TAGShield: Persistent Tagging for Robust Stack Memory Error Protection

Michele Grisafi
University of Trento, Italy
michele.grisafi@unitn.it

Carlo Ramponi
University of Trento, Italy
carlo.ramponi@unitn.it

Silviu Vlasceanu
Huawei Research, Germany
silviu.vlasceanu@huawei.com

Mahmoud Ammar
Huawei Research, Germany
mahmoud.ammar@huawei.com

Bruno Crispo
University of Trento, Italy
bruno.crispo@unitn.it

Abstract

Stack memory errors such as buffer overflows remain a major security challenge for software written in memory unsafe languages like C and C++. Arm Memory Tagging Extension (MTE) offers a promising foundation for detecting spatial violations by associating tags with both pointers and memory objects. However, existing MTE-based runtime defenses do not preserve pointer tag integrity. In particular, pointer tags can be inadvertently corrupted or adversarially altered during pointer arithmetic operations, which are common in system-level code, thereby rendering the security guarantees of current solutions questionable.

This paper presents TAGShield, a runtime exploit mitigation mechanism that leverages MTE to provide persistent and deterministic protection against stack spatial memory errors while enforcing pointer tag integrity. TAGShield combines a transparent compile time tagging scheme with lightweight software instrumentation that checks and maintains tag correctness. Evaluation results demonstrate that TAGShield introduces a geometric mean runtime overhead of less than 10% on representative benchmarks, including SPEC CPU2006 and nginx, while effectively mitigating all stack spatial memory vulnerabilities in the Juliet C/C++ test suite.

CCS Concepts

• **Security and privacy** → **Systems security**.

Keywords

Memory Tagging, ARM MTE, Tag Integrity, Memory Safety

ACM Reference Format:
Michele Grisafi, Carlo Ramponi, Silviu Vlasceanu, Mahmoud Ammar, and Bruno Crispo. 2026. TAGShield: Persistent Tagging for Robust Stack Memory Error Protection. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '26), June 1–5, 2026, Bangalore, India.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3779208.3785279

ASIA CCS '26, Bangalore, India

ACM ISBN 979-8-4007-2356-8/26/06
https://doi.org/10.1145/3779208.3785279

1 Introduction

Memory corruption vulnerabilities, and spatial memory errors in particular, continue to dominate the landscape of modern software exploits. Recent findings from Google [1] show that spatial memory errors account for more than 40% of the vulnerabilities reported in 2024, highlighting the need for effective defenses. These errors occur when a program accesses an object's memory beyond its intended bounds, enabling powerful primitives that can be used by adversaries to manipulate program execution and launch sophisticated exploits, such as control-flow hijacking [2–7] and data-only attacks [8–12]. Stack memory errors are especially prime targets for such attacks and their exploitation continues to pose a significant challenge [13–15].

Despite the long-standing recognition of stack-based spatial memory errors, first emphasized in the Anderson report [16], they remain a critical threat in modern software security [1]. Existing mitigations have raised the bar for some exploit classes by protecting return addresses, first with stack canaries [17] and more recently with software- or hardware-enforced shadow stacks [18–22]. However, these defenses cover only a subset of stack errors and leave many stack objects and memory accesses exposed, enabling corruption of local variables and unauthorized reads and writes [10, 13]. Coarse-grained probabilistic defenses such as ASLR further complicate exploitation but are often bypassed once attackers can exploit relative layout information on the stack to disclose addresses [23, 24]. More comprehensive approaches for spatial safety exist, ranging from isolation[13, 25, 26] and randomization [27] to rewriting [28] and bounds checking [29–31], but they frequently incur substantial performance overheads, introduce compatibility issues, or provide limited security guarantees [12, 32]. The sustained efforts of major industry players, exemplified by Apple's 2023 `-fbounds-safety` LLVM proposal [33] and Google's 2024 STL hardening solution [34], highlight the importance of this problem and continue to demand attention.

Problem Statement. Recent hardware primitives offer an appealing path toward balancing security and efficiency. Among these primitives, ARM Memory Tagging Extensions (MTE) [35] is a significant step toward mitigating spatial errors. MTE associates each pointer and a memory granule with a 4-bit tag and performs a hardware check on dereference, permitting access only when the pointer and memory tags match. MTE-based defenses [36–40] typically reduce performance overhead compared to software-only bounds checking mechanisms [30, 31, 41, 42]. However, several proposals rely on hardware-generated random tags and therefore

provide only probabilistic protection, which remains susceptible to brute force guessing and tag collisions [36–38]. More recent work [39, 40] addresses these limitations by adopting a different approach that leverages deterministic and persistent memory tagging to reduce retagging overhead and improve protection against contiguous overflows. However, current MTE-based approaches share a fundamental limitation: *they do not guarantee pointer tag integrity*. In other words, existing MTE-based solutions are primarily tailored for *sanitization* rather than runtime exploit mitigation, even though some attempt to address the latter [39, 40]. This limitation stems primarily from failing to preserve pointer tag integrity, leaving the unused upper bits of pointers, where part of the tag is stored, vulnerable to manipulation via malicious pointer arithmetic. Considering the prevalence of pointer forgery and substitution attacks, many of which stem directly from such arithmetic manipulations, this limitation poses a significant barrier to the adoption of these solutions, as the gap remains substantial [43, 44].

Contribution. In this paper, we present `TAGShield`, a runtime exploit mitigation mechanism designed to defend against stack-based spatial memory errors. `TAGShield` combines memory tagging with selective software instrumentation to address the limitations of prior techniques, providing deterministic protection while preserving pointer tag integrity.[1] This approach mitigates pointer corruption attacks, including those that rely on arithmetic manipulation, and strengthens protection against stack-based spatial memory errors. The design of `TAGShield` is guided by two key principles:

- **Transparent Memory Coloring Scheme:** `TAGShield` employs an efficient memory coloring mechanism to selectively tag unsafe pointers and objects, ensuring spatial isolation and preventing tag collisions between adjacent regions.
- **Lightweight Software Instrumentation:** `TAGShield` introduces a lightweight software layer that persistently enforces pointer tag integrity, making tags effectively immutable and resistant to corruption. This layer thwarts adversarial attempts to manipulate tags.

`TAGShield` balances performance and security through a selective protection strategy that focuses on vulnerable stack allocations. It also leverages compile-time tag generation to reduce runtime overhead while maintaining strong security guarantees. Notably, `TAGShield` introduces no compatibility issues, minimizing barriers to adoption in real-world applications that can tolerate its modest overhead budget, that is, less than 10% runtime overhead (geometric mean). The source code of `TAGShield` is available at [45].

2 Background & Related Work

Section 2.2 and Section 2.3 also serve as motivation for proposing `TAGShield`.

[1] In this paper, "deterministic" means that tags are assigned according to a fixed compile-time pattern (e.g., cycling through the 4-bit space per function) rather than at random; "persistent" means that the authentic tag is maintained and enforced across pointer operations for the lifetime of the allocation, preventing tag manipulation via malicious arithmetic.

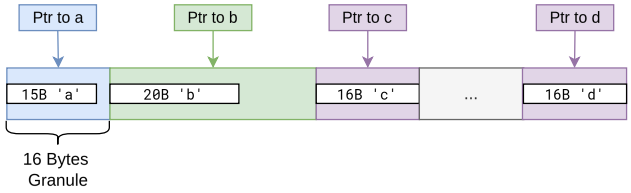

Figure 1: Illustration of MTE. Four memory allocations and four pointers, with two of them sharing the same color (tag).

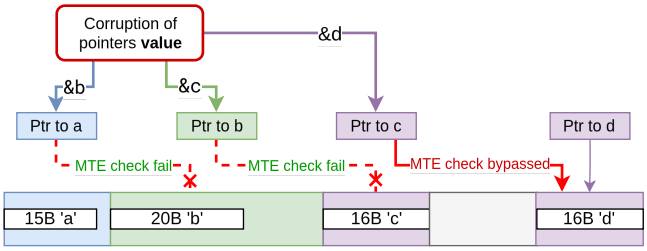

Figure 2: Corruption of pointers with the address of other memory areas (&c = *address of 'c'*). MTE detects the corruption if and only if there is a color mismatch.

2.1 ARM Memory Tagging Extension (MTE)

MTE is a hardware-based implementation of tagged memory in recent ARM architectures (v8.5 and later). It introduces two types of 4-bit tags (also known as colors): address tags, which are stored at the top of each pointer utilizing the Top-Byte Ignore (TBI) feature, and memory tags, which are associated with each aligned 16-byte region of application memory (memory granule).

MTE operates on a key-lock principle, where each pointer is tagged with a color (key) corresponding to the tagged (locked) memory it references, as shown in Figure 1. To realize MTE, ARM extends the Instruction Set Architecture (ISA) with new instructions that assign tags to both memory and pointers. During execution, the MTE-enabled CPU detects and prevents any color mismatches, such as when a pointer attempts to access memory tagged with a different color, as shown in Figure 2. This mechanism supports both synchronous and asynchronous error detection modes.

2.2 Stack Spatial Memory Errors

A spatial memory error occurs when a program accesses memory outside the bounds of the *intended* referent, violating one of the memory safety principles. Such errors are prevalent in low-level programming languages like C and C++, where developers have direct control over memory management. The intended referent of a pointer is the object from which the pointer's base address was derived, as defined by Austin et al. [46]. When a pointer accesses memory outside this defined boundary, it can lead to unauthorized reads or writes, commonly known as buffer overread and buffer overflow, respectively.

