Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags

Floris Gorter*, Taddeus Kroes‡, Herbert Bos* and Cristiano Giuffrida*
*‡*Vrije Universiteit Amsterdam*
‡*taddeuskroes@gmail.com*
*{*f.c.gorter,h.j.bos,c.giuffrida*}*@vu.nl*

***Abstract*—Spatial memory errors such as buffer overflows still rank among the top vulnerabilities in C/C++ programs. Despite much research in the area, the performance overhead of (even partial) mitigations is still too high for practical adoption. To reduce the cost, recent solutions are shifting towards hardware-assisted techniques such as Arm's Memory Tagging Extension (MTE). Unfortunately, state-of-the-art MTE solutions incur high overhead due to frequent memory (re)tagging, especially on the stack. Moreover, they rely on the secrecy of random memory tags and offer *probabilistic* security guarantees.**

In this paper, we first provide evidence that *random* tagging offers limited protection as attackers can deduce the memory tags by means of speculative probing. We then present StickyTags, a *deterministic* MTE solution that efficiently mitigates bounded spatial memory errors. By organizing the stack and heap layout into per-size-class regions, we can apply *persistent* memory tags to each region in a predetermined pattern. Hence, the memory tags need only be initialized once, after which they can be reused by objects of the same size class. This eliminates the need for costly memory retagging and allows for a fixed, round-robin assignment of the tags, surrounding every object with large *implicit* spatial guards. While the size of such guards is bounded by the 4-bit MTE entropy (16 tags), the protection is *efficient* and *deterministic*. Indeed, we show StickyTags significantly outperforms existing solutions with realistic runtime overheads for practical adoption ($\leq 4\%$ on SPEC CPU2006), while fully mitigating 7 out of 8 spatial CVEs evaluated by a recent probabilistic MTE solution.

1. Introduction

Spatial memory errors remain a common and impactful security concern. The 2023 CWE ranking lists *out-of-bounds writes* as the most severe software weakness [1]. Anecdotally, the GWP-ASan project has already found over thirty buffer overflows in the live build of Google Chrome [2], highlighting the need for exploit mitigations. While many existing tools can *detect* such bugs during software testing [3], [4], [5], [6], [7], [8], [9], [10], post-deployment solutions have found little applicability in the field due to their high overhead. Recent reports indicate that mitigations only see real-world deployment if their performance overhead stays below 5% [11]—rendering existing bounds checking solutions impractical [12], [13], [14], [15], [16]. In response, contemporary memory error detection and mitigation systems are shifting towards hardware-assisted solutions to reduce the overhead [6], [7], [17], [18], [19].

In particular, Arm's Memory Tagging Extension (MTE) is a strong contender to provide spatial memory error mitigation on the cheap. MTE associates every memory location and every pointer with a tag, with the hardware disallowing any dereference if the pointer and memory tags do not match. Unfortunately, even state-of-the-art MTE solutions remain costly due to the need for frequent memory (re)tagging: LLVM's MemTagSanitizer [20] incurs average and worst-case overheads on SPEC CPU2006 of 15.2% and 3.67x (respectively) for the stack alone (Section 8).

Moreover, existing MTE solutions [20], [21], [22], [23], [24] heavily rely on *random tags* provided by the hardware [25] (which, in turn, impose expensive retagging costs). While such tags are not trivial to predict, at best they offer probabilistic security guarantees with low entropy. In particular, tag collisions between near or neighboring objects may leave the application vulnerable to *contiguous* overflows and *bounded* overflows such as type confusion bugs. Moreover, the low (4-bit) entropy of MTE tags leaves applications trivially vulnerable to brute-force attacks against a variety of crash-resistant targets, such as servers [26], web browsers [27], and even kernels, for instance Linux with the default *oops* mechanism [28].

Unfortunately, the situation is even worse, since, as we show, attackers can find pointer / memory tag matches through *speculative probing* [29]. More specifically, we show attackers can use a *contention-based* side channel to deduce whether or not a tag check results in a violation (i.e., tag mismatch). These results confirm, for the first time, conjectures in the community that MTE is vulnerable to side channels [25], [30], [31] and contradict a recent analysis by Google [32]. The net result is that such speculative oracles broaden the brute-force attack surface to non-crash-resistant targets (e.g., Linux *without* the *oops* mechanism [29]).

In this paper, we present StickyTags, an *efficient* and *deterministic* spatial memory error mitigation for the stack and the heap. Rather than aiming for the classic random

‡*Now at Google*

(re)tagging-based design to detect generic spatial and temporal errors with only *probabilistic* guarantees, StickyTags focuses on mitigating a specific (but widespread) class of vulnerabilities (bounded spatial errors) with *strong* performance and security guarantees. In particular, StickyTags offers production-ready overheads below 5% and deterministic security guarantees (where all tags are public and known) bounded by the number of MTE-provided tags.

To minimize memory tagging costs, we build on Arm's recommendation to limit the number of (de)allocations [33]. Rather than rewriting the application, we divide the heap and stack in *per-size-class* regions—assigning objects to predetermined slots with *persistent* memory tags already in place. We make sure to initialize the tags only *once*, and keep them in memory ready to be reused for objects of the same size class. The synergy between organizing memory into size classes and the underlying persistent tags allows us to achieve high performance by eliminating the need for memory retagging. This is especially beneficial on the stack, where the allocation (and hence tagging) frequency is high. As an added advantage and in contrast to state-of-the-art solutions [20], our stack tagging design also offers increased backwards compatibility with legacy (non-MTE) devices. We detail our compatibility guarantees in Section 5.

To provide deterministic security guarantees, we assign tags in a round-robin fashion in each region such that the tag of any object cannot collide with that of a known number of neighboring slots. As a result, our prototype StickyTags effectively creates *implicit* spatial guards around each object. Even with the current tag size of 4 bits, StickyTags efficiently mitigates spatial memory errors with under- and overflow guards of 15 times the size class. Since the smallest class contains objects of 16 bytes (the MTE tagging granularity), each object in this class is protected by implicit spatial guards of 240 bytes in both directions. The spatial guards for larger size classes are proportionally larger.

Our evaluation shows that StickyTags significantly outperforms state-of-the-art spatial memory error mitigations. On SPEC CPU2006 and 2017, StickyTags incurs geomean overheads of $\leq 4\%$ measured both with MTE analogs [34] and MTE devices, bringing spatial memory protection within reach of production systems for the first time. StickyTags is 12x faster on average than MemTagSanitizer (stack tagging) and nearly 2x faster than the Scudo allocator (heap tagging), while fully mitigating 7 out of 8 spatial CVEs evaluated by the recent (probabilistic) MTSan [17].

Contributions. We make the following contributions:

- We present the first on-device evidence that speculative probing can leak MTE pointer / memory tag matches, questioning random tagging as a mitigation strategy even for applications not prone to classic brute forcing.
- We present a design for deterministic memory tagging for the stack and the heap that uses *persistent* tags to enable efficient spatial memory guards with MTE. We further study the applicability of persistent spatial guards to x86 architectures, using lightweight compiler instrumentation to compensate for the lack of MTE.
- We evaluate our StickyTags prototype and show that StickyTags provides production-ready overheads.

Availability. https://github.com/vusec/stickytags

2. Background

Spatial memory errors. Spatial memory errors such as buffer overflows occur when a derived pointer erroneously accesses a different object than its base pointer. To prevent exploitation of such bugs, bounds checkers [12], [13], [14], [15], [16], [35] retrofit programs with checks that disallow such illegal accesses. Unfortunately, bounds checkers have not seen widespread adoption due to high overheads. To lower the overhead, other solutions reduce the scope of bounds checking and instead rely on *explicit spatial guards* bracketing each memory object [2], [3], [4], [9], [36], [37]. Such guards, implemented by means of *guard pages* [2], [38] or compiler-enforced *redzones* [3], [4], [36], [37], can detect invalid out-of-bounds reads/writes up to the guard size. This can mitigate *contiguous overflows* and *bounded non-contiguous overflows*, such as off-by-N errors and type confusion. In the latter case, unsafe type casts allow attackers to replace an object pointer with a pointer to a larger object type, yielding out-of-bounds reads/writes at a bounded offset up to the largest difference in confusable object sizes [39]. Unfortunately, existing guard-based solutions still incur high overheads and have only found practical adoption in offline testing [3], [37] or online sampling [40].

Memory Tagging Extension. Memory Tagging Extension (MTE) is an Armv8.5+ feature to detect memory errors. It introduces a 'lock' and 'key' mechanism, with the hardware only permitting reads/writes if the pointer tag (key) matches the memory tag (lock). Checks are supported both in *synchronous* and *asynchronous* mode. Pointer tagging relies on Arm's TBI (Top-Byte Ignore) feature to store a tag in the upper pointer bits. The 4-bit memory tags (16 values in total) are stored separately from application data. Existing MTE solutions [17], [20], [21], [22], [23], [24] often rely on the Arm `IRG` instruction to assign a random tag to each allocation/deallocation, which scales poorly due to frequent (re)tagging and provides probabilistic security.

3. Speculatively Probing for Random Tags

For probabilistic MTE solutions based on random tagging [20], [21], [22], [23], [24], the assumption is that, even if attackers manage to hijack a tagged *victim pointer* (e.g., via a buffer overflow) to reference a *target object*, they cannot predict whether the tag of the target object matches the pointer tag—hindering reliable exploitation. However, even without brute-forcing capabilities [26], if attackers can deduce which tags are assigned at runtime, then the random source of the tags has no added benefit. This is because

Listing 1 Tested probe gadgets to leak tag matches.

```
1       flush(signal);
2       if (/* mispredict */) {
3   #ifdef DEP_LOAD
4           idx = *oob_ptr; // target tag check
5           *(signal+idx);  // dependent load
6   #else
7           *oob_ptr;       // target tag check
8           *signal;        // independent load
9   #endif
10      }
11      reload(signal);     // is signal cached?
```

attackers can massage memory [41] until the victim pointer's random tag happens to match the one of the target object—and only then trigger the vulnerability to achieve reliable exploitation in spite of random tagging.

We now show attackers can indeed deduce tag assignments via side channels and bypass probabilistic MTE solutions that rely on secret random tags, confirming conjectures from the community [25], [30], [31] with the first evidence on real MTE hardware. Specifically, we show that attackers can leak whether the tag of a given victim pointer matches the tag of a target object using *speculative probing* [29], [42], [43]. This is possible by repeatedly probing different pointer / (massaged) object pairs using a *probe gadget* until microarchitectural side channels leak a tag match.

