Cryptographically Enforced Memory Safety

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

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

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

Pascal Nasahl
pascal.nasahl@iaik.tugraz.at
Graz University of Technology
Graz, Austria

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

ABSTRACT

C/C++ memory safety issues, such as out-of-bounds errors, are still prevalent in today's applications. The presence of a single exploitable software bug allows an adversary to gain unauthorized memory access and ultimately compromise the entire system. Typically, memory safety schemes only achieve widespread adaption if they provide lightweight and practical security. Thus, hardware support is indispensable. However, countermeasures often restrict unauthorized access to data using heavy-weight protection mechanisms that extensively reshape the processor's microarchitecture and break legacy compatibility.

This paper presents cryptographically sealed pointers, a novel approach for memory safety based on message authentication codes (MACs) and object-granular metadata that is efficiently scaled and stored in tagged memory. The MAC cryptographically binds the object's bounds and liveness information, represented by the corresponding address range and memory tag, to the pointer. Through recent low-latency block cipher designs, we are able to authenticate sealed pointers on every memory access, cryptographically enforcing temporal and spatial memory safety. Our lightweight ISA extension only requires minimal hardware changes while maintaining binary compatibility. We systematically analyze the security and efficacy of our design using the NIST Juliet C/C++ test suite. The simulated performance overhead of our prototype implementation showcases competitive results for the SPEC CPU2017 benchmark suite with an average overhead of just 1.3 % and 9.5 % for the performance and efficiency modes, respectively.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Security in hardware**; • **Computer systems organization → Architectures**.

KEYWORDS

Memory Safety, Low-latency Cryptography, Memory Tagging

1 INTRODUCTION

The recent report on software memory safety by the U.S. National Security Agency (NSA) [1] strongly emphasizes the need for immediate actions to mitigate C/C++ memory errors. Such efforts are especially relevant for system software, ranging from (cloud) operating systems to native applications, as memory errors are the leading cause of software attacks [30, 57]. Especially C/C++ programs are notoriously susceptible to memory bugs (*i.e.*, temporal and spatial memory safety errors [81]) due to the flexible handling of memory references. This often leads to programming errors, resulting in exploitable software. Spatial memory safety violations, like Heartbleed [25] (CVE[1]-2014-0160), allow attackers to remotely leak or modify data of arbitrary memory locations by exploiting out-of-bounds (OOB) memory errors. Temporal memory safety violations such as use-after-free (UAF) errors, e.g., in Google Chrome [30] (CVE-2016-1635), are introduced by dereferencing a *dangling pointer* to an already freed memory object. A single exploitable software bug can create a large attack surface that allows for a variety of attacks. Sophisticated exploitation techniques allow an adversary to hijack the program's control flow [10, 15, 18, 67, 76], acquire unauthorized access to resources in memory [8, 16, 34, 37], or execute arbitrary code [4, 28]. To protect systems against these powerful attacks, *temporal* and *spatial* memory safety is required.

Security mechanisms have been proposed using software-only and hardware-enforced approaches. Software-based approaches [13, 39, 63, 79] suffer from large runtime overheads (e.g., SoftBound [61] and CETS [60] incur a 116 % performance overhead) and possible race conditions in multi-threaded environments [85, 90]. Thus, the focus in academia and industry shifts more and more to hardware-enforced mechanisms for memory safety. ISA extensions [22, 41, 59, 64, 69, 78, 91, 94] gain momentum due to reasonable performance results. Typically, these countermeasures, such as CHERI [27, 84, 88], use a *fat pointer* approach to enforce bounds-checks in hardware. Such hardware-enforced mechanisms heavily rely on fundamental microarchitectural modifications and cannot preserve ABI or binary compatibility, thus hindering widespread adoption. Contrarily, commercial products, like the ARM memory tagging extension (MTE) [74] and SPARC application data integrity (ADI) [2, 75], suffer from limited protection capabilities, *i.e.*, 4-bit security that yields high tag collision probabilities of 1/16.

In order to find suitable trade-offs between security and efficiency, cryptography has turned out to be a promising approach. PointGuard [21], ARM pointer authentication (PAuth) [82], and CCFI [55] utilize cryptographic primitives to provide pointer integrity (a subset of memory safety for code and data pointers [36, 44, 49]). However, ARM PAuth and CCFI only protect pointers *stored in memory*. Academic designs like C^3 [48] and CrypTag [62] enforce memory safety through memory encryption. Yet, without

[1]Common vulnerabilities and exposures (CVE) by MITRE [19] define and classify publicly disclosed cybersecurity vulnerabilities such as C/C++ memory safety issues.

authentication, both countermeasures could suffer from silent data corruption (*i.e.*, program execution with garbled data).

Contributions. In this paper, we present cryptographically sealed pointers, a novel architectural mechanism to detect and mitigate memory safety violations with cryptographic strength. **We cryptographically bind the object's bounds and liveness to the pointer itself**. For this, we encode the object's size and message authentication code (MAC) into the upper bits of the pointer. We compute the MAC using the address, encoded length information, and object metadata from tagged memory to cryptographically link the bounds and liveness information to the pointer. In contrast to ARM PAuth and CCFI, our approach utilizes low-latency MACs in combination with efficient tagged memory to provide strong security guarantees for both temporal and spatial memory safety. Newly developed low-latency block cipher designs allow us to perform *cryptographic access checks during every memory operation*. We significantly enhance security by introducing the novel concept of tag accumulation that, unlike ARM MTE and SPARC ADI, enables us to *efficiently scale our tagged architecture*. Our approach has several advantages: First, the MAC binds the pointer with a specific address range and memory tag, limiting access solely to the data stored within that range. Similar to capability-based systems [84, 88], these pointers grant only access to the memory location within the address range rather than the entire address space where two memory tags could coincide. Second, the (larger) MAC expands the tag space of the underlying tagged memory architecture, as the attacker must accurately guess the correct MAC of the corresponding address to mount a successful attack. Third, the MAC computation prevents tag forging attacks, mitigating security issues when tagging policies rely on deterministically chosen memory tags.

Our ISA extension only requires minor hardware changes to enforce *lightweight and transparent protection while preserving binary compatibility*. We introduce two modes of operation for our design: the efficiency and the performance mode. Our efficiency mode provides area-efficient memory safety. While our performance mode, on the other hand, utilizes multiple cipher instances to minimize the impact on the system performance. We provide an extensive security analysis of our design by systematically analyzing all classes of memory safety vulnerabilities. Moreover, we analyze the efficacy of our design using the NIST Juliet C/C++ [11] test suite, empirically validating the protection against temporal and spatial memory exploitation. Our performance evaluation, based on the gem5 [9, 52] system simulator, as well as our area and memory utilization analysis, highlight the practicality of our design.

In summary, our contributions are as follows:

(1) **We present cryptographically sealed pointers.** A novel architectural mechanism that enforces temporal and spatial memory safety by cryptographically binding the object's bounds and liveness information to the pointer itself.
(2) **We introduce the new concept of tag accumulation.** Tag accumulation enables us to efficiently scale our tagged architecture using low-latency cryptography, substantially enhancing security while minimizing the memory overhead.
(3) **We prototype our design.** Our proof-of-concept implementation consists of a hardware model of our lightweight ISA extension and a custom memory allocator.
(4) **We systematically analyze the security of our design.** Our comprehensive analysis, including an empirical evaluation based on the NIST Juliet C/C++ test suite, underlines the strong security guarantees and efficacy of our design.
(5) **We demonstrate the practicability of our design.** Our performance evaluation showcases competitive results for the performance and efficiency modes.

Outline. The paper is structured as follows: Section 2 discusses the background on memory safety. Section 3 defines our threat model and assumptions. Section 4 and Section 5 present our design and implementation. Section 6 provides a systematic and empirical security analysis. Section 7 evaluates the performance, area, and memory utilization. Section 8 discusses related and future work. Section 9 concludes this work.

2 BACKGROUND

This section elaborates on memory safety, cryptographic computing, and tagged memory architectures.

2.1 Memory Safety Vulnerabilities

Software built with memory-unsafe programming languages such as C/C++ is highly susceptible to memory errors [1]. More specifically, memory safety vulnerabilities introduced by programming bugs allow for powerful software attacks. Such programming errors frequently occur in large and complex software projects, as recent surveys from Microsoft [57] and Google [30] indicate. Their surveys highlight that approximately 70 % of security bug fixes in production software are classified as memory safety issues. Depending on the underlying cause, memory safety errors can be categorized into two classes: *temporal* and *spatial* safety vulnerabilities [81].

Spatial Memory Vulnerabilities. Spatial memory errors allow an adversary to leak or modify data stored in *adjacent* or *non-adjacent* memory locations. These errors occur whenever a pointer is dereferenced outside the intended boundaries of the corresponding memory object. For example, a heap buffer overrun (CWE[2]-122) occurs when dereferencing a pointer using an incorrect array index. Such overruns can cause overflows into adjacent memory objects. Furthermore, an arbitrary access violation, like an out-of-bounds (OOB) memory access (non-adjacent), can be misused by an adversary to attack and ultimately compromise the entire system. Besides adjacent and non-adjacent access violations, intra-object memory access violations occur if data is modified within an object outside the intended internal boundaries, e.g., a linear overflow into an adjacent member of a C structure.

Temporal Memory Vulnerabilities. Temporal memory violations allow an adversary to manipulate data of already freed or reallocated memory objects. Typically, pointers referring to already freed memory locations are called *dangling* or *stale pointers*. An adversary can perform so-called use-after-free (UAF) attacks (CWE-416) by misusing a dangling pointer to access the data of a freed memory object. Depending on the state of the memory, *i.e.*, freed or reallocated, the UAF violation can be used to tamper with data of another object located on the same chunk of memory. Furthermore, uninitialized memory accesses (CWE-457) leak data of an

[2]Common weakness enumeration (CWE) by MITRE [20] is a categorized list of security weaknesses, including software vulnerabilities such as C/C++ memory safety issues.

object previously located at the same memory location. In addition, double-free violations (CWE-415) tamper with the free list of the heap allocator or, even more severely, free a memory object currently used by another part of the program.

2.2 Cryptographic Computing

Cryptographic building blocks, such as encryption and authentication, can be leveraged in the context of memory safety, memory integrity, and isolation technologies [21, 48, 55, 62, 70, 82]. Multiple academic and commercial designs enforce memory protection via cryptographic primitives.

