Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging

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

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

Robert Schilling
robert.schilling@iaik.tugraz.at
Graz University of Technology
Graz, Austria

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

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

ABSTRACT

Memory safety vulnerabilities are a severe threat to modern computer systems allowing adversaries to leak or modify security-critical data. To protect systems from this attack vector, full memory safety is required. As software-based countermeasures tend to induce significant runtime overheads, which is not acceptable for production code, hardware assistance is needed. Tagged memory architectures, e.g., already offered by the ARM MTE and SPARC ADI extensions, assign meta-information to memory objects, thus allowing to implement memory safety policies. However, due to the high tag collision probability caused by the small tag sizes, the protection guarantees of these schemes are limited.

This paper presents Multi-Tag, the first hardware-software co-design utilizing a multi-granular tagging structure that provides strong protection against spatial and temporal memory safety violations. By combining object-granular memory tags with page-granular tags stored in the page table entries, Multi-Tag overcomes the limitation of small tag sizes. Introducing page-granular tags significantly enhances the probabilistic protection capabilities of memory tagging without increasing the memory overhead or the system's complexity. We develop a prototype implementation comprising a gem5 model of the tagged architecture, a Linux kernel extension, and an LLVM-based compiler toolchain. The simulated performance overhead for the SPEC CPU2017 and nbench-byte benchmarks highlights the practicability of our design.

CCS CONCEPTS

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

KEYWORDS

Memory Safety, Tagged Memory Architecture, Multi-Granular Tags

1 INTRODUCTION

According to Microsoft [38] and Google [21], 70 % of all security bug fixes in Windows and Chrome are related to memory safety vulnerabilities. Such vulnerabilities are exploited to leak [18] or modify [24, 49] security-critical data or even to gain remote code execution [4]. Memory safety violations are generally categorized into spatial violations, e.g., accessing an array index out of bounds, and temporal violations like use-after-free (UAF) or double-free errors [55]. Memory corruption allows an adversary to perform powerful exploitation techniques like ROP [49], JOP [7], or DOP [24] attacks. To mitigate these attacks, it is necessary to prevent the exploitation of spatial and temporal memory safety bugs, *i.e.*, full memory safety is required.

Existing research shows that implementing memory safety countermeasures using software-based approaches is expensive in terms of performance. Software-based memory safety schemes (such as SoftBound+CETS [40, 39]) lead to 116 % and more in runtime overhead. Thus, they are impractical to use. To cope with the performance overhead, hardware support for memory safety schemes is needed. ISA extensions [15, 16, 30, 41, 42, 45, 50, 58–60] relying on custom hardware changes provide memory safety with a reasonable performance overhead. However, many of these countermeasures require intrusive hardware and ABI changes (e.g., fat pointers) to enforce their security policies. In contrast, commercial products like the ARM memory tagging extension (MTE) [34, 48] and SPARC application data integrity (ADI) [1] are promising hardware features based on tagged memory. Both protection mechanisms utilize additional metadata (so-called *memory tags*) to provide hardware-enforced detection of memory safety violations. However, tagged memory architectures, including ARM MTE and SPARC ADI, suffer from significant limitations. Memory tagging is a probabilistic approach, and its protection is limited by the available tag size (*i.e.*, 4 bits for MTE and ADI) of the underlying architecture. Due to the small tag sizes, tag collisions likely occur as the same tag must be re-assigned for several memory objects (e.g., MTE and ADI provide only 16 distinct tags). When the tags of the exploited object and the target objects coincide, *i.e.*, a tag collision occurs, an attack cannot be mitigated or even detected.

Contributions. In this paper, we present Multi-Tag, a novel mechanism that combines object-granular memory tagging with page-granular tags for memory safety. Multi-Tag differs from conventional tagged memory schemes that associate each memory object with a single tag. By introducing a multi-granular tagging structure, we significantly increase the detection probability of memory safety violations, thus fortifying the system against memory-based attacks. Instead of storing the page-granular tags separately in memory, we utilize the page table entries (PTE) for this purpose. Using free bits in the PTEs allows us to extend the tag space without

*The work was done while the author was at Lamarr Security Research.

incurring any additional memory overhead or increased system complexity.

Furthermore, we showcase that Multi-Tag's design methodology can be integrated into existing tagged memory architectures. More specifically, we demonstrate how Multi-Tag can be used with ARM MTE, a feature of upcoming ARMv8.5-A systems [34]. Here, our evaluation shows that using Multi-Tag on ARM MTE systems considerably increases the memory safety guarantees. Finally, we discuss how security properties can be improved by leveraging a platform-specific hardware feature, *i.e.*, ARM pointer authentication (PA) [56]. We highlight ARM PA-based defense mechanisms which further enhance the detection probabilities of memory safety issues.

We implement a functional prototype of Multi-Tag's hardware mechanism consisting of a modified gem5 [6] system simulator, a custom Linux kernel, and a toolchain containing an LLVM-based [29] compiler and an instrumented heap allocator. Moreover, we provide an in-depth security analysis of Multi-Tag. Highlighting the practicability of our approach, we extensively evaluate our prototype in terms of performance and memory overhead, yielding a geometric mean performance overhead of 11.7 % for the SPEC CPU2017 [9] benchmarks and 1.7 % for the nbench-byte [36] benchmarks. For both settings, we maintain a constant memory overhead of 6.25 %.

Summarized, we make the following key contributions:

- We present Multi-Tag, the first hardware-software co-design utilizing object- and page-granular tags for memory safety. Multi-Tag applies memory tagging to enforce fine-grained security policies on the object level in combination with page-granular tags stored in the page table entries. Our multi-granular tagging approach significantly improves the probabilistic detection of memory safety violations without incurring additional memory overhead.
- We adapt our design for ARM MTE, which will be broadly available on upcoming ARMv8.5-A systems. Combining Multi-Tag with MTE increases the security guarantees significantly while requiring minimal hardware changes. Moreover, we showcase how to exploit a platform-specific hardware feature (*i.e.*, ARM PA) to further enhance system security.
- We provide an in-depth security analysis of Multi-Tag and evaluate our design regarding runtime overhead and memory usage, highlighting a low performance impact.
- We implement a prototype, including a gem5-based tagged memory architecture, a modified Linux kernel, and a compiler toolchain based on the Clang/LLVM framework.

Outline. The paper is structured as follows. Section 2 provides the background on memory safety, pointer encoding, and memory tagging. Section 3 specifies the threat model and requirements. Section 4 and Section 5 describe the design and implementation of Multi-Tag. Section 6 provides the security and performance evaluation of our design. Section 7 adapts Multi-Tag's design for ARM platforms and discusses synergies with ARM PA. Section 8 compares related work, and Section 9 concludes this work.

2 BACKGROUND

In this section, we discuss memory safety and the required background on pointer encoding and memory tagging.

2.1 Memory Safety

Programs written in unsafe programming languages like C and C++ are prone to memory safety errors. An attacker can exploit memory safety issues in order to corrupt security-critical data in memory or even take over the system. This security-critical data comprises return addresses, function pointers, and even non-control data allowing the adversary to perform ROP [49], JOP [7], or DOP [24] attacks to hijack the control flow.

Memory safety vulnerabilities can be categorized into spatial and temporal memory issues. Spatial violations are introduced by out-of-bounds memory accesses, e.g., a buffer overflowing into an adjacent memory object. Furthermore, temporal violations are introduced by so-called *dangling pointers*, which are pointers that refer to an already freed memory object. An adversary can misuse dangling pointers to perform use-after-free (UAF) attacks. In addition, uninitialized memory accesses and double-free attacks are also considered temporal violations.

Different countermeasures have been developed to mitigate memory safety violations in unsafe program languages. Software-based approaches [39, 43] utilize compile-time transformations to instrument bounds checks before dereferencing a pointer to detect and counteract spatial violations. The required metadata information is stored in a table managed during the program's runtime. Temporal memory safety issues can be handled by using a garbage collector [8] or by tracking the liveness of memory objects [40]. The major drawback of software-based memory safety, *i.e.*, bounds-checking and liveness tracking, is the significant performance overhead they introduce which is unacceptable for production code. To achieve better performance results, countermeasures utilizing architectural features are required. Hardware-assisted mechanisms [16, 44, 58] implement the bounds-checking and metadata handling in hardware, thus keeping the introduced runtime overhead relatively low. However, hardware-enforced bounds-checking schemes require intrusive hardware and ABI changes (e.g., increased pointer size), thus limiting their applicability for commodity systems.

