is negligible. Thus, when using 4 KiB pages and 6-bit keyIDs, the overhead is about 0.125x of protected heap memory pages. We consider this an upper bound since a 4 KiB page can not hold more than $2^6 = 64$ different allocations that are 64 B or more. Depending on the actual tagging policy, fewer aliases may be used. The *Tripwires* policy, for example, only requires a single additional alias per page, which results in a 0.002x overhead. Furthermore, both numbers can be further reduced by a factor of 512 if 2 MiB mappings are used instead. However, Partition-

TABLE 3. COMPARISON OF *IntegriTag* WITH EXISTING WORK ON MEMORY SAFETY AVAILABLE ON COMMODITY HARDWARE.

		Spatial	Temp.	Performance Overhead	Memory Overhead
AArch64	ASan [54]‡	●	●	81 %	374 %
	HWASan [66]‡	●	●	108 %	67 %
	MemTagSanitizer [36]	●	●	–	–
	PACMem [32]	●	●	68 %	106 %
x86-64	ASan [54]‡	●	●	86 %	221 %
	Memcheck [46]‡	●	●	1649 %	241 %
	Softbound+CETS [41], [42]‡	●	●	367 %	83 %
	CUP [7]	●	●	158 %	–
	LowFat [12], [13]‡	●	○	237 %	230 %
	BOGO [47], [79]	●	●	60 %	17 %
	MEMES [52]	◐†	◐†	27 %	17 %
	MarkUs [2]	○	●	10 %	16 %
	xTag [4]	○	●	24 %	–
	DangZero [17]	○	●	22 %	25–75 %
	FFmalloc [72]	○	●	2 %	60 %
	CRCount [58]	○	●	22 %	18 %
	IntegriTag*	●	●	24–40 %	2 %
	IntegriTag* w/ quarantining	●	●	29–41 %	14 %

‡ Performance numbers are taken from [32] for better comparability.

† No detection capabilities, computing with garbled data.

* Using 2–6 bits with the policies *Temporal* and *Tripwires + Temporal*.

Alloc would require significant rework to allow committing/decommitting huge pages.

8. Related Work

In this section, we compare *IntegriTag* with prior work on memory safety and memory sanitization. Table 3 compares *IntegriTag* with existing work on memory safety available on off-the-shelf commercially available commodity hardware. The reported performance and memory overheads offer a rough overview of state-of-the-art mechanisms. Note that the evaluations may vary regarding used benchmarks and execution platforms, reducing direct comparability.

Traditionally, spatial memory safety violations are mitigated using bounds-checking mechanisms. Existing research like SoftBound [41], CUP [7], and LowFat [12], [13] highlights the flexibility of software-based solutions achieved through program recompilation to address spatial issues. However, software-based countermeasures tend to introduce high runtime overheads. For instance, SoftBound+CETS [41], [42] introduces a 367 % runtime overhead, potentially impractical for various software systems.

As a result, hardware-supported countermeasures are introduced to potentially achieve better performance results. The CHERI ISA [74] provides spatial memory safety using *fat-pointers* incorporating the required bounds information of the corresponding memory object. Nevertheless, CHERI requires intrusive hardware changes, and the fat-pointer approach leads to ABI and binary incompatibility. In contrast, *IntegriTag* provides a binary and ABI-compatible solution based on commodity hardware.

Temporal safety is typically achieved through metadata that represents the object's liveness. For instance, CETS [42] introduces a unique identifier with each object, which is checked during the program's runtime for every memory access. Moreover, Cornucopia [14] enforces temporal safety for CHERI heaps utilizing heap quarantining and memory scanning. Other countermeasures, like MarkUs [2], also rely on memory quarantining. By managing a quarantine list for freed objects, memory is only reallocated if no dangling pointers referring to the quarantined memory blocks exist. FFmalloc [72] works similarly, but they never reuse allocation slots. In contrast, *IntegriTag* (with the temporal policy) allows the reuse of memory slots multiple times before memory scanning is required. Furthermore, xTag [4] utilizes page aliasing in combination with pointer tagging to achieve temporal safety. DangZero [17] prevents UAF errors using direct page table access. Compared to xTag and DangZero, *IntegriTag* performs the access checks in hardware and can also support spatial safety.

Memory Sanitization. Address Sanitizer (ASan) [54] and Valgrind's Memcheck [46] are widespread memory sanitizers used for debugging and testing C/C++ software. In addition, PACMem [32] provides memory sanitization using the ARM pointer authentication in combination with bound checks. Typically, the performance and memory overheads of memory sanitizers are too high for runtime safety. In contrast, *IntegriTag* achieves efficient memory sanitization and runtime protection by leveraging the Intel® TME-MK hardware feature.