Cryptographic Pointer Integrity. Pointer encryption or authentication [36, 49] mitigates attacks that rely on the manipulation of pointers stored in memory (a subset of memory safety) to hijack the control flow of the program. PointGuard [21] uses pointer encryption to prevent attackers from tampering with code pointers. ARM pointer authentication (PAuth) [82] utilizes message authentication codes (MACs) encoded into the upper bits of the pointers to enforce pointer integrity. Control flow hijacking attacks are mitigated by signing and verifying the pointer, detecting any manipulation by an adversary while the pointer is stored in memory. Similarly, CCFI [55] uses MACs to provide integrity for code pointers.

Cryptographic Memory Safety. Recent academic work proposes the use of cryptographic primitives to enforce full memory safety. CrypTag [62] uses pointer tagging (*i.e.*, encoding metadata in the upper bits of a pointer) to enforce memory safety through cache line granular encryption. Every memory object is padded to the cache line size, and the pointer tag is used as additional input for the memory encryption engine. Memory safety violations lead to garbled data or an authentication error, depending on the underlying mode of operation (*i.e.*, encryption-only or authenticated encryption).

Cryptographic capability computing (C^3) [48] leverages low-latency pointer and memory encryption to mitigate software attacks. By partly encrypting data pointers, C^3 generates a cryptographic address (CA) that provides pointer integrity. The CA acts as a nonce for the keystream generator used to encrypt and decrypt the accessed data. This procedure introduces two layers of defense: First, forging an arbitrary pointer most likely leads to a garbled pointer and, thus, in most cases to a page fault and program termination. Second, if the pointer decryption leads to a valid result, the data decryption using the CA will result in garbled data with high probability. However, due to its stateless design, C^3 is vulnerable to XOR-based attacks exploiting the garbled data, and UAF attacks introduced by the low entropy of the 4-bit version field [33].

2.3 Tagged Memory Architectures

Tagged memory constitutes a promising approach for mitigating a wide range of memory safety issues. In general, memory tagging [75] associates meta-information with every memory location, thus allowing the enforcement of different security policies [87, 93]. The metadata, called *memory tags*, is software-controlled and parameterizable by the tag size and granularity. The chosen tag size and tag granularity directly determine the incurred memory overhead as well as the range of implementable policies. Tagged memory is a versatile tool and can be implemented in software or with hardware support. Various tagged architectures [2, 38, 71, 74, 87, 93] have been proposed, ranging from single-bit to multi-bit memory tags and static to (partly) programmable security policies.

Memory Safety Policies. Commercial products, such as ARM memory tagging extension (MTE) [74] and SPARC application data integrity (ADI) [2], are ISA extensions integrating memory tagging into the system architecture. Specifically, ARM MTE and SPARC ADI utilize a 4-bit tag size at the granularity of 16 B and 64 B, respectively. Both ISA extensions use memory tagging to enforce security policies for memory safety with a so-called *lock-and-key* approach. At its core, a pseudorandomly chosen tag is assigned to a memory location and the corresponding pointer during allocation. The tag gets encoded into the upper bits of the pointer (the *key*) using hardware address masking, e.g., ARM's top-byte ignore (TBI) feature. Subsequently, the tag gets assigned to the corresponding memory location (the *lock*) using the tagged memory architecture. During every memory operation, the key and the lock are compared. Only possession of the correct key grants access to the memory location, *i.e.*, the pointer tag (*key*) and the memory tag (*lock*) match.

3 THREAT MODEL AND ASSUMPTIONS

Similar to prior work [41, 48], we consider an adversary that attempts to exploit one or several memory safety vulnerabilities present in the target user space program. This includes spatial safety issues such as linear buffer overruns (adjacent) and out-of-bounds (OOB) errors (non-adjacent). Additionally, temporal safety issues such as use-after-free (UAF), uninitialized memory, and double-free errors are included in the threat model. Our work focuses on heap protection, as most memory safety vulnerabilities only concern heap memory [57]. While our proof-of-concept implementation enforces heap memory safety, the design can also be adapted to protect the stack and global memory. In accordance with related work [41, 48], we exclude intra-object overflows from our threat model since these overflows are hard to exploit and only account for around 1 % of observed vulnerabilities [56].

We assume that the adversary knows the address space layout and that the OS is benign and free of exploitable bugs. Moreover, we assume that the attacker cannot bypass the heap allocator due to logical errors. Furthermore, we assume that signing instructions, which have to exist in the heap allocator, cannot be misused, e.g., through selectively protecting the heap library [29, 77]. As modern systems enforce a strict no-execution policy on writable memory, we assume that Write-XOR-Execute is enabled. Thus, the attacker cannot inject and execute custom code. We consider side-channel attacks [65, 92], microarchitectural attacks [43, 50], and software fault attacks [42, 58] to be out-of-scope of this work.

4 DESIGN

In this section, we present cryptographically sealed pointers, a novel architectural mechanism that uses low-latency cryptography in combination with object-granular metadata that is efficiently scaled and stored in tagged memory to enforce memory-safe execution.

4.1 Overview

The fundamental idea of our design is to *cryptographically enforce temporal and spatial memory safety by binding the object's bounds*

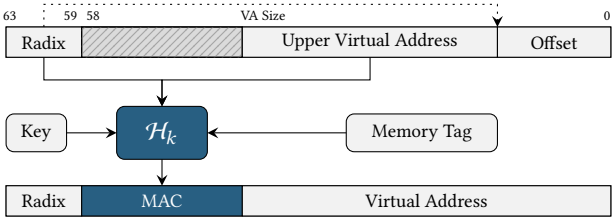

Figure 1: The pointer layout and MAC generation. We utilize a radix field to distinguish between the fixed upper virtual address of a pointer and the offset used for pointer arithmetic operations. For the MAC generation, we use the fixed part of the pointer's address, the radix, and the object-specific memory tag in combination with a secret per-process key.

and liveness to the pointer. For this, we leverage message authentication codes (MACs) embedded into the pointer and associated metadata (*i.e.*, memory tag) stored in tagged memory to perform fine-grain access checks. In order to link the bounds and liveness information to the pointer, we compute the MAC using a memory object's address and size, as well as an associated memory tag.

Our hardware architecture synchronously verifies the MAC encoded in the pointer during every memory access. Thus, *every memory access implicitly authenticates the pointer* enforcing memory safety for our scheme. This inherently mitigates any potential race conditions in multi-threaded environments. We utilize low-latency cryptographic primitives for the MAC computation, as the frequent verification must be highly efficient. Successful verification of a pointer guarantees that the memory access is within the *object's boundaries* and attests the *liveness* of the referred object. A failed verification denotes that the pointer is invalid, e.g., an out-of-bounds (OOB) error or a memory access using a dangling pointer. Thus, our design provides strong probabilistic detection of temporal and spatial memory safety violations. We maintain binary compatibility while achieving low overhead and transparent protection.

4.2 Cryptographically Sealed Pointers

We use a specially designed pointer layout that cryptographically binds the object's bounds and liveness to the pointer using a MAC. Computing a MAC over certain fields of the pointer in combination with tag metadata effectively seals the pointer, making it unforgeable (within cryptographic bounds) for an attacker. Figure 1 shows the pointer layout and MAC generation of our design.

Radix Encoding. In order to utilize our MAC-based approach, we need to adapt the pointer layout to support pointer arithmetic operations. Similar to the M-Machine [14] and C^3 [48], we encode a radix field into the pointer, dividing the address into two parts: A fixed upper field and a variable offset field used for pointer arithmetics. The field sizes are determined by the base-2 logarithm of the size corresponding to the object's memory. The 5-bit radix allows us to encode object sizes ranging from 16 B to 16 GB. Larger allocation sizes could be supported by increasing the radix by a single bit at the cost of decreasing either the virtual address space or the MAC size. Alternatively, Huffman codes [35] could be used to compress the radix for frequently used radix sizes. This would increase the MAC size, and thus the security, for commonly used allocation sizes (at the cost of higher decoding complexity).

MAC Generation and Authentication. The radix field divides the virtual address into a fixed upper part and an offset part enabling pointer arithmetic operations. The fixed upper part of the virtual address must be unforgeable by an adversary to mitigate spatial memory violations. We compute the MAC over the fixed part of the address, the radix, and the memory tag, which gets encoded into the upper bits of the pointer. By including the radix field, the coarse-grain memory range of the allocation influences the resulting MAC. Fine-granular access checks (within the bounds of the radix) are enforced by the tag metadata included in the MAC computation. Note that accessing a memory location that only differs in the memory tag will cause a MAC mismatch. In addition, the key used for MAC computations is unique per process and pseudorandomly generated by the OS (through `exec`-like system calls).

The pointers are used for cryptographically secured memory accesses during runtime, *i.e.*, all sealed pointers are verified for *every* memory operation. The hardware architecture implicitly verifies the MAC using the fixed upper virtual address of the pointer, the radix, and the object-specific memory tag. Legacy pointers, *i.e.*, pointers that are not protected by our design, are easily identified as their radix and MAC fields are either set to all 0 or 1 for user space and kernel space addresses, respectively. For legacy pointers, the hardware verifies that the corresponding memory tags are *zero*, which is the default value of our tagged memory architecture. Notably, there is no co-location of legacy data and protected data within a single cache line, *i.e.*, the memory tag of the entire cache line is set to zero. This is sufficient to guarantee the security of all sealed pointers since protected memory always has non-zero memory tags. Thus, even if an attacker strips the MAC from a sealed pointer, it cannot be used to access the corresponding data. A stripped pointer will be interpreted as a legacy pointer and the check whether the memory tag is zero will fail.

4.3 Memory Tagging and Tag Accumulation

As described above, the radix allows the MAC computation to bind the approximate object size to the pointer. However, to ensure spatial (and temporal) safety *within* the radix, additional object-granular access checks are necessary. We solve this by associating tag metadata with the memory of each allocation. On memory allocation, the memory tag is pseudorandomly chosen and assigned to the memory location. Our design uses a 4-bit tag per 16 B of memory, *i.e.*, 16 bits per cache line with a granularity of 64 B. This combination of tag size and granularity allows us to enforce fine-grain access policies while keeping the memory overhead low.

Conventional tagged memory architectures suffer from limited tag spaces and relatively high tag collision probabilities (*i.e.*, 1/16 in the case of ARM MTE [74]). In contrast, our approach uses the MAC to bind the memory tag to the radix memory region, where the correct tag only grants access *within* the power-of-two memory region. As a result, tag collisions (for large allocations) can only occur within the radix range rather than in the entire address space. Note that for small allocations, the radix field still protects

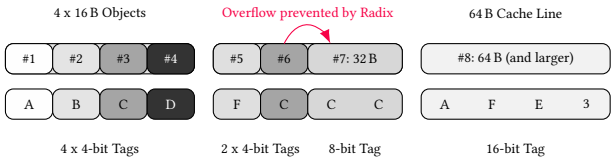

Figure 2: Tag accumulation. Depending on the object's size, we use 4-bit, 8-bit, or 16-bit tags. The co-located memory tags are used for intra-radix isolation and liveness information.