2.2 Pointer Encoding and Memory Tagging

Defense mechanisms based on tagged pointers repurpose the upper bits of the pointers by reducing the virtual address space on 64-bit architectures. These upper bits of pointers can then be utilized to encode meta-information efficiently without the need for fat pointers. Intel linear address masking (LAM) [14] and ARM top-byte ignore (TBI) [34] provide this functionality by reducing the available virtual address space, thus freeing a number of bits in each pointer. This allows the programmer to embed arbitrary metadata, e.g., to track object information in managed runtimes.

ARM pointer authentication (PA) [56] was introduced with the ARMv8.3-A architecture and was gradually updated with newer instruction set architectures (ISA). The idea of ARM PA is to cryptographically sign a pointer and store the truncated MAC, *i.e.*, the pointer authentication code (PAC), within the unused upper bits of the pointer. This pointer signing mechanism uses a static key k, a 64-bit modifier, and the virtual address to seal a pointer. Depending on the virtual addressing mode, the PAC size can range from 3 up to 31 bits [56]. For Linux, using the 39-bit virtual addressing mode, the

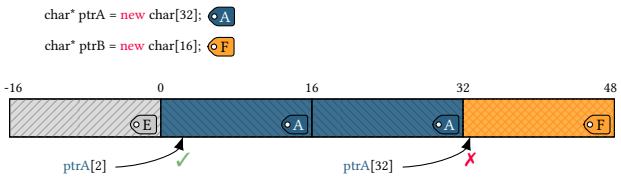

Figure 1: ARM memory tagging extension enforces probabilistic memory safety by comparing memory tags during every memory access.

size of the PAC can range up to 24 bits. ARM PA provides the ability to authenticate the pointer by verifying the integrity of previously PAC-ed pointers. Suppose the pointer is corrupted, e.g., due to a memory vulnerability. In that case, the authentication instruction of ARM PA detects the corruption and sets a dedicated error bit such that subsequent memory accesses using the corrupted pointer lead to an exception. Pointer authentication is currently used by the LLVM compiler to protect return addresses by signing them during the function prologue and verifying their integrity in the epilogue. Moreover, PARTS [32] uses ARMv8.3-A pointer authentication to protect the integrity of code and data pointers.

Memory Tagging. Another promising hardware mechanism for memory safety is memory tagging. Memory tagging is a versatile building block utilized by various countermeasures to enforce memory safety [47, 52, 54, 58]. For tagged memory architectures, two properties are essential; The *tag size* specifies the available tag space in memory, while the *tag granularity* specifies the granule of the tagged memory architecture. The choice of tag size and granularity strongly influences the possible policies and the introduced storage overhead of tagged memory architectures. Memory tagging is implemented on different architectures tailored toward specific use cases. Several dynamic information flow tracking (DIFT) [51, 54] schemes use a single-bit memory tag to implement taint tracking. Capability-based architectures [19] like CHERI [57, 58] and the M-Machine [12] use a single-bit tag to protect capabilities in memory. Additionally, the lowRISC tagged memory architecture [52] uses a 4-bit tag per 8 bytes of memory which is partly configurable, enforcing different security policies.

The ARM memory tagging extension (MTE) [34, 48] is a hardware feature for the ARMv8.5-A architecture, which assigns a 4-bit tag to every 16 bytes of memory. MTE utilizes a *lock-and-key* approach, where the key, *i.e.*, the tag, is also stored in the upper byte of the pointer (enabled by TBI), pointing to that location in the memory. A memory access only succeeds when the tag stored in the pointer (the *key*) matches with the one in memory (the *lock*). Otherwise, an exception is raised, and the program aborts. Figure 1 depicts two memory allocations aligned to the 16-byte granularity followed by two memory accesses. The memory access within the bounds succeeds, while an out-of-bounds memory access to a memory object using a different memory tag fails. Similarly, the SPARC M7 [1] processor series includes the application data integrity (ADI) feature, which offers tagged memory with a tag granularity of 64 bytes and a tag size of 4 bits. HWASan [47] uses ARM's TBI feature to provide memory tagging in software utilizing shadow memory. Similar to ARM MTE and SPARC ADI, HWASan also assigns a tag for the corresponding memory object but checks the tag for every memory access in software.

3 THREAT MODEL AND REQUIREMENTS

In this section, we define our threat model and derive our requirements for the introduced system.

Threat Model. Our threat model is consistent with related work [28, 30, 35, 42, 55], where an attacker has arbitrary read and write capabilities and knows the address space layout. We assume Write-XOR-Execute is enabled, thus, the attacker cannot write code. We assume the operating system is trusted and free of exploitable bugs. Furthermore, we assume the absence of logical vulnerabilities, *i.e.*, programming errors, that would allow an attacker to bypass the heap allocator. In addition, side-channel and fault attacks are out of the scope of this work.

Requirements. We derive the following requirements for our system. Our goal is to mitigate the exploitation of memory safety vulnerabilities. This includes the prevention of spatial safety issues such as adjacent and non-adjacent memory access violations. Furthermore, we aim to mitigate use-after-free (UAF), uninitialized memory, and double-free vulnerabilities to achieve temporal safety. To minimize the performance overhead, we leverage architectural hardware features. We aim to maximize the security guarantees of the underlying hardware building blocks, *i.e.*, probabilistic spatial and temporal security guarantees. Our solution must be compatible with existing source code, *i.e.*, allow existing C code to be used without source code modifications.

4 DESIGN

In this section, we present Multi-Tag, the first scheme leveraging a multi-granular tagging strategy to provide strong security guarantees against spatial and temporal memory safety vulnerabilities. Multi-Tag provides probabilistic protection and withstands various software attacks defined in our threat model by introducing a hardware-software co-design based on object-granular memory tagging combined with page-granular tags. This combination allows Multi-Tag to overcome the limitations of commercial tagged memory architectures like ARM MTE and SPARC ADI incurred by small tag sizes while maintaining a moderate memory overhead.

4.1 System Architecture

We introduce a hardware-software co-design based on multi-granular tags. Here, we utilize fine-grained object-granular memory tags in combination with coarse-grained page-granular tags to enforce access policies.

Overview. In Multi-Tag, we embed the multi-granular tag, consisting of the object- and page-granular tag, into the upper virtual address bits of a pointer. Figure 2 illustrates the pointer encoding of our design. While the object-granular tag is stored in memory, we store the page-granular tag in the PTE, thus avoiding additional memory pressure and memory storage overhead. Thereby Multi-Tag is capable of assigning a unique tag for each object on a 4 kB page without inducing infeasibly large memory overheads.

During memory load and store operations, different hardware primitives use the multi-granular tag to perform access permission

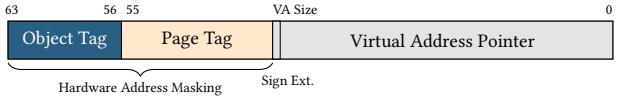

Figure 2: The encoded pointer layout. Multi-Tag utilizes a combination of an object-granular and a page-granular tag to perform memory access checks. The tagged memory architecture provides the object-granular tag. The page-granular tag is stored in the PTE of the corresponding page.

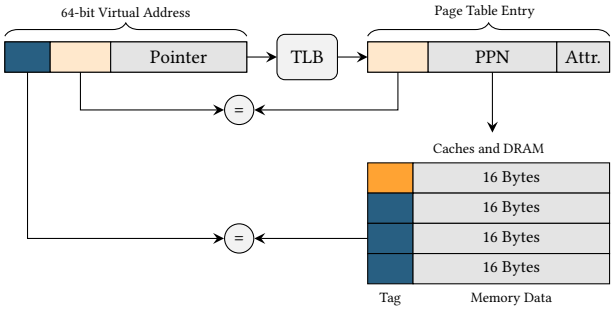

Figure 3: Memory access checks in hardware. The tag comparison of the object- and page-granular tags enforced during every memory operation consists of two parts. First, a comparison between the object-granular tag and the tagged memory architecture. Second, a comparison of the page-granular tag and the PTE.

checks. As shown in Figure 3, the hardware compares the fine-granular part of the multi-granular tag encoded into the pointer with the object-granular tag stored in memory during every memory access. Moreover, the coarse-granular part of the multi-granular tag is compared to the page-granular tag embedded in the PTE. Access to a distinct memory location is only granted if both checks succeed.