For our evaluation, we conducted experiments on rooted Samsung Galaxy S22 and Google Pixel 8 Pro devices, supporting sync/async MTE mode. Our first experiment was on the former device in sync mode, using the standard Spectre *probe gadget* in Listing 1 (`DEP_LOAD` case). The gadget speculatively issues an out-of-bounds load via the (tagged) victim pointer (`oob_ptr`, line 4) followed by a load dependent on the loaded value (at `signal+idx`, line 5). The dependent load, if completed, fills a cache line and transmits 1 bit of information via a classic Flush+Reload *covert channel* [44] (i.e., "cache hit" as revealed by timing).

We initially expected two possible scenarios for failed (i.e., mismatching) MTE checks on the speculative path: (i) they are *fully synchronous* and prevent the victim load from passing data to the dependent load, resulting in a 0% cache hit rate; (ii) they are *fully asynchronous* and allow data to be passed to the dependent load, resulting in a 100% cache hit rate similar to the successful checks. The former scenario would allow the MTE implementation to guarantee *speculative memory safety*—since the checks hinder any invalid speculative access—but also incur *tag leakage*—since the attacker can distinguish correct/incorrect tag pairs based on the cache hit rate. The latter scenario, in turn, would yield opposite guarantees (i.e., no tag leakage, no speculative memory safety). However, our first experiment revealed a high cache hit rate (suggesting checks are *asynchronous*), but not as high as for successful checks. Hence, we can still leak a tag match based on the cache hit rate.

Our next question was why the *failed* check causes the subsequent dependent load to *sometimes* not complete in the speculation window. One hypothesis is that failed checks occasionally cut the speculation window short. Another is that they create *contention* on the memory subsystem (by having to act upon the tag violation), causing other memory operations to occasionally stall and fail to complete within the window. To answer this question, we designed another experiment with the simpler probe gadget in Listing 1 (`not DEP_LOAD` case), which, unlike standard Spectre, drops the second dependent load in favor of an arbitrary (*independent*) one. Switching to the independent load did not affect our original results and neither did switching to async MTE mode (where checks are *very* asynchronous even architecturally) or replacing the independent load with an independent store. This all seems to confirm our second hypothesis, with contention on the memory subsystem affecting the cache hit rate (and allowing for tag match leaks).

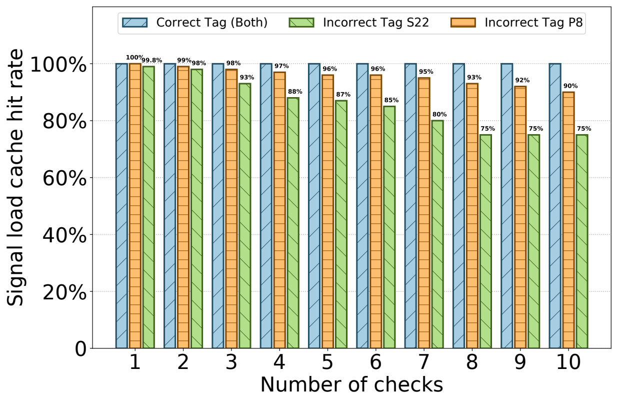

Figure 1: Cache hit rates of a single *independent* load for correct/incorrect tag match on the Samsung S22 and Google Pixel 8 Pro.

Figure 1 presents our results when repeatedly triggering the simpler probe gadget for matching/mismatching tag pairs as we increase the number of checked out-of-bounds loads in the gadget (by duplicating line 7 in Listing 1). As expected, correct pointer tag / object tag pairs consistently score a cache hit rate of 100%. Incorrect tag pairs, on the other hand, result in an increasingly lower rate as we increase the number of (failed) checks and thus the contention. Moreover, one check is sufficient to leak a tag match (0.2% cache hit rate difference). The figure also includes results for the Pixel 8 Pro, on which we reproduced the behavior discussed for the S22, except that contention seems lower and our probe gadget can identify the correct tag starting from the contention caused by *two* (rather than one) checks.

In summary, the contention caused by tag mismatches provides attackers with a convenient side channel to determine whether a tag mismatch occurred. Crafting probe gadgets is relatively simple: an attacker needs to trigger the target software vulnerability on a speculative path [29] and, unlike standard (and mitigated) Spectre [44], observe a microarchitectural signal from *any independent* memory operation within the speculation window. On some devices (Pixel 8), additional failed checks within the window may be required, but, since invalid memory accesses are common on speculative paths, this is a relatively minor hurdle for attackers to overcome. Our results provide concrete evidence that tag leakage attacks are possible with easy-to-craft probe gadgets and question the use of probabilistic MTE-based solutions that rely on random tagging as a mitigation—even with lack of brute-forcing capabilities [26]. Furthermore, our findings contradict a recent analysis on MTE by Google, which found no side channels on their tested devices [32].

4. Threat Model

We consider an adversary seeking to exploit an existing spatial memory error in a victim program. We assume attackers can exhaust the tag entropy through classic brute forcing for crash-resistant targets [26], [27], [28] or speculative probing for non-crash-resistant ones [29]. For speculative probing, the standard Spectre [44] threat model applies, with a local attacker mounting cross-privilege (e.g., user-to-kernel or guest-to-host) or in-domain (e.g., JavaScript sandbox) attacks [45]. For the latter, attackers also need to bypass any deployed (browser) timer mitigations [46], for instance by crafting their own high-resolution timers [47], [48], [49], [50] or mounting timerless attacks [51], [52].

We consider overflows (and underflows) within the size of our spatial guards. This includes contiguous overflows and bounded non-contiguous overflows, such as constrained out-of-bounds accesses via type confusion, etc. Attackers can launch their spatial memory attacks not just through buffers on the heap, but also on the stack. In addition, they may attack both confidentiality and integrity (reads and writes). We consider temporal memory errors out of scope and subject of extensive literature on orthogonal defenses [53], [54], [55], [56], [57], [58], [59].

5. StickyTags

As later evidenced by our evaluation, frequent memory tagging (normally done for every allocation/deallocation) is a major performance bottleneck of existing MTE-based solutions. To reduce this overhead, StickyTags decreases the number of times it needs to tag memory, by reorganizing memory into *regions* each containing objects of a particular *size class* (Figure 2). It tags memory at the *first use* of an object slot, allowing the tag to *persist* across the lifetimes of different objects allocated in the same slot.

Our persistent memory tags follow a *deterministic* pattern that assigns tags to slots in round-robin fashion, so for N-bit tags, each tag repeats every 2^N slots. This way, StickyTags protects against buffer under- and overflows bounded by the number of tags times the slot size. Conceptually, each memory slot has two *implicit spatial guards* consisting of the $2^N - 1$ surrounding objects on both sides with different tags. The effective size of the guards depends on the size class S: each object is protected by $(2^N - 1) \times S$ guard bytes. With MTE featuring tags of $N = 4$ bits, this amounts to $15 \times S$. To quantify this: our smallest (16 bytes) and largest (262 KB) size classes provide bi-directional guards of 240 bytes and 3.75 MB, respectively.

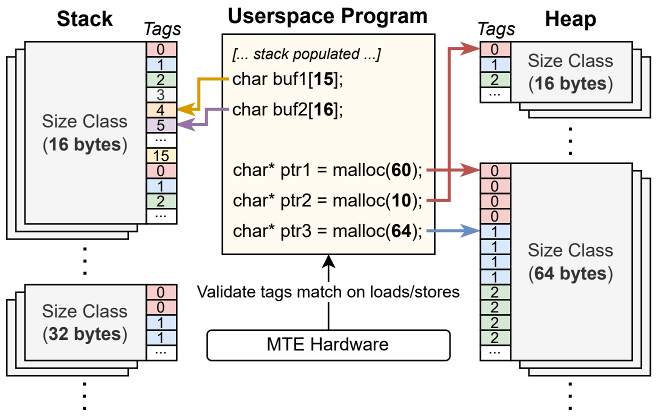

Figure 2: Memory organization in StickyTags. The tags are persistent: they remain in place when new objects reuse the memory.

Within a size class the objects always use the same slot size, hence the tag layout in a region is constant. After an object is deallocated, a new object can reuse the slot while the underlying memory tags remain unchanged. The tags are repeated to match the size class, e.g., 64-byte objects require four consecutive identical tags. On object allocation, we tag the returned object pointer such that it matches the persistent tag of the corresponding memory location. The MTE hardware compares address tags with memory tags and generates an interrupt upon accesses in case of mismatch.

While size classes are common in modern heap allocators [60], [61], and previous work has split the stack into separate regions per *variable type* to combat type confusions [62], [63], to the best of our knowledge the stack has never been divided into size classes. It is precisely this combination of a size class-based (stack and heap) allocator and persistent memory tags that allows StickyTags to deliver much higher performance than existing solutions. Specifically, StickyTags eliminates the need for retagging memory, because of which it performs well on both the stack, where the allocation (and hence retagging) frequency is typically high, and the heap, where large allocations are not uncommon and hence retagging is costly.

5.1. Persistent Memory Tag Initialization

While StickyTags' one-time tag initialization is key to its performance, it is also challenging. A naive solution is to maintain metadata to track whether the memory tag of a slot has already been initialized, and check it at object allocation time—initializing the tags only if needed. However, this strategy would introduce checks and metadata tracking on the fast path, severely impacting performance. Another option is to immediately tag an entire memory area every time a whole region (i.e., stack/heap chunk) is allocated. However, doing so may result in severe *overtagging* for memory that is never used, incurring both runtime and memory overhead. This is especially likely because modern applications and allocators tend to keep large amounts of unused memory around for future use.

To avoid such shortcomings, StickyTags initializes memory tags only once the memory receives *backing*. In particular, StickyTags relies on user-level page fault handling to lazily initialize memory tags: upon accessing a memory page for the first time, the hardware triggers a page fault that StickyTags handles by initializing the predetermined memory tags for the page. For this purpose, it only needs to know the base address and size class of the region containing the page, which it obtains from per-region metadata maintained by the allocator. Using this information, StickyTags determines what tags to apply to the page based on the distance of the current page to the base of the region, because the region always starts with tag zero and tags cycle up deterministically (round-robin) from that point.

5.2. Size Classes

Stack. For the stack, StickyTags allocates one *region* per size class of stack objects. Allocating the stack regions using the heap allocator (described below) allows StickyTags to deduce the size class at runtime when handling page faults by consulting the heap metadata. StickyTags uses stack size classes that are multiples of two, which simplifies pointer tagging, as we will explain in the next section. While there is at most one instance of each size class in a single-threaded application, there can be multiple in the case of multi-threading. StickyTags instruments each unsafe stack allocation in the program (as determined by static analysis [64]) to use the base pointer of the associated stack region instead of the regular stack. The regular stack is still used for return addresses, statically safe allocations, and stack objects in uninstrumented libraries.