Spatial memory errors have been a well-documented issue in system security for more than five decades [16]. Despite this long-standing awareness, these errors remain a significant challenge to detect and mitigate effectively. Automated static analyzes often fail to catch subtle spatial violations due to the complex control flow

and data structures in modern software systems [47]. Similarly, dynamic analysis techniques such as fuzzing may uncover only those spatial memory errors that occur along specific execution paths, leaving many vulnerabilities latent in the codebase [48].

A notable subset of spatial memory errors involves stack-based buffer overflows, where a program writes more data to a stack buffer than it can hold, corrupting adjacent memory. These errors can be particularly dangerous, as they often lead to control-flow hijacking attacks [2], where an attacker manipulates the corrupted stack to execute arbitrary code. Out-of-bounds (OoB) writes, which mainly include stack buffer overflows, rank second in the CWE Top 25 for 2024, following cross-site scripting (XSS) and preceding SQL injection [49]. This list, based on high-severity vulnerabilities reported between mid-2023 and mid-2024, highlights the critical impact of buffer overflow vulnerabilities on software security. For instance, as reported by Rahul et al. [15], researchers have identified over 200 stack buffer overflow vulnerabilities in the past three years, with 122 of these classified as high-severity, scoring 7 or above on the Common Vulnerability Scoring System (CVSS) scale.

Recent examples of stack buffer overflow vulnerabilities further highlight the pervasiveness of spatial errors across different software domains. For instance, CVE-2023-21610 affected Adobe Acrobat Reader, allowing an attacker to execute arbitrary code through a specially crafted PDF file, leading to a stack-based buffer overflow. Similarly, CVE-2023-22243 impacted Adobe Animate, where a stack buffer overflow could be exploited to achieve remote code execution. In the realm of cryptographic software, CVE-2022-3786 was a vulnerability in OpenSSL, one of the most widely used libraries for secure communication. This flaw allowed a buffer overflow in the X.509 certificate verification process, which adversaries could exploit to cause a denial of service or execute arbitrary code. In the Linux kernel, CVE-2024-1151 exposed a vulnerability in the Open vSwitch (OVS) component, where recursive operations could lead to a stack overflow, potentially causing system crashes or other denial-of-service conditions. Another recent issue, CVE-2024-24861, was identified in the Xceive XC4000 silicon tuner device driver, where an integer overflow could trigger a stack-based buffer overflow, leading to system instability and potential crashes.

2.3 Spatial Memory Error Defenses

Defending against spatial memory errors has been a central focus of extensive research, resulting in a variety of bounds-checking techniques that can be broadly categorized into tripwire, pointer-based, and object-based approaches.

Tripwire defenses, such as AddressSanitizer [50] and RangeSanitizer [51], employ guard blocks (e.g., red-zones or invalid memory regions) around memory objects to detect OoB accesses. These techniques are particularly effective against contiguous overflow vulnerabilities. However, they are incomplete because they cannot detect cases where pointers go out of bounds but land in another valid allocation. This incompleteness, combined with high performance overhead, restricts their practical use to debugging and sanitization scenarios.

Pointer-based defenses, such as CCured [52] and Cyclone [53], use fat pointers—pointers augmented with metadata—to track bounds information for each pointer, enabling precise checks on dereference. However, because fat pointers alter the fundamental representation of pointers by transforming them from simple addresses into composite structures (e.g., including base address and size), they face significant compatibility issues with uninstrumented libraries [30, 31, 54]. Additionally, type casts can corrupt propagated metadata, further complicating their application. Subsequent work prioritized memory-layout compatibility by decoupling metadata from the pointer representation and storing it in a disjoint metadata space (e.g., a shadow bounds table), as in software-based mechanisms such as SoftBound [29] and hardware-oriented solutions such as Intel Memory Protection Extensions (MPX) [55]. While these approaches offer better compatibility than fat-pointer-based ones, they still fall short when instrumented pointers interact with uninstrumented code that modifies them without tracking metadata updates [54]. This requires either additional wrappers to maintain consistent metadata updates or recompilation of code that may exist only in binary form or is otherwise inaccessible for direct modification.

Object-based approaches, such as Baggy Bounds [30] and other follow-up proposals [56–58], mitigate some of these limitations by associating bounds information directly with objects rather than pointers. These techniques operate under the assumption that all pointers are correctly derived from their intended referents and therefore enforce checks during pointer arithmetic rather than on memory accesses (i.e., pointer dereferences). While this design resolves memory-layout compatibility issues that hindered early pointer-based approaches [52, 53], it can introduce false positives in pointer-arithmetic checks. For instance, a pointer may be flagged as OoB during arithmetic operations even though it remains valid according to the language specification (e.g., the C standard) and is never dereferenced, as in common programming constructs such as for-loops [57, 59].

Another limitation of certain object-based approaches is their inability to detect overflows that span across objects. When a pointer derived from one object is manipulated through arithmetic and ends up within the bounds of a different valid object, the access may be incorrectly considered legitimate. This issue is especially relevant in designs that rely on low-fat pointers [31, 60], which encode broad allocation boundaries instead of precise object-level bounds. Unlike pure pointer-based approaches [29, 52], these techniques do not guarantee the retrieval of accurate bounds information, potentially allowing some spatial memory violations to remain undetected.

To enhance performance, OptiSAN [15] employs a hybrid design that dynamically selects between tripwire defenses, such as AddressSanitizer [50], and object-based bounds-checking mechanisms such as Baggy Bounds [30], based on the specific scenario. This selective approach enables OptiSAN to balance performance and security by choosing the most efficient method for detecting stack memory errors in different contexts. However, its primary focus is on optimizing performance, leaving other limitations, such as false positives for OoB pointers, largely unaddressed.

More recently, mitigation techniques that leverage the Pointer Authentication (PA) feature introduced in the ARMv8.3 architecture have been proposed to provide both spatial and temporal safety guarantees. PACMem [61] and CryptSan [62] utilize cryptographic

pointer authentication codes (PACs) to ensure memory safety without fat pointers or intrusive metadata management. Despite their promise, these techniques still encounter performance and compatibility challenges, particularly when interacting with legacy code and uninstrumented libraries, which may hinder their adoption.

MTE-based solutions. The advent of ARM's Memory Tagging Extensions (MTE) provides a promising hardware building block for improving the compatibility and performance of spatial memory error defenses [35]. Unlike traditional software-based tagging defenses [42], MTE leverages hardware support to perform these checks, significantly reducing runtime overhead. In principle, native MTE enablement, which relies on assigning random tags via the `IRG` instruction, provides only probabilistic security guarantees [36–38]. This approach is vulnerable to high collision rates due to the limited entropy of the tag space, potentially allowing adversaries to bypass the mechanism. As a result, most MTE-based defenses are proposed primarily as *sanitizers*, where the focus is on detecting errors in the pre-deployment phase rather than providing runtime security guarantees.

Some solutions have attempted to mitigate these issues by proposing exploit mitigation techniques. For instance, ZomeTag [40] introduces a two-layer approach to deterministically assign unique tags to each object by dividing memory into non-overlapping zones using a tripwire mechanism. Each zone can contain up to 16 objects, each with a unique tag. To further reduce performance overhead, StickyTags [39] proposes persistent tagging, where memory retagging is not required each time an object is created. Instead, stack and heap memory layouts are organized into per-size-class regions with fixed tags. Despite these advancements, these runtime defenses still operate in a *sanitization* mode and lack tag integrity, meaning that tags can be manipulated at runtime through pointer arithmetic operations without detection. This limitation creates a significant gap that adversaries can exploit to bypass these defenses.

3 TAGShield

Overview. `TAGShield` is a runtime exploit mitigation mechanism designed to address stack-based spatial memory errors by combining memory tagging with selective software instrumentation. It uses an efficient memory coloring scheme to tag unsafe objects, preventing adjacent memory collisions and isolating vulnerable regions. `TAGShield` also ensures the integrity of pointer tags through lightweight software enforcement, making them resistant to manipulation from malicious pointer arithmetic operations.

3.1 Adversary Model

We assume a powerful software-based adversary capable of exploiting any stack-based spatial memory error to mount a wide range of attacks, including control-flow hijacking and data-only attacks. The adversary has full control over pointer manipulation and can perform brute-force attacks to modify pointer contents, targeting arbitrary memory locations.

The adversary is assumed to have access to the target application's source code, binary, and compiler information of the target application, enabling detailed analysis of the target's memory layout and behavior. However, the adversary cannot tamper with the source code, binary, or compiler prior to deploying the target binary. Additionally, the adversary may attempt speculative execution attacks to infer memory tags [39], but cannot directly modify MTE tags or alter the MTE configuration, as the adversary operates in user mode and lacks the necessary privileges.

While `TAGShield` can be extended to address heap-based spatial memory vulnerabilities, such scenarios are considered out of scope for this work and are only briefly discussed in Section 6.2. Similarly, temporal memory vulnerabilities are excluded from this threat model, as they are addressed by orthogonal defenses specifically designed for such threats [60, 63–65].