Memory Tagging. Memory tagging is a core building block for various security countermeasures [1], [53], [55], [56], [69], [71]. Several memory tagging schemes employ the lock-and-key approach for memory safety in software and hardware. For instance, HWASan [66] utilizes ARM's TBI feature to support software-based memory tagging using shadow memory. However, HWASan suffers from relatively high performance and memory overheads.

In contrast, ARM MTE [55] and SPARC ADI [1], [56] provide memory tagging integrated into the system architecture. In particular, ARM MTE and SPARC ADI use a 4-bit memory tag with a granularity of 16 B and 64 B, respectively. The chosen tag size and tag granularity directly influence the resulting memory overhead. Both tagged memory architectures are used for probabilistic memory safety using pseudorandom memory tags. For instance, the MemTagSanitizer [36] integrated into the Clang/LLVM [30] compiler provides memory sanitization by leveraging ARM MTE. Importantly, *IntegriTag* provides significantly better detection capabilities of up to 15-bit (99.997% detection probability), directly compared to the 4-bit (93.75 % detection probability) security of ARM MTE and SPARC ADI.

StarScan [16] utilizes memory quarantining in combination with ARM MTE in order to enforce temporal memory safety. Similar to our temporal tagging policy, StarScan puts memory objects into a quarantine list after reusing them 16 times. Notably, our approach allows us to reuse the memory objects significantly more often, *i.e.*, 15-bit key space compared to the 4-bit memory tags of ARM MTE.

Cryptographic Computing. Several memory safety countermeasures rely on the usage of cryptographic primitives for security [31], [33], [43], [52], [68].

CrypTag [43] leverages cache line granular encryption to enforce temporal and spatial memory safety for the RISC-V architecture. Specifically, CrypTag uses pointer tagging (*i.e.*, lock-and-key approach) to encode a pseudorandom memory tag within the pointer, similar to ARM

MTE and SPARC ADI. However, instead of storing the memory tags in a tagged memory architecture, CrypTag uses the tag metadata as additional input for the memory encryption engine. Thus, every memory object needs to be padded to cache line granularity. Depending on the selected mode of the memory encryption engine (*i.e.*, encryption-only or authenticated encryption), accessing memory using the wrong memory tag (*i.e.*, a memory safety violation) leads to garbled data, or an authentication error triggers a hardware exception.

Cryptographic capability computing (C^3) [31] uses a combination of pointer and memory encryption to mitigate the exploitation of memory safety vulnerabilities. C^3 derives a so-called cryptographic address (CA) by encrypting the upper parts of the pointer. Additionally, the CA is used for a keystream generator responsible for encrypting and decrypting the data stored in the data cache. A memory safety violation then either leads to the decryption of a mangled pointer (and likely to a page fault) or the decryption of garbled data (by using the wrong CA). In contrast to our work, in the second case, C^3 cannot detect memory safety violations, resulting in program execution with garbled data.

MEMES [52] uses aliasing in combination with memory encryption but only provides limited memory exploit mitigation. Without integrity, MEMES cannot detect memory safety violations, and program execution with garbled data is still possible. EC-CFI [44] provides cryptographic control flow integrity utilizing Intel® TME-MK with virtualization by encrypting functions with different keys and switching keys using `vmfunc` when entering/exiting a function. Calling a function with a wrong key leads to decoding garbled data as instructions, which likely leads to illegal instruction exceptions. Thus, EC-CFI mitigates fault attacks trying to hijack the program's control flow.

9. Conclusion

This paper presented *IntegriTag*, a novel mechanism for memory safety by leveraging the existing memory encryption feature of modern CPUs. By assigning different keyIDs to objects (enabled by aliasing), we are able to implement memory tagging for commodity x86 CPUs. Our software-based approach introduces memory tagging capabilities, similar to ARM MTE and SPARC ADI, for the x86 architecture. Unlike ARM MTE and SPARC ADI, our approach does not rely on a separate tagged memory architecture that increases system complexity and memory overhead. Instead, we perform implicit memory access checks based on the Intel® TME-MK memory encryption engine. Additionally, *IntegriTag* provides significantly better detection capabilities of up to 15-bit (99.997% detection probability), directly compared to the 4-bit (93.75 % detection probability) security of ARM MTE and SPARC ADI.