Since stack objects are allocated on each function entry, they require a highly efficient replacement scheme. Therefore, StickyTags decides at compile time which stack region pointer (or simply *stack pointer*) to use for each stack object and stores the stack pointer in thread-local storage (TLS). Note that StickyTags creates stack regions only for the size classes that it statically determines to be required. Dynamically-sized stack objects (e.g., calls to `alloca`) cannot benefit from this scheme and, like heap objects, require the allocator to find a region for their size class. Since this is already done for heap objects, StickyTags moves these objects to the heap by transforming them into `malloc` calls. It inserts calls to `free` at the end of the object's lifetime, which it determines using dominance frontiers [65]. In our evaluation we rarely observe unsafe dynamically-sized stack objects. Therefore, moving these to the heap incurs negligible runtime overhead.

Heap. For heap allocations, commodity high-performance memory allocators already provide a suitable organization with size classes. For instance, an allocator such as TCMalloc [60] uses slab allocation to efficiently implement per-thread caching of heap objects. StickyTags piggybacks on these efforts and ensures the allocator uses only size classes that are a multiple of 16 bytes—the MTE tagging granularity. See Appendix A for the exact size classes.

Listing 2 Tagging stack pointers upon allocation.

```
// assuming: 'ptr' is target allocation
region_base = ptr & (~((1 << 24)-1));
distance = ptr - region_base;
jumps = distance >> size_class_power;
tag = jumps & 15;
ptr = ptr | (tag << 56);
```

5.3. Tag Calculation

Whenever StickyTags allocates a memory object, it determines the tag to use for the pointer (i.e., the address) based on the location of the underlying memory. Additionally, upon a page fault, StickyTags applies the appropriate tagging pattern to the faulting page, in accordance to the associated size class. The key insight for tagging is that StickyTags can deterministically *calculate* the correct tags for all allocation pointers and faulting pages such that both correspond to the same tagging layout.

Stack. Since the stack is designed for frequent allocations, it is crucial to optimize *pointer* tagging even with efficient persistent *memory* tags. For our purposes, we considered three possible design options. The first maintains an explicit (per-size-class) *tag pointer* in TLS similar to the stack pointer, which we forward through function calls and increase/decrease accordingly, allowing StickyTags to trivially calculate the next tag to use based on the current tag pointer value. The second option relies instead on the stack pointer to calculate the current (per-size-class) tag value just-in-time. The third option is a hybrid design. Specifically, if we ensure that all stack allocations are performed unconditionally (i.e., we perform *allocation hoisting*), then we can assume complete linearity of the allocations within each size class in a function. As a result, we do not need to recalculate the tag for each allocation (since the tag layout is fixed), and instead need only calculate the first tag in the function for each size class, cache it, and offset subsequent allocations accordingly. With this approach, we effectively calculate and maintain an explicit function-local *tag pointer*.

After inspecting preliminary benchmark results, we quickly discarded the two options based on explicit tag pointer management, as any increased pressure on the register allocators caused by propagating variables proved to undermine performance. Instead, we focused on the second option, optimizing the performance of just-in-time tag calculations as much as possible. Listing 2 presents how StickyTags performs just-in-time pointer tag calculations for stack allocations based on the object address. In particular, by computing the distance of the current address to the base of the region, StickyTags can deduce how many tag cycles fit in this distance, and hence determine the next tag to use.

To optimize the calculation, we align the base of each stack region to 16 MB and limit the size of each region to 16 MB. As a result, we can find the base of a stack region by masking (i.e., cutting down) any arbitrary stack object's pointer to the 16 MB boundary, instead of having to perform a memory load (line 2 of Listing 2). Next, a simple subtraction yields the distance of the object to the base (line 3). We compute how many objects fit in this distance by dividing it by the size class of the region (as a power of two exponent). The computation is efficient: the size class of the allocation (and the corresponding region) is known at compile time, while the stack's size classes are a multiple of two, allowing us to use right shifts rather than divisions (line 4). By knowing how many objects fit before the current one, StickyTags computes the next tag to use by performing a modulo 16 (the tag cycle size) operation, which is optimized to a bitwise `AND` (line 5). The last step applies the tag to the upper bits (56-63 with Arm TBI) of the pointer (line 6).

Now that StickyTags has tagged the pointers of the allocations, it must ensure that the underlying memory follows the same tagging layout. To apply these *persistent* memory tags to the stack upon page faults, it uses an algorithm that closely resembles Listing 2. Starting from a faulting address, it obtains the corresponding region base and size class from the heap metadata (since StickyTags allocates stack regions using the heap allocator) and computes the first tag of the page through the same steps as in Listing 2 (lines 3 to 5). Finally, it applies the memory tags by repeatedly executing the `STG` (store tag) MTE instruction, looping over the page in chunks of the size class, and increasing the tag by one for every object—wrapping around after tag 15.

It is important to note that StickyTags tags *stack memory* upon page faults in a runtime library and hence the inline stack instrumentation (see Listing 2) does not require any MTE instructions. More specifically, StickyTags' stack tagging instructions do not require `STG` to store memory tags, which is instead done by the page fault handler, nor `LDG` to load tags from storage, since the pointer tags are computed based on the stack addresses (and applied using TBI). This comes with the added benefit of providing backwards compatibility with legacy Armv8 devices (supporting TBI, but not MTE). Indeed, if a device does not support MTE, StickyTags can simply not register the page fault handler, thereby making memory tagging fully conditional. In contrast, random tagging solutions such as MemTagSanitizer [20] require unconditional insertion of MTE instructions on the stack, thereby breaking the application binary interface (ABI). As acknowledged by Google, retaining ABI compatibility is crucial to deploy stack tagging in practice—and this is especially the case for systems targeting a wide variety of Arm devices such as Android [66].

Heap. On the heap, StickyTags tags pointers by piggybacking on the existing heap metadata to retrieve the base address and size class of the object's region. In contrast to the stack, the heap uses size classes that are multiples of 16 bytes to avoid excessive memory overhead as well as potential performance penalties due to internal fragmentation of power-of-two heap allocators [16]. Since the allocation patterns on the heap are generally less intensive, we trade off better memory locality for a slightly more expensive tag calculation. The main complication is that some size classes do not fit perfectly in the memory page granularity, because dividing the page size by a size class that is not a multiple of two results in a non-integer object distribution (e.g., $4096/48$). StickyTags addresses this by offsetting the memory tagging initialization accordingly, such that it accounts for memory objects that are partially tagged by a (prior or future) neighboring page fault.

The algorithm for tagging heap pointers follows the same structure as for the stack (Listing 2), with the following minor differences: (1) the region base and size class are determined through a metadata lookup, and (2) the `jumps` calculation (line 4) is a division instead of a right shift (as the size classes are not always a power of two). The result of this division is always an integer, since the offset from the start of the allocation to the region base (`distance`) is guaranteed to be a multiple of the size class.

While StickyTags' lazy tagging strategy reduces the tagging costs for large objects within a region, the one-time tagging and subsequent checks still incur some residual overhead for *huge* heap objects. To reduce the cost of both memory tagging and the checks that MTE performs (only) on tagged memory, StickyTags includes an optimization where huge objects are allocated in separate (guarded) memory regions and remain untagged. Huge object allocations (>262 KB) already constitute a special case in TCMalloc: each resides in its own dedicated region (also called a *span*). Since such huge objects do not have any neighboring objects inside the region, we can simply spatially fence their regions using inaccessible *guard pages* [10], [38]. Similar optimizations are also present in modern allocators, for instance in the Scudo allocator (Android), which does not tag objects bigger than a threshold (64 KB on Android, 131 KB by default) and instead relies on guard pages [67]. While Scudo does this to avoid (frequently) tagging large regions of memory [67], StickyTags primarily aims to reduce the residual *tag checking* overhead, since huge objects can reuse previously tagged memory when available.

6. Persistent Spatial Guards on x86

In this section, we show that the principle of *persistent* spatial guards along with one-time initialization extrapolates well to the x86 architecture, even though x86 does not have memory tagging capabilities. In the absence of MTE, we rely on providing spatial memory error mitigation through more traditional compiler-inserted checks and (padded) explicit spatial guards, commonly called *redzones*. As before, we reorganize the address space into regions containing objects of the same size class, but now additionally we place redzones at fixed intervals within each region (see Figure 3). By doing so we can optimize redzone management, even eliminating the need for out-of-bound metadata such as a shadow memory. Compared to the implicit spatial guards of MTE, where tags are associated separately from the memory, in the case of x86 the persistent guards are *explicit*, as they weave between objects to spatially separate them. As a result, each object size (and hence the size class) is inflated by including the redzone on its right side. Note that the redzone on the left is always the right redzone of the

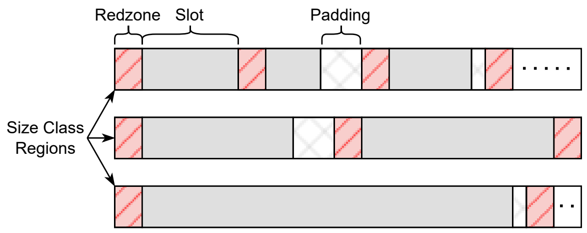

Figure 3: On x86: Memory is organized in size classes, each containing equally sized slots of a given size class, interleaved by redzones. Objects are padded to fit the slot.

Algorithm 1 High-level bounds check. The actual implementation reuses the loaded value for reads instead of calling LOAD_BYTE, and unrolls the loop for up to 8 fast checks. REDZONE_DISTANCE is different for the heap and the stack.

```
$offset \leftarrow 0$
while $offset < num\_accessed\_bytes$ do
    if $\text{LOAD\_BYTE}(address + offset) = guard\_value$ then
        $region \leftarrow \text{GET\_REGION}(address)$
        $sc \leftarrow \text{GET\_SIZECLASS}(region)$
        $dist \leftarrow \text{REDZONE\_DISTANCE}(region, address, sc)$
        if $dist < 0 \lor num\_accessed\_bytes > dist$ then
            RAISE_ERROR(out of bounds)
    $offset \leftarrow offset + redzone\_size$
```

previous object, except for the first object, for which the left redzone is created along with the region initialization.