3.2 Isolation of Safe Objects

Although MTE tag checks are relatively low-cost due to their hardware-based implementation, tagging memory can still introduce non-negligible overhead [39, 66]. To mitigate this, `TAGShield` selectively instruments only *unsafe* objects. Specifically, it leverages compile-time static analysis to classify stack allocations into safety categories. Objects and their associated pointers deemed safe are efficiently isolated by retaining the default tag, typically `0x0`. In contrast, *unsafe* objects and their associated pointers are assigned a non-zero tag chosen from the remaining values in the 4-bit tag space. `TAGShield` not only ensures that *unsafe* allocations are fully isolated from safe ones by preventing the forgery of the default tag (`0x0`), but also guarantees that *unsafe* allocations cannot forge each other's tags by protecting their tags' integrity.

We define an allocation as memory-safe if all pointer dereferences based on that allocation are either safe or cause the application to terminate. Specifically, we adopt the approach of Volodymyr et al. [25], which considers an allocation safe if it is accessed exclusively through safe dereferences, i.e., dereferences that only access the memory on which their pointer is based. Conversely, an allocation is *unsafe* if it is accessed at least once by an unsafe dereference. For instance, if an object on the stack is accessed exclusively by compiler computed addresses (e.g., Stack Pointer (SP)-based), without ever storing its address on memory, then such an object is classified as safe. To classify objects, `TAGShield` integrates LLVM's `StackSafetyAnalysis` [67], a static analysis pass that determines whether stack allocations are accessed exclusively through safe pointer dereferences. This analysis enables `TAGShield` to distinguish between safe and *unsafe* stack allocations at compile-time, ensuring that only potentially exploitable allocations receive non-zero tags. Intuitively, if all allocations in a program are safe, the program itself is memory-safe.

3.3 Deterministic Tagging of Unsafe Objects

The core principle behind `TAGShield`'s coloring scheme is simple: whenever a function is invoked, we explicitly assign a unique tag to each of its stack allocations and propagate this tag to the corresponding pointer. A known limitation of MTE is its low tag entropy (4-bit), which makes it impractical to assign a truly unique tag to every *unsafe* allocation. Unlike many existing memory tagging implementations [38, 68, 69], we do not rely on random tags. While randomness can deter trivial attacks, it provides no security advantage under a stronger threat model (Section 3.1) where an attacker can read memory tags. Instead, we use statically computed

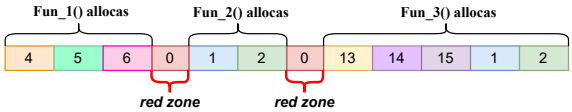

Figure 3: TAGShield 's coloring scheme for allocations. Three nested functions cycle through the available colors, starting from a random tag. A 0-tagged red-zone is inserted between functions.

tags determined at compile time and hard-code them within the application. Figure 3 illustrates TAGShield's coloring scheme.

In the function prologue, we assign tags to all *unsafe* allocations by cycling through the available tag values, ensuring that adjacent allocations receive different tags. To accommodate for the 16 bytes granule of MTE, every allocation is padded to a multiple of 16, thus ensuring each allocation has its own color. The first tag in each function is chosen randomly from the set of non-zero tags, after which subsequent allocations are assigned tags by cycling through the remaining values, re-using previous ones if necessary. Finally, these tags are cleared in the function epilogue when the corresponding stack frame is removed. To further reduce the risk of tag collisions across function boundaries, TAGShield's instrumentation pass inserts a 16-byte padding granule (i.e., a red zone) after each stack frame. This prevents the first tag assigned in a new stack frame from inadvertently matching the last tag of the previous frame. This red zone is assigned the default tag `0x0`, ensuring that no unsafe pointer can access it.

While alternating tag values mitigate contiguous (i.e., linear) memory overflows, they do not protect against non-contiguous (i.e., non-linear) overflows, which often arise from pointer arithmetic — a common pattern in real-world applications [43, 44]. An adversary can exploit this limitation to manipulate tags to access unintended memory objects without detection. Addressing this issue is a key objective of TAGShield, as detailed in Section 3.4.

Furthermore, due to the low entropy of tag values (only 4 bits), multiple live allocations may share the same tag (or color), either across different functions or within the same stack frame. This could allow a vulnerable pointer to access other allocations (memory objects) with matching tags if pointer arithmetic modifies only the pointer's address while leaving the tag intact. Tag collisions are an inherent limitation of memory tagging approaches, and since MTE does not inherently bind tag values to specific addresses, this issue remains unresolved in all pure MTE-based approaches, including TAGShield. One potential mitigation is to enforce checks on pointer arithmetic, as in object-based bounds-checking mechanisms [30]; however, doing so would increase runtime overhead. Given that TAGShield primarily targets *unsafe* allocations, the risk of exploiting non-adjacent tag collisions is already reduced. We therefore leave fully addressing non-linear buffer overflow attacks that exploit identical tags to future work.

3.3.1 Selective Tag Checks. To further improve the performance of TAGShield, we optimize memory tagging by differentiating between types of memory accesses rather than applying tagging uniformly to all *unsafe* allocations. Stack memory can be accessed in

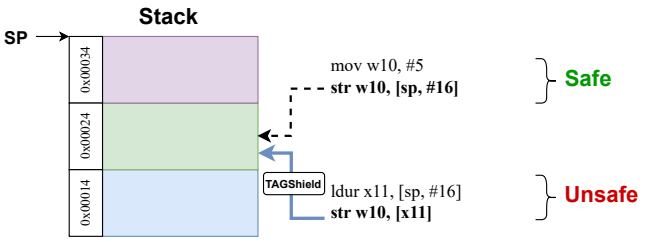

Figure 4: Two types of memory access in the AArch64 architecture: the first relies on stack pointer addresses and is therefore safe, whereas the second uses generic pointer addresses, is unsafe, and is protected by TAGShield.

two ways: via stack pointer (SP) or other pointers. SP-based accesses use the stack pointer with a static offset to reference specific memory locations. Given the adversary model described in Section 3.1, instructions using SP-based accesses are considered *write-safe* because they directly target fixed memory locations, leaving no opportunity for an adversary to redirect them to unintended addresses. In contrast, pointer-based accesses rely on pointers stored in memory, making them inherently more susceptible to exploitation. While pointers enhance program flexibility by enabling operations such as pointer arithmetic and dynamic array accesses, they also introduce security risks. An adversary could exploit a memory vulnerability to manipulate a pointer stored on the stack, causing subsequent pointer-based accesses to reference unintended memory locations. Therefore, pointer-based accesses are classified as *write-unsafe*: an adversary who gains control over a pointer could leverage it to corrupt arbitrary memory locations. Since SP-based accesses cannot be hijacked to target unintended locations, TAGShield focuses exclusively on protecting pointer-based accesses to *unsafe* allocations. Figure 4 illustrates these two types of memory access.

3.4 Enforcing Integrity of Tags

In the case of MTE, a corrupted pointer has a theoretical probability of $^{14}/_{15}$ of triggering a tag mismatch, with a $^{1}/_{15}$ chance of going undetected. To mitigate tag collisions, several works [38–40] alternate tag values in memory, ensuring that contiguous overflows trigger a tag mismatch. However, this approach does not fully eliminate tag collisions. Non-contiguous buffer overflows can still target distant memory locations that share the same tag, without detection.

In practice, pointer arithmetic, which is common in programming constructs, not only allows adversaries to manipulate addresses to target allocations with matching tags (as shown in Figure 5) but also enables them to forge tags themselves, as MTE does not protect tags. For instance, consider a scenario where an adversary gains full control over a pointer, `'a'`, which points to `A[1]`. [2] If the adversary's goal is to overwrite the memory locations `B[2]` and `C[1]`, she can do so by injecting the target address into `'a'` and modifying its tag to match the target location's tag. This behavior is illustrated in Figure 5, which also shows that, when the target location already carries the same tag, modifying the address alone

[2]We use capitalized letters to indicate memory allocations and square brackets to indicate the tag of the allocation.

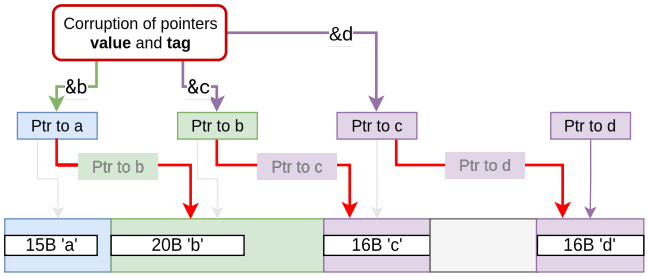

Figure 5: Bypassing MTE by corrupting both the pointer tag and the pointer value (no TAGShield).

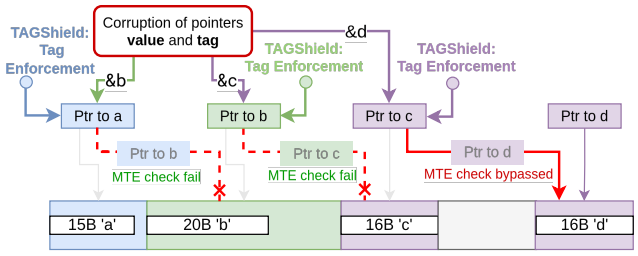

Figure 6: Adversary corrupting both the pointer tag and value with TAGShield enabled. TAGShield prevents the bypass when the corrupted tag mismatches the authentic tag.

suffices. This situation arises, for example, when a pointer 'c' is maliciously altered to target a location referenced by pointer 'd', which shares the same color. As a result, the adversary obtains an arbitrary memory read and write primitive, thereby bypassing MTE checks entirely.