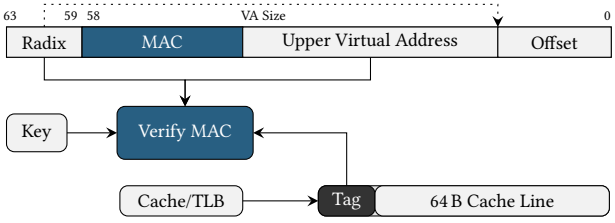

Figure 3: Cryptographically secured memory access. Every memory operation verifies the pointer's MAC using the upper virtual address and radix of the pointer in combination with the memory tag. The upper virtual address and tag accumulation are determined using the radix encoded into the pointer. The object-specific memory tag co-located within every cache line allows for fine-grain access checks.

co-located objects within a single cache line, even when multiple allocations within a cache line may share the same tag (cf. Figure 2).

Tag Accumulation. For a memory safety scheme, it is important to support object-granular memory sizes. To accommodate a wide range of allocation sizes, our smallest supported tag granularity is 16 B. Existing research [75] shows that the alignment of allocations to 16 B has a negligible impact on system performance and memory overhead. Contrary, for the majority of software, a granularity of 32 B or 64 B would drastically increase the runtime and memory alignment overheads. In addition, to keep the chance of tag collisions reasonably low, larger tag sizes (e.g., 16-bit) are desirable. However, 16-bit tag sizes quadruple the memory overhead compared to the 4-bit tags used by ARM MTE, when implemented naïvely. To solve this issue, our design introduces a novel concept called *tag accumulation*. We increase the tag space by utilizing the 16-bit tag metadata of an entire cache line for objects that have the size (and alignment) of a cache line or larger. Our pointer layout, which includes the object's size (*i.e.*, the radix), allows for flexible tag sizes depending on the object's actual size. We *accumulate* up to four 4-bit tags per cache line for a combined 16-bit tag, which drastically reduces the probability of tag collisions for larger objects. However, we can still authenticate smaller (intra-cache line) allocations. Depending on the object's size, *i.e.*, 16 B, 32 B, or 64 B (and larger), our scheme utilizes 4-bit, 8-bit, or 16-bit tags, respectively (cf. Figure 2). At runtime, during the pointer authentication, the radix part of the pointer decides which parts of the full 16 tag bits are used as an input for the MAC. To unambiguously select the correct bits, we align small (16 B and 32 B) allocations according to their respective radix, and larger allocations to 64 B.

Figure 2 shows an overview of our tag accumulation. There, despite the matching tags, allocation #6 cannot overflow into the adjacent buffer (#7), due to the radix within the pointer. Specifically, allocations up to 64 B in size are always aligned and padded to their radix. Since this means that there only exists a single allocation within its radix, spatial memory safety issues will always be detected, and the tag metadata (for small allocations) is only used for temporal protection. Thus, a 4- or 8-bit tag is sufficient to enforce temporal safety for small allocations. Alternatively, we could systematically protect small allocations (*i.e.*, 16 B and 32 B) by assigning each tag value only once (and not reusing it afterward). Thereby we prevent UAF errors due to tag mismatches, despite a slight decrease in usable virtual address space for small allocations over time. For larger allocations, on the other hand, it is possible that multiple memory objects exist within a single radix. In this case, the tag metadata is not only used for temporal protection but also to distinguish between different memory objects within the radix. Due to tag accumulation, these larger allocations benefit from the increased tag size of 16-bit, which provides a vast improvement in temporal safety compared to regular 4-bit security.

Apple's recent M1 CPU reports a cache line size of 128 B (using `sysctl -a`). There, using the same memory overheads, we could also obtain 32-bit metadata for objects that are cache line-sized or larger. This would vastly improve the number of available unique tags (2^{32} instead of 2^{16}). However, throughout this paper, we use 64 B since this is the most commonly used cache line size.

4.4 Cryptographic Memory Safety

The hardware architecture enforces implicit memory safety checks by authenticating the cryptographically sealed pointers during every memory operation. Figure 3 illustrates the cryptographically secured memory access of our design. The authentication procedure re-computes the MAC from the fields of the dereferenced pointer in combination with the memory tag. The re-computed MAC and the encoded MAC are checked for equivalence. A matching MAC implies a successful authentication, while a mismatch denotes a memory safety violation and triggers a hardware exception.

Spatial Violations. In the context of spatial memory safety, we distinguish between spatial violations outside and within the object's radix bounds. A spatial violation outside the radix bounds manipulates the fixed upper part of the virtual address, leading to an authentication error within the cryptographic security bounds. Moreover, a spatial violation within the radix bounds is detected through the mismatching memory tag, e.g., the tag mismatches by overflowing into an adjacent memory object. The object-specific memory tag is used as an input for the MAC computation, thus, a tag mismatch is detected within the cryptographic bounds. Given that the MAC is chosen sufficiently large, it is highly unlikely that a spatial violation passes the authentication.

Temporal Violations. We counteract temporal safety violations by generating and assigning a pseudorandom memory tag on object allocation. Generating a fresh memory tag binds the liveness of the corresponding object (using the MAC computation) to the sealed pointer. In this way, the tag metadata mitigates UAF errors on a probabilistic basis. Additionally, uninitialized memory errors are

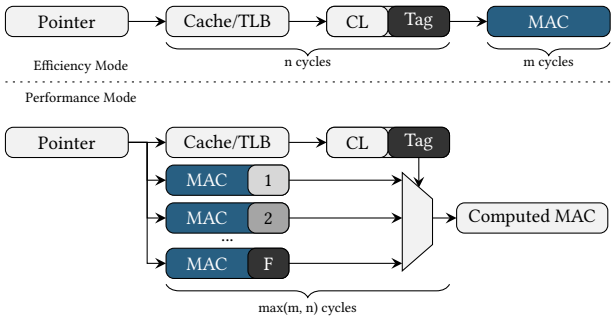

Figure 4: The memory access latency of the introduced modes. The efficiency mode introduces additional memory access latency due to the MAC verification (m cycles). The performance mode allows to pre-compute the MAC from all possible tag values in parallel to the cache lookup. This effectively eliminates any performance overhead if the MAC is faster than the cache lookup.

prevented by our design since setting tags always zeroes the data of the corresponding memory location.

4.5 Hardware Architecture

Our design integrates MAC computations and memory tagging into the architecture, which causes additional latency at three distinct places in the system. First, the memory tag has to be fetched from the cache before the MAC can be calculated. This has the same latency as a regular (L1) cache lookup, which is typically only a few cycles. Second, once the tag value is known, the MAC is then re-computed and compared to the MAC from the pointer. Third, the memory has to be tagged with its respective tag value when the object is allocated. This is done using a new instruction, which sets the tag and zeroes the data of the corresponding memory location. In the following, we detail these architectural aspects of our design and discuss optimizations that minimize the introduced latency.

Note that we do not limit our design to a specific cipher design or cache architecture. Hence, in this section, we describe the added latencies as follows. A normal cache hit, or cache lookup, takes n cycles. Computing and comparing the MAC takes m cycles. In Section 7, we perform a case study with concrete numbers for our cache hierarchy and different cipher latencies, evaluating the impact when using real-world lightweight ciphers.

Tagged Memory Modes. Our tagged memory architecture fetches the cache line and its associated tags from the DRAM and stores them together in the cache. Our *classic* tagged memory architecture issues additional DRAM requests to fetch memory tags on every cache miss. Additionally, memory tags can also be stored in ECC memory [32, 45, 80], allowing for zero-overhead tag fetches. These two approaches represent the worst-case and the best-case overhead in terms of system performance. Thus, in Section 7, we evaluate the performance impact of our design with both tagged memory architectures. In practice, however, different trade-offs are possible by using advanced tag caches and different techniques for storing the tags in memory [40, 66], improving system performance.

Efficiency and Performance Modes. For a concrete hardware implementation, we suggest two different instantiations of our design called the *efficiency* and the *performance* mode. These modes differ in terms of area/power overhead and cache access latency.

In Figure 4, we illustrate both modes. The efficiency mode (top) targets lightweight and efficient memory protection in terms of area overhead, power consumption, and system complexity. It only requires a single cipher instance per core. There, we require n cycles for a cache lookup and an additional m cycles for the MAC calculation, resulting in an overall overhead of $n + m$ cycles.

The performance mode (bottom) aims for the lowest possible impact on system performance. There, we do not wait for the cache lookup to receive the tag required for the MAC computation. Instead, we compute the MAC for all possible tag values in parallel to the cache access. Thus, for a 4-bit tag, we use $2^4 - 1 = 15$ cipher instances since the *zero* tag is reserved. After the n cycles for the cache access, we select the computed MAC according to the fetched tag value. Thus, at the cost of additional area and power, this mode of operation shortens the total cache access (and pointer verification) time to $max(m, n)$ cycles. Assuming a MAC computation that is faster than the (L1) cache access, this mode completely masks the latency of the MAC, resulting in zero performance overhead.

Naturally, there is a limit to the number of cipher instances before the trade-off is no longer worth it. In this paper, we assume that only 4-bit tags are computed in parallel. This also means that, in this mode, our tag accumulation concept does not apply.

Speculation and Prediction. Alternatively, an obvious solution to eliminate the MAC latency (while enabling tag accumulation) would be to only compute frequently-used, previously-used, or otherwise predicted tag values. There, it may only be required to have very few cipher instances to minimize the performance overhead to a negligible level. While the potential for performance and area/power savings is undeniable, one must be careful not to introduce any new attack surfaces. E.g., related work [48] uses similar prediction tables with a hit rate of 99.85 %. However, in this paper, we do not use any speculation or prediction, as this could introduce new side-channels similar to the PACMAN [68] attack.

5 IMPLEMENTATION

In this section, we elaborate on our proof-of-concept implementation. We model the microarchitecture of our design using the gem5 [9, 52] system simulator. Moreover, we use an instrumented memory allocator to enforce our memory safety policies.

5.1 Microarchitecture

We utilize the open-source gem5 [9, 52] system simulator, which is widely used in academia [41, 45, 54, 83, 86], for our implementation and evaluation. Related work (*i.e.*, C^3 [48]) uses Simics [53], a functional full-system simulator. Note that Simics is a closed-source simulator by Intel and only supports functional simulation. In addition, C^3 uses the generated Simics traces as input for a proprietary cycle-accurate simulator to evaluate the performance impact of their design. Thus, Simics is unsuitable for this work.