Multi-Granular Tag Sizes and Tag Granularities. Similar to related work [1, 48], we encode the tag metadata into the upper bits of the pointer. Depending on the size of the virtual address space, e.g., **39-** or **48-bit**, the topmost 25 or 16 bits in the virtual address are unused by default. Hence, different sizes for the multi-granular tag are possible. In a **39-bit** virtual address space configuration, Multi-Tag utilizes 24 of the topmost bits, which are ignored during the address translation, for the multi-granular tag and a single bit for the sign extension. Here, our tagged memory architecture could use 8 of these 24 bits as an object-granular tag and 16 bits for the page-granular tag. By tagging every 16 bytes of memory stored in DRAM with an 8-bit tag, we can tag each of the 256 possible memory objects in a 4 kB page uniquely, thus preventing tag collisions within a page. We tag each page with a 16-bit tag, hence the probability of a tag collision for data objects located on distinct pages is $2^{-8} \cdot 2^{-16} = 2^{-24}$. As the page-granular tag is only stored in the unused bits in the PTE, the memory overhead introduced by Multi-Tag is determined by the 8-bit tag per 16 bytes, which equals an overhead of 6.25 %. For a **48-bit** configuration, the number of available bits Multi-Tag can utilize in the PTE decreases from 24 to 15 bits. By using an 8-bit object-granular tag, all 256 16-byte memory objects in a 4 kB can still be uniquely tagged. However, the probability of having a tag collision over different pages increases to $2^{-8} \cdot 2^{-7} = 2^{-15}$.

Note that the tag sizes can be chosen according to the security and memory overhead requirements. For instance, Multi-Tag can be instantiated with 4-bit object-granular tags, similar to ARM MTE, which halves the introduced memory overhead. Moreover, the size of the page-granular tags can be chosen to reduce the required amount of PTE bits. The BIOS or the operating system can set both tag sizes at boot time.

System Requirements. To enable multi-granular tagging, we only require minimally-intrusive system changes: First, an instruction set extension (ISE) is needed to set the tag bits for memory objects in the pointer. Furthermore, a new system call is added to configure the page-granular tag in the PTE. Moreover, the TLB is modified to perform the page granular tag comparison. Finally, the memory system needs to be aware of the object-granular memory tags and needs to check them on each memory access.

Tag Integrity Protection. Pointer arithmetic poses a threat to schemes that encode metadata into the upper bits of the pointer. Precisely, pointer arithmetic operations overflowing into the tag bits of the encoded pointer can lead to tag forgery attacks. To overcome this problem, there exist different approaches. First, tag-aware pointer arithmetic instructions or an instruction sequence that provides integrity for the tag bits prohibit or detect an overflow. Second, a single-bit memory tag (similar to CHERI [58]) could be utilized to mark all pointers in the register file and in memory. Subsequently, the ALU uses this type-information to avoid writing into the tag bits. In this work, we provide tag integrity for encoded pointers by introducing separate instructions for pointer arithmetic that do not overflow into the tagged upper bits.

4.2 Enforcing Memory Safety with Multi-Tag

By assigning and managing tags in the memory allocator, memory safety can be enforced with Multi-Tag. In general, the memory allocator implements the following procedure: On allocation, the allocator aligns the size of the memory object to the tag granularity, e.g., 16 bytes. Additionally, the allocator tags the pointer to a memory object by generating and storing the tag in the upper bits of the pointer. The same tag gets assigned to the memory location as associated metadata information utilizing the ISA extension of the tagged memory architecture. By zeroing the initialized memory during this tag assignment, uninitialized memory accesses attacks are mitigated. Similarly, the page-granular tag gets encoded into the upper bits of the pointer. The memory allocator manages the page-granular tag using system calls. Depending on the tag assignment policy, different memory safety guarantees can be enforced. Precisely, we introduce two policies. The first policy enforces spatial memory safety, while the second policy enforces both spatial and temporal safety. Implementing two policies allows us to evaluate their performance overhead separately.

Spatial Memory Safety. In this tag assignment policy, we assign each memory object within a page a unique object-granular tag

by tracking already assigned tag values. To ensure this property, a tag granularity of 16 bytes and an object-granular tag of 8 bits are one possible tag size and tag granularity selection. With this combination of parameters, all 256 16-byte memory objects on a 4 kB page can be tagged with a distinct tag. This prevents any adjacent or non-adjacent spatial memory safety violations, e.g., buffer overflows, within the same page.

For the page-granular tag, we assign a pseudorandom tag to every page. To prevent adjacent spatial memory safety violations over page boundaries, neighboring pages are assigned different page-granular tags. For objects larger than a single page, an identical page-granular tag is used for the pages containing the object. Through the use of pseudorandom tags, non-adjacent memory safety violations over page boundaries are prevented on a probabilistic basis.

Spatial and Temporal Memory Safety. The page-granular tag can be utilized to counteract spatial *and* temporal memory safety vulnerabilities at the same time. Similar to the previous tag assignment policy, we assign unique tags for every object co-located on the same page enforcing spatial security. We prevent the misuse of dangling pointers using two separate techniques. First, we make the memory location inaccessible by tagging it with the reserved *zero* object-granular memory tag. And second, to prevent future allocations in that area from having the same object-granular memory tag, we exclude this value for future allocations on that page until it has a new page-granular tag. We also prevent double-free attacks by checking if the tags of the corresponding memory location are valid before freeing. Memory reallocation is immediately possible if unused tags within the page are available. Otherwise, the page needs to be re-tagged. For this, we keep track of the quarantined memory and wait until all objects within that page are freed. Once that is the case, the page gets re-tagged using a different pseudorandom page-granular tag, ensuring that future memory allocations have a different tag.

5 IMPLEMENTATION

Our prototype implementation consists of a tagged memory architecture integrated into the gem5 [6] simulator, a kernel extension to provide page-granular tagging, and an LLVM-based [29] toolchain for instrumenting heap allocations. Multi-Tag is a scheme utilizing memory tagging in combination with page-granular tags stored in the PTEs. While our design is ISA-agnostic, we provide a prototype implementation for the x86-64 instruction set architecture (ISA) since the gem5 model for this architecture is the most mature.

5.1 Tagged Memory Architecture

We base the implementation of our tagged memory architecture on the gem5 simulator to provide an accurate performance model. Memory tagging incurs overhead by introducing additional DRAM requests required to fetch the tag metadata from memory. While Multi-Tag allows for a variety of combinations of tag sizes and granularities, our prototype implements a variant that associates an 8-bit tag with every 16 bytes of memory. We reserve a dedicated memory region of the DRAM for storing the tags in memory. The size of the reserved region is proportional to the overall system memory size. In our prototype implementation, we select the size

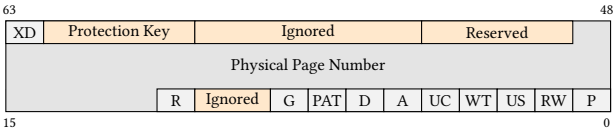

Figure 4: Page table entries on the Intel x86-64 architecture. We utilize the unused 9 bits (bit 58:52 and bit 10:9) and repurpose the 4 Protection Key bits for our design. Moreover, we could reduce the addressable physical memory leading up to 16 bits for the page-granular tag.

of the special memory region such that all of the remaining system memory can be associated with a tag. Reading and writing the tags is only possible by using special instructions, and the tag memory region is inaccessible for regular memory load and store operations. Our custom memory allocator uses these special instructions to enforce our defined security policies. To access data and the corresponding tags at every cache access, we increase the size of each cache line to store the associated memory tags. Furthermore, access checks are performed during every memory operation, preventing memory accesses on a tag mismatch. In addition, tagged memory architectures require additional DRAM fetches for every cache miss to load the tag metadata from memory. For every cache miss, the memory controller issues one additional DRAM request to load the corresponding tag from memory.

A tag store operation can either result in a cache hit or a cache miss. In the case of a cache hit, the cache line holding the data and the corresponding tag is marked as dirty as a write to a tag has to be handled identically to a write aimed at regular data. A cache miss will cause a load that fetches the tag and the associated data from DRAM, followed by a subsequent write to the cache line that now contains the tag. When reading a tag using the dedicated tag read instruction, a cache miss will cause a load from the main memory that caches the tag and the corresponding data.