As mentioned earlier, defending against the latter case, where only pointer addresses are manipulated to target allocations with matching tags, is beyond the capability of any MTE-based approach, given the low entropy of 4 bits. Therefore, TAGShield focuses on the former case, which is largely overlooked by existing approaches. Mitigating this issue eliminates a significant attack surface. To achieve this, TAGShield enforces that a pointer can only be dereferenced if its tag matches the authentic tag that is assigned at compile-time. Figure 6 illustrates the tag integrity enforcement mechanism of TAGShield.

Mechanism. Upon accessing a tagged pointer, TAGShield ensures that its tag is authentic. Enforcing tag integrity, however, is not always straightforward and depends on the type of pointer. In particular, we distinguish between two types of pointers: *root level* and *non-root level*. A pointer is considered root level when it directly references a stack allocation without any prior dereference. In LLVM IR, where TAGShield's passes are implemented, every `alloca` instruction generates a root level pointer (e.g., `%a` and `%ptr` in Listing 3.1). Conversely, non-root level pointers are derived by pivoting from another root or non-root level pointer. In C/C++, this is typically expressed through pointer dereferencing, forming a chain of memory accesses. For instance, the operation `*(ptr)` consists of two memory accesses:

```
1 //Root level pointers
2 %a = alloca i32, align 4
3 %ptr = alloca ptr, align 4
4
5 //Non-root level pointers
6 %0 = load ptr, ptr %ptr, align 4
```

Listing 3.1: LLVM IR encoding of the C instructions 'int a, *ptr;', showing an example of root level and non-root level pointers.

```
1 call @enforceStaticTag(%a, #tag)
2 store #40, ptr %a, align 16
```

Listing 3.2: Enforcement of a root level tag for a variable, e.g., %a = alloca i32. The call to enforceStaticTag is an intrinsic, i.e., an inlined series of assembly instructions.

- The first access involves the root level pointer, `*(ptr)`, namely `%ptr` in Listing 3.1.
- The second access pivots from the root pointer `%ptr` by dereferencing a non-root level pointer `*(ptr)`, which corresponds to `%0` in Listing 3.1.

Notably, all non-root level accesses must originate from root level accesses: first, the pointer value (root level) is loaded, and only then it is dereferenced (non-root level).

For every unsafe `alloca` instruction that generates a root level pointer, TAGShield's plugin in the compiler toolchain generates and embeds a distinct tag. The integrity of this tag is safeguarded by hard-coding it in the preamble of the corresponding stack frame and enforcing it with every access, as shown in Listing 3.2. In other words, whenever the pointer is accessed, we reset its tag to the authentic one computed at compile-time, ensuring that it remains persistent throughout the pointer's lifecycle.

Non-root pointers, however, introduce additional challenges since their authentic tags may not be known at compile-time. Consider the code listing *App Instructions* in Figure 7, where three variables (`int x`, `int y`, `int * ptr`) are statically tagged with red, blue and green, respectively. Accesses to the root level pointer `ptr`, such as ❶ and ❸, are handled statically by enforcing its green tag. However, determining the correct tag for the non-root pointer `*(ptr)` at ❹ is non-trivial. Since `*(ptr)` may point to either `x` or `y`, each with a different tag, the appropriate tag must be dynamically determined. In general, when a pointer is updated to reference a new variable, its tag must match that of the referenced variable at the point of dereference.

Interestingly, since MTE tags are embedded in pointers without integrity guarantees, altering a pointer's content automatically updates its tag to match the source pointer. Nevertheless, TAGShield must track these tag changes to ensure tag integrity at runtime. Specifically, at any point during execution, TAGShield must determine the correct tag for non-root level pointers. We refer to the collection of all non-root level pointer tags as TAGShield *state*. TAGShield maintains this state using one reserved register

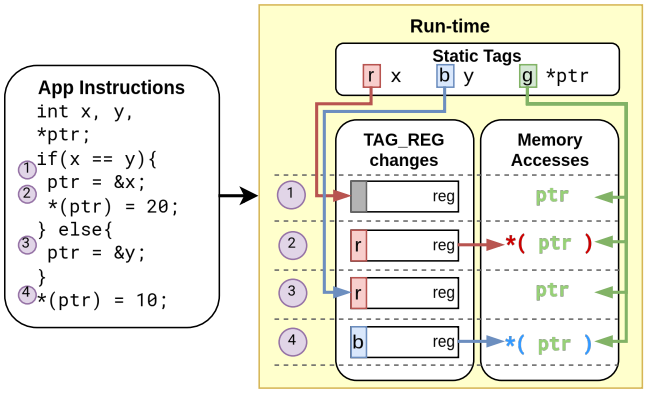

Figure 7: Representation of how pointer accesses are protected by TAGShield. When a pointer content is changed, the source tag is updated in TAG_REG. Each root level access is then protected with the static tag, while each non-root access is protected with the tag saved in TAG_REG.

(TAG_REG) within its software instrumentation layer. For each non-root pointer in a function, TAGShield reserves 4 bits in TAG_REG. For instance, given the variables `int a, *pa, **ppa`, TAGShield allocates 4 bits each for `*(pa)`, `*(ppa)` and `*(*(ppa))`. Whenever a non-root pointer is assigned a new address, its corresponding TAG_REG entry is updated with the tag of the source pointer. Since AArch64 registers are 64-bit wide, TAGShield can encode up to 16 non-root pointers per function. Our analysis (see Section 5) shows that this is sufficient for many real-world applications, as functions rarely require tracking more than 16 non-root pointers. In cases where a function makes a call to another function that also utilizes TAG_REG, the register is spilled to memory in the function prologue of the callee and restored in the epilogue, ensuring consistent tracking across function boundaries. In the code snippet in Figure 7, TAGShield reserves 4 bits to track the tag of `*(ptr)` and updates this tag at lines ❶ and ❸. The effects of each update are reflected in the subsequent lines. Listing 3.3 illustrates the TAG_REG update mechanism at the LLVM IR level. By tracking non-root pointer tags, TAGShield instruments each non-root dereference to enforce the authentic (aka correct) tag, retrieving it from TAG_REG as shown in Listing 3.4. During compilation, TAGShield determines the location of each non-root pointer's tag in TAG_REG, enabling efficient runtime enforcement.

3.4.1 Optional Tag Forgery Detection. Section 3.4 described how TAGShield tracks various tags at runtime and ensures their persistence, thereby providing lightweight tag integrity. While this design prevents corrupted pointers from being misused, it does not explicitly detect instances of tag corruption. For instance, if an adversary forges a tag, TAGShield will restore it to its authentic value without logging the forgery attempt.

Although modifying a pointer and its tag will typically trigger an MTE tag mismatch exception, this exception does not distinguish whether the corruption originated from the tag itself or the pointer's content. To enable tag forgery detection rather than prevention, TAGShield could be extended to compare pointers' tags against those stored in its state. Any mismatch could then trigger a

```
call @enforceDynamicTag(%ptr, #reg_index)
store %y, ptr %ptr, align 16
call @updateState(#tag_y, #reg_index_ptr)
```

Listing 3.3: TAG_REG update mechanism after a store instruction on a non-root pointer.

```
call @enforceStaticTag(%ptr, #tag)
%0 = load ptr, ptr %ptr
call @enforceDynamicTag(%0, #reg_index)
store #40, ptr %0, align 16
```

Listing 3.4: Enforcement of tags for a pointer dereference. The first root level access to the variable is protected with a static tag. The second non-root access is protected by enforcing the tag saved on TAG_REG.

dedicated exception, explicitly signaling tag forgery. However, this extension would introduce additional instrumentation overhead, potentially affecting performance. Thus, we leave it for future work.

3.5 Inter Functions operations

Tags are inherently local to a function and should remain active throughout its execution, i.e., as long as the function retains a stack frame in the stack. Accordingly, TAGShield assigns static tags to both pointers and memory in the function prologue and removes them in the epilogue. This ensures the consistency of memory protection for the duration of the function's execution. Likewise, the TAGShield state must be preserved throughout the function's lifetime to allow subroutines to manage their own independent TAGShield states without interfering with the caller's state.

To achieve this, TAGShield employs a state preservation mechanism. In particular, it spills the content of TAG_REG to the stack in the callee's prologue, treating it as a safe allocation tagged with the default tag (`0x0`) to ensure protection. This safeguards the caller function's TAGShield state before its overwritten by the callee. The state is then restored when the callee returns. Crucially, this backup and restore process is performed only when necessary, specifically, when the invoked function requires maintaining a TAGShield state, which typically occurs when the callee function contains unsafe dereferences that necessitate memory protection.

4 Implementation

To implement TAGShield, we leverage LLVM 18 [70], a widely used compiler toolchain in both industry and academia, known for its modularity and extensibility. TAGShield's implementation consists of two major LLVM components: a front-end pass and a back-end module. The front-end pass operates at the function-level of LLVM IR of a target application to detect pointer aliases and instrument it with custom intrinsics: special instructions that manage both static and dynamic tags at runtime. The front-end is implemented using LLVM's new pass manager and consists of approximately 1500 lines of code (LoC). To perform various analyses, our pass builds on several existing LLVM passes, including StackSafetyAnalysisPass,

AAResultsWrapperPass, StackSafetyGlobalInfoWrapperPass, TargetLibraryInfo-WrapperPass, and MemoryDependenceWrapperPass.