Existing redzoning solutions (e.g., AddressSanitizer [3]) use a shadow memory to record which memory is accessible, storing one bit of accessibility information for each byte of application memory. This fine-grained metadata management is expensive in both runtime and memory usage, but is necessary for a design in which a memory location containing an object (or padding bytes) can later be used to store a redzone, and vice-versa. Especially if we want the redzones to be reasonably large (e.g., 256 bytes) to approximate the lower-bound security guarantees of our MTE solution, the frequent construction and destruction of redzones is expensive. In contrast, persistent redzones entirely eliminate the need for a shadow memory. The base address and size class of the containing region are known for each memory location, and can be used to determine whether a pointer points to a redzone. Additionally, since the redzones only need to be initialized with guard values (see below) once, we avoid redzone creation becoming a severe performance bottleneck.

Accessibility Checks. We insert accessibility checks before each memory read/write. We leverage redzone-aware static analysis at the compiler level to skip unnecessary checks and merge checks of adjacent memory ranges, similar to state-of-the-art compiler optimizations as seen in related work [36]. The checks consult the metadata of the region containing the accessed memory address, and use its base and size class to determine whether the pointer falls within a redzone. This introduces three memory loads (metadata, region base, and size class) and some arithmetic/branching operations which together cause high runtime overhead. We therefore use a *guard value*, as done by LBC [4], to quickly filter benign memory accesses: a single byte value that is stored in each redzone byte. To avoid writing guard values to a redzone twice, we lazily initialize guard values upon a page fault.

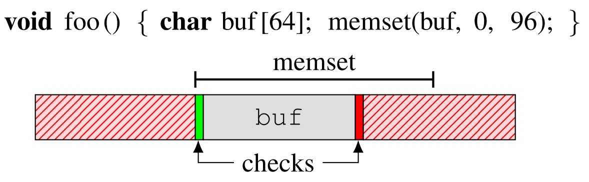

Figure 4: Fast checks on a stack object with 64-byte redzones. The access spans more than a redzone and is checked by fast-checking every 64 bytes, failing on the second check.

Each accessibility check first performs a "fast" check, comparing the byte value at the accessed location to the guard value, only resorting to a regular "slow" check based on region metadata if the values match. Algorithm 1 shows this in detail. Because a memory access can access more bytes than fit in a redzone, one fast check is emitted for each $redzone\_size$ bytes of the access (see Figure 4). Otherwise, an attacker might abuse a large memory access that starts before the redzone and thus does not contain the guard value in the first byte, but crosses the entire redzone to access the next object slot. As a result, for very large memory accesses, the performance gain of doing fast checks is overcome by the number of fast checks that are needed. Hence, we only emit fast checks if the required number is still beneficial for performance (up to eight, determined experimentally). To optimally benefit from fast checks, the guard value should be uncommon in regular application memory.

In summary, on x86 we use explicit persistent spatial guards (i.e., redzones) that allow us to scale to relatively large guard sizes, since we avoid (frequent) redzone creation/destruction becoming a bottleneck by only having to initialize the guards once. Due to a lack of hardware assistance, we rely on compiler-inserted checks to validate memory accesses. As we will show in our evaluation, (large) explicit persistent spatial guards are indeed significantly more efficient than their non-persistent counterparts. However, the use of compiler checks still results in residual overheads unsuitable for production use.

7. Implementation

We have implemented our prototypes on Linux on top of the LLVM [68] compiler infrastructure and the TCMalloc [60] memory allocator. We apply our compiler passes after link-time optimizations. This ensures that inserted instrumentation does not interfere with any analysis during optimizations. Our implementations of implicit guards (MTE tags) and explicit guards (redzones on x86) share the memory reorganization and page fault handling logic, and mainly differ with respect to the application of guards.

Size Classes. We use TCMalloc [60] as the basis for our allocator. TCMalloc organizes objects into size classes by default. A compiler pass, based on LLVM's internal SafeStack [69], creates size classes for the stack and also applies the pointer tags. We modified the pass to support one "unsafe" stack per size class. We rely on TCMalloc to allocate memory areas for the stack regions and to determine object size classes at runtime. Large stack objects that do not fit in any of the precomputed size classes are assigned a new, unique size class. This rarely occurs in practice.

Page Fault Handling. A dedicated poller thread catches page faults in user mode using Linux' `userfaultfd` system call, initializing memory tags and redzones on x86 in the faulting page when it is accessed for the first time. We derive where to apply the tags and redzones from per-region metadata maintained by TCMalloc. Execution of the poller/application threads is interleaved: during page fault handling, the faulting application thread waits for the poller thread to finish. Because only one thread runs at a time, there is no offloaded overhead on a separate core.

8. Evaluation

In this section, we evaluate the performance and security of StickyTags. We measure the runtime and memory overhead using the SPEC CPU2006 and CPU2017 benchmarking suites, and compare this to state-of-the-art solutions. To quantify the security impact of StickyTags, we use the Juliet Test Suite [70], existing CVEs [17], and a type confusion vulnerability analysis. Additionally, we investigate the performance accuracy of existing MTE *analogs*. For additional information and experiments we refer to the Appendix.

8.1. Experimental Setup

For the core of our experiments we use a rooted Google Pixel 8 Pro with MTE support. The device contains 12 GB RAM and runs a chroot Debian 12 distribution. We further make use of a Samsung Galaxy S22 (8 GB RAM) and a MacBook Pro (Apple M2, 16 GB RAM, Asahi Linux 6.3, Debian 12). All singlethreaded benchmarks are pinned to a single core. Each measurement reported is the median of five iterations of the same program (using the reference workload for SPEC CPU). For the baseline, we enabled link-time optimizations and used an unmodified TCMalloc as the memory allocator. Note that default TCMalloc has an average speedup of 11.7% compared to the default (non-MTE) Scudo allocator (and 8% to the default GNU heap allocator) and up to 73% for a single benchmark (471.omnetpp), while consuming 3% more memory on average. Unfortunately, we have to exclude SPEC CPU2017 from most of our experiments, because the baseline runs out of memory. The system requirements for SPECspeed 2017 state 16 GB of physical memory, and the Pixel 8 and S22 do not meet this.

MTE Hardware. Recent work concerning MTE uses *analogs* to approximate the overhead of memory tagging since MTE hardware was not widely available [17], [19], [34], [71]. In our evaluation, we measured the performance of StickyTags with actual MTE hardware. First, the Google Pixel 8 (Tensor G3, android14-5.15) supports MTE and allows the feature to be enabled through its developer options. Second, by rooting a Samsung S22 and deploying a custom Exynos (Linux 5.10) kernel, we manage to activate its MTE hardware by explicitly ignoring the `nomte` kernel parameter. However, on the S22 the locked-down boot monitor does not reserve backing (physical) memory to store the memory tags for uncached data. Consequently, tagging memory works as expected, reading the target data into the cache and setting the memory tags in the cache hierarchy accordingly. However, as soon as the data leaves the cache, the memory tags cannot be swapped to backing memory and effectively vanish. Subsequent accesses to the memory cause a segmentation fault by the MTE checks (since the pointer still has the tag). While the Pixel 8 serves as the main target for evaluation (with completely functional MTE), the S22 nonetheless provides another data point as MTE implementation, allowing us to gain further insights into the performance of MTE's memory tagging and our design. Additionally, to paint a complete picture with respect to existing work, we also conduct performance experiments with the existing MTE *analogs* on the S22 (and Apple M2).

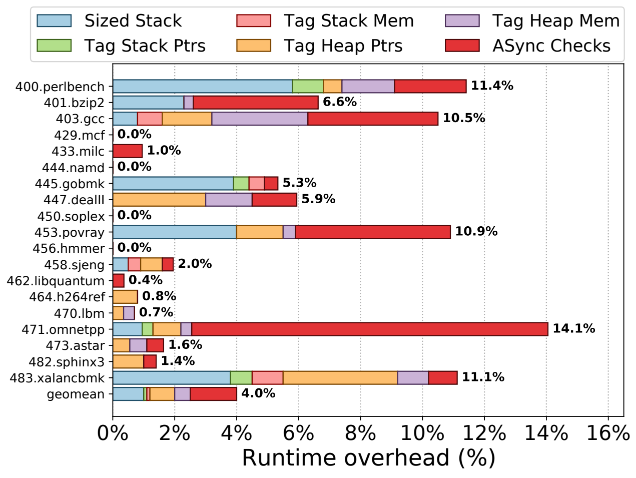

Figure 5: Runtime overhead buildup of different components of StickyTags on SPEC CPU2006 using MTE hardware (Pixel 8).

8.2. Performance Buildup

For our performance evaluation we configured MTE on the Pixel 8 to perform *asynchronous* checks, which Arm recommends for production usage [72]. Figure 5 displays the runtime overhead of StickyTags on each individual benchmark of the SPEC CPU2006 suite. In total, StickyTags incurs a geomean runtime overhead of 4.0%. The figure breaks down this overhead into six distinct components: using size classes on the stack (1.0%), tagging stack pointers (0.1%), tagging stack memory (0.1%), tagging heap pointers (0.8%),

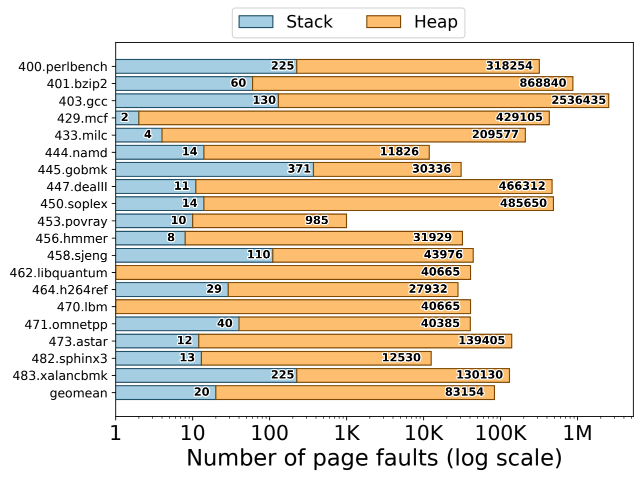

Figure 6: Number of (4 KB) page faults in SPEC CPU2006.

tagging heap memory (0.5%), and the asynchronous checks (1.5%). Note that the heap and stack memory tagging components constitute the overhead of using `userfaultfd` to one-time initialize the persistent tags upon page faults.