Our proof-of-concept prototype is based on the gem5 simulator version 22.1.0.0, allowing us to implement a functional and timing-accurate hardware model of our design. The prototype consists

```
Sign(ptr, tag, key) {
  return AddMAC(ptr, tag, key)
}
Settag(ptr, tag) {
  assert(tag < TAG_SIZE)
  assert(ptr == Align(ptr))
  R = ptr[63:59] // get radix
  paddr = TranslateAddr(ptr)
  // zero memory location
  StoreMem(paddr, R) = 0
  // store tag in memory
  StoreTag(paddr, R) = tag
}
```

```
AddMAC(ptr, tag, key) {
  assert(tag != 0)
  assert(tag < TAG_SIZE)
  assert(ptr[58:VA_SIZE] == 0)
  // align to encoded radix
  sh = ptr[63:59] + 4 // 16B offset
  p = (ptr >> sh) << sh;
  // compute and set MAC
  MAC = ComputeMAC(p, tag, key)
  res = ptr // copy ptr
  res[58:VA_SIZE] = Trunc(MAC)
  return res
}
```

```
HwAuth(ptr, tag, key) {
  if CheckLegacyMem(ptr, tag)
    return
  // sealed pointer
  MAC = ptr[58:VA_SIZE] // get MAC
  p = ptr // copy ptr
  p[58:VA_SIZE] = 0 // remove MAC
  // re-compute and get MAC
  ptr_auth = AddMAC(p, tag, key)
  MAC_auth = ptr_auth[58:VA_SIZE]
  // compare re-computed MAC
  assert(MAC == MAC_auth)
}
```

Figure 5: Pseudocode description of our ISA extension, including the new instructions and the hardware authentication.

of a custom instruction set extension (ISE), hardware models of the efficiency and performance modes, and the tagged memory architecture supporting two modes of operation. The gem5 implementation is fully parameterizable with regard to the additional latencies, tag sizes, and tag granularities. Although our design is ISA agnostic, we implement the prototype for the x86-64 architecture.

Instruction Set Extension. Our prototype integrates two new instructions: one to generate cryptographically sealed pointers (on object allocation) and another to interact with the tagged memory architecture. Both instructions are implemented using previously unused three-byte opcodes. Figure 5 provides a pseudocode description of our ISA extension, including the new instructions.

First, the `Sign(ptr, tag, key)` instruction calculates the MAC for a pointer based on a given radix and tag value provided by our memory allocator. The pointer, stored in a general-purpose register (GPR), contains the encoded radix, while the tag value is passed through another GPR. When executed, the instruction computes the MAC in `AddMAC(ptr, tag, key)` for the provided input using the secret per-process key and encodes the resulting MAC back into the pointer stored in the GPR. Precisely, we compute the MAC using `ComputeMAC(p, tag, key)`, which calculates the MAC for the given input (*i.e.*, the radix-aligned pointer, memory tag, and secret key), and truncate the resulting output.

Second, the `Settag(ptr, tag)` instruction sets the memory tag for a given memory address while simultaneously zeroing the corresponding memory location. Similar to the `Sign` instruction, the radix is encoded within the pointer (passed through a GPR) of the corresponding memory location. Depending on the radix, the instruction zeroes 16 B, 32 B, or 64 B of memory and sets the memory tag (*i.e.*, 4-bit, 8-bit, or 16-bit) for the cache line.

Hardware Authentication. The MAC encoded within our cryptographically sealed pointers is implicitly authenticated during every memory access, *i.e.*, every load and store operation. Figure 5 illustrates the pseudocode description of the hardware authentication in `HwAuth(ptr, tag, key)`. The hardware authentication first verifies whether the memory operation accesses a legacy memory location (*i.e.*, zero tag) using a legacy pointer. If the memory access uses a sealed pointer, the hardware re-computes the MAC from the sealed pointer and the corresponding memory tag. Subsequently, the re-computed MAC is compared to the encoded MAC, and in case of a MAC mismatch, a hardware exception is triggered.

The memory access latency varies depending on the selected mode, *i.e.*, efficiency or performance mode. In the efficiency mode, the MAC computation latency is added to the L1 cache latency, impacting system performance. Contrarily, the performance mode utilizes multiple cipher instances in parallel to eliminate the performance overhead, albeit introducing a higher area overhead and increased power consumption.

Cipher Instances. Several low-latency ciphers, such as `BipBip` [7], `QARMA` [6], or `SPEEDY` [47], are potential candidates for our design. As the MAC is computed over a fixed-size input, we can construct our MAC from a single (tweakable) block-cipher instance. This cipher encrypts the fields of a pointer (along with the tag value), and the resulting output is truncated to the desired MAC size M. Given that the used cipher is secure, the probability that the MAC of a modified pointer matches the original MAC is 2^{-M}.

For our proof-of-concept implementation and evaluation, we select `BipBip` as the underlying cipher. Note that the choice of the secure cipher only influences the performance overhead of our design. With `BipBip`, each MAC computation takes 3 cycles at a 4.5 GHz clock frequency [7]. The hardware verifies the MAC encoded in the pointer (enabled by hardware address masking) on every memory access.

As the MAC resides in the topmost bits of the pointer, the MAC size depends on the virtual address mode of the system. Our pointer tagging approach allows us to use 25 bits and 16 bits for metadata encoding using the 39-bit and 48-bit virtual addressing modes, respectively. We require 5 pointer bits for the radix encoding, which leads to MAC sizes of 20 bits and 11 bits for 39-bit and 48-bit virtual addressing, respectively.

Tagged Memory Architecture. Our proof-of-concept implementation supports two modes: the *classic* tagged memory mode and the *ECC-based* tagged memory mode. Tagged memory architectures allow the co-location of data and memory tags by increasing the cache line size to store the associated tag metadata.

For the classic tagged architecture, additional DRAM requests (issued by the memory controller) are required for every cache miss to load the corresponding tag from memory. Therefore, our prototype reserves a dedicated DRAM memory region (proportional to the overall system memory) to enable tagging of the remaining system memory. This region is inaccessible for regular memory operations, and tags are solely set through the `Settag` instruction. Notice that

our classic tagged memory implementation is a worst-case representation, and further optimizations, such as integrating an advanced tag cache [40, 66], can significantly reduce the overhead of fetching the memory tags.

In contrast, the ECC-based memory tagging approach stores the memory tags in the additional chips found on ECC DRAM modules, as suggested by previous research [32, 45, 80]. This mode allows the memory controller to load the associated tags in parallel with the data, eliminating the need for additional DRAM requests. Our prototype implements the tagged architecture with a 16-bit tag size at a cache line granularity of 64 B. This combination is practical since commercial products (e.g., ARM MTE [74]) incur the same memory overhead.

5.2 Memory Allocator

We base our prototype implementation on the GNU C/C++ standard library (glibc 2.37). The memory allocator is instrumented to create cryptographically sealed pointers by leveraging our ISE (*i.e.*, using the `Settag` and `Sign` instructions). For every allocation, our memory allocator pads and aligns the memory to 16 B, which is our smallest granule. This is also the smallest granule used by most existing allocators to achieve maximum compatibility.

Subsequently, the allocator generates a non-zero pseudorandom tag for the allocation utilizing the on-chip random number generator (via `RDRAND`). We then calculate the radix (from the object's size) and set the tag for the corresponding memory of the newly allocated object. Notably, the zero tag is reserved for legacy memory locations. After allocating and tagging, we cryptographically seal the pointer using the tag metadata. The allocator returns the cryptographically sealed pointer and subsequent memory accesses are authenticated in hardware. During object deallocation, the memory allocator first authenticates the sealed pointer before freeing the object by zeroing the content and tag of the corresponding memory location. On memory reallocation, a new non-zero pseudorandom tag is generated and set for the reallocated memory location.

Special considerations are required for small allocations co-located within the same cache line (*i.e.*, 16 B and 32 B). There, we ensure that small allocations are also aligned to their radix. This alignment ensures that there are no two allocations within the same radix (for small allocations). Thus, each intra-cache line object is protected by the radix.

6 SECURITY ANALYSIS

In this section, we analyze the security of our design. First, we present a systematic analysis showcasing the protection against all classes of memory safety violations and other attack vectors. Second, we provide an empirical security evaluation using the NIST Juliet C/C++ [11] test suite.

6.1 Systematic Analysis

In the following, we provide a detailed discussion of the derived security properties of our design. We analyze our design regarding memory safety, *i.e.*, the temporal and spatial security guarantees (cf. Section 2.1). Furthermore, we discuss fundamental security properties such as thread safety and pointer forgery prevention.

Definitions. Our approach is a probabilistic scheme for which the detection of memory safety violations depends on the MAC size and the tag size. Our prototype implementation uses a MAC size of $M = 20$ bits and tag sizes of $T = 4$ to 16 bits. However, we analyze the general security of our design for a MAC size of M bits and for T-bit tags (assuming $M > T$).

In general, a MAC collision occurs at a probability of 2^{-M} and a tag collision at $1/(2^T - 1) \approx 2^{-T}$. This is because we reserve one tag value to support legacy memory locations. Apart from this, our allocator chooses the tag values pseudorandomly using the on-chip random number generator with the hardware instruction `RDRAND`. Thus, the values of the MACs, as well as the tags, are expected to be uniformly distributed. To be more precise, if an attacker already knows k MAC values that are not possible, the probability of correctly guessing the MAC is $1/(2^M - k)$. E.g., when the radix is set to all 0 or 1, then $k = 1$, since the MAC value of all 0 or 1 is not possible for the respective radix. However, in the remainder, we approximate this to 2^{-M}.

Spatial Memory Safety. For our spatial memory safety analysis, we distinguish between (i) adjacent memory violations, (ii) non-adjacent memory violations, (iii) arbitrary memory access violations, and (iv) intra-object memory violations.

(i) Our memory allocator guarantees that two consecutive allocations receive different tags. Since these tags are covered by the MAC, the collision probability is 2^{-M}.

(ii) For non-adjacent memory accesses within the same radix, a collision of the tags could occur. There, another object within the radix could have the same tag value. Thus, given a valid sealed pointer, an attacker can access different objects within the same radix with a probability of 2^{-T}. In practice, this can be further improved, similar to (i), by using unique tags within a radix.

(iii) For arbitrary memory accesses (outside the bounds of the radix), e.g., out-of-bounds (OOB) violations, the MAC's inputs (upper address bits, radix, tags) change, which leads to an unpredictable MAC. Thus, the collision probability is 2^{-M}.