Although LLVM IR is architecture-independent, TAGShield's implementation is tailored for AArch64, requiring architecture-specific instrumentation. The IR generated by the front-end must be translated into AArch64 assembly, which is handled by TAGShield 's back-end module. This module extends the AArch64 LLVM back-end with custom lowering mechanisms, converting TAGShield 's intrinsics into efficient AArch64 assembly instructions. While our implementation includes several optimizations to generate performance-efficient instruction sequences, most intrinsics are lowered using just two key AArch64 instructions: movk and bfm. The former only updates 16-bit segments of a register, while the latter moves arbitrary bit subsequences from one register to another. Listing 4.1 illustrates how TAGShield enforces both static and dynamic tags at the assembly level. The back-end module comprises approximately 700 LoC.

```
1 ldr x8, [sp, #8] ; root-level address
2 movk x8, #tag, 56 ; set tag
3 ldr x9, [x8]  ; first-level address
4 bfm TAG_REG, x10, #immr, #imms
5 bfm x10, x9, #immr, #imms
6 str x11, [x9] ; use the pointer
```

Listing 4.1: Enforcement of tags for non-root-level dereferences. The number of bfm operations and their parameters vary depending on where the tag is stored within TAG_REG.

5 Evaluation

Although ARM introduced MTE in 2019, as of this writing, only a few CPUs and devices support it. While emulating the MTE extension is common practice [37, 40, 71–74], we evaluated TAGShield on real MTE-capable hardware to obtain a more accurate performance assessment. Specifically, we tested TAGShield on a Google Pixel 8 that contains 12GB of RAM and features the Google Tensor G3 SoC with 4 ARM Cortex-A715 and 4 ARM Cortex-A510 cores, all supporting MTE. The tests were conducted on a rooted device within a Docker container running Gentoo Linux [75], on a single core set to maximum frequency. We chose Gentoo over native Android because it fully supports the standard C library, which we used in its original form without any TAGShield instrumentation.

Benchmarks Selection. Our benchmark selection was primarily influenced by the capabilities of the Google Pixel 8. Among widely used academic benchmarks, we chose NBench-Byte [76] and SPEC CPU 2006 [77] because their memory requirements remain under 12GB, making them suitable for our test environment. We excluded SPEC CPU 2017 due to its higher RAM demands.

To assess TAGShield 's impact on real-world applications, we also benchmarked nginx, one of the most widely used web servers, powering 33.8% of known websites [78]. Additionally, we evaluated the impact of TAGShield on cryptographic operations using a standalone SHA-512 crypto engine [79] and an OP-TEE [80] AES Trusted Application.

All benchmarks were compiled with the -O3 optimization flag. MTE was configured in synchronization mode, ensuring that each run triggered an exception on any tag mismatch. Performance comparisons were made between the original binaries and their TAGShield-protected counterparts. Reported results represent the median of 10 different test runs. For SPEC CPU 2006, we used the reference dataset. Three applications from the SPEC CPU2006 benchmark suite, namely 400.perlbench, 453.povray, and 471.omnetpp, are excluded from the evaluation because they fail due to a bug in one of the leveraged built-in LLVM passes, which prevents correct untagging of stack frames under certain conditions. Further details are provided in [45].

5.1 Performance evaluation

Memory Overhead. Table 1 shows the impact of TAGShield on code size (".text" section), data size (".data" section), and the number of objects identified as unsafe across the selected benchmarks. On average 97.6% of objects in SPEC CPU2006 were classified as unsafe and tagged with non-zero values when applying TAGShield. Similarly, nginx reported 96.1% of unsafe objects, while all other benchmarks and test applications contained only unsafe objects. Notably, the -O3 optimization is the main responsible for the low number of safe objects, with SPEC CPU2006 having only 23% of unsafe objects when compiled with -O0.

Regarding code size overhead, which depends on the number of instrumentation instructions added, TAGShield incurred an average overhead of 5.84% for SPEC CPU2006. The benchmark 403.gcc reported the smallest increase (1.31%), while 470.lbm showed the largest (18.9%). Similarly, the other benchmarks and test applications remained below 5%.

The data memory overhead in TAGShield is mostly attributed to the additional symbols introduced by the MTE configuration routine. Larger applications, which rely heavily on the stack, tend to experience lower relative overhead, as the additional data introduced by TAGShield is less pronounced compared to applications with fewer function calls. On average, TAGShield incurred a 3.34% increase in data memory usage in SPEC CPU2006. The highest overhead was observed in 470.lbm at 10.59%, while the lowest was in 445.gobk which reported 0.003%. The overhead is negligible also on the rest of the benchmarks, with TAGShield incurring 2.61% overhead on NBench-Byte and less than 1% on the test applications.

Runtime Overhead. Figure 8 illustrates the runtime overhead introduced by TAGShield in various benchmarks. In NBench-Byte, TAGShield incurs minimal overhead, with a geometric mean of 0.46%. Notably, nearly all tests exhibit negligible performance degradation (less than 1%), except for Huffman coding, which experiences an overhead of 5.93%.

For SPEC CPU2006, the geometric mean overhead across the evaluated C and C++ applications is 9.86%, with the lowest overhead observed in 447.dealII (1.39%) and the highest in 458.sjeng (123.73%). The variation in overhead primarily depends on the frequency and necessity of tagging and retagging objects during execution. While the impact of TAGShield on benchmarks is not uniform — showing minimal overhead in some cases and more significant slowdowns in others — benchmark results often tend to highlight edge cases. To better assess TAGShield 's suitability in real-world environments, we evaluated its impact on other applications.

Table 1: Statistics about the various benchmarks and real-world applications that are tested with TAGShield.

	Benchmarks	Code memory size (original)	Code memory size (TAGShield)	Overhead %	Data memory size (original)	Data memory size (TAGShield)	Overhead %	Total objects	Unsafe objects	Ratio %
SPEC CPU 2006	401.bzip2	75.5 KB	78.8 KB	4.26%	3.73 KB	3.81 KB	2.09%	49	48	98.0%
	403.gcc	3360 KB	3404 KB	1.31%	158.86 KB	158.92 KB	0.04%	1594	1424	89.3%
	429.mcf	12.9 KB	14.9 KB	15.7%	704 B	768 B	9.09%	13	13	100%
	433.milc	135 KB	143 KB	5.88%	1.79 KB	1.86 KB	3.48%	206	206	100%
	444.namd	202 KB	216 KB	6.99%	1.10 KB	1.16 KB	4.96%	378	378	100%
	445.gobmk	2063 KB	2098 KB	1.70%	1942 KB	1942 KB	0.00%	1078	1059	98.2%
	447.dealII	3244 KB	3427 KB	5.63%	65.7 KB	65.8 KB	0.14%	6858	6595	96.2%
	450.soplex	354 KB	367 KB	3.47%	8.74 KB	8.84 KB	1.07%	561	551	98.2%
	456.hmmer	293 KB	303 KB	3.50%	4.72 KB	4.76 KB	0.83%	299	275	92.0%
	458.sjeng	141 KB	147 KB	4.16%	3.09 KB	3.15 KB	1.77%	123	123	100%
	462.libquantum	39.6 KB	42.1 KB	6.21%	764 B	828 B	8.38%	33	31	93.9%
	464.h264ref	715 KB	728 KB	1.70%	12.41 KB	12.48 KB	0.57%	217	217	100%
	470.lbm	11.7 KB	13.9 KB	18.9%	680 B	752 B	10.59%	17	17	100%
	473.astar	41.0 KB	44.5 KB	8.52%	908 B	988 B	8.81%	51	51	100%
	482.sphinx3	192 KB	199 KB	3.29%	3.43 KB	3.49 KB	1.59%	165	160	97.0%
	483.xalancbmk	4226 KB	4319 KB	2.21%	280.88 KB	280.96 KB	0.03%	5268	5173	98.2%
	average			**5.84%**			**3.34%**			**97.6%**
	NBench-byte	61 KB	64.4 KB	5.55%	2144 B	2200 B	2.61%	59	59	100%
	nginx	474 KB	483 KB	2.02%	34.00 KB	34.09 KB	0.25%	380	365	96.1%
	SHA2-512	9.15 KB	9.30 KB	1.71%	660 B	660 B	0%	6	6	100%
	AES-TA	2949 B	2965 B	0.54%	0 B	0 B	-	2	2	100%

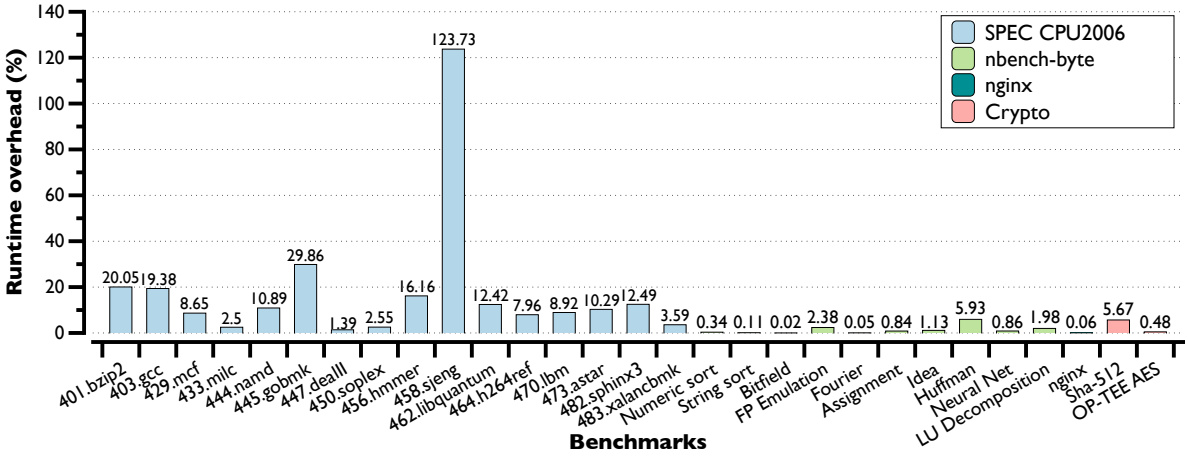

Figure 8: Runtime Overhead of TAGShield on various benchmarks when executed on a Google Pixel 8.