Our gem5 prototype implements this basic behavior without an advanced tag cache design, providing a worst-case performance approximation using a basic tagged architecture. Note that advanced designs [26] for tagged memory architectures can significantly improve system performance. The memory tags for the tagged memory architecture are encoded into the topmost byte of the pointer. We utilize the remaining upper bits of the pointer as a page tag, which is stored without memory overhead in the PTEs. These upper bits are ignored by our system during the address translation procedure. Our implementation utilizes a 39-bit virtual memory configuration of the Linux system resulting in up to 24 bits that can be used for metadata. Precisely, we split up the 24 bits into an 8-bit object-granular and 16-bit page-granular tag.

5.2 Kernel Extension

The operating system is responsible for managing the tags in the PTEs and memory during allocation and for special system calls.

Page Table Entries. Figure 4 illustrates the layout of Intel x86-64 page table entries, highlighting 7 bits (bit 58-52) that are currently unused. In combination with bits 9 and 10, which are also unused, the total amount of unused bits in the PTE is 9. Furthermore, the bits

62 to 59 are used for the Intel memory protection keys (MPK) [46], which we repurpose for our design. Additionally, we can use 3 of the bits reserved for the physical page number, summing to up to 16 bits for the page-granular tag on a 39-bit virtual address Linux system. While this reduces the physical address space to 49 bits, which results in a total of 512 TB of addressable memory, this is still enough for most applications. However, our design introduces a dynamic trade-off between memory and security by reducing the size of the page-granular tags, which allows larger ranges of addressable DRAM.

System Calls. We extend the system calls `mmap` and `mprotect` to tag requested pages accordingly by inserting the corresponding tag into the PTE. In the case of Linux, like the other permission bits, the kernel also stores the tag in the `vm_area_struct`. The page-granular tag is stored in the PTE and checked in hardware during the TLB access (cf. Figure 3). This approach does not increase the memory overhead and does not introduce any performance overhead as the tag comparison is done in hardware in parallel to the other PTE permission (e.g., read, write, execute) checks.

5.3 Toolchain

In our paper, we focus on protecting spatial and temporal memory safety on the heap. Hence, our toolchain consists of an LLVM-based compiler and an instrumented musl-libc heap allocator.

Compiler Extension. We base our prototype on the LLVM 14 framework [29], which is extended to support tagging instructions to control the tagged memory architecture. Furthermore, the compiler is responsible for using the tag-aware instructions for pointer arithmetic operations. These instructions ensure the integrity of the upper tag bits when performing arithmetic operations on the pointer. To achieve this, the compiler lowers all `getelementptr` (GEP) instructions to our newly introduced tag-aware pointer arithmetic operations.

Our design requires dedicated instructions to perform operations on the object-granular tag value stored in memory. Precisely, we utilize store tag instructions that set the dedicated memory tag with and without initializing the corresponding memory location. Additionally, we implement a load tag instruction to retrieve the tag value from a dedicated memory location corresponding to the tagged address. Finally, the instructions to perform tag-aware pointer arithmetics, e.g., addition and subtraction, are added.

Heap Allocator. The instrumented heap allocator is responsible for enforcing the previously defined security policies (cf. Section 4.2) using multi-granular tagging. To provide spatial memory safety, the allocator tags the corresponding memory locations using the object- and page-granular tags. The allocator keeps track that tags of adjacent memory locations are distinct, thus protecting against linear overflows. Non-adjacent allocations are protected by the object- and page-granular tag. Therefore, the allocator uses the following procedure: First, the heap allocator aligns allocations to the tag granularity of 16 bytes. Second, the allocator tags the memory location using our tagging instructions and encodes the tag into the topmost byte of the pointer. The set tag instruction assigns the tag and zeroes the data stored at this memory location, effectively mitigating attacks based on uninitialized memory accesses. Afterward, the memory allocator checks whether the current page is already tagged. If the page is not tagged, the allocator requests the operating system to tag the allocated page using our modified `mprotect` system call. Once a page-granular tag is assigned, this tag is set into the page-granular tag field in the pointer.

Besides spatial security, our allocator uses memory tagging to enforce temporal memory safety for the system. Our heap allocator utilizes per-page metadata in order to prevent the misuse of dangling pointers. Such UAF attacks are mitigated by assigning every object-granular memory tag within a page only once. After each distinct tag (8 tag bits map to 256 unique tags) on a page is used, the allocator starts to put freed memory objects within this page into quarantine. Before allocating new memory on such a page, the allocator waits until all objects within the quarantined page are freed. Afterward, the allocator can re-tag the quarantined page and reuse it for future memory allocations. Additionally, double-free attacks are mitigated since we check if the tag is not zero before freeing the memory. Precisely, we instrument the `malloc`, `calloc`, `realloc`, and `free` functions to include this functionality.

6 EVALUATION

This section provides an in-depth security analysis of Multi-Tag. Additionally, we analyze the overhead of our design in terms of performance and memory usage.

6.1 Security Analysis

In this section, we analyze the security guarantees of Multi-Tag in terms of spatial and temporal heap memory safety. In general, our design provides probabilistic memory safety utilizing a hardware-software co-design based on multi-granular tags. This means that the exact detection probability depends on the implemented tag size of the tagged memory architecture. Since our prototype only instruments heap memory, we exclude globals and stack memory vulnerabilities for this analysis. Stack memory is explicitly discussed in Section 8.1.

Spatial Security Analysis. Multi-Tag prevents the exploitation of spatial memory safety vulnerabilities by utilizing the underlying tagged memory architecture in combination with page-granular tags. For our analysis, we distinguish between adjacent, non-adjacent, and intra-object memory accesses. Memory objects are assigned both, an object- and a page-granular tag on allocation, and accessing these tagged objects requires that the pointer used for the access contains the correct tag. The memory allocator assigns a unique tag for every object within a page, thus eliminating intra-page memory corruption. To deterministically protect against linear overflows across page boundaries, the memory allocator assigns different tags to adjacent memory objects. For arbitrary memory accesses, e.g., non-linear buffer overflows, the protection capabilities of Multi-Tag are only limited by the tag size of the underlying tagged memory architecture and the page-granular tag. The page-granular tag improves the security guarantees of Multi-Tag by increasing the available tag space. By combining a fine-granular 8-bit memory tag and a coarse-grain 16-bit page tag, the tag collision probability (*i.e.*, the tags of two different objects coincide) is $2^{-8} \cdot 2^{-16} = 2^{-24}$. Our flexible design allows to dynamically select the size of the page-granular tags. Thus, it is possible to permit a larger addressable physical memory if needed.

We resize memory allocations to a multiple of the tag granularity to ensure that the entire object can be protected using the underlying tagged memory system. Hence, linear out-of-bounds accesses within the borders of the resized object cannot be detected. However, from a security point of view, this is uncritical since there is no data stored, and new allocations start with the next tag granular address. Currently, Multi-Tag cannot prevent intra-object overflows, for example within C structs, as the struct is a single memory object. Similar to related work [60], source-to-source transformations could be used to promote C struct members to separate allocations and provide tagging for them.

To prevent tag corruption due to arithmetic operations, the compiler instruments them using our tag-aware instructions. However, code sequences where the type information is lost, e.g., due to casts, cannot be instrumented. Such casts allow the attacker to guess the tag encoded in the upper bits of the pointer. We argue that such code sequences where the attacker could add arbitrary user input to an integer-casted pointer are fairly uncommon, and the attacker still needs to guess the correct page-granular tag to access an arbitrary memory location.

Temporal Security Analysis. For our temporal security analysis, we distinguish between uninitialized memory, double-free, and use-after-free (UAF) attacks. Uninitialized memory vulnerabilities are inherently prevented since every allocation is zero-initialized during the tag assignment. The heap allocator checks the liveness of every object before freeing the memory using the special memory tag zero to protect against double-free attacks. If an object with the memory tag zero gets freed, the double-free is detected and handled accordingly.