(iv) One special case of spatial violation is the intra-object memory violation. Heap allocators lack the information about the object's internal structure to support intra-object memory safety. However, intra-object memory violations are typically hard to exploit and only account for around 1 % of observed vulnerabilities in recent studies [56]. Similar to the prior work No-FAT [94], C structure members could be promoted to separate allocations to enforce protection with our design. Note that this promotion might cause compatibility issues if the source code makes assumptions about the memory layout of the C structure and requires program recompilation. Additionally, In-Fat [91] addresses intra-object violations by encoding a subobject index into the pointer for bounds retrieval.

Temporal Memory Safety. In our temporal safety analysis, we distinguish between (i) use-after-free (UAF) errors, (ii) uninitialized memory accesses, and (iii) double-free errors.

(i) UAF errors allow an adversary to tamper with data in memory by misusing dangling pointers. For precise security guarantees, we need to further distinguish between two UAF states, the freed state and the reallocated state. In the freed state, the allocator sets the object's tags, as well as the memory content, to zero, thus deterministically preventing any memory accesses using a dangling

pointer. This is because memory with tag values of zero can only be accessed with plain pointers where the MAC and radix fields are all zero or one. On memory reallocation, the allocator assigns a new pseudorandom tag to the memory location of the reallocated object. Thus, the probability of successful memory accesses using a dangling pointer in the reallocated state is 2^{-T}.

(ii) Uninitialized memory accesses are deterministically mitigated since our `Settag` instruction zeroes the associated memory.

(iii) Besides UAF and uninitialized memory accesses, double-free errors can lead to undefined behavior of the program by corrupting the allocator's metadata or freeing a memory object currently used by another part of the program. Again, we distinguish between the freed state and reallocated state. Before freeing a memory object, our allocator performs a memory access on the object. This implicitly verifies the MAC and ensures that the memory has not been freed or reallocated yet. In the freed state, the tags are set to zero, deterministically detecting double-free errors. In the reallocated state, a double-free error can be detected with a probability of $1 - 2^{-T}$, since the new memory object likely has a different tag.

Thread Safety. One important property of a security countermeasure is thread safety. Countermeasures that do not enforce this property can be vulnerable to time-of-check-to-time-of-use (TOCTTOU) [85] attacks. In general, thread safety is an important property that is hard to enforce using software-based approaches [90]. Our approach is thread-safe by design as we perform our cryptographic memory access checks before every memory operation. When the CPU requests memory from the DRAM, the memory controller always fetches its associated tags. Tags are always stored within the cache next to the fetched cache line. When tags are written, they are also updated within the cache line. Like for ARM MTE and SPARC ADI, the cache coherence protocol ensures consistency across all cores and CPUs, thus providing thread safety for data and tags simultaneously.

Pointer Forgery. An attacker might strip a tagged pointer to bypass the verification procedure. This could be done using a pointer arithmetic operation or a gadget that allows the attacker to craft a custom pointer, e.g., using arithmetic and logical operations on a casted pointer. Such attacks are inherently mitigated by the design since legacy memory locations are marked with the tag zero. Thus, accessing protected data by stripping the MAC from the pointer is impossible. Besides removing the MAC, harvesting or leaking valid (non-dangling) sealed pointers could lead to MAC-reuse attacks. An adversary could use the harvested sealed pointer to perform MAC-reuse attacks while the corresponding memory object is live. Related work, like C^3 [48] and ARM PAuth [82], also suffers from this problem, which cannot be easily mitigated. However, we argue that leaking and harvesting pointers are hard to perform for an attacker. An attacker cannot simply scan the memory without first forging individual pointers for every memory location with different tags. Thus, a valid pointer can only be harvested if it is stored within a memory object to which the attacker already has access through a valid sealed pointer or when it is stored in a legacy memory region. Countermeasures like PACTight [36] bind the MAC authentication to the memory address where the pointer is stored to counteract this problem. However, this leads to severe compatibility issues since pointers cannot be copied, e.g., using `memcpy`.

Settag Gadgets. An attacker that has access to a set-tag gadget could misuse it to gain access to a distinct memory location. If the attacker would guess the tag value of their own allocation, they could use the set-tag gadget to set tags to the entire memory area within their own power-of-two radix, and then be able to read any potential secret values within this radix. We mitigate this threat on several layers. First, our `Settag` instruction always zeroes the data stored at the memory location. Thus, secret data of the corresponding memory location cannot be leaked. Second, the attacker has to guess the correct value (with a probability of 2^{-T}), otherwise, the MAC verification would fail, and the program execution would stop. Third, within our modified standard library, this instruction exists only once and can only be called when memory is allocated or freed. Additionally, our ISE only utilizes three-byte opcodes to minimize the risk of introducing any gadgets through misaligned opcodes. However, we suggest that exploitable set-tag gadgets should generally be absent from the program.

Signing Gadgets. Similarly, an attacker that has access to a signing gadget could misuse this gadget to forge arbitrary pointers. We emphasize that arbitrary pointer signing gadgets should be absent in the program outside of our allocation library. However, we mitigate this attack vector since the `Sign` instruction takes the tag as an argument and checks if the tag matches the one in memory. Thus, an attacker would need to guess the correct memory tag in order to forge a pointer using a signing gadget.

System Calls. One known method to brute-force stack canaries is to misuse a `fork` gadget. One process continuously uses the `fork` system call, effectively duplicating the calling process. This child process is then misused by the attacker to brute-force the stack canary. A similar methodology could be applied to brute-force MAC collisions of sealed pointers. Every security violation signals the operating system, which can then decide how to handle the situation, e.g., by killing the child and parent process.

Besides this `fork` gadget, an attacker that has the capability to execute arbitrary system calls could also simply spawn a shell for arbitrary code execution. As such, this must be mitigated by orthogonal countermeasures (e.g., by system call filtering [72]).

Microarchitectural Attacks. Side-channel and microarchitectural attacks are considered out-of-scope for our threat model. Nevertheless, new architectural features should avoid introducing attack surfaces based on side-channels. Recently, PACMAN [68] showcased that timing variations in the authentication procedure can break the security of the ARM PAuth hardware primitive. Thus, our design performs the MAC authentication synchronously on every memory access without timing variations. In our belief, performance optimizations (e.g., by using a prediction table similar to C^3) require a thorough analysis of the microarchitecture to ensure that no leakage occurs. Otherwise, it might lead to new side-channel attack vectors (similar to PACMAN), ultimately compromising the provided security guarantees.

Cryptographic Primitive. In general, the security of our design does not depend on the chosen cipher, given that it is infeasible to revert it and its output is uniformly distributed. In Section 7, we evaluate the impact of using different low-latency ciphers in our design. Our proof-of-concept implementation uses the low-latency tweakable block cipher `BipBip` [7], specifically designed for the

Table 1: The empirical security evaluation using CWE-122 and CWE-416 of the NIST Juliet C/C++ test suite.

CWE	Description	Number of Test Cases	Passed
CWE-122	Heap buffer overrun	63 Variants	100 %
		2985 Test Cases	100 %
CWE-416	Use-after-free	22 Variants	100 %
		459 Test Cases	100 %

C^3 countermeasure, with an excellent latency/area ratio. `BipBip` is a 24-bit tweakable block cipher with a 40-bit tweak input, and the sizes in combination with the low latency make it a suitable candidate for our use case. `BipBip` is secure against an attacker that has the capability of reading and choosing plaintexts, ciphertexts, and tweaks [7]. Note that for our design, parts of the tweak (*i.e.*, tag metadata) cannot be read or chosen by the attacker. Thus, the attacker's capabilities are even more limited than assumed in the security analysis of `BipBip`. Our approach is covered by the security claim of `BipBip` since we use the cipher for its envisioned use case. We use `BipBip` to calculate a MAC that gets truncated and encoded into the pointer. Hence, the chance of predicting a MAC is influenced by the truncation and equals 2^{-M} per guess for an M-bit MAC. As our MAC size is never larger than the output of the cipher, we can use a single cipher instance to compute the MAC.

On detection of a memory safety violation, the hardware immediately triggers an exception and the program aborts. Thus, the adversary can never repeat a forgery attempt. A new per-process key is generated for subsequent program execution, thus further limiting the attacker. Note that these per-process keys used for the MAC calculations are secret, thus, not known by the attacker.

6.2 Empirical Security Analysis

We provide an empirical security analysis to practically analyze the efficacy of our design. We evaluate common types of memory safety vulnerabilities, such as heap buffer overruns (CWE-122) and UAF errors (CWE-416) described in Section 2.1. For our evaluation, we used the NIST Juliet C/C++ test suite version 1.3.

Juliet C/C++ Test Suite. The NIST Juliet C/C++ test suite consists of test case pairs separated into good and bad types of a CWE (subdivided into various variants). Every variant is duplicated several times with obfuscation since the test suite is designed for static analysis tools. However, Juliet also allows us to test the runtime security of our design. Particularly, we evaluate the effectiveness of our design using CWE-122: *heap buffer overrun* and CWE-416: *use-after-free*. Juliet is originally designed for static analysis. Thus, some test cases are not applicable or do not provide an input stimulus to trigger a memory safety violation. Additionally, some test programs already crash without modifications or protection. We exclude the test cases for intra-object memory violations since they are currently not supported by our prototype.

We use the Address Sanitizer (ASan) [73] to select the relevant test cases that actually trigger heap memory safety violations. We consider a distinct test case as passed if our architecture detects the memory violation and an authentication exception is raised. For the evaluation of CWE-122, some test cases need to be adapted due to the 16 B memory alignment of our design to actually trigger a memory safety violation. Following our methodology, we detect all relevant test cases of CWE-122 (heap buffer overflow) and CWE-416 (use-after-free), as shown in Table 1. This highlights that our approach effectively protects against spatial (CWE-122) and temporal (CWE-416) memory safety violations.

Table 2: The gem5 configuration of our prototype.

Parameter	Configuration
Core	3 GHz, TimingSimpleCPU model
L1-I Cache	16 kB, 8-way, 5-cycle latency, 64 B line (private)
L1-D Cache	64 kB, 8-way, 5-cycle latency, 64 B line (private)
L2 Cache	8 MB, 16-way, 17-cycle latency, 64 B line (shared)
MAC	1–4 cycles depending on the cipher
DRAM	8 GB, DDR4-2400

7 EVALUATION

In this section, we evaluate our prototype in terms of system performance and approximate the area and memory utilization.

7.1 Performance Evaluation

We evaluate our introduced modes using the SPEC CPU2017 [12] benchmark suite. All benchmarks are compiled using our instrumented glibc with optimization level `-O3`. We execute all benchmarks on our gem5 prototype running Linux (kernel 5.15.67).