From the overhead buildup figure we can conclude that tagging memory is cheap (see geomean bar), which is a logical consequence from our persistent tags design. Moreover, we see that using a stack with size classes can be the largest contributor of overhead in some benchmarks (400.perlbench, 445.gobmk), while overall the slowdown is modest. For the CPU2006 benchmarks, we create an average (geomean) of 7 stack regions (each dedicated to a distinct size class). Then, on average, a maximum of 4 are used per function. Note that these numbers are statically computed, meaning that some stack regions may be unused at runtime depending on the execution path. Excluding the checks, the remaining overhead originates from tagging pointers on the stack and the heap, which correlates with the memory intensity of the applications. For instance, 447.dealII and 483.xalancbmk are known to be relatively heap-intensive, and hence these accordingly experience more overhead from tagging heap pointers. As touched upon before, we may choose to only use size classes that are a multiple of two on the heap, which accelerates the pointer tag calculation, but this may come with other drawbacks such as memory fragmentation.

We observe that the overhead implications of enabling asynchronous MTE checks are low but non-negligible. On average, the checks comprise the most significant overhead component, however this is not unexpected, because StickyTags focuses on eliminating tagging overhead. Moreover, the checks clearly show up as the dominant overhead factor in multiple programs. The 471.omnetpp benchmark stands out the most, where the checks incur more than 11% overhead. Upon further inspection with `perf` [73], we found that for this benchmark the CPU experiences a 19% increase in stalled cycles in the backend, which is likely the result of the asynchronous checks creating additional contention.

To better understand the characteristics of our memory tagging design, we measured the number of page faults that occur at runtime for both the heap and the stack. Figure 6 displays the results of this experiment. Looking at the aggregate numbers in the figure, it is clear that the stack experiences much fewer page faults than the heap, with the geomeans being 20 and 83,154, respectively. Moreover, we see a logical correlation between the overhead of tagging heap memory being relatively expensive for 403.gcc and the large number of heap page faults for this benchmark. In contrast, 401.bzip2 also experiences a relatively large number of heap page faults, but the heap objects are effectively all huge (>262 KB), which means our huge objects optimization leaves them untagged. Without this optimization, the runtime overhead of 401.bzip2 grows from 6.6% to 7.7%. Additionally, the low number of page faults for the stack highlights the efficacy of our persistent tagging design on the stack. On average, only 20 memory pages need to be tagged throughout the entire lifetime of the evaluated applications, regardless of the intensity of their allocation patterns.

System	Heap	Stack	Both
StickyTags	3.1%	1.2%	4.0%
MemTagSan + Scudo	5.8%	15.2%	20.2%

TABLE 1: Runtime overhead comparison between StickyTags, MemTagSanitizer, and Scudo using SPEC CPU2006 (Pixel 8).

8.3. Comparison to the State of the Art

In order to put the performance of StickyTags into perspective, we compared its overhead to state-of-the-art systems. We measured runtime and memory overhead using the SPEC CPU2006 benchmarking suite and evaluated against LLVM's MemTagSanitizer (stack) and the Scudo heap allocator. The primarily probabilistic Scudo allocator assigns a random tag to every heap allocation and retags the memory upon deallocation. Additionally, Scudo guarantees neighboring objects to have different tags by employing an odd-even tag masking pattern. MemTagSanitizer tags every stack allocation with a "random" tag at the start of its lifetime and resets the tag at the end of it. To avoid scalability issues with random tags requiring an extra live register for each variable, the tags are not completely random. Instead, MemTagSanitizer generates a random base tag for each function, with the following stack variables receiving a tag derived from the base tag. Note that the primary use case of MemTagSanitizer is deployment in production binaries [20]. Unfortunately, MemTagSanitizer causes false positive tag mismatches due to untagged pointers accessing tagged stack memory. Therefore, for the faulting programs (400.perlbench and 471.omnetpp) we modified MemTagSanitizer to only use tag zero to avoid these non-trivial crashes.

Table 1 shows the geomean runtime overhead of StickyTags, MemTagSanitizer, and Scudo on the SPEC CPU2006 suite. The table contains the isolated heap and stack overhead, as well as the combination of both. Most notably, we observe that MemTagSanitizer's stack instrumentation incurs 15.2% overhead, while StickyTags' stack overhead is more than twelve times lower at 1.2%. Moreover, we see that StickyTags incurs 3.1% overhead for protecting the heap, which is nearly half of the 5.8% overhead of Scudo. When MemTagSanitizer is combined with Scudo to protect both the stack and the heap, the overhead becomes a combined total of 20.2%. As a result, with 4.0% overhead StickyTags is over five times faster at protecting both the heap and stack compared to the existing state-of-the-art solutions.

Regarding memory consumption, we measured that StickyTags increases memory usage (i.e., the RSS–resident set size) by 15.7%, while Scudo and MemTagSanitizer only slightly (1%) increase memory consumption. StickyTags raises memory consumption by creating size classes on the stack and through internal modifications to TCMalloc, for instance to adhere to the 16 byte MTE granularity for object sizes and to support proper memory separation for the huge objects optimization. While the relative memory consumption of StickyTags is higher compared to enabling MTE with existing allocators, the absolute peak RSS value for StickyTags is lower for a few benchmarks, including two heap-intensive workloads: 400.perlbench and 471.omnetpp.

Additionally, we conducted an experiment to obtain overhead results on SPEC CPU2017, for which the Pixel 8 lacks the prerequisite physical memory. Therefore, we make use of MTE analogs on a MacBook Pro (containing an Apple M2 CPU with Arm Top-Byte Ignore for pointer tagging) to benchmark the memory tagging costs on SPECspeed 2017. The details concerning this experiment are displayed in Section A in the Appendix. The takeaway from this experiment aligns with our results in Table 1, where the tagging performance of StickyTags is greater than existing non-persistent tagging techniques. For instance, using analogs on SPEC CPU2017 we measure an overhead of 8.8% for MemTagSanitizer, and only 1.0% for the stack instrumentation of StickyTags.

Next, we put StickyTags in perspective to related research, such as software-only systems like Delta Pointers [12], TailCheck [10], Low Fat Pointers [14], [35], and MEDS [74], as well as a hardware-assisted solution like PACMem [6]. These existing techniques still report overheads hovering between 30% and 55% (or higher), which leads to the conclusion that the overhead is too high to be considered for a post-deployment mitigation. Although we note some systems also provide temporal error protection, their runtime overhead remains impractically high for live deployment. Additionally, Color My World [71] is an MTE-based solution that relies on deterministic stack tagging, however this technique still incurs a reported runtime overhead of 13.6% (with MTE analogs) on SPEC CPU2017 for protecting the stack, which is more than the 8.8% we measured for MemTagSanitizer, and significantly more than the 1.0% stack overhead of StickyTags (see Appendix A).

MemTagSanitizer Overhead Analysis. Although MemTagSanitizer incurs a sizeable runtime overhead on several CPU2006 benchmarks (41% on 453.povray, 36% on 400.perlbench, etc.), the benchmark that stands out the most is the (chess variants playing) 458.sjeng program with an overhead of 267% (3.67x), even with checks disabled. In order to better understand the source of the overhead and the potential bottlenecks surrounding MTE, we used `perf` [73] to profile the 458.sjeng binary. We discovered that the program spends almost all of its execution time inside a recursive function called `search`. Aside from local variables that are statically proven to be safe, this function contains three arrays: one struct array of 512 elements, where the size of the struct is 24 bytes (6 integers), and two 32-bit integer arrays of 512 elements each. In total, this results in exactly 16 KB of 'unsafe' stack data. As a consequence of MemTagSanitizer's per-request memory tagging design, this entire region needs to be (un)tagged for every recursive call. In contrast, when the recursive call pattern re-iterates through previously used call depth, StickyTags only needs to calculate which tag to assign to the pointers of the stack allocations, while the underlying memory tags remain persistent. As a result, StickyTags successfully avoids this memory tagging bottleneck and only incurs a small runtime overhead of 2% on 458.sjeng. Additionally, we further confirmed that *tagging memory* clearly dominates the overhead of random tagging by measuring a 14.5% geomean runtime overhead of MemTagSanitizer on SPEC CPU2006 with the *checks disabled*. Tagging memory therefore makes up 95% of the total 15.2% (see Table 1) random tagging runtime overhead.

[Type] System	**Stack**	**Stack + Heap**
`[MTE]` MemTagSanitizer + TC	22.8%	25.7%
`[ANL]` MemTagSanitizer + TC	14.0%	16.1%
`[MTE]` StickyTags	1.4%	2.7%
`[ANL]` StickyTags	1.4%	2.7%

TABLE 2: SPEC CPU2006 geomean runtime overhead summary of memory tagging costs on a Samsung S22 for StickyTags and MemTagSanitizer using MTE hardware. `ANL` means MTE analogs, `TC` stands for TCMalloc with per-request (non-persistent) tagging.

8.4. Memory Tagging Performance

In this section, we specifically evaluate the costs of *tagging memory*, which is the bottleneck we aim to relieve with StickyTags. Therefore, we exclude the overhead of tag *checks*, since this concerns an overhead that is orthogonal to our design. To this end, we use the Samsung S22 for these experiments, which is capable of tagging memory (but not performing checks; see Section 8.1), and therefore provides another data point for tagging costs. MTE provides the Tag Check Override instruction and we confirmed that the cost of memory tagging remains the same even when the checks are disabled. We also confirmed that applying tags forces the memory into the cache, which provides further evidence that the S22 is representative for the cost of setting tags.

Since we are interested in quantifying the performance of *persistent* memory tags, we conducted the following experiments using SPEC CPU2006. For our main configuration we measured the *stack* tagging overhead of StickyTags and MemTagSanitizer, because the stack concerns relatively frequent tagging behavior, and hence should reveal potential tagging bottlenecks. Next, we measured the performance difference between persistent and non-persistent tagging on the *heap* by evaluating against the TCMalloc heap allocator with a similar tagging pattern but with no one-time memory tag initialization. We assign matching pointer and memory tags upon each heap allocation and cycle through the tags deterministically in a round-robin fashion. Lastly, we investigated the overhead accuracy of existing MTE analogs by repeating both of these experiments using analogs.

Table 2 displays the geomean runtime overhead of MemTagSanitizer and StickyTags on SPEC CPU2006. From these results we conclude that StickyTags provides significantly higher tagging performance than MemTagSanitizer. Specifically, StickyTags incurs a runtime overhead of 1.4% for tagging the stack, while MemTagSanitizer is over 16 times slower with 22.8% overhead. MemTagSanitizer is still heavily impacted by the 458.sjeng binary, incurring a slowdown of over 6x on the S22. Next, we consider the additional runtime overhead when including persistent and non-persistent tagging on the heap. For StickyTags, we measure a 2.7% overhead for tagging both the stack and the heap, attributing 1.3% to the heap. Note that this corresponds to the tagging overhead we measured on the Pixel 8 (see Figure 5). For the non-persistent tagging design (MemTagSanitizer + TCMalloc), the combined overhead is 25.7%, with an increase of 2.9% caused by the heap. These results indicate that persistent tagging is approximately twice as fast on the heap compared to its non-persistent counterpart.