First, we tested nginx by measuring the average delay when serving a static page under a workload of 12 threads and 400 concurrent connections over a 30-second period. The observed overhead was just 0.06%, indicating virtually no impact on performance.

Next, we evaluated TAGShield 's impact on cryptographic operations using an open-source SHA2-512 crypto engine [79]. We computed digests for 1 MB, 10 MB, 100 MB, and 1 GB of random data, with measured performance overheads of 5.76%, 6.37%, 5.14%, and 5.41%, respectively. On average, these measurements result in an overhead of 5.67%, as shown in Figure 8. These results confirm that TAGShield can be integrated into cryptographic workloads without significantly affecting performance.

Finally, we examined the effect of TAGShield on a TrustZone-based system running the open-source OP-TEE OS [80]. Instead of instrumenting the entire TEE OS, we focused on a specific Trusted Application (TA) responsible for AES encryption. In general, OP-TEE functions as an OS managing multiple TAs, which execute in the Secure World and interact with the Rich Execution Environment (REE) via GlobalPlatform APIs [81]. Figure 9 illustrates the system architecture. When a Client Application (CA) requests an encryption service, the OP-TEE core forwards the request to the relevant TA, which processes it and returns the result to the CA. Since deploying OP-TEE on the TrustZone of our Pixel 8 is not feasible, we created a mock OP-TEE setup running entirely in

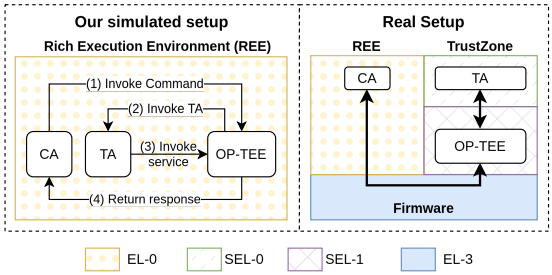

Figure 9: `TAGShield` setup for the evaluation of an OP-TEE Trusted Application (TA). On the left, our setup, with Client Application (CA), TA and OP-TEE running inside the REE. On the right, a real setup where the CA runs in REE and the TA with OP-TEE in TrustZone.

the REE. This setup consists of three components: a `TAGShield`-instrumented TA, a minimal OP-TEE core, and a CA invoking the TA's AES encryption service. Unlike a real deployment, where the TA runs in TrustZone, our setup executes all components at user level, as shown in Figure 9.

To assess performance impact, the CA was configured to request 300 AES encryptions of a 4 KB buffer. `TAGShield` introduced an average overhead of 0.48%. Notably, this setup simplifies system interactions by eliminating privilege-level transitions and Secure/Non-Secure world switches. While our measurements do not fully reflect real-world performance, they provide a lower-bound estimate of `TAGShield`'s impact, as system-level interactions—being outside the scope of `TAGShield` instrumentation—remain unaffected.

5.2 Security Evaluation

To validate our implementation, we evaluated it using a subset of the Juliet C/C++ 1.3 Test Suite from NIST [82], a widely used benchmark for assessing memory defense mechanisms. The suite contains 64,099 test cases, systematically categorized by CWE identifiers, specific vulnerability types, and unique variants. For our analysis, we focused on CWE-121 (Stack-Based Buffer Overflows), excluding irrelevant tests such as 32-bit-only bugs, Windows-specific cases, and non-deterministic scenarios, resulting in 2,362 distinct variants.

As summarized in Table 2, our implementation successfully triggered a tag mismatch in 47 test cases (covering 1,925 variants), with no missed detections. Additionally, 9 test cases (spanning 359 variants) were mitigated by the `TAGShield` allocator, which aligns allocations to MTE tag granularity — effectively preventing off-by-one overflows in non-multiple-sized allocations.

We note that this evaluation does not capture the full protection scope of `TAGShield`, as Juliet does not include test cases specifically targeting tag corruption attacks. To further assess the effectiveness of `TAGShield`, we tested it on a sample application [45] and demonstrated that it significantly improves the protection of pointer tags by mitigating a broader range of attacks. The application implements a simple substring extraction function, where the user inputs a string and an index, and the program returns the corresponding substring. As shown in Listing 5.1, this application contains a non-linear buffer overflow that can be exploited to corrupt the `name` pointer by supplying a large index.

Table 2: Juliet test cases: $N(M)$ denotes N unique cases, each with M different flow variants.

Fixed by allocator	Detected	Missed
9 (359)	47 (1925)	0 (0)

```
1 fgets(input, sizeof(input), stdin);
2 offset = strtoul(input, NULL, 10);
3 printf("Sub:%s\n", &name[offset]);
```

Listing 5.1: Example of non-linear buffer overflow vulnerability that can be defended with `TAGShield`.

The impact of this vulnerability varies depending on the target application. In our experiment, we simulate two common attack primitives: (1) arbitrary memory reads and (2) arbitrary memory writes, both of which can be leveraged for advanced exploitation techniques such as Direct Data Manipulation (DDM) [83], Data-Oriented Programming (DOP) [11], or Block-Oriented Programming (BOP) [84]. We consider two different adversaries:

- **Naïve adversary**: modifies the pointer to reference an unintended memory location.
- **Sophisticated adversary**: not only redirects the pointer but also forges a matching MTE tag, bypassing basic memory tagging protections.

We tested these attacks on three versions of the application, each hardened with a different level of protection:

- **Unprotected binary (no memory tagging)**: Both adversaries successfully corrupted the pointer and accessed unintended memory locations without detection, illustrating the risks posed by the lack of data-oriented defenses.
- **Binary hardened with MemTagSanitizer [38]**: The naïve adversary was successfully thwarted due to an MTE tag mismatch. However, the sophisticated adversary, by forging a valid tag for the target memory location, successfully bypassed the MemTagSanitizer protection. This demonstrates the practical feasibility of tag forgery attacks against MTE-based defenses.
- **Binary protected by `TAGShield`**: Like the previous case, `TAGShield` prevented the naïve adversary from corrupting the pointer. Crucially, it also neutralized the sophisticated adversary. Even when both the pointer's tag and its contents were successfully manipulated, `TAGShield` restored the correct tag before permitting any memory access, thereby blocking the attack.

These results provide empirical evidence of the weaknesses in existing MTE-based defenses and highlight `TAGShield` 's effectiveness in mitigating tag corruption attacks. We also attempted to evaluate a fourth approach using StickyTags [39], but the open-source implementation lacked proper documentation, making it difficult to integrate into our build process. Due to missing instructions and persistent build errors, we were unable to harden our binaries with StickyTags for direct comparison.

6 Discussion

6.1 Comparison with State-of-the-Art Work

ASan [50] detects buffer overflows by tagging every 8 bytes of application memory with a 1-byte tag stored in shadow memory, while HWASAN (Hardware-Assisted AddressSanitizer) [85] leverages the TBI feature in 64-bit architectures to embed tags within pointer metadata. However, both incur substantial runtime overheads, typically ranging from 1.5x to 3x [66], with a geomean overhead of at least 86% on SPEC CPU2006 [86], compared to TAGShield 's 10%. Moreover, TAGShield significantly reduces code size overhead, adding only 5.84% overhead, compared to at least 40% in HWASAN. MemTagSanitizer [38] implements an optimized MTE-based version of HWASAN, achieving a higher runtime overhead (22.8%) than TAGShield while benefiting from hardware-assisted tagging. However, unlike TAGShield, it does not provide tag integrity protection, making it susceptible to tag corruption attacks, limiting its applicability to debugging and testing purposes only.

In terms of runtime exploit mitigation, Low-Fat-based stack protection [31] incurs a higher overhead (26%) and faces compatibility issues, particularly due to its incomplete handling of benign out-of-bounds pointers. Similarly, Delta Pointers [42], which repurpose unused bits in 64-bit architectures to implement a tagging-based stack buffer overflow protection mechanism, exhibit a geomean overhead of 35% on SPEC CPU2006 while failing to provide protection against stack underflows.

Among MTE-based solutions, StickyTags [39] deterministically assign tag values based on an object's memory location. This approach effectively creates deterministic red zones of 16×object_size bytes, preventing both linear buffer overflows and certain non-linear overflows. StickyTags achieves excellent performance, with an average overhead of less than 4%, but suffers from two key limitations. First, it does not protect pointer tags, making it vulnerable to tag forgery attacks, which TAGShield explicitly prevents. Second, its design may facilitate stack temporal memory errors. For example, consider the scenario discussed in Listing 6.1, which is taken from DataGuard [13] and further analyzed there. In this scenario, a pointer (buf) references a local variable (lbuf). Upon function return, TAGShield properly untags the related memory, causing accesses to crash upon mismatched tag checks. However, in StickyTags, the pointer retains its tag indefinitely, allowing access to the same memory region throughout its lifetime, leading to potential use-after-return vulnerabilities.