For dangling pointers that may lead to UAF attacks, we differentiate between three cases. In the first case, the dangling pointer points to a previously freed memory location that is not currently used by a subsequent allocation. As the freed memory location is tagged with the special zero tag, we deterministically detect such cases. The second case occurs if the dangling pointer points to a memory location that has been reused by a following memory allocation. As the allocator guarantees that all object-granular tags on the same page are unique, the stale tag of the dangling pointer will always cause a tag mismatch on access. Finally, a dangling pointer may refer to a memory location on a page from which all allocations have been freed. There, our allocator would then assign a new pseudorandom page-granular tag, which means that all object-granular tags can be reused. Thus, since the page-granular tag is chosen pseudorandomly, subsequent allocations on this page can have the same tag as the dangling pointer. The probability of such a collision is 2^{-24}. This is significantly better than using only object-granular tags, *i.e.*, ARM MTE, which has a much higher collision probability of 2^{-4}.

Alternatively, our object-granular tagging strategy can also be applied to pages: Instead of choosing the page-granular tags pseudorandomly, they can be chosen systematically, such that each tag is only assigned once. For this, a 16-bit maximum-length LFSR that generates all possible $2^{16}-1$ values for the page-granular tag can be used. This guarantees that no collision between page-granular tags will occur before the period of the LFSR is reached. By initializing the state to a value that depends on the address of the page, we can check if a page has used up all available unique tags. Once the period of the LFSR is reached, the (virtual) page will be put into quarantine and not be reused again. While this slightly decreases the usable virtual address space over time, this strategy can be used to deterministically prevent any UAF vulnerability using Multi-Tag.

Multi-threading. Concurrency-based attacks are particularly hard to mitigate. Many software-based approaches accept a time window where an attacker could mount a time-of-check to time-of-use (TOCTTOU) attack. Tagged memory systems like Multi-Tag do not have this flaw since the tags are always checked when accessing the data. Since the operating system always ensures TLB coherency, it automatically ensures coherency for our page-granular tags.

6.2 Performance Evaluation

To evaluate the performance of Multi-Tag, we utilize the SPEC CPU2017 [9] and the nbench-byte [36] benchmark suite. Our gem5 model in full system mode provides an accurate simulation of the memory subsystem (including memory access latencies based on real-world DRAM timings). As the `ref` input of SPEC leads to infeasible simulation times, we use the smaller inputs to measure the performance impact. Our toolchain currently does not support C++, so we are limited to benchmarks that are written in C. Similar to the methodology of related work [32], we run each test case of nbench-byte using a fixed number of iterations. Furthermore, all benchmarks of both benchmark suites are compiled with optimization level `-O3`.

Configuration. We configure our gem5 model in full system mode to represent a commodity computing system. We use the x86-64 ISA and set the clock frequency to 3 GHz. Our model uses an 8-way set associative L1 instruction and data cache with 16 kB and 64 kB, respectively. Moreover, we use a 16-way set associative L2 cache of 256 kB in combination with a single-channel 2 GB 2400 MHz DDR4 DRAM. We integrate a custom module generating tag requests between the memory controller and the memory bus. For our simulation, we use the timing CPU model of gem5.

Results. Figure 5 illustrates the simulated performance overheads for the SPEC CPU2017 and nbench-byte benchmarks. We evaluate *(i)* the overhead incurred by the tagged memory architecture, *(ii)* the overhead of spatial memory safety, and *(iii)* the overhead of spatial and temporal memory safety using Multi-Tag's design and compare the relative performance overhead. The overhead of the tagged memory architecture largely depends on the number of memory accesses that result in a cache miss in both, the L1 and the L2 cache. As each access to DRAM that is not a tag load or store causes an additional tag fetch, memory access patterns that cause high cache pressure result in larger performance overheads. Spatial and temporal memory safety incurs more overhead since the allocator tags all memory at allocation time. This causes the memory of the entire allocation to be fetched into the caches before it is actually needed, leading to increased cache pressure.

For some SPEC benchmarks, the performance overhead of all evaluated configurations is nearly equivalent. We suspect that in such cases, the majority of tag load and store instructions result in cache hits, thus causing only a small amount of additional fetches from main memory. Note that the larger architectural overhead for the `505.mcf` benchmark is due to noise in the measurements.

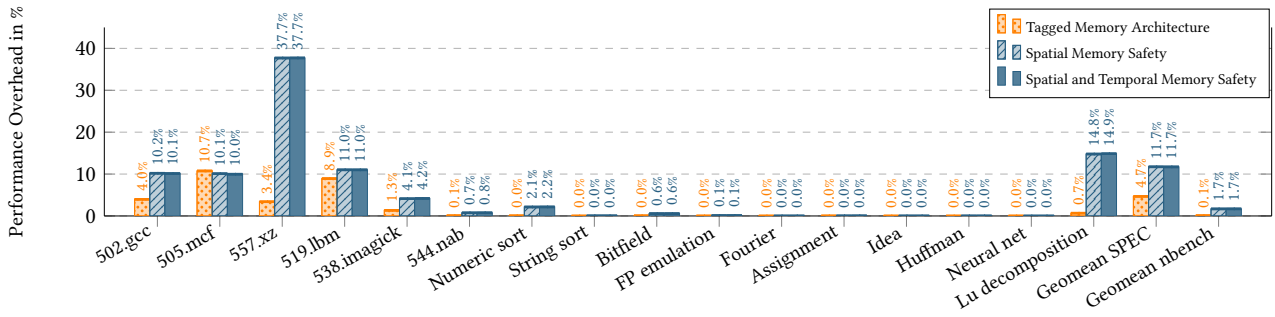

Figure 5: Simulated performance overhead of the SPEC CPU2017 and nbench-byte benchmarks.

For the nbench-byte benchmark suite, we see very low overheads that are close to zero most of the time. We accredit this good performance to the fact that the allocations of most nbench-byte benchmarks are small compared to SPEC. Small allocations do not cause large amounts of tag store operations, and the cache miss rate is lower.

6.3 Memory Usage and Overhead

Multi-Tag increases the memory usage of the program due to the alignment to 16 bytes for every allocated object. However, the increased memory usage is typically less than 0.1 % for the evaluated benchmarks.

Our scheme co-locates the memory data as well as the tag metadata in the cache. Furthermore, the tag metadata is stored in a dedicated DRAM region. We associate an 8-bit tag with every 16 bytes of memory, resulting in a memory overhead of 6.25 %. One possible optimization to minimize the required memory overhead is to increase the tag granularity. E.g., SPARC ADI uses 4-bit per 64 bytes. Similarly, we could utilize 6-bit per 64 bytes, which also allows assigning a unique tag per page and results in a memory overhead of 1.17 %. The memory overhead can be further reduced to effectively 0 % by leveraging ECC DRAM modules [22, 26, 58]. This also has another advantage since it eliminates the performance overhead introduced for every cache miss by the extra DRAM requests for fetching the tags.

7 CASE STUDY

In this section, we first *(i)* analyze how to implement Multi-Tag on an existing tagged memory architecture. Here, we focus on ARM as the upcoming memory tagging extension (MTE) for ARMv8.5-A systems will soon become broadly available. We showcase that Multi-Tag only requires minimal hardware changes on systems already featuring tagged memory. Then *(ii)*, we discuss the security benefits of integrating Multi-Tag into ARM MTE systems. Finally *(iii)*, we demonstrate and evaluate how these security guarantees can be further enhanced by combining Multi-Tag with platform-specific hardware features, *i.e.*, ARM PA.

7.1 Multi-Tag on ARM MTE Platforms

ARM MTE introduces tagged memory for ARMv8.5-A systems. There, 16 bytes of memory are tagged with a 4-bit memory tag, which is also stored in the unused upper bits of the virtual address. For our multi-granular tagging approach, we utilize these 4 bits as the object-granular tag. As ARM systems have 4 unused bits in the PTE, we can utilize them for page-granular tagging and also include them in the upper bits of the virtual address. With the ARM TBI feature, the hardware already ignores the topmost byte during address translation. The only hardware change we need is the comparison of the page-granular tag. Similar to our base design, we instrument the heap allocator to apply the object- and page-granular tag during allocation. Additionally, LLVM provides the `AArch64StackTagging` pass, which is utilized to apply ARM MTE on the stack. Mitigating tag forgery attacks introduced by pointer arithmetic operations can either be tackled in hardware or software. Introducing dedicated tag-aware pointer arithmetic operation instructions prevents tag manipulation. Alternatively, a special instruction sequence can ensure the tag integrity in software during arithmetic operations. However, dedicated instructions are usually a more favorable solution as they do not induce any performance impact.