We developed a software framework supporting various memory tagging policies using a custom heap allocator and a kernel patch. The proof-of-concept prototype provides effective heap memory safety while maintaining binary compatibility. Moreover, we evaluate several security policies based on our encryption-based lock-and-key mechanism, enforcing temporal and spatial memory safety on the architectural level. Our extensive security analysis, including a security evaluation based on the NIST Juliet C/C++ test suite, highlights the strong detection capabilities and efficacy of our design. Our performance evaluation, across all C/C++ SPEC CPU2017 benchmarks, highlights competitive results of a 32–41% overhead for both temporal and spatial memory safety.

Data Availability

We release our implementation as open-source at github.com/IntelLabs/TME-MK-Fine-Grained-Encryption-Integrity. The repository includes all three parts of our implementation (*i.e.*, patches for glibc, the Linux kernel, and PartitionAlloc) as well as instructions on how to compile it, run it, and verify that the necessary CPU feature(s) are correctly set up.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback which improved this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087). Additional funding was provided by a generous gift from Intel.

References

[1] Kathirgamar Aingaran, Sumti Jairath, Georgios K. Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, and Thomas Wicki. M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro*, 2015.

[2] Sam Ainsworth and Timothy M. Jones. MarkUs: Drop-in use-after-free prevention for low-level languages. In *S&P'20*, 2020.

[3] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL'03*, 2003.

[4] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davi. xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In *EURO S&P'22*, 2022.

[5] Hans-Juergen Boehm and Mark D. Weiser. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exp.*, 1988.

[6] Tim Boland and Paul E. Black. Juliet 1.1 C/C++ and Java Test Suite. *Computer*, 2012.

[7] Nathan Burow, Derrick Paul McKee, Scott A. Carr, and Mathias Payer. CUP: Comprehensive User-Space Protection for C/C++. In *AsiaCCS'18*, 2018.

[8] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX Demystified: A Top-Down Approach. *CoRR*, 2023.

[9] Intel Corporation. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2020.

[10] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. In *ASPLOS'08*, 2008.

[11] Baozeng Ding, Yeping He, Yanjun Wu, Alex Miller, and John Criswell. Baggy Bounds with Accurate Checking. In *23rd IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Dallas, TX, USA, November 27-30, 2012*, 2012.

[12] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *CC'16*, 2016.

[13] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *NDSS'17*, 2017.

[14] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *S&P'20*, 2020.

[15] Google. Efficient and safe allocations everywhere! https://blog.chromium.org/2021/04/efficient-and-safe-allocations-everywhere.html, 2021. Accessed: 2023-07-26.

[16] Google. Retrofitting temporal memory safety on c++. https://security.googleblog.com/2022/05/retrofitting-temporal-memory-safety-on-c.html, 2022. Accessed: 2023-07-26.

[17] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In *CCS'22*, 2022.

[18] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX'08*, 2008.

[19] Intel. Intel MKTME enabling. https://lore.kernel.org/lkml/20190731150813.26289-8-kirill.shutemov@linux.intel.com/t/, 2019. Accessed: 2023-09-13.

[20] Intel. Intel Architecture Memory Encryption Technologies. https://www.intel.com/content/www/us/en/content-details/679154/intel-architecture-memory-encryption-technologies-specification.html, 08 2022. Revision 1.4, Accessed: 2023-01-31.

[21] Intel®. 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, 2020. Accessed: 2023-01-30.

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

[23] Samuel Jero, Nathan Burow, Bryan C. Ward, Richard Skowyra, Roger Khazan, Howard E. Shrobe, and Hamed Okhravi. TAG: Tagged Architecture Guide. *ACM Comput. Surv.*, 2023.

[24] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, 2002.

[25] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Hardware-based Always-On Heap Memory Safety. In *MICRO'20*, 2020.

[26] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA'14*, 2014.

[27] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P'19*, 2019.

[28] Intel Labs. c3-glibc. https://github.com/IntelLabs/c3-glibc/, 2023. Accessed: 2023-09-14.

[29] Lukas Lamster, Martin Unterguggenberger, David Schrammel, and Stefan Mangard. HashTag: Hash-based Integrity Protection for Tagged Architectures. In *USENIX'23*, 2023.

[30] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*, 2004.

[31] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. Cryptographic Capability Computing. In *MICRO'21*, 2021.

[32] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *CCS'22*, 2022.

[33] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX'19*, 2019.

[34] ARM Limited. Arm architecture reference manual for armv8, for armv8-a architecture profile. https://developer.arm.com/documentation/ddi0487/ea, 2019. Accessed 2023-09-01.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX'18*, 2018.

[36] LLVM Project. Memtagsanitizer. https://releases.llvm.org/11.0.0/docs/MemTagSanitizer.html, 2020. Accessed: 2023-07-26.