Next, we examined how representative MTE analogs are for the overhead of memory tagging through a direct comparison. We first point out that both tagging systems are nearly twice as slow on MTE hardware compared to the MTE analogs on the Apple M2 (see Table 4 in the Appendix). However, we measure that the overhead of MemTagSanitizer (plus the heap) using MTE analogs is nearly equivalent across the two devices: 16.1% on the S22 vs. 15.8% on the M2. Since the overhead we measure with MTE hardware (25.7%) is notably higher, our results suggest that the overhead approximation of the MTE analogs is too optimistic for this particular MTE implementation. We also point out that the MTE analogs currently do not incorporate the overhead of *checks*, which we measured are non-negligible (see Figure 5). For StickyTags the overhead with MTE hardware and analogs is equivalent on the S22, but this is not unexpected since most of StickyTags' overhead originates from sources that are not tagging memory (Figure 5), hence switching to analogs has a smaller impact.

In conclusion, StickyTags can successfully provide performance that is suitable for deployment on live systems. On the Samsung S22 StickyTags introduces an average memory tagging overhead of 2.7% for the stack and the heap combined. When including the tag checks, we measured a 4.0% overhead on the Pixel 8. Our evaluation shows that StickyTags improves tagging performance both on the heap and the stack, with the stack being the more significant component. It is expected that the stack is the most susceptible to MTE overhead [33] and therefore also permits the most room for improvement, due to the inherently high allocation frequency. Furthermore, since memory tagging does not require explicit interleaving to introduce spatial guards, the memory overhead of StickyTags, but also of MTE solutions in general, is modest overall.

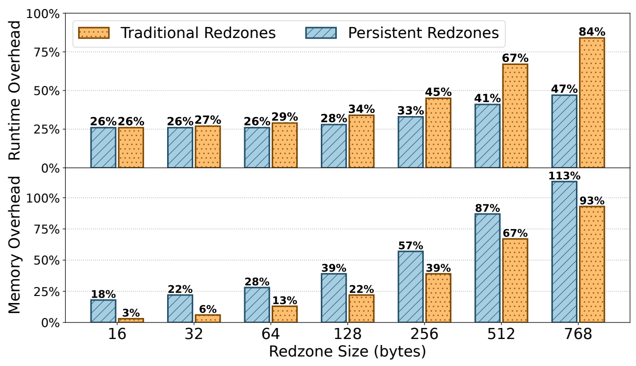

Figure 7: Geomean overhead on SPEC CPU2006 of increasing redzone size for traditional and persistent (one-time) initialization.

8.5. Generalizability to x86

In this section, we evaluate the overhead of our design generalized to x86. Recall this involves a software-only design that relies on compiler-based checks and explicit spatial guards. For our experiments, we make use of an Intel Xeon E5-2630v3 machine with 16 cores running CentOS 7.4.1708. We select the guard value `223` to fill the redzone bytes for performing fast checks. We find this value to be suitable by analyzing SPEC CPU2006's memory accesses, and determine it consistently occurs sparsely across the different programs. Reasons for this include: it is not a printable ASCII character, it is not a power of two, and it is a prime number. We refer to Figure 9 in the Appendix for more details on the prevalence of different guard values.

Since we are interested in to what degree we can scale redzones to match the implicit spatial guards of MTE, we evaluate our design with different redzone size configurations. For reference, the first three size classes of StickyTags on the heap (16, 32, and 48 bytes) result in implicit guards of 240, 480, and 720 bytes. Additionally, we also investigate to what extent our design of *persistent* guards provides performance benefits, compared to the traditional method of initializing redzones upon each allocation. We optimize this traditional method by only initializing the *right* redzone of each object, since the left redzone is also the right redzone of the preceding slot and is therefore already initialized (in our memory layout) when the current slot is allocated. Note that for the traditional redzones we do not incorporate the costs of destructing redzones upon deallocation (to derive the most efficient baseline for comparison), which is normally necessary for non-persistent redzones, for example in ASan.

Figure 7 displays the geomean runtime and memory overhead of our design on SPEC CPU2006 for an increasing redzone size, both with and without persistent redzones. For a redzone size of 16 bytes, we reach a runtime overhead as low as 25%, for both redzone initialization types. Although this overhead may appear relatively high, especially compared to StickyTags with MTE, this is a significant speedup compared to ASan, for which we measured an 86% runtime overhead when configured to detect only spatial memory errors. Considering other state-of-the-art spatial memory error mitigation techniques, we measured an overhead of 67% for Low-Fat Pointers [14], and 35% for Delta Pointers [12] (which is limited to protecting only *upper* bounds).

Regarding scalability, we observe that for 16-byte redzones, runtime performance does not benefit from persistent redzones because repeatedly initializing a redzone is very efficient—, requiring a single instruction that writes a 128-bit value from an SSE register. As the redzone size increases, however, the runtime overhead increases significantly less with persistent redzones. Indeed, up to 64 bytes, the overhead does not increase at all. Even at 768 bytes, the runtime overhead is only 47%—versus 84% with lazy initialization disabled, an improvement of 44%, as seen in Figure 7.

Naturally, the memory overhead of explicit guards is more pronounced than implicit guards. Starting with a redzone size of 16 bytes, we observe a memory overhead of 18% when using persistent redzones and 3% using traditional redzones. The memory overhead then scales upward with the increase in redzone size, up to 113% and 93% when using redzones of 768 bytes. The memory overhead for persistent redzones is raised due to the potential overprovisioning of redzones at the page granularity. Additionally, the instrumentation also protects global variables, which further increases the overall memory footprint of the redzones.

In conclusion, we show that our design of persistent spatial guards can translate from implicit guards with a memory tagging backend to explicit guards through more traditional redzones and compiler-based checks. The resulting runtime overhead is significantly lower than ASan and other existing compiler-based techniques, and we also highlight the performance gain with respect to traditional redzone initialization. Furthermore, we show that this design can scale towards large redzone sizes and does so more efficiently than traditional redzone initialization, by nearly halving the runtime overhead when using redzones of size 768. Unfortunately, unlike our MTE solution, the residual runtime overheads are still too high for practical adoption.

8.6. Security

The spatial security guarantees of StickyTags are generally comparable to those of redzoning solutions [3], [4], [36] for a given spatial guard size. Both techniques mitigate all contiguous buffer overflows (and underflows) and bounded non-contiguous out-of-bounds accesses up to the guard size. The main difference is that typical detection systems such as ASan [3] detect (non-exploitable) accesses to padding bytes within the object memory slot, while StickyTags' mitigation-focused checks do not. Furthermore, our tagging layout is deterministic and therefore fully predictable. For protection, we do not rely on the attacker being unbeknownst to the tag used (prone to tag leakage attacks), but instead on the relatively large spatial guards. Whereas in probabilistic solutions neighboring objects may share the same tag (sometimes one neighbor is guaranteed to have a different tag), we guarantee that the next fifteen objects cannot be accessed using the tagged pointer of the current object.

Description (CWE)	Total	Detected	Mitigated
Stack buffer overflow (121)	69	56 (81%)	69 (100%)
Heap buffer overflow (122)	69	56 (81%)	69 (100%)
Buffer underwrite (124)	21	21 (100%)	21 (100%)
Buffer overread (126)	13	12 (92%)	13 (100%)
Buffer underread (127)	21	21 (100%)	21 (100%)

TABLE 3: Juliet Test Suite detection and mitigation results.

Juliet Test Suite. We use the NIST Juliet Test Suite (v1.3) [70] to show the vulnerability detection and mitigation efficacy of StickyTags. This test suite consists of a large number of programs containing memory safety issues. We limit ourselves to the bug categories concerning spatial errors. Furthermore, we deduplicate the test cases (since many are equivalent at runtime), and we omit test cases that do not (deterministically) contain an error, such as bugs that rely on `rand` or 32-bit systems.

Table 3 displays the total number of test cases for each category along with the number of bugs detected and mitigated by StickyTags. Overall, we observe that StickyTags can *detect* most of the bugs and *mitigate* all of them. The cases that go undetected all display the same erroneous behavior: they contain a bounded overflow that lands in the padding bytes as a result of StickyTags respecting the 16-byte MTE granularity for object sizes. For instance, one bug concerns a 10 bytes allocation followed by an off-by-1 access. Since StickyTags allocates and tags 16 bytes of memory for this object, the out-of-bounds access lands in the padding bytes, thereby mitigating (although not detecting) the overflow since it cannot corrupt a neighboring object.

CVE Analysis. To further evaluate the real-world security guarantees of StickyTags, we gathered the CVEs used in the security evaluation of MTSan [17]—a recently published MTE-based sanitizer—and show that StickyTags can mitigate these test cases. We selected all relevant (and reproducible) cases: heap and stack buffer overflows, resulting in a total of eight distinct CVEs (see Table 5 in the Appendix for details). When executing the test cases on the Pixel 8, StickyTags successfully detects (i.e., observes a tag mismatch and causes a segmentation fault) the spatial errors in all eight CVEs, which we confirm by obtaining a bug stacktrace that is identical to the ground truth provided by ASan [3]. To assess whether StickyTags also fully mitigates the exploitation of the CVEs (i.e., can detect the underlying vulnerability no matter what out-of-bounds offset the attacker can force the program to use), we resorted to manual code inspection. Our analysis revealed that five of the bugs concern linear overflows, and two are bounded non-linear overflows with a constant offset on an uncontrollable pointer—all of which StickyTags completely neutralizes. We

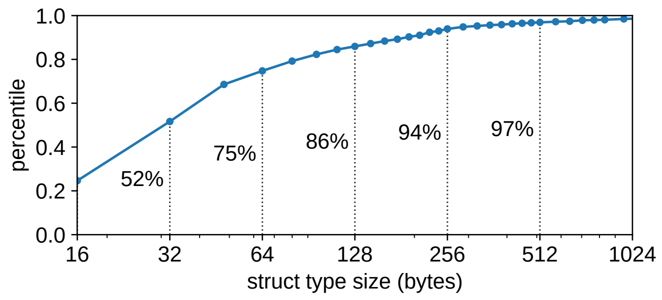

Figure 8: Distribution of struct type sizes in SPEC CPU2006 rounded up to the nearest power of two (on a log scale). Part of the graph is omitted for readability on the right side, containing a small number of very large object sizes up to 100 KB.

could not easily ascertain the set of possible out-of-bounds offsets for the remaining CVE. Hence, we conservatively ascribe it to the non-linear, unbounded overflow category, which StickyTags cannot deterministically mitigate.