ZomeTag [40], another MTE-based approach, reserves large 4GB-aligned zones, within which up to 14 objects can be simultaneously allocated, each tagged with a unique ARM MTE tag. Additionally, ZomeTag applies efficient arithmetic constraints to prevent pointers from escaping their assigned 4GB zones, partially preventing tag manipulation, similar to the TDI approach [87]. ZomeTag achieves a a higher overhead than TAGShield on stack allocations (20% vs. 10% geomean overhead). Furthermore, its zone-based memory layout is highly sparse, significantly increasing TLB pressure due to under-utilization of available memory. Additionally, like StickyTags [39], ZomeTag remains vulnerable to certain tag forgery cases, an issue TAGShield effectively mitigates through its tag integrity enforcement.

```
1 void example(int c, char **buf){
2   int lct = BUF_SIZE;
3   char lbuf[lct];
4   if(ct < lct){      // (1) spatial error
5     strlcpy(lbuf, *buf, (size_t) ct);
6   }
7   *buf = lbuf;       // (2) temporal error
8 }
```

Listing 6.1: Example function demonstrates: (1) bounds error that enabled overread of buf, and (2) temporal error as *buf references local variable lbuf after return.

```
1 int * heapPtr = malloc(10 * sizeof(int));
2 heapPtr[20] = 40;
```

Listing 6.2: C code that creates a new heap variable and perform an out-of-bound array access.

6.2 Extending to the heap

TAGShield was primarily designed to address stack-based spatial memory errors, given the severe impact of their exploitation [13–15]. However, spatial heap memory errors can still be exploited with similar consequences, as illustrated by the buffer overflow in Listing 6.2. While orthogonal defences for heap memory exist, we argue that TAGShield could be extended to protect heap allocations through persistent tags, similar to its handling of stack allocations. Currently, root pointers holding heap references, such as the %heapPtr stack variable in Listing 6.3, which stores the actual heap address, are natively protected by TAGShield. In contrast, the heap memory itself is managed by the allocator, which currently applies no tags.

From a theoretical perspective, heap variables should always be treated as unsafe and thus require tagging, since their addresses are fetched from memory and may be corrupted. Our current implementation, however, is integrated into the compiler and does not intervene in the heap allocator. To extend protection to heap memory, the compiler could compute a tag for each allocation and apply it after malloc returns, as shown in Listing 6.4. For greater efficiency, the allocator itself could be modified to accept an additional, transparent argument to malloc, namely, the tag to apply to memory, and perform the tagging operations internally.

To provide guarantees equivalent to those for stack variables, dynamic heap tags must also be stored in TAG_REG, allowing us to track any changes to an allocation's tag. Furthermore, the free function must be instrumented to apply a special free-tag (e.g., 0x0) to released memory.

6.3 Limitations

In general, TAGShield inherits the well-known limitations of MTE-based solutions [39], including potential compatibility issues with MTE-unaware external software modules that implement custom pointer tagging. However, TAGShield does not suffer from any fundamental design flaws; any constraints it faces are primarily

```
%heapPtr = alloca ptr, align 16
%call = call noalias ptr @malloc(i64 noundef 40)
store ptr %call, ptr %heapPtr, align 16

%0 = load ptr, ptr %heapPtr, align 16
%arrayidx = getelementptr inbounds i32, ptr %0, i64 20
store i32 40, ptr %arrayidx, align 4
```

Listing 6.3: LLVM IR representation of Listing 6.2.

```
%heapPtr = alloca ptr, align 8
%call = call noalias ptr @malloc(i64 noundef 40)
call @tagMemory(ptr %heapPtr, align 8)
store ptr %call, ptr %heapPtr, align 8

%0 = load ptr, ptr %heapPtr, align 8
%arrayidx = getelementptr inbounds i32, ptr %0, i64 20
call @enforceDynamicTag(ptr %arrayidx, #tag_index)
store i32 40, ptr %arrayidx, align 4
```

Listing 6.4: Instrumentation of Listing 6.3, illustrating pointer tag enforcement for heap allocations.

engineering challenges rather than inherent limitations. Furthermore, similar to other works in the literature [37, 39, 40, 71–74], TAGShield does not enforce tag integrity at the sub-object level within structs. Instead, it treats them as part of the same allocation and applies a uniform tag across the entire structure.

Additionally, the current TAGShield's tag integrity mechanism is constrained in handling pointer aliasing [3] in complex programs. Consider the code snippet in Figure 10, where at line ❷, an alias of `ptr` is created by storing its address in `pptr`, making `pptr` an alias of `ptr`. Once this alias exists, it can reference the original pointer, as seen in line ❸, where it modifies `ptr` through its alias. TAGShield tracks and enforces tag integrity for non-root pointers. This means that when '`ptr`' is modified at line ❸, TAGShield should update its internal state to ensure that subsequent dereferences of both `(ptr)` and `(*(pptr))` (e.g., on lines ❺ and ❹, respectively) enforce the correct tag, since both refer to the same memory region. In other words, after line ❸, the tag states for all aliases should remain synchronized. Failing to synchronize pointer aliases when persisting tag states can result in false negatives during MTE validation. For instance, on line ❺, if TAGShield incorrectly enforces the red tag on `*(ptr)`—based on its outdated state from line ❶—the protection mechanism fails to detect a violation introduced at line ❸.

Effectively detecting pointer aliasing remains an open research challenge, and maintaining synchronized tag states introduces significant instrumentation overhead. To mitigate these challenges, TAGShield uses a conservative pointer analysis with no false negative, defaulting to its baseline coloring scheme once aliasing is detected. This means that after a pointer is aliased, TAGShield ceases enforcing tag integrity for that pointer and its aliases, falling back to standard memory tagging protections. We also consider a

[3] A pointer alias occurs when a pointer's address is stored in another pointer, effectively creating multiple references to the same memory location.

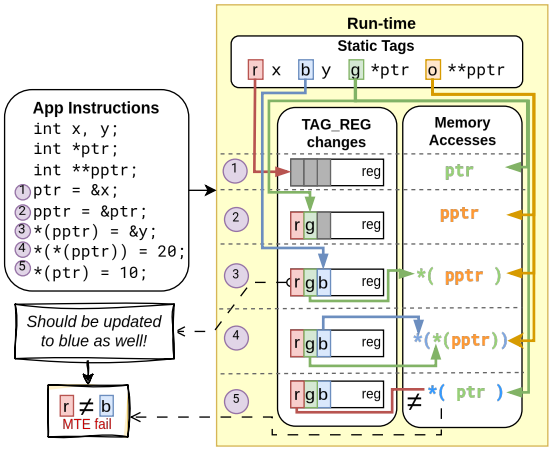

Figure 10: Tag integrity issue with pointer aliasing. After the `ptr` alias created at ❷, instruction ❸ should change the `TAGShield` state for both `*(ptr)` and `*(*(pptr))`. Failing to do so will cause an MTE failure on ❺.

pointer as aliased as soon as it leaves a function, e.g., if it is passed as a parameter. Notably, we did not observe any pointer aliases in the SPEC CPU2006 benchmark suite when compiling with the `-O3` optimization flag.

Similarly, TAGShield falls back to the standard protection when interacting with non-instrumented code, such as external libraries. When a pointer is passed to such code, their content and tag could be manipulated without TAGShield knowing. Therefore, as soon as a pointer is passed to external code, its tag is not tracked anymore.

Finally, we acknowledge that TAGShield may introduce a high runtime overhead in certain applications. Nevertheless, as no single approach achieves an optimal balance between performance and security across all scenarios, TAGShield remains a practical solution for applications where its security guarantees outweigh the associated performance costs.

7 Conclusion and Future Work

This paper presented TAGShield, a runtime exploit mitigation mechanism that leverages Arm MTE to protect stack spatial memory while preserving pointer tag integrity, addressing a key limitation of existing MTE-based defenses. TAGShield assigns deterministic tags to unsafe stack allocations at compile-time and enforces persistent tag integrity at runtime by restoring the authentic tag at each pointer dereference, thereby preventing MTE bypasses enabled by tag forgery through pointer arithmetic. Our evaluation on real MTE capable hardware demonstrates that TAGShield provides robust protection with practical performance costs, incurring an average geometric mean runtime overhead of 10% across representative benchmarks.

Future work includes extending TAGShield to heap allocations through allocator integration and exploring deployment opportunities on emerging tagged memory architectures, including RISC-V.

Acknowledgments

This work was partially funded by the European Union under the Horizon Europe Programme (GA 101070537 — CrossCon) and by NextGenerationEU PRIN 2022 S2: Safe and Secure Industrial Internet of Things (Prot. No. 202297YF75). Part of this work was conducted during the first author's internship at the Huawei Munich Research Center, Germany.

References

[1] Google. Retrofitting Spatial Safety to Hundreds of Millions of Lines of C++, 2024. URL https://security.googleblog.com/2024/11/retrofitting-spatial-safety-to-hundreds.html.

[2] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[3] Mahmoud Ammar, Ahmed Abdelraoof, and Silviu Vlasceanu. On Bridging the Gap between Control Flow Integrity and Attestation Schemes. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6633–6650, 2024.

[4] Per Larsen and Ahmad-Reza Sadeghi. *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, 2018.

[5] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*, pages 30–40, 2011.

[6] Mahmoud Ammar, Adam Caulfield, and Ivan De Oliveira Nunes. SoK: Integrity, Attestation, and Auditing of Program Execution. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3255–3272. IEEE, 2025.