7.2 Security of Multi-Tag using ARM MTE

In this section, we compare the security guarantees of native ARM MTE with Multi-Tag combined with ARM MTE.

Security of ARM MTE. MTE provides probabilistic protection based on the size of the tag. Precisely, MTE utilizes a 4-bit memory tag at the granularity of 16 bytes to tag the memory. While this design choice of using 4-bit tags minimizes the overhead of storing the tags in memory, it also limits security. First, the probability that two memory objects receive the same tag is, with a probability of $1/16 = 6.25\,\%$, high. The memory allocator could guarantee that consecutive allocations are assigned a different tag, thus, linear buffer overflows are prevented. However, non-linear buffer overflows, as well as temporal memory safety vulnerabilities, cannot easily be prevented with such a small tag size. Second, an adversary with access to a memory object containing a tagged pointer can overwrite this pointer using pointer arithmetics and guess a correct tag with a high probability. Due to these limitations, MTE is currently primarily used as a debugging feature, e.g., as part of the MemTagSanitizer.

Security of Multi-Tag combined with ARM MTE. Integrating Multi-Tag into a platform already featuring ARM MTE greatly

enhances security guarantees as the multi-granular tagging approach increases the available tag space. As the object-granular tag size here is 4 bits, assigning a unique tag for each memory object on a page (cf. Section 4) is not possible. Hence, the memory allocator generates a pseudorandom tag and assigns it to the distinct memory region. For spatial safety, assigning different tags to adjacent memory objects can prevent the exploitation of linear buffer overflows deterministically. For non-linear buffer overflows, *i.e.*, arbitrary read-write vulnerabilities, the protection guarantees of Multi-Tag are limited by the size of the object-granular tag determined by the underlying tagged memory architecture. On ARM systems, the 4-bit object-granular tag yields a collision probability of 2^{-4} when accessing arbitrary memory objects within the same page. For objects allocated on different pages, the inter-page tag collision probability for non-linear buffer overflows is $2^{-4} \cdot 2^{-4} = 2^{-8}$ since the object- and the page-granular tag need to coincide. In terms of temporal safety, UAF attacks are prevented on a probabilistic basis due to the pseudorandom memory tag. The limited tag space of MTE leads to the problem that pages need to be re-tagged more frequently. MTE can mitigate uninitialized memory accesses by assigning the memory tags and zeroing the data of the memory location. Furthermore, a special memory tag for freed memory can be reserved on ARM platforms. However, this might prove too costly since only 16 distinct memory tags are available on ARM.

7.3 Combination with ARM PA

Since temporal memory safety violations cannot be efficiently prevented due to the limitations of the underlying ARM hardware, we propose additional memory protection based on ARM pointer authentication (PA). ARM PA is a compatible architectural feature that can be used to sign and authenticate a pointer's address and tag (cf. Section 2). The TBI hardware feature uses the topmost byte for metadata and allows up to 16-bit PAC for a 39-bit virtual address system. In the following, we discuss possible combinations with ARM PA used for pointer integrity and revocation. Depending on the size of the authentication code, PA provides strong probabilistic protection for code and data pointers.

ARM Pointer Authentication. We distinguish between two applications for ARM PA. First, we provide pointer integrity by utilizing static modifiers, and second, pointer revocation using pseudorandom modifiers assigned at runtime and stored in shadow memory. Similar to related work [32], we protect all data pointers using a static modifier. Therefore data pointers get sealed before storing them in memory, providing an additional layer of security for pointers. Moreover, we use ARM PA for pointer revocation by associating a modifier to track the object's liveness. The memory allocator assigns the object- and page-granular tags during allocation for spatial memory protection. Furthermore, we mitigate UAF and double-free attacks using ARM PA and shadow memory. Pointer authentication uses a modifier as nonce input, a key, the address, and the TBI metadata, to compute its PAC. Therefore, a pseudorandom modifier is generated and stored in the shadow memory during the allocation of stack and heap objects. Since the PAC is truncated to 16 bits, we also use a 16-bit modifier (instead of the maximum of 64 bits) stored in the shadow memory for our evaluation. To access the modifier value, we derive the index of the shadow memory using the address of the corresponding pointer. On pointer authentication, first, the modifier is fetched from the shadow memory. Then, the PAC is verified using the modifier, the sealed pointer, and the key. On successful authentication, the plain pointer is retrieved by striping the PAC from the pointer. When an adversary manipulates a pointer, the authentication procedure fails, which is detected by the system. On deallocation, we first check whether the corresponding modifier in the table is valid. If the modifier is valid, we invalidate the modifier, and the memory gets freed by the allocator. We mitigate the exploitation of possible dangling pointers remaining in memory by invalidating the modifier. If the modifier is already invalid, a double-free attack is detected and handled accordingly.

We have different options to verify the integrity of the pointer, which is associated with the liveness of the object. For example, we can authenticate after loading the pointer from memory or before the pointer dereference. Authentication after loading the pointer from memory increases the system performance significantly. However, the drawback is the introduced time window for possible TOCTTOU attacks. Our prototype implementation verifies data pointers after being fetched from memory to minimize the performance penalty as much as possible.

Enhanced Temporal Safety using ARM PA. Multi-Tag on ARM thwarts temporal memory safety vulnerabilities using the combination of memory tagging, pointer authentication, and shadow memory. When a memory object is deallocated, the modifier stored in the table gets deleted and, thus, invalidated. Hence, if an adversary tries to load a dangling pointer stored in memory, the authentication with PA fails, as the corresponding modifier used for sealing the pointer is not available anymore. The received modifier is either zero if the corresponding memory was not reallocated or is a new pseudorandom modifier if another memory object was allocated using the referred memory. This decreases the probability of a UAF vulnerability from 2^{-4}, by using plain MTE, to the combined probability of a collision of the modifiers and the MTE tags, *i.e.*, 2^{-20}. In addition, the heap allocator checks the liveness of every object before freeing the memory to protect against double-free attacks. Uninitialized memory vulnerabilities are prevented since the memory locations get zeroed during the tag assignment. Furthermore, we protect the shadow memory by reserving a dedicated page-granular tag for the shadow memory, making them inaccessible to an attacker.

TOCTTOU. The ARM PA revocation introduces a time window for TOCTTOU attacks, depending on when data pointers are checked. Authenticating the pointer after loading it from memory results in significant performance improvements compared to authentication before every pointer dereference. Various other performance optimizations are discussed in related work [23, 31, 39], effectively removing redundant access checks. However, these performance optimizations also increase the exploitable time window for shared objects of multi-threaded programs, introducing a trade-off between performance and security. Multi-Tag is a probabilistic scheme and targets runtime protection. We accept this time window since the performance overhead would make the countermeasure impractical. Even if an attacker manages to mount a TOCTTOU attack, we fall back to the security guarantees of ARM MTE.

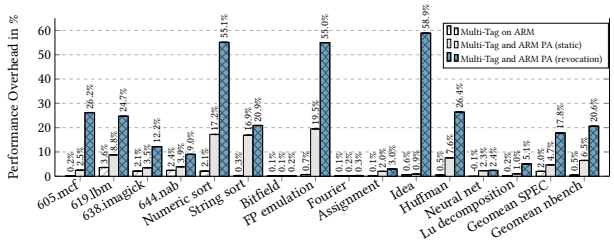

Figure 6: The relative runtime overhead of the ARM evaluation using the SPEC CPU2017 and nbench-byte benchmarks.

We argue that access to shared resources is typically tied to a locking mechanism, reducing the attack window if the pointer check is within the lock. The compiler could enforce to re-load and authenticate pointers, binding the authentication to the locking mechanism. Thus, already authenticated pointers in the register file only exist when the corresponding lock is held.

7.4 ARM Performance Evaluation

In this section, we evaluate the performance of Multi-Tag on ARM platforms. We use SPEC CPU2017 and nbench-byte on an ARM-based Apple M1 processor, which already implements the ARM PA feature. For ARM MTE, we estimate the overhead of this feature (cf. Appendix A). Figure 6 shows the performance results. We evaluate our ARM prototype implementation for memory tagging in combination with Multi-Tag's page-granular tags. The memory tagging overhead depends on the number of objects and their size since every allocated object must be tagged. The ARM MTE approximation highlights a negligible performance overhead for tagging stack and heap objects. Additionally, we evaluate combinations with pointer integrity and pointer revocation using ARM PA. The overhead strongly depends on the program's pointer usage. For example, the `Fourier` benchmark only uses a small number of data pointers resulting in a minimal performance overhead. Contrary, benchmarks like `Numeric sort`, `FP emulation`, and `Huffman` make extensive use of data pointers, resulting in higher runtime overheads.