Type Confusion Vulnerabilities. Recall StickyTags introduces spatial guards of size $(2^N - 1) \times S$ bytes. To further quantify the effectiveness of StickyTags' protection, we consider bounded non-contiguous overflows caused by type confusion bugs. StickyTags can protect against non-contiguous bugs caused by type confusion as long as the guard size is at least as big as the size difference between the confused structure types. When this is the case, any member of the larger object type will either reside in the smaller object type (i.e., no out-of-bounds primitive) or in a neighboring mismatched tag.

Figure 8 shows the sizes of all structure types in SPEC CPU2006 rounded up to the next multiple of two. To easily reason over the attack surface reduction, we focus on the worst-case scenario: an object of the smallest size class (16 bytes) being confused with larger object types. As shown in the figure, even by focusing only on the smallest size class with guards of 240 bytes, StickyTags protects against type confusion for 94% of all object sizes (i.e., those up to 256 bytes). In comparison, ASan's default configuration of 32-byte redzones handles around 52% of object sizes. Unlike ASan, our design on x86 can also scalably support large (e.g., 240-byte) redzones with realistic overheads.

8.7. Beyond Spatial Memory Errors

Finally, we also briefly study more holistic designs that include use-after-free (UAF) protection. One option is to enable orthogonal measures for UAF, such as MarkUs [75], DangZero [54], or FFmalloc [57]. Alternatively, in more favorable threat models, where brute forcing and speculative probing capabilities are unavailable, defenders may wish to switch to random tagging on the heap. In such a scenario, combining StickyTags on the stack with random tagging on the heap can provide a balance between runtime overhead and (limited) temporal security guarantees. We implemented this strategy by combining StickyTags' stack tagging with random tagging support in TCMalloc. To implement the latter, we inserted a random tag on every heap allocation and changed the tag to another (distinct) random tag upon deallocation (for UAF). For this hybrid deterministic/random tagging design, we measured a geomean runtime overhead of 7.5% on SPEC CPU2006. We also confirmed this setup successfully detects the UAF test cases of the Juliet Test Suite (CWE416). Compared to the performance results presented in Table 1, we observe that, thanks to StickyTags' efficient stack tagging strategy, the hybrid design remains significantly faster than Scudo with MemTagSanitizer (20.2%), but it is also notably slower than default StickyTags (4.0%).

9. Limitations

Currently, neither MemTagSanitizer nor StickyTags implements tagging for global variables, although this is not a fundamental limitation. Engineering efforts in this direction have already been discussed concerning MemTagSanitizer [76]. For a direct comparison between the two solutions, we decided against implementing support for globals for StickyTags. Nonetheless, to show it is feasible to support globals in a persistent guard design, we did implement support for globals variables in our x86 implementation—which introduced no noticeable performance overhead. This is done at compile time by moving each variable into a newly inserted global array containing slots for its size class.

In addition, StickyTags uses all the MTE tags for spatial memory error mitigation and spares no tags for temporal errors. This is to maximize the size of our guards, but also because the potential temporal guarantees are not strong to begin with, as memory massaging allows attackers to exhaust the tag entropy through repeated memory reuse—dashing any hope to use tags for deterministic temporal mitigations. As discussed earlier, even random tags are insufficient as attackers can probe for matching pointer / memory tags using classic brute forcing or speculative probing. To overcome this limitation, one can complement StickyTags with temporal error mitigations [54], [56], [57], [58], [75]. Alternatively, for weaker threat models (i.e., no tag entropy exhaustion possible), one can consider a hybrid deterministic/random tagging solution that enhances StickyTags with (some) heap use-after-free protection (Section 8.7).

Finally, StickyTags also shares all the limitations of MTE-based and SafeStack-based solutions. For instance, for MTE we need to reserve the upper pointer bits for the pointer tag. This may introduce compatibility problems with MTE-unaware applications that implement their own custom pointer tagging. On the SafeStack side, known compatibility limitations are with applications relying on low-level stack manipulations. Moreover, compiling dynamic libraries with SafeStack is still not supported [69]. None of these limitations are fundamental, but addressing them requires engineering effort and/or application code changes.

10. Related work

Existing MTE Solutions. Most existing MTE solutions rely on randomly generated tags for their inner workings. The main exception is Color My World [71], which performs extensive static analysis to deterministically protect the stack through tag forgery prevention. However, this design results in a reported overhead that is higher even than what we measured for MemTagSanitizer and is hence unsuitable for production use. MemTagSanitizer [20] also features a deterministic component, in that it generates a base tag for every stack frame and then cycles upwards deterministically from that point. Since the base tag is random, MemTagSanitizer attains only deterministic intra-stack frame protection. The security guarantees remain probabilistic across frames and can be breached by an attacker armed with stack massaging.

The remaining MTE solutions rely largely on random tagging. More specifically, the SLUB allocator in KASAN [22], glibc's allocator [21], and PartitionAlloc [24] are all completely probabilistic. Nonetheless, other solutions feature partially deterministic behavior. Scudo [23] places 16 bytes of tag zero (reserved) before and after allocations, covering determinism up to off-by-16 bytes, and also features an odd-even tag masking mode to ensure that adjacent objects have different tags, hence providing deterministic off-by-1 object detection. Similarly, MTSan [17] ensures that neighboring objects have different tags. In StickyTags, we provide completely deterministic tag assignment, thereby creating large implicit spatial guards bounded only by the MTE tag entropy. The resulting design incurs low runtime overhead, therefore making it suitable for production use.

Spatial Memory Errors. Over the past decades, many different solutions have been studied to address spatial memory errors, including *sanitizers* [3], [8], [9], [10], [36], [37] and *bounds checkers* [4], [12], [13], [14], [15], [16], [35], [77]. The former are mostly employed in offline testing scenarios, while the latter have been proposed as a mitigation. While existing bounds checkers can completely protect against spatial memory errors, they unfortunately remain impractically expensive on commodity hardware, with the lowest reported overhead being Delta Pointers [12] with a slowdown of 35%.

In contrast, with StickyTags, we focus on the opposite design point: an efficient mitigation that offers protection bounded by the MTE tag entropy. Historically, mitigations that found production deployment mostly focused on cheap methods to reduce the attack surface, since providing full protection was too costly. Examples include: Address Space Layout Randomization (ASLR), No-EXecute (NX), Position Independent Executables (PIE), Read-only relocation (RELRO), and stack canaries.

11. Conclusion

Spatial memory errors remain a major vulnerability class in C and C++ programs. Despite decades of research, proposed mitigations are still not considered for production use due to the impractical overheads they introduce. In this paper, we argued Arm's Memory Tagging Extension (MTE) is a strong contender for production-ready mitigation against spatial memory errors. We first showed that existing *probabilistic* MTE solutions incur important limitations in performance (i.e., due to frequent memory retagging) and security (i.e., due to random tagging being vulnerable to tag leakage attacks). Then, we presented StickyTags and demonstrated that *persistent* memory tags can provide *deterministic* protection against spatial memory errors with high performance. Our evaluation shows that StickyTags is efficient with both MTE hardware and analogs, incurring a low overhead of $\leq 4\%$ to protect the stack and the heap.

Disclosure. We disclosed speculative probing of random tags to Arm, which further disclosed to affected licensees. In response, Arm published an advisory [78] to offer guidance on the impact of speculative oracles on memory tagging.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. Special thanks to Jakob Koschel and Nikolaos Chalkiadakis (MTE on the S22), Raphael Isemann (LTO magic), Johannes Blaser (LLVM guidance), Mathé Hertogh (Spectre hacking), and Chris Ouwehand (early x86 prototyping). This work was supported by Intel Corporation through the "Allocamelus" project, by NWO through project "INTERSECT" and "Theseus", and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

References

[1] CWE, "2023 CWE Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023.

[2] V. Tsyrklevich, "GWP-ASan: Sampling heap memory error detection in-the-wild," Online, https://www.chromium.org/Home/chromium-security/articles/gwp-asan/.

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX ATC*, 2012.

[4] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight Bounds Checking," in *Code Generation and Optimization (CGO)*, 2012.

[5] T. Kroes, K. Koning, C. Giuffrida, H. Bos, and E. van der Kouwe, "Fast and generic metadata management with mid-fat pointers," in *10th European Workshop on Systems Security (EuroSec)*, 2017.

[6] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication," in *ACM CCS*, 2022.

[7] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory Tagging and how it improves C/C++ memory safety," 2018.

[8] J. Ba, G. J. Duck, and A. Roychoudhury, "Efficient Greybox Fuzzing to Detect Memory Errors," in *IEEE/ACM ASE*, 2022.

[9] F. Gorter, E. Barberis, R. Isemann, E. Van Der Kouwe, C. Giuffrida, and H. Bos, "FloatZone: Accelerating Memory Error Detection using the Floating Point Unit," in *USENIX Security*, 2023.

[10] A. U. S. Gopal, R. Soori, M. Ferdman, and D. Lee, "TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers," in *OSDI*, 2023.

[11] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *IEEE S&P*, 2019.

[12] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *Thirteenth EuroSys Conference*, 2018, pp. 1–14.

[13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *PLDI*, 2009.

[14] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack Bounds Protection with Low Fat Pointers," in *NDSS Symposium*, 2017, pp. 1–15.

[15] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cup: Comprehensive user-space protection for c/c++," in *ASIA CCS*, 2018.

[16] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, vol. 10, 2009.

[17] X. Chen, Y. Shi, Z. Jiang, Y. Li, R. Wang, H. Duan, H. Wang, and C. Zhang, "MTSan: A Feasible and Practical Memory Sanitizer for Fuzzing COTS Binaries," in *USENIX Security Symposium*, 2023.

[18] K. Hohentanner, P. Zieris, and J. Horsch, "Cryptsan: Leveraging arm pointer authentication for memory safety in c/c++," in *SAC*, 2023.

[19] J. Seo, J. You, D. Kwon, Y. Cho, and Y. Paek, "Zometag: Zone-based memory tagging for fast, deterministic detection of spatial memory violations on arm," *IEEE TIFS*, 2023.

[20] LLVM, "MemTagSanitizer," Online, https://llvm.org/docs/MemTagSanitizer.html.