[37] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP'13*, 2013.

[38] Microsoft. 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, 2019. Accessed: 2023-02-26.

[39] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P'20*, 2020.

[40] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ISCA'12*, 2012.

[41] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. SoftBound: highly compatible and complete spatial memory safety for c. In *PLDI'09*, 2009.

[42] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In *ISMM'10*, 2010.

[43] Pascal Nasahl, Robert Schilling, Mario Werner, Jan Hoogerbrugge, Marcel Medwed, and Stefan Mangard. CrypTag: Thwarting Physical and Logical Memory Vulnerabilities using Cryptographically Colored Memory. In *ASIACCS'21*, 2021.

[44] Pascal Nasahl, Salmin Sultana, Hans Liljestrand, Karanvir Grewal, Michael LeMay, David M. Durham, David Schrammel, and Stefan Mangard. EC-CFI: Control-Flow Integrity via Code Encryption Counteracting Fault Attacks. In *HOST'23*, 2023.

[45] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *POPL'02*, 2002.

[46] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07*, 2007.

[47] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *SIGMETRICS'18*, 2018.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA'06*, 2006.

[49] Alex Rebert and Christoph Kern. Secure by design: Google's perspective on memory safety. Technical report, Google Security Engineering, 2024.

[50] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. HeapCheck: Low-cost Hardware Support for Memory Safety. *ACM Trans. Archit. Code Optim.*, 2022.

[51] Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. Practical Byte-Granular Memory Blacklisting using Califorms. In *MICRO'19*, 2019.

[52] David Schrammel, Salmin Sultana, Karanvir Grewal, Michael LeMay, David M. Durham, Martin Unterguggenberger, Pascal Nasahl, and Stefan Mangard. MEMES: Memory Encryption-Based Memory Safety on Commodity Hardware. In *Proceedings of the 20th International Conference on Security and Cryptography, SECRYPT 2023, Rome, Italy, July 10-12, 2023*, 2023.

[53] David Schrammel, Moritz Waser, Lukas Lamster, Martin Unterguggenberger, and Stefan Mangard. SPEAR-V: Secure and Practical Enclave Architecture for RISC-V. In *ASIACCS'23*, 2023.

[54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC'12*, 2012.

[55] Kostya Serebryany. ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. *login Usenix Mag.*, 2019.

[56] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory Tagging and how it improves C/C++ memory safety. *CoRR*, 2018.

[57] Rasool Sharifi and Ashish Venkat. CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities. In *ISCA'20*, 2020.

[58] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *NDSS'19*, 2019.

[59] Matthew S. Simpson and Rajeev Barua. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. In *Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010, Timisoara, Romania, 12-13 September 2010*, 2010.

[60] Kanad Sinha and Simha Sethumadhavan. Practical Memory Safety with REST. In *ISCA'18*, 2018.

[61] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *S&P'16*, 2016.

[62] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for Security. In *S&P'19*, 2019.

[63] Stefan Steinegger, David Schrammel, Samuel Weiser, Pascal Nasahl, and Stefan Mangard. SERVAS! Secure Enclaves via RISC-V Authenticryption Shield. In *ESORICS'21*, 2021.

[64] Michael B. Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W. Keckler. Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC. In *ISCA'23*, 2023.

[65] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *S&P'13*, 2013.

[66] The Clang Team. Hardware-assisted addresssanitizer design documentation. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html. Accessed 2023-10-05.

[67] Tolga Yalcin - Google. Need for low-latency ciphers: A comparative study of nist lwc finalists. https://csrc.nist.gov/Presentations/2022/need-for-low-latency-ciphers-a-comparative-study-o. Accessed 2023-10-23.

[68] Martin Unterguggenberger, David Schrammel, Lukas Lamster, Pascal Nasahl, and Stefan Mangard. Cryptographically Enforced Memory Safety. In *CCS'23*, 2023.

[69] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging. In *ASIACCS'23*, 2023.

[70] Jinpeng Wei and Calton Pu. 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*, 2005.

[71] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *NDSS'19*, 2019.

[72] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, Jungwon Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *USENIX'21*, 2021.

[73] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *ASPLOS'02*, 2002.

[74] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA'14*, 2014.

[75] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches. In *S&P'23*, 2023.

[76] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *ASPLOS'21*, 2021.

[77] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX'14*, 2014.

[78] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *OSDI'08*, 2008.

[79] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *ASPLOS'19*, 2019.

[80] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-FAT: Architectural Support for Low Overhead Memory Safety Checks. In *ISCA'21*, 2021.