8 DISCUSSION

In this section, we discuss the limitations of our prototype implementation, possible future work, and existing related work.

8.1 Limitations and Future Work

Currently, our prototype only instruments C programs, but toolchain support for C++ could be added in future work. Therefore, the C++ memory allocator (e.g., `new`, `delete`) needs to be instrumented to apply the object- and page-granular tags. In this work, we focussed on the protection of heap objects since dynamic memory is one of the most common attack targets [21, 38]. Additionally, our prototype could be extended to protect stack and global data as well. However, there are many different approaches to choose from, which adds significant implementation complexity. For example, a safe stack could be implemented by building on top of LLVM's `SafeStackAnalysis`, and we can protect the safe stack with a distinct tag and apply different tags for normal stacks. Alternatively, each stack object could be tagged separately. However, these objects would also need to be aligned and padded to our minimum tag granularity. Furthermore, this is complicated by the fact that these allocations can have a dynamic size and need to be un-tagged at function return. A third approach could be to only tag the return addresses. Due to the wide range of design decisions and implications, we defer this stack analysis to future work. Nevertheless, Multi-Tag, similar to other tagging schemes, could be used for stack protection. Furthermore, our approach can also be used to protect global data. Global memory behaves like a simplified heap, requiring tagging of the global data during the program startup. Moreover, our design is tailored towards 4 kB pages and currently does not support huge pages. Different design and security trade-offs need to be explored to support huge pages. However, current operating systems, like Linux, use 4 kB pages by default. Moreover, virtual page aliasing similar to xTag [5] in combination with our (virtual) page-granular tags could be further explored for different tagging strategies.

8.2 Related Work

C and C++ memory safety issues are already well-studied research topics. However, the previous solutions tend towards high overheads, which is often unacceptable for mass deployment. On the other hand, hardware-assisted countermeasures solve this problem efficiently but require intrusive hardware and ABI changes. Researchers have proposed various countermeasures over recent years, considering the problem from different perspectives [2, 3, 10, 11, 13, 17, 25, 43, 54].

Bounds-checking in Software and Hardware. Many memory safety schemes are based on additional bound checks, enforced either in software or hardware. For example, software solutions like SoftBound+CETS [40, 39] achieve the goal of full memory safety by performing compile-time transformations. The SoftBound compiler instruments additional bounds checks for load and store operations which might lead to out-of-bounds memory accesses. Therefore, SoftBound utilizes a disjoint hash table to store the pointer base address and the object bounds. For temporal safety, CETS extends this concept and introduces a unique identifier associated with the object's liveness to mitigate dangling pointers for heap and stack objects. The significant drawback of software solutions like SoftBound+CETS is their tendency to introduce significant performance overheads. Henceforth, Hardbound [16], Watchdog [41], Intel MPX [44], and CHERI [58] enforce the bounds-checking mechanism in hardware. These countermeasures introduce different types of data structures to store the additional metadata, e.g., inline, adjacent, or disjoint metadata storage. While offering a reasonable performance overhead, hardware-enforced bounds-checking requires intrusive hardware and ABI changes, which are hard to deploy on a large scale. At its core, the register file gets extended to hold the pointer bounds information, which is additionally checked during every memory access. AOS [27] enforces memory safety for heap objects based on bounds metadata utilizing ARM PA. All pointers are signed using the PA instruction, and the PAC is further used to access the metadata efficiently. During every memory access, the system additionally checks the object bounds in hardware using the PAC to index the required bounds metadata. In contrast, Multi-Tag utilizes an efficient hardware-software co-design

based on multi-granular tags to enforce memory safety. Our design enforces memory safety based on lightweight hardware changes, transparent to the programmer.

Tagged Memory in Software and Hardware. HWASan [47] uses memory tagging to detect memory safety errors. This scheme assigns a memory tag during allocation and stores this tag in the upper 8 bits of the pointer by utilizing ARM's TBI feature. Upon every memory operation, the tag of the pointer is checked against the tag stored in the metadata table. Similarly, MemTagSanitizer applies the same procedure as HWASan but uses ARM MTE to perform the access check in hardware. Both schemes are primarily used for debugging purposes since the schemes cannot withstand the standard adversary model (cf. Section 3). An attacker could simply overwrite the memory tag stored in the upper bits of the pointer to forge the tag and address of the pointer. Compared to HWASan and MemTagSanitizer, Multi-Tag utilizes memory tagging for security by providing tag integrity and introducing a high detection probability. Furthermore, HWASan introduces a high runtime overhead, which makes it impractical for production code. Color My World [33] utilizes ARM MTE to enforce memory safety for objects that can be statically proven safe during compile time. Moreover, CrypTag [42] uses memory tagging in combination with memory encryption. Precisely, the tag encoded into the pointer is used to tweak the encryption for the dedicated memory location. The disadvantage of this approach is the substantial performance penalty introduced by the authenticated memory encryption [53].

ARM PA-based Protection Mechanisms. PACSafe [23] provides memory safety by utilizing counter values stored in shadow memory. PACSafe can be seen as a software-based tagging scheme that uses ARM PA to authenticate the tags stored in shadow memory. During every memory access, the counter values are used to authenticate the PAC value stored in the upper bits of the pointer. For temporal safety, the counter value is checked and removed on free to mitigate UAF and double-free attacks. Use-after-return protection is achieved by an additional compiler pass, transforming unsafely accessed stack variables into heap objects. PTAuth [20] introduces temporal memory safety for heap objects by instrumenting an object identifier into the allocated memory. This identifier is used for the PAC computation as the modifier, which mitigates temporal memory issues like UAF and double-free attacks. Additionally, HAKC [37] introduces software compartmentalization for code and data in the kernel using ARM PA and MTE.

9 CONCLUSION

In this paper, we presented Multi-Tag, a probabilistic memory safety scheme preventing the exploitation of spatial and temporal memory safety vulnerabilities. At its core, Multi-Tag is a hardware-software co-design based on tagged memory utilizing multi-granular tags. In particular, our design leverages a tagged memory architecture for fine-grained access control on the object level. Combined with page-granular tags stored in the PTEs, we provide flexible software-controlled security policies for memory protection. In contrast to conventional tagged architectures, page-granular tags increase the available tag space without incurring additional memory overhead or bus pressure. Additionally, page-granular tags do not increase the system's complexity.

To demonstrate the feasibility of our design, we create a prototype implementation of Multi-Tag, consisting of a modified gem5 simulator, an extension of the Linux kernel, and a compiler toolchain. Our compiler toolchain can automatically protect C-programs without user interaction. The security evaluation highlights Multi-Tag's strong security guarantees with a geometric mean performance overhead of 11.7 % for the SPEC CPU2017 benchmark suite and 1.7 % for the nbench-byte benchmarks.

ACKNOWLEDGMENTS

We thank 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 a generous gift from Intel.

REFERENCES

[1] Aingaran et al. 2015. M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro* (2015).
[2] Akritidis et al. 2008. Preventing Memory Error Exploits with WIT. In *S&P'08*.
[3] Akritidis et al. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX'09*.
[4] Iván Arce. 2004. The Shellcode Generation. *IEEE Secur. Priv.* (2004).
[5] Bernhard et al. 2022. xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In *EURO S&P'22*.
[6] Binkert et al. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* (2011).
[7] Bletsch et al. 2011. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS'11*.
[8] Hans-Juergen Boehm and Mark D. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exp.* (1988).
[9] Bucek et al. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*.
[10] Burow et al. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *AsiaCCS'18*.
[11] Burow et al. 2019. SoK: Shining Light on Shadow Stacks. In *S&P'19*.
[12] Carter et al. 1994. Hardware Support for Fast Capability-based Addressing. In *ASPLOS'94*.
[13] Castro et al. 2009. Fast byte-granularity software fault isolation. In *SOSP'09*.
[14] Intel Corporation. 2022. Intel Architecture Instruction Set Extensions and Future Features. https://www.intel.com/content/www/us/en/developer/tools/isa-extensions/overview.html. Accessed 2022-06-01.
[15] Cowan et al. 2003. PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *USENIX'03*.