[21] G. glibc, "38.7 Memory Related Tunables," Online, https://www.gnu.org/software/libc/manual/html_node/Memory-Related-Tunables.html.

[22] Linux, "The Kernel Address Sanitizer (KASAN)," Online, https://docs.kernel.org/dev-tools/kasan.html.

[23] pcc, "scudo: Add initial memory tagging support," Online, https://reviews.llvm.org/D70762.

[24] R. Townsend, "feat: basic MTE support for the partition allocator," Online, https://chromium-review.googlesource.com/c/chromium/src/+/2695355.

[25] A. Partap and D. Boneh, "Memory tagging: A memory efficient design," *arXiv preprint arXiv:2209.00307*, 2022.

[26] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *IEEE S&P*, 2014.

[27] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling client-side crash-resistance to overcome diversification and information hiding." in *NDSS*, vol. 16, 2016, pp. 21–24.

[28] LWN.net, "Averting excessive oopses," Online, https://lwn.net/Articles/914878/.

[29] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the Spectre era," in *CCS*, 2020.

[30] J. Bialek, K. Johnson, M. Miller, and T. Chen, "Security analysis of memory tagging," Online, https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf.

[31] S. Jero, N. Burow, B. Ward, R. Skowyra, R. Khazan, H. Shrobe, and H. Okhravi, "Tag: Tagged architecture guide," *ACM Computing Surveys*, vol. 55, no. 6, pp. 1–34, 2022.

[32] M. Brand, "MTE As Implemented, Part 1: Implementation Testing," Online, https://googleprojectzero.blogspot.com/2023/08/mte-as-implemented-part-1.html.

[33] Arm, "Armv8.5-A Memory Tagging Extension," Online, https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[34] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with HAKC," in *Network and Distributed System Security Symposium (NDSS)*, 2022.

[35] A. Kwon, U. Dhawan, J. M. Smith, T. F. K. Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *ACM CCS*, 2013.

[36] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating Address Sanitizer," in *USENIX Security*, 2022.

[37] Y. Jeon, W. Han, N. Burow, and M. Payer, "FuZZan: Efficient Sanitizer Metadata Design for Fuzzing," in *USENIX ATC*, 2020.

[38] B. Perens, "Electric Fence," 1987, https://elinux.org/Electric_Fence.

[39] M. Labs, "Pwn2own 2013 write-up: Webkit exploit," https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-webkit-exploit, 2013.

[40] K. Serebryany *et al.*, "GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production," *arXiv:2311.09394*, 2023.

[41] A. Sotirov, "Heap Feng Shui in JavaScript," *Black Hat Europe*, 2007.

[42] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus, "Bypassing memory safety mechanisms through speculative control flow hijacks," in *EuroS&P*, 2021.

[43] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: attacking arm pointer authentication with speculative execution," in *ISCA*, 2022.

[44] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019.

[45] Intel, "Refined speculative execution terminology," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html.

[46] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security*, 2019.

[47] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *NDSS*, 2017.

[48] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript," in *Financial Crypto*, 2017.

[49] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *USENIX Security*, 2016.

[50] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the GPU," in *S&P*, 2018.

[51] J. Kim, S. van Schaik, D. Genkin, and Y. Yarom, "iLeakage: Browser-based timerless speculative execution attacks on Apple devices," in *CCS*, 2023.

[52] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses," in *USENIX Security*, 2021.

[53] H.-J. Boehm, "Bounding space usage of conservative garbage collectors," in *POPL*, 2002.

[54] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, "DangZero: Efficient Use-After-Free Detection via Direct Page Table Access," in *CCS*, 2022.

[55] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *USENIX Security*, 2010.

[56] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers," in *USENIX Security Symposium*, 2017, pp. 815–832.

[57] B. Wickman, H. Hu, I. Yun, D. Jang, J. Lim, S. Kashyap, and T. Kim, "Preventing Use-After-Free Attacks with Fast Forward Allocation," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[58] R. M. Farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal Memory Safety via Robust Points-to Authentication," in *USENIX Security*, 2021.

[59] M. Erdős, S. Ainsworth, and T. M. Jones, "Minesweeper: a "clean sweep" for drop-in use-after-free prevention," in *ASPLOS*, 2022.

[60] S. Ghemawat and P. Menage, "TCMalloc: Thread-caching malloc," 2009.

[61] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," in *APLAS*. Springer, 2019.

[62] 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)*. IEEE, 2022.

[63] E. Van Der Kouwe, T. Kroes, C. Ouwehand, H. Bos, and C. Giuffrida, "Type-after-type: Practical and complete type-safe memory reuse," in *ACSAC*, 2018, pp. 17–27.

[64] LLVM, "Stack Safety Analysis," Online, https://llvm.org/docs/StackSafetyAnalysis.html.

[65] S. Muchnick, *Advanced Compiler Design and Implementation*, 1997.

[66] "Private email communication with Google (MTE) engineers."

[67] LLVM, "Scudo source," https://llvm.googlesource.com/scudo/+/966620155350ba9e3d09b6bc70b9babc4d222027/combined.h#388.

[68] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[69] "Safestack," https://clang.llvm.org/docs/SafeStack.html.

[70] F. B. Jr. and P. Black, "Juliet 1.1 C/C++ and Java Test Suite," in *IEEE Computer*, 2012, pp. 88–90.

[71] H. Liljestrand, C. Chinea, R. Denis-Courmont, J.-E. Ekberg, and N. Asokan, "Color My World: Deterministic Tagging for Memory Safety," *arXiv preprint arXiv:2204.03781*, 2022.

[72] Arm, "Arm Memory Tagging Extension," Online, https://source.android.com/docs/security/test/memory-safety/arm-mte#async-mode.

[73] Linux, "Profiling with performance counters."

[74] W. Han, B. Joe, B. Lee, C. Song, and I. Shin, "Enhancing memory error detection for large-scale applications and fuzz testing," in *Network and Distributed Systems Security (NDSS) Symposium*, 2018.

[75] S. Ainsworth and T. M. Jones, "Markus: Drop-in use-after-free prevention for low-level languages," in *S&P*, 2020.

[76] M. Phillips, "Globals Tagging - Discussion," Online, https://groups.google.com/g/llvm-dev/c/FAR7zKNkWh4/m/FIddvBRQAgAJ.

[77] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," *ACM SIGOPS Operating Systems Review*, 2008.

[78] Arm, "Arm Memory Tagging Extension: Security Update," Online, https://developer.arm.com/Arm%20Security%20Center/Arm%20Memory%20Tagging%20Extension.

[79] S. Singh and M. Awasthi, "Memory centric characterization and analysis of spec cpu2017 suite," in *ICPE*, 2019.

Appendix

Apple M2: StickyTags with MTE Analogs

System	SPEC CPU2006	SPEC CPU2017
StickyTags-stack	0.7%	1.0%
MemTagSan-stack	14.2%	8.8%
StickyTags-heap	0.5%	0.7%
Non-persistent-heap	1.6%	1.4%
StickyTags-both	1.2%	1.9%
MemTagSan + TC	15.8%	10.4%

TABLE 4: SPEC CPU runtime overhead summary using MTE analogs for StickyTags, TCMalloc with non-persistent tagging, and MemTagSanitizer.

Comparison to the state-of-the-art. In this section, we compare the overhead of StickyTags, TCMalloc with a non-persistent deterministic tagging scheme (described in Section 8.4), and MemTagSanitizer. We make use of MTE analogs on the Apple M2 and the SPEC CPU2006 and 2017 benchmarking suites. Note that the M2 uses pages of size 16 KB, which reduces the number of page faults StickyTags has to handle compared to a 4 KB page size. Unfortunately, the 657.xz_s benchmark is known to exhibit an extremely large memory footprint [79], and we have to omit it because the M2 runs out of memory (16 GB). Although the system requirements for SPECspeed 2017 state 16 GB of physical memory, this is insufficient on our machine.

Table 4 showcases the geomean runtime overhead we measured. The table contains the isolated heap and stack overhead, as well as the combination of both. Most notably, we observe that MemTagSanitizer's stack instrumentation incurs 14.2% and 8.8% runtime overhead on CPU2006 and 2017, respectively, while StickyTags' stack overhead is significantly lower at 0.7% and 1.0%. Moreover, for the heap we measure a runtime overhead of 1.6% and 1.4% on CPU2006 and 2017 for the non-persistent tagging design, while StickyTags manages to (more than) halve this overhead to 0.5% and 0.7%. With the stack and heap combined, StickyTags incurs a low overhead of 1.2% and 1.9%, compared to existing techniques with 15.8% and 10.4%, for SPEC CPU2006 and 2017, respectively. These data points for both the stack and the heap highlight the benefit of our design for persistent memory tags, with which we manage to relieve the pressure of frequent memory tagging.

Listing 3 MTE analog for setting and clearing memory tags. The analog is adapted from the original [34] to be inlined.

```
#define MTE_SET_TAG_INLINE(ptr, size) asm volatile ( \
    "mov x2, %0             \n"\
    "mov x3, %1             \n"\
    "mov x17, %0            \n"\
    "cbz %1, 2f             \n"\
"1:                         \n"\
    "mov x16, %0            \n"\
    "lsr x16, x16, #56      \n"\
    "and x16, x16, #0xFUL   \n"\
    "strb w16, [x17, #0x0]  \n"\
    "add %0, %0, #16        \n"\
    "sub %1, %1, #16        \n"\
    "add x17, x17, 1        \n"\
    "cbnz %1, 1b            \n"\
"2:                         \n"\
    "mov %0, x2             \n"\
    "mov %1, x3             \n"\
:: "r"(ptr),"r"(size) : "x16","x17","x2","x3","memory")
```

Vulnerability	Type	Project	Program	Version
CVE-2016-10270	heap	libtiff	tiffcp	4.0.1
CVE-2016-10271	heap	libtiff	tiffcrop	4.0.1
CVE-2017-8786	heap	pcre2	pcre2test	10.23
CVE-2017-14408	stack	mp3gain	mp3gain	1.5.2
CVE-2018-20004	stack	mxml	testmxml	2.12
CVE-2020-21675	stack	fig2dev	fig2dev	93795dd
CVE-2020-21050	stack	libsixel	img2sixel	2df6437
CVE-2021-20294	stack	binutils	readelf	2.35

TABLE 5: Program details of the CVE analysis.

Heap and stack size classes

For the stack we use a total of 15 size classes, all of them being a multiple of two, and the smallest being the MTE tagging granularity. The