Configuration. Table 2 shows the core parameters of our gem5 prototype. We use the TimingSimpleCPU model for our evaluation and configure the system to a clock frequency of 3 GHz. Moreover, we use a private 8-way set associative L1 instruction (16 kB) and data (64 kB) cache with 64 B cache line size. We use a shared 16-way set associative L2 cache with a size of 8 MB, which is shared within the cores. The system's main memory is an 8 GB dual-channel 2400 MHz DDR4 DRAM module. The cache access latencies for the L1 and L2 cache are configured to 5- and 17-cycles, respectively. These parameters are derived from recent Intel processors, which are in line with prior work [41, 48].

Timing Model. The system performance is simulated according to our timing model. For the ISE, our `Settag` instruction accesses the cache line and zeroes the memory data in addition to setting the tag for the corresponding memory location, which is simulated accurately in the cache hierarchy. Depending on the mode, the MAC authentication is performed serial or parallel. For the performance mode, no additional L1 cache access overhead is required since the MAC computation is performed in parallel by multiple cipher instances. Contrarily, the efficiency mode increases the L1 cache access latency by the duration of the MAC computation.

Both modes are evaluated using different implementations of the tagged memory architecture. The classic tagged memory architecture issues additional DRAM requests to fetch and store tags on each memory access that is not directly served by the cache hierarchy. Our optimized tagged memory implementation utilizes ECC memory to co-locate data and memory tags [32, 45, 80]. Thus, no additional tag fetches are required.

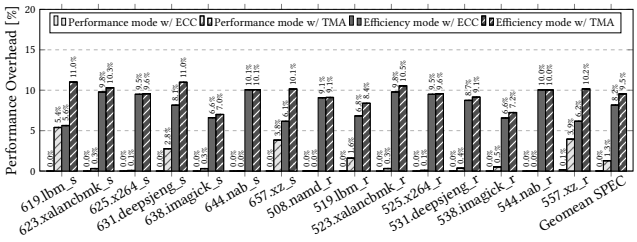

Figure 6: The simulated relative runtime overhead of the performance and efficiency modes using the SPEC CPU2017 benchmark suite. (Efficiency mode +3-cycle L1 delay)

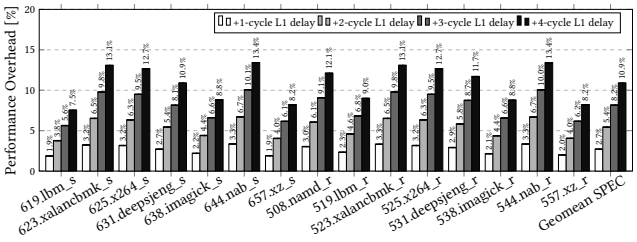

Figure 7: The simulated relative runtime overhead of the efficiency mode with ECC tagged memory using different cipher latencies for the SPEC CPU2017 benchmark suite.

We use the TimingSimpleCPU model for our evaluation, similar to existing work [45, 54, 83, 86] that utilizes the gem5 simulator. Additionally, gem5 provides an O3CPU model supporting out-of-order execution. However, the O3CPU model drastically increases the simulation time (at least by a factor of x10) for larger workloads. Consequently, evaluating the O3CPU model for larger SPEC CPU2017 benchmarks is infeasible due to the simulation times. We report our experimental results for a subset of SPEC CPU2017 benchmarks in Appendix A. Note that out-of-order execution primarily masks memory access latencies. Thus, our simulation results based on the TimingSimpleCPU represent a worst-case approximation, as reflected in the experimental results, since the main source of overhead arises from the increased L1 cache access latency.

Simulation Results. We use an unmodified gem5 model executing the SPEC CPU2017 benchmarks compiled with an uninstrumented glibc as our baseline. Our evaluation setup measures the execution time of each workload. Thus, we use the execution time as our performance metric when comparing the different system configurations. We evaluate all possible combinations of our modes (efficiency and performance modes), cipher latencies, and tagged memory architectures (ECC-based and classic tagged memory). Note that some SPEC CPU2017 benchmarks are excluded from our evaluation due to toolchain issues that either resulted in compilation failures or runtime errors, even for the baseline binaries.

Figure 6 illustrates the simulated runtime overhead of our design for the performance and efficiency modes using the ECC-based tagging and the classic tagged memory implementation. Besides the runtime overhead introduced by memory tagging, *i.e.*, setting the tags in memory and the tag propagation in hardware, the performance mode effectively conceals the overhead of the MAC calculation in the microarchitecture. We find that the geomean of the relative runtime overhead equals 0.013 % when using ECC-based memory tagging in the performance mode. For the unoptimized tagged memory implementation, we measure an average overhead of 1.286 %. Moreover, we report the runtime overhead of our design in the efficiency mode supporting tag accumulation. In this mode, the MAC computation delays each access to the L1 cache. The overall memory access latency is increased by the combination of MAC latency and the latency of the tagged memory architecture. When using `BipBip`, the MAC computation introduces a 3-cycle latency for every L1 memory access. Here, the geomean of the relative runtime overhead ranges from 8.157 % to 9.542 %, depending on the implementation of memory tagging.

Table 3: The latency and area overhead of suitable ciphers.

Cipher	Latency	Cycles‡	Area	Manufacturing Process
`BipBip` [7]	0.622 ns	2 / 3 / 4	5.74 kGE	Intel 10 nm FinFet [5]
`SPEEDY-5` [47]	0.300 ns	1 / 2 / 2	34.64 kGE	NanGate OCL 15 nm
`SPEEDY-7` [47]	0.431 ns	2 / 3 / 3	48.72 kGE	
`QARMA-5` [47]	0.385 ns	2 / 2 / 2	14.88 kGE	NanGate OCL 15 nm
`QARMA-8` [47]	0.608 ns	2 / 3 / 4	22.76 kGE	

‡ Cycle numbers are for 3 GHz / 4.5 GHz / 5 GHz, respectively.

To showcase the impact of the MAC latency, we evaluate a range of ciphers and their corresponding MAC delays. With the selected ciphers, we simulate the runtime overheads for additional L1 latencies ranging from 1 to 4 cycles. Figure 7 shows the relative runtime overhead for different L1 delays, highlighting the performance impact of different ciphers. In the slowest configuration with a 4-cycle delay, our design introduces a reasonable average overhead of 10.874 %. With the fastest configuration of a 1-cycle delay, we measure an average performance overhead of 2.707 %.

Our evaluation shows that the performance overhead scales approximately linearly with the delay introduced by the MAC computation. We find that the performance mode, in combination with ECC-based memory tagging, introduces negligible performance overheads that are below 1 %.

7.2 Area and Memory Utilization

Memory Overhead. Our tagged memory architecture requires the alignment of allocations to 16 B. However, most software and heap allocators (including the one we chose for our implementation) already adhere to this alignment. Thus, the additional memory overhead for alignment or padding is negligible.

Furthermore, we need to store 16-bit tag metadata per 64 B cache line. Similar to ARM MTE, this has a memory overhead of 3.125 %. However, existing work already showed that memory tags could be stored within the ECC memory [32, 45, 80], effectively reducing this overhead to zero.

Area and Power Estimation. In our design, we introduce either one or multiple instances of a MAC per core. In Table 3, we give an overview of the most relevant low-latency ciphers that can be

used for this purpose. In the case of BipBip, this would roughly add 5.74 kGE for our efficiency mode and $15 \cdot 5.74\,\text{kGE} = 86.1\,\text{kGE}$ for the performance mode for each core. Depending on the cache architecture, more instances might be required to support multiple ports. Intel's 10nm process achieves around 100.8 M transistors/mm². Thus, one BipBip instance with 5.74 kGE, or roughly 22.9 k transistors, would require around 227 µm². Recent Intel Alder Lake 16-core CPUs have a similar manufacturing process and their dies are 215.25 mm² in size. With one instance per core, BipBip requires 3632 µm², or around 0.000016873x the size of the CPU. Hence, we claim that our added area overheads, even in our performance mode which multiplies the overhead by 15, are insignificant. BipBip claims a power requirement of just 15.91 mW [7]. If we pessimistically assume that cache hits, and thus MAC calculations, happen in every single cycle, this is equal to around 0.01 % of the TDP of the same CPU.

Table 4: Comparison of hardware-based memory safety countermeasures. Legend: ● Deterministic mitigation, ◐ Pointer integrity, ◓ Probabilistic mitigation, ○ None.

Mechanism	OOB	UAF	Uninit.	Hardware Extensions
ARM PAuth [82]	◐	◐	○	Ptr. Auth. Instr. (e.g., QARMA)
CCFI [55]	◐	◐	○	Intel x86 AES-NI Instr.
CHERI [88]	●	○	○*	Cap. Regs + Tag$ 1-bit/16–32 B
Cornucopia [27]	○	●	○*	Cap. Regs + Tag$ 1-bit/16–32 B
C^3 [48]	◓‡	◓‡	◓‡	Ptr. Enc. + Keystream Gen.
CrypTag [62]	◓	◓	●	Tag$ + Mem. Enc. Engine
ARM MTE [74]	◓	◓	○*	Tag$ 4-bit/16 B
SPARC ADI [2]	◓	◓	○*	Tag$ 4-bit/64 B
Our work	◓	◓	●	Ptr. Auth. + Tag$ 16-bit/64 B

‡ Silent data corruption possible (garbled data), * Memory zeroing can be enforced

8 DISCUSSION

In this section, we compare our design to related work on memory safety and discuss possible future work.

8.1 Related Work

In the following, we compare our design to state-of-the-art memory safety countermeasures and directly related prior work. Table 4 compares hardware-based memory safety countermeasures, including their protection capabilities and required hardware extensions.

Memory Protection. Various protection mechanisms utilize additional bounds [22, 23, 24, 41, 59, 61, 63, 64, 69, 79, 91, 94] metadata inlined, co-located, or disjoined with pointers to enforce spatial memory safety. Moreover, different strategies for temporal memory safety based on additional liveness metadata [26, 60] or quarantine lists [3, 27, 31, 89] are proposed. These additional access checks are either entirely implemented in software [13, 60, 61, 39, 63, 79] or utilize hardware extensions [22, 41, 59, 64, 69, 78, 91, 94].

Software-based countermeasures, like CCured [63] and SoftBound [61], provide spatial memory safety by instrumenting additional bounds checks for every memory access. While software-based approaches provide a flexible solution based on compile-time transformations, these countermeasures typically introduce a high impact on system performance. For instance, SoftBound+CETS [60, 61] incur a runtime overhead of 116 % for temporal and spatial protection, thus, often not applicable for runtime deployment.

Hardware-enforced mechanisms [22, 41, 59, 69, 94] typically achieve better performance results. However, they often require intrusive hardware and ABI changes, which increase the overall system complexity and are difficult to deploy on a large scale.

For example, CHERI [84, 88] requires so-called *fat pointers* to co-locate the object's bounds information with the pointer. This breaks ABI compatibility: software libraries must be recompiled, and the operating system needs to support it. Additionally, Cornucopia [27] and CHERIvoke [89] provide temporal memory safety for the CHERI architecture. In contrast, we require only minimal hardware changes and maintain binary compatibility.

ISA extensions, like AOS [41] and In-Fat [91], adapt pointer tagging for bounds retrieval. Particularly, In-Fat encodes an index to the base and bounds metadata achieving subobject granular spatial protection. However, compared to our design, In-Fat cannot provide temporal safety. Moreover, our design aims for lightweight and hardware-enforced protection based on efficient cryptography.

Cryptographic Memory Protection. Several countermeasures aim to enforce memory safety using cryptographic primitives [21, 48, 55, 62, 70, 82] (cf. Section 2.2). Specifically, PointGuard [21], ARM PAuth [82], and CCFI [55] enforce pointer integrity for code pointers, data pointers, or both. In comparison to mechanisms such as ARM PAuth and CCFI, we enforce temporal and spatial memory safety instead of only focusing on pointer protection.

Furthermore, CrypTag [62] enforces memory safety by utilizing pointer tagging in combination with memory encryption. Every memory object gets a key assigned during allocation, which is used as additional metadata to tweak the memory encryption procedure. Depending on the used encryption mode (*i.e.*, encryption-only or authenticated encryption), memory accesses using the wrong key lead to garbled data or lead to an authentication error immediately triggering a hardware exception. Compared to CrypTag, we do not require a memory encryption engine, which also introduces memory overhead to store the MAC values for authentication. Furthermore, the key metadata must be stored in the cache, *i.e.*, large key sizes result in large cache size overheads.

Cryptographic capability computing (C^3) [48] uses a combination of pointer and data encryption. The pointer encryption generates a cryptographic address (CA), preventing arbitrary pointer forgery. Moreover, the CA is used to derive a keystream that gets XORed with the accessed data in memory. Accessing memory without the correct CA leads to garbled data. Due to its stateless design, C^3 is susceptible to specific attacks [33] misusing the XOR-based keystream generator. Specifically, an adversary can mount attacks that exploit the silent data corruption. For example, the garbled data can be misused to leak secrets of memory locations initialized with known values [33]. Besides XOR-based attacks, UAF attacks are introduced by the low entropy of the 4-bit version field. In contrast, our approach offers stronger security properties, such as tangible detection capabilities (and appropriate error handling) instead of program execution with garbled data.

Memory Tagging. Tagged memory is a versatile building block used to enforce different security policies [2, 38, 71, 74, 87, 93] (cf. Section 2.3). Currently, the Clang/LLVM [46] compiler supports two

memory sanitizers that rely on memory tagging: HWASan [75] and MemTagSanitizer [51]. HWASan applies software-based memory tagging to detect memory errors utilizing 8-bit tags. The memory tags are encoded into the pointer and are stored in shadow memory. However, HWASan is primarily used for debugging purposes due to the relatively high runtime overhead.

MemTagSanitizer adopts a comparable implementation strategy based on ARM MTE [74] (4-bit tag size at the granularity of 16 B) and is used for debugging as well as safeguarding production code during runtime. MemTagSanitizer generates a pseudorandom tag for the first object within a stack frame, and subsequent objects receive incremented tag values.

In contrast to schemes based on ARM MTE, our approach offers significantly stronger protection. First, we enhance the probabilistic detection substantially, achieving a spatial security of up to 20-bit using the MAC (*i.e.*, 99.9999 % detection probability). Moreover, we enable stronger temporal security by using tag accumulation for 16-bit memory tags (*i.e.*, 99.998 %) while maintaining the same memory overhead as ARM MTE, which only provides 4-bit security (*i.e.*, 93.75 %). Second, our design enforces pointer integrity by cryptographically binding the protection to the pointer. The pointer (and encoded tags) of ARM MTE are forgeable when tagging policies rely on deterministically chosen memory tags.

For instance, MemTagSanitizer increments the memory tag for adjacent memory locations, which can easily be exploited, e.g., by forging the incremented tag value using pointer arithmetics. Additionally, Google's StarScan [31] provides UAF protection using ARM MTE by incrementing tags before memory reallocation. In comparison, our design utilizes the uniformly distributed output of the MAC computation, which effectively mitigates this issue, as an adversary cannot predict MACs or leak memory tags.

Similar to ARM MTE, SPARC ADI [2] implements memory tagging in hardware with a 4-bit tag size at a granularity of 64 B. Besides the outlined issues of ARM MTE (*i.e.*, low detection probability), SPARC ADI also requires padding to 64 B, leading to caching effects that introduce a non-negligible performance overhead [75].

Furthermore, we evaluate the performance impact of tagged memory using a classic tagged memory architecture that duplicates every DRAM request to fetch the corresponding tag value (a worst-case approximation). However, our design could benefit from an enhanced tagged memory architecture incorporating an advanced tag cache, as proposed by prior work [40, 66], which would improve system performance.

In addition, we evaluate our design utilizing memory tags stored in ECC memory [32, 45, 80], which has been widely recognized as a viable option, as seen in SPARC ADI [75], Intel TDX [17], and CHERI [88]. Recent research, such as HashTag [45], showcases how memory tags can be encoded into ECC using a hash-based integrity scheme. Also, implicit memory tagging [80] provides memory tagging using alias-free tagged ECC. Both approaches propose efficient implementations of tagged memory architectures that are fully compatible with our design.

8.2 Future Work

In this work, we introduced two modes to present possible trade-offs between the runtime overhead of the system and area efficiency. However, the approach of our performance mode is only applicable for small tag sizes due to the area overhead of several cipher instances. Future research could investigate solutions to reduce the cache access latency introduced by the authentication check of the memory accesses. Particularly, we could use a prediction table (e.g., similar to C^3 [48] that reports a prediction rate of 99.85 %) to cache recent MAC calculations. We anticipate that our mechanism with a prediction table would yield comparable runtime overheads for the efficiency mode to those observed by the performance mode demonstrated in our evaluation. Nevertheless, such a prediction table would require extensive analysis of the microarchitecture to guarantee the absence of possible side-channel attacks like PACMAN [68], allowing an attacker to bypass the security mechanism.

Moreover, this paper focuses on the microarchitecture of our design and uses our hardware feature to implement lightweight and efficient heap memory safety. Similarly, stack and global memory objects could be protected using our mechanism. Future compiler support could be investigated to protect stack and global data. However, then our heap allocator cannot be linked to existing binaries anymore, *i.e.*, losing binary compatibility. Note that implementing such a compiler extension requires engineering effort beyond the scope of this paper.

9 CONCLUSION

In this paper, we presented cryptographically sealed pointers, a novel architectural mechanism that cryptographically enforces temporal and spatial memory safety. Our lightweight ISA extension provides memory safety based on message authentication codes (MACs) and object-granular metadata efficiently scaled and stored in tagged memory enabled by our new concept of tag accumulation. We use the MAC to cryptographically bind the object's bounds and liveness to the pointer. Precisely, we compute the MAC of the address and encoded length in combination with the memory tag to associate the pointer with the corresponding memory. The hardware architecture implicitly enforces memory access checks by authenticating the sealed pointers on every dereference.

Our comprehensive security analysis, including an empirical evaluation using the NIST Juliet C/C++ test suite, highlights the strong protection capabilities of our design. We implemented a proof-of-concept prototype consisting of a hardware model based on gem5 and a custom memory allocator and evaluated the impact on the system performance. The performance evaluation, using the SPEC CPU2017 benchmark suite, showcases the practicability of our design, introducing a negligible performance overhead of 1.3 % and 9.5 % for the *performance* and *efficiency* modes, respectively.

ACKNOWLEDGMENTS

We thank Robert Primas and the anonymous reviewers for their valuable feedback that improved this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087) and the AWARE project (FFG grant number 891092). Additional funding was provided by generous gifts from Intel and from SGS.

REFERENCES

[1] National Security Agency. 2022. Software memory safety - cybersecurity information sheet. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF. Accessed: 2023-02-26. (2022).

[2] Kathirgamar Aingaran, Sumti Jairath, Georgios K. Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, and Thomas Wicki. 2015. M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro*, 35, 36–45.

[3] Sam Ainsworth and Timothy M. Jones. 2020. MarkUs: Drop-in use-after-free prevention for low-level languages. In *S&P'20*, 578–591.

[4] Iván Arce. 2004. The Shellcode Generation. *IEEE Security & Privacy*, 2, 72–76.

[5] C. Auth, A. Aliyarukunju, M. Asoro, D. Bergstrom, V. Bhagwat, J. Birdsall, N. Bisnik, M. Buehler, V. Chikarmane, G. Ding, Q. Fu, H. Gomez, W. Han, D. Hanken, M. Haran, M. Hattendorf, R. Heussner, H. Hiramatsu, B. Ho, S. Jaloviar, I. Jin, S. Joshi, S. Kirby, S. Kosaraju, H. Kothari, G. Leatherman, K. Lee, J. Leib, A. Madhavan, K. Marla, H. Meyer, T. Mule, C. Parker, S. Parthasarathy, C. Pelto, L. Pipes, I. Post, M. Prince, A. Rahman, S. Rajamani, A. Saha, J. Dacuna Santos, M. Sharma, V. Sharma, J. Shin, P. Sinha, P. Smith, M. Sprinkle, A. St. Amour, C. Staus, R. Suri, D. Towner, A. Tripathi, A. Tura, C. Ward, and A. Yeoh. 2017. A 10nm high performance and low-power cmos technology featuring 3rd generation finfet transistors, self-aligned quad patterning, contact over active gate and cobalt local interconnects. In *2017 IEEE International Electron Devices Meeting (IEDM)*.

[6] Roberto Avanzi. 2017. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Transactions on Symmetric Cryptology*, 2017, 4–44.

[7] Yanis Belkheyar, Joan Daemen, Christoph Dobraunig, Santosh Ghosh, and Shahram Rasoolzadeh. 2023. BipBip: A Low-Latency Tweakable Block Cipher with Small Dimensions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023, 326–368.

[8] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. 2018. Smashing the Stack Protector for Fun and Profit. In *SEC'18*, 293–306.

[9] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Computer Architecture News*, 39, 1–7.

[10] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS'11*, 30–40.

[11] Tim Boland and Paul E. Black. 2012. Juliet 1.1 C/C++ and Java Test Suite. *Computer*, 45, 88–90.

[12] James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *ICPE'18*, 41–42.

[13] Nathan Burow, Derrick Paul McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *AsiaCCS'18*, 381–392.

[14] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-based Addressing. In *ASPLOS'94*, 319–327.

[15] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *CCS'10*, 559–572.

[16] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. 2021. Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches. *ACM Transactions on Privacy and Security*, 24, 26:1–26:36.

[17] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2023. Intel TDX Demystified: A Top-Down Approach. *CoRR*, abs/2303.15540.

[18] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *CCS'15*, 952–963.

[19] MITRE Corporation. 1999-2023. Common vulnerabilities and exposures. https://cve.mitre.org/. Accessed: 2023-02-26. (1999-2023).

[20] MITRE Corporation. 2006-2023. Common weakness enumeration. https://cwe.mitre.org/. Accessed: 2023-02-26. (2006-2023).

[21] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *USENIX Security Symposium'03*.

[22] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *ASPLOS'08*, 103–114.

[23] Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *CC'16*, 132–142.

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

[25] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter of Heartbleed. In *IMC'14*, 475–488.

[26] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *USENIX Security Symposium'21*, 1037–1054.

[27] 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. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *S&P'20*, 608–625.

[28] Aurélien Francillon and Claude Castelluccia. 2008. Code injection attacks on harvard-architecture devices. In *CCS'08*, 15–26.

[29] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. *CoRR*, abs/2303.16353.

[30] Google. 2021. An update on memory safety in chrome. https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html. Accessed: 2023-02-26. (2021).

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

[32] Richard H. Gumpertz. 1983. Combining Tags With Error Codes. In *ISCA'83*, 160–165.

[33] Mohamed Tarek Bnziad Mohamed Hassan. 2022. *Hardware-Software Co-design for Practical Memory Safety*. Columbia University.

[34] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *S&P'16*, 969–986.

[35] David A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40, 9, 1098–1101.

[36] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2022. Tightly Seal Your Sensitive Pointers with PACTight. In *USENIX Security Symposium'22*, 3717–3734.

[37] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *CCS'18*, 1868–1882.

[38] Samuel Jero, Nathan Burow, Bryan C. Ward, Richard Skowyra, Roger Khazan, Howard E. Shrobe, and Hamed Okhravi. 2023. TAG: Tagged Architecture Guide. *ACM Computing Surveys*, 55, 124:1–124:34.

[39] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX ATC'02*, 275–288.

[40] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey D. Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *ICCD'17*, 641–648.

[41] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2020. Hardware-based Always-On Heap Memory Safety. In *MICRO'20*, 1153–1166.

[42] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA'14*, 361–372.

[43] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P'19*, 1–19.

[44] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *OSDI'14*, 147–163.

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

[46] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*, 75–88.

[47] Gregor Leander, Thorben Moos, Amir Moradi, and Shahram Rasoolzadeh. 2021. The SPEEDY Family of Block Ciphers Engineering an Ultra Low-Latency Cipher from Gate Level for Secure Processor Architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021, 510–545.

[48] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. 2021. Cryptographic Capability Computing. In *MICRO'21*, 253–267.

[49] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX Security Symposium'19*, 177–194.

[50] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium'18*, 973–990.

[51] LLVM. 2020. Memtagsanitizer. https://releases.llvm.org/11.0.0/docs/MemTagSanitizer.html. Accessed: 2023-07-26. (2020).

[52] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. *CoRR*, abs/2007.03152.

[53] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A Full System Simulation Platform. *Computer*, 35, 50–58.

[54] Evgeny Manzhosov, Adam Hastings, Meghna Pancholi, Ryan Piersma, Mohamed Tarek Ibn Ziad, and Simha Sethumadhavan. 2022. Revisiting Residue Codes for Modern Memories. In *MICRO'22*, 73–90.

[55] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *CCS'15*, 941–951.

[56] Microsoft. 2020. Security Analysis of CHERI ISA. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf. Accessed: 2023-02-26. (2020).

[57] Microsoft. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf. Accessed: 2023-02-26. (2019).

[58] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P'20*, 1466–1482.

[59] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ISCA'12*, 189–200.

[60] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *ISMM'10*, 31–40.

[61] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *PLDI'09*, 245–258.

[62] Pascal Nasahl, Robert Schilling, Mario Werner, Jan Hoogerbrugge, Marcel Medwed, and Stefan Mangard. 2021. CrypTag: Thwarting Physical and Logical Memory Vulnerabilities using Cryptographically Colored Memory. In *AsiaCCS'21*, 200–212.

[63] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: type-safe retrofitting of legacy code. In *POPL'02*, 128–139.

[64] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *SIGMETRICS'18*, 111–112.

[65] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA'06*, 1–20.

[66] Aditi Partap and Dan Boneh. 2022. Memory Tagging: A Memory Efficient Design. *CoRR*, abs/2209.00307.

[67] Marco Prandini and Marco Ramilli. 2012. Return-Oriented Programming. *IEEE Security & Privacy*, 10, 84–87.

[68] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: attacking ARM pointer authentication with speculative execution. In *ISCA'22*, 685–698.

[69] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. 2022. HeapCheck: Low-cost Hardware Support for Memory Safety. *ACM Transactions on Architecture and Code Optimization*, 19, 10:1–10:24.

[70] David Schrammel, Salmin Sultana, Karanvir Grewal, Michael LeMay, David M. Durham, Martin Unterguggenberger, Pascal Nasahl, and Stefan Mangard. 2023. MEMES: Memory Encryption-Based Memory Safety on Commodity Hardware. In *SECRYPT'23*, 25–36.

[71] David Schrammel, Moritz Waser, Lukas Lamster, Martin Unterguggenberger, and Stefan Mangard. 2023. SPEAR-V: Secure and Practical Enclave Architecture for RISC-V. In *AsiaCCS'23*, 457–468.

[72] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium'22*, 936–952.

[73] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC'12*, 309–318.

[74] Kostya Serebryany. 2019. ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. *login Usenix Mag.*, 44.

[75] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *CoRR*, abs/1802.09517.

[76] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS'07*, 552–561.

[77] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *HASP'19*, 8:1–8:11.

[78] Rasool Sharifi and Ashish Venkat. 2020. CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities. In *ISCA'20*, 762–775.

[79] Matthew S. Simpson and Rajeev Barua. 2010. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. In *SCAM'10*, 199–208.

[80] Michael B. Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W. Keckler. 2023. Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC. In *ISCA'23*, 67:1–67:13.

[81] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *S&P'13*, 48–62.

[82] Qualcomm Technologies. 2017. Pointer authentication on armv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf. Accessed: 2023-02-26. (2017).

[83] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. 2023. Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging. In *AsiaCCS'23*, 177–189.

[84] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P'15*, 20–37.

[85] Jinpeng Wei and Calton Pu. 2005. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. In *FAST'05*.

[86] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium'19*, 675–692.

[87] Emmett Witchel, Josh Cates, and Krste Asanovic. 2002. Mondrian memory protection. In *ASPLOS'02*, 304–316.

[88] 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. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA'14*, 457–468.

[89] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel Wesley Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *MICRO'19*, 545–557.

[90] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. 2023. Warpattack: bypassing cfi through compiler-introduced double-fetches. In *S&P'23*.

[91] Shengjie Xu, Wei Huang, and David Lie. 2021. In-fat pointer: hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *ASPLOS'21*, 224–240.

[92] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium'14*, 719–732.

[93] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *OSDI'08*, 225–240.

[94] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. 2021. No-FAT: Architectural Support for Low Overhead Memory Safety Checks. In *ISCA'21*, 916–929.

A SPEC CPU2017 EXPERIMENTAL SETUP

In our gem5 performance evaluation, we utilize the in-order TimingSimpleCPU model. While gem5 also provides an O3CPU model supporting out-of-order execution, we observed a drastically increased simulation time for large workloads like SPEC CPU2017 when using the O3CPU model. Specifically, depending on the benchmark, we can report an increase by a factor of x10 to x18 compared to the TimingSimpleCPU model.

Benchmarking the TimingSimpleCPU took several weeks, primarily due to large simulation times with individual benchmarks lasting several days and due to the comprehensive set of different configurations of our design. Consequently, given the extensive simulation time required, evaluating the O3CPU for all SPEC CPU2017 benchmarks is infeasible. Nevertheless, we want to provide experimental results for a subset of SPEC CPU2017 benchmarks.

Figure 8 shows the simulated relative runtime overhead of the performance and efficiency modes using the O3CPU model for a subset of SPEC CPU2017 benchmarks. These smaller workloads provide insights into the performance of our design using the O3CPU model and reveal a performance overhead reduction. For instance, the `538.imagick_r` overhead decreased from 7.2 % (with TimingSimpleCPU) to 5.9 % (with O3CPU) for the efficiency mode using the classic tagged memory architecture. Moreover, the overhead of `544.nab_r` decreased from 10.0 % (with TimingSimpleCPU) to 1.1 % (with O3CPU) using the efficiency mode.

Additionally, Figure 9 illustrates the simulated relative runtime overhead of the efficiency mode with ECC tagged memory using the O3CPU model with different cipher latencies for a subset of SPEC CPU2017 benchmarks. In comparison to the TimingSimpleCPU, the experimental results indicate that the out-of-order execution of the O3CPU masks the increased L1 cache access latency introduced by our design. Thus, our simulation results based on the TimingSimpleCPU provide a worst-case approximation since the main source of overhead is the increased L1 cache latency.

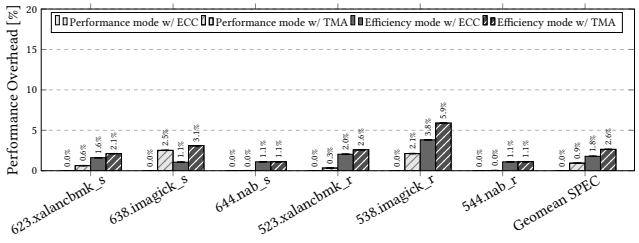

Figure 8: The simulated relative runtime overhead of the performance and efficiency modes using the O3CPU model for a subset of SPEC CPU2017 benchmarks. (Efficiency mode +3-cycle L1 delay)

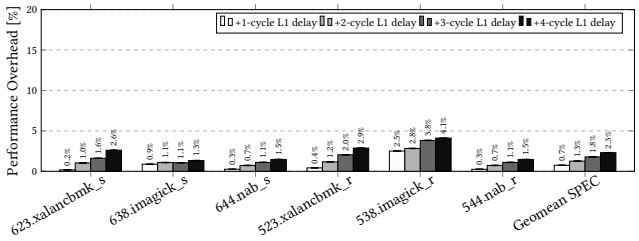

Figure 9: The simulated relative runtime overhead of the efficiency mode with ECC tagged memory using the O3CPU model with different cipher latencies for a subset of SPEC CPU2017 benchmarks.