[16] Devietti et al. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *ASPLOS'08*.

[17] Duck et al. 2017. Stack Bounds Protection with Low Fat Pointers. In *NDSS'17*.

[18] Durumeric et al. 2014. The Matter of Heartbleed. In *IMC'14*.

[19] Robert S. Fabry. 1974. Capability-Based Addressing. *Commun. ACM* (1974).

[20] Farkhani et al. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *USENIX'21*.

[21] Google. 2021. An update on Memory Safety in Chrome. https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html. Accessed: 2022-05-14.

[22] Richard H. Gumpertz. 1983. Combining Tags With Error Codes. In *ISCA'83*.

[23] Hohentanner et al. 2022. PACSafe: Leveraging ARM Pointer Authentication for Memory Safety in C/C++. *CoRR* (2022).

[24] Hu et al. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *S&P'16*.

[25] Jim et al. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*.

[26] Joannou et al. 2017. Efficient Tagged Memory. In *ICCD'17*.

[27] Kim et al. 2020. Hardware-based Always-On Heap Memory Safety. In *MICRO'20*.

[28] Kuznetsov et al. 2014. Code-Pointer Integrity. In *OSDI'14*.

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

[30] LeMay et al. 2021. Cryptographic Capability Computing. In *MICRO'21*.

[31] Li et al. 2022. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *CCS'22*.

[32] Liljestrand et al. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX'19*.

[33] Liljestrand et al. 2022. Color My World: Deterministic Tagging for Memory Safety. *CoRR* (2022).

[34] ARM Limited. 2019. Arm Architecture Reference Manual for Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/ea. Accessed 2022-06-01.

[35] Mashtizadeh et al. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *CCS'15*.

[36] Uwe F. Mayer. 1996. Linux/Unix nbench. https://www.math.utah.edu/~mayer/linux/bmark.html. Accessed 2022-05-23.

[37] McKee et al. 2022. Preventing Kernel Hacks with HAKCs. In *NDSS'22*.

[38] Microsoft. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. BlueHat IL 2019.

[39] Nagarakatte et al. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *PLDI'09*.

[40] Nagarakatte et al. 2010. CETS: compiler enforced temporal safety for C. In *ISMM'10*.

[41] Nagarakatte et al. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ISCA'12*.

[42] Nasahl et al. 2021. CrypTag: Thwarting Physical and Logical Memory Vulnerabilities using Cryptographically Colored Memory. In *ASIACCS'21*.

[43] Necula et al. 2002. CCured: type-safe retrofitting of legacy code. In *POPL'02*.

[44] Oleksenko et al. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *SIGMETRICS'18*.

[45] Saileshwar et al. 2022. HeapCheck: Low-cost Hardware Support for Memory Safety. *ACM Trans. Archit. Code Optim.* (2022).

[46] Schrammel et al. 2020. Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86. In *USENIX'20*.

[47] Serebryany et al. 2018. Memory Tagging and how it improves C/C++ memory safety. *CoRR* (2018).

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

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

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

[51] Song et al. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *S&P'16*.

[52] Wei Song, Alex Bradbury, and Robert Mullins. 2015. Towards general purpose tagged memory. In *Proceedings of the RISC-V Workshop*, Vol. 2015. Citeseer.

[53] Steinegger et al. 2021. SERVAS! Secure Enclaves via RISC-V Authenticryption Shield. In *ESORICS'21*.

[54] Suh et al. 2004. Secure program execution via dynamic information flow tracking. In *ASPLOS'04*.

[55] Szekeres et al. 2013. SoK: Eternal War in Memory. In *S&P'13*.

[56] Qualcomm Technologies. 2017. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf. Accessed: 2022-05-14.

[57] Watson et al. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P'15*.

[58] Woodruff et al. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA'14*.

[59] Xu et al. 2021. In-fat pointer: hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *ASPLOS'21*.

[60] Ziad et al. 2021. No-FAT: Architectural Support for Low Overhead Memory Safety Checks. In *ISCA'21*.

A ARM EVALUATION METHODOLOGY

For the ARM evaluation, we distinguish between performance and functional evaluation as there is currently no processor available that implements the ARMv8.5-A MTE feature. To validate the functional correctness, we conducted the functional evaluation on QEMU 6.2, natively supporting the ARM PA and MTE hardware features. We compiled the benchmarks with our custom toolchain and successfully executed the instrumented programs on QEMU.

A.1 M1 Performance Evaluation

To measure the performance impact of Multi-Tag on real hardware, we executed several micro- and macrobenchmarks on a 2020 Apple Mac Mini (8 GB) with an ARM-based Apple M1 processor supporting the PA hardware extension. We ported our toolchain to compile the benchmarks to Apple macOS 11.2.3. Since the M1 processor only supports the PA instructions but not ARM MTE, we approximated the performance impact of the MTE instructions. We further could not integrate the tagging of pages in the kernel for Multi-Tag, as macOS does not allow to modify the kernel. Hence, we approximated this using the normal `mprotect` system call.

ARM MTE Approximation. For the performance evaluation, we introduce a worst-case approximation of the memory tagging overhead. Memory tagging requires several steps, for which MTE uses dedicated instructions. First *(i)*, the pointer needs to be tagged with a pseudorandom tag. Here, ARM provides the `IRG` instruction to create a pseudorandom tag and store it into the register. We estimate this instruction by using a bit operation to add bits into the TBI field of the pointer. Besides pointer tagging, the corresponding memory location also needs to be tagged *(ii)*. This is done using the `STG`, `STZG`, `ST2G`, `STZ2G`, and `STGP` instructions. We model these instructions as regular store instructions to the desired memory location. Contrary, the `LDG` instruction used to load the tag *(iii)* from a memory location gets estimated using a regular load operation. `ADDG` and `SUBG` are arithmetic instructions *(iv)* responsible for pointer arithmetic operations. We approximate these instructions using typical ALU instructions used for addition and subtraction. The overhead introduced for the tag checks *(v)* on each memory access cannot be reliably estimated. However, we expect that ARM uses architectural optimizations, such as a dedicated tag cache, to minimize this overhead. Furthermore, the `RNDR` system register used by Multi-Tag to generate a pseudorandom modifier *(vi)* is not available on the Apple M1 silicon. We estimate this overhead by reading from another system register.

Microbenchmarks. To evaluate the performance impact of the PA instructions and the instructions used for our MTE approximation, we benchmarked them on the M1 processor. To measure the execution time of each instruction, we used the internal performance counters of the CPU and averaged the timing over 100 M execution runs. As shown in Table 1, the execution times for signing and authenticating pointers using the `PAC` and `AUT` instructions on the M1 are identical.

Macrobenchmarks. In addition to the microbenchmark, we compiled the nbench-byte [36] and SPEC CPU2017 [9] benchmarks with our custom toolchain and executed the instrumented binaries on the M1 processor. As the current Multi-Tag prototype only supports C programs, we excluded all non-C-based SPEC benchmarks.

Table 1: Microbenchmarks of the used architectural security features. Microbenchmarks measuring the execution time of PA and estimated MTE instructions on the M1 processor.

	Instruction		ISA	Execution time
PA	PACIA	PAC Generation	ARMv8.3-A	2.19 ns
	PACDA	PAC Generation	ARMv8.3-A	2.19 ns
	AUTIA	PAC Authentication	ARMv8.3-A	2.19 ns
	AUTDA	PAC Authentication	ARMv8.3-A	2.19 ns
MTE	IRG	Estimated as ORR	ARMv8.5-A	0.31 ns
	STG	Estimated as STR	ARMv8.5-A	1.55 ns
	LDG	Estimated as LDR	ARMv8.5-A	1.55 ns
	ADDG	Estimated as ADD	ARMv8.5-A	0.31 ns
	RNDR	Estimated as MRS	ARMv8.5-A	0.32 ns

To execute the PA instruction on macOS, the binaries are compiled for the `arm64e` ABI. Since we use the provided macOS libc implementation, the code of the C standard library remains unprotected for the performance evaluation. We used the identical set of SPEC benchmarks related work [23] also executed in their performance measurement on macOS.

Received 15 December 2022; accepted 22 March 2023