[7] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *Proceedings of the 18th European Workshop on Systems Security*, pages 40–48, 2025.

[8] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015.

[9] Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Data-Only Attack Generation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1401–1418, 2024.

[10] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–36, 2021.

[11] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[12] Adriaan Jacobs and Stijn Volckaert. Not Quite Write: On the Effectiveness of Store-Only Bounds Checking. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, pages 171–187, 2024.

[13] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The Taming of the Stack: Isolating Stack Data from Memory Errors. In *NDSS*, 2022.

[14] Kaiming Huang, Jack Sampson, and Trent Jaeger. Assessing the Impact of Efficiently Protecting Ten Million Stack Objects from Memory Errors Comprehensively. In *2023 IEEE Secure Development Conference (SecDev)*, pages 67–74. IEEE, 2023.

[15] Rahul George, Mingming Chen, Kaiming Huang, Zhiyun Qian, Thomas La Porta, and Trent Jaeger. OPTISAN: Using Multiple Spatial Error Defenses to Optimize Stack Memory Protection within a Budget. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7195–7212, 2024.

[16] James P Anderson et al. Computer Security Technology Planning Study. Technical report, Citeseer, 1972.

[17] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[18] The Linux Kernel. Control-flow Enforcement Technology (CET) Shadow Stack, 2024. URL https://docs.kernel.org/next/x86/shstk.html.

[19] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. FLAShadow: A Flash-based Shadow Stack for Low-end Embedded Systems. *ACM Transactions on Internet of Things*, 5(3):1–29, 2024.

[20] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[21] Kernel Development Community. Guarded Control Stack support for AArch64 Linux. URL https://docs.kernel.org/arch/arm64/gcs.html.

[22] Wonwoo Choi, Minjae Seo, Seongman Lee, and Brent Byunghoon Kang. SuM: Efficient shadow stack protection on ARM Cortex-M. *Computers & Security*, 136: 103568, 2024.

[23] Lorenzo Binosi, Gregorio Barzasi, Michele Carminati, Stefano Zanero, and Mario Polino. The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1360–1374, 2024.

[24] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. CoDaRR: Continuous Data Space Randomization against Data-Only Attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 494–505, 2020.

[25] Kuznetsov Volodymyr, Szekeres Laszlo, Payer Mathias, Candea George, and R Sekar. Code-pointer Integrity. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[26] Michele Grisafi, Mahmoud Ammar, and Bruno Crispo. On the (in)security of Memory Protection Units : A Cautionary Note. In *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 157–162. IEEE, 2022.

[27] Seongman Lee, Hyeonwoo Kang, Jinsoo Jang, and Brent Byunghoon Kang. SaVioR: Thwarting Stack-Based Memory Safety Violations by Randomizing Stack Layout. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2559–2575, 2021.

[28] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In *NDSS*, 2015.

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[30] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security Symposium*, volume 10, page 96, 2009.

[31] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *NDSS*, volume 17, pages 1–15, 2017.

[32] Myoung Jin Nam. *Inline and Sideline Approaches for Low-Cost Memory Safety in C*. PhD thesis, University of Cambridge, 2021.

[33] Yeoul Na. -fbounds-safety. Enforcing Bounds Safety for Production C Code. *EuroLLVM Developers' Meeting*, May 2023. URL https://llvm.org/devmtg/2023-05/slides/TechnicalTalks-May11/01-Na-fbounds-safety.pdf.

[34] Google. Retrofitting spatial safety to hundreds of millions of lines of C++. https://security.googleblog.com/2024/11/retrofitting-spatial-safety-to-hundreds.html, 2024.

[35] ARM. Learn the Architecture - Providing Protection for Complex Software. https://developer.arm.com/documentation/102433/0100, 2022.

[36] Andreas Hager-Clukas and Konrad Hohentanner. DMTI: Accelerating Memory Error Detection in Precompiled C/C++ Binaries with ARM Memory Tagging Extension. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1173–1185, 2024.

[37] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. MTSan: A Feasible and Practical Memory Sanitizer for Fuzzing COTS Binaries. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 841–858, 2023.

[38] LLVM Project. Memtagsanitizer, . URL https://llvm.org/docs/MemTagSanitizer.html.

[39] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 217–217. IEEE Computer Society, 2024.

[40] Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek. ZOMETAG: Zone-Based Memory Tagging for Fast, Deterministic Detection of Spatial Memory Violations on ARM. *IEEE Transactions on Information Forensics and Security*, 2023.

[41] Benjamin Orthen, Oliver Braunsdorf, Philipp Zieris, and Julian Horsch. SoftBound+CETS Revisited: More Than a Decade Later. In *Proceedings of the 17th European Workshop on Systems Security*, pages 22–28, 2024.

[42] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta Pointers: Buffer Overflow Checks Without the Checks. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[43] Mohamed Tarek Ibn Ziad, Miguel A Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan. ZeRO: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 999–1012. IEEE, 2021.

[44] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 378–387. IEEE, 2005.

[45] Authors. TAGShield Source Code. URL https://github.com/MicheleGrisafi/TAGshield.

[46] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[47] David R Chase, Mark Wegman, and F Kenneth Zadeck. Analysis of Pointers and Structures. *ACM SIGPLAN Notices*, 25(6):296–310, 1990.

[48] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[49] CWE. 2024 CWE Top 25 Most Dangerous Software Weaknesses, 2024. URL https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html.

[50] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[51] Floris Gorter and Cristiano Giuffrida. RangeSanitizer: Detecting Memory Errors with Efficient Range Checks. In *USENIX Security*, 2025.

[52] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[53] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, USA, 2002. USENIX Association. ISBN 1880446006.

[54] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything You Want to Know about Pointer-based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2015.

[55] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[56] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 145–156, 2010.

[57] Gregory J Duck, Yuntong Zhang, and Roland HC Yap. Hardening Binaries Against More Memory Errors. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 117–131, 2022.

[58] Olatunji Ruwase and Monica S Lam. A Practical Dynamic Buffer Overflow Detector. In *NDSS*, volume 2004, pages 159–169, 2004.

[59] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. Into the Depths of C: Elaborating The De-Facto Standards. *ACM SIGPLAN Notices*, 51(6):1–15, 2016.

[60] Gregory J Duck and Roland HC Yap. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[61] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1901–1915, 2022.

[62] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. CryptSan: Leveraging ARM Pointer Authentication for Memory Safety in C/C++. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 1530–1539, 2023.

[63] Yu-Chang Chen and Shih-Wei Li. HeMate: Enhancing Heap Security through Isolating Primitive Types with Arm Memory Tagging Extension. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pages 1–11, 2024.

[64] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054, 2021.

[65] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. Top of the Heap: Efficient Memory Error Protection of Safe Heap Objects. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1330–1344, 2024.

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

[67] LLVM. Stack Safety Analysis. https://llvm.org/docs/StackSafetyAnalysis.html.

[68] Richard Earnshaw. GLIBC MTE Support. URL https://sourceware.org/git/?p=glibc.git;a=commit;h=d27f0e5d889f4bf4a796fe2a883b2f264bf40c12.

[69] The Linux Kernel. KASAN. URL https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[70] LLVM Project. Llvm 18.1.8 release notes, . URL https://releases.llvm.org/18.1.8/docs/ReleaseNotes.html.

[71] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKCs. In *NDSS*, pages 1–17, 2022.

[72] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 177–189, 2023.

[73] Aditi Partap and Dan Boneh. Memory tagging: A memory efficient design. *arXiv preprint arXiv:2209.00307*, 2022.

[74] Hans Liljestrand, Carlos Chinea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N. Asokan. Color My World: Deterministic Tagging for Memory Safety, 2022.

[75] Inc. Gentoo Foundation and Förderverein Gentoo e.V. Gentoo linux. URL https://www.gentoo.org/.

[76] Uwe F. Mayer. Linux/unix nbench. URL https://www.math.utah.edu/~mayer/linux/bmark.html.

[77] System Performance Evaluation Cooperative (SPEC). SPEC CPU® 2006. URL https://www.spec.org/cpu2006/.

[78] Q-Success W3Techs. Usage statistics of Nginx. URL https://w3techs.com/technologies/details/ws-nginx.

[79] Timothy Vaccarelli. SHA256-512, 2017. URL https://github.com/LeFroid/sha256-512.

[80] TrustedFirmware.org. About OP-TEE. URL https://optee.readthedocs.io/en/latest/general/about.html.

[81] Hassaan Janjua, Mahmoud Ammar, Bruno Crispo, and Danny Hughes. Towards a Standards-Compliant Pure-Software Trusted Execution Environment for Resource-Constrained Embedded Devices. In *Proceedings of the 4th Workshop on System Software for Trusted Execution*, pages 1–6, 2019.

[82] Tim Boland and Paul E Black. Juliet 1. 1 C/C++ and java test suite. *Computer*, 45 (10):88–90, 2012.

[83] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX security symposium*, volume 5, page 146, 2005.

[84] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882, 2018.

[85] Android Open Source Project. Hardware-assisted AddressSanitizer, . URL https://source.android.com/docs/security/test/hwasan.

[86] Google. AddressSanitizer Performance Numbers, 2015. URL https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers.

[87] A. Milburn, E. van der Kouwe, and C. Giuffrida. Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation. In *2022 IEEE Symposium on Security and Privacy (S&P)*, pages 259–275, Los Alamitos, CA, USA, May 2022. IEEE Computer Society. doi: 10.1109/SP46214.2022.00016. URL https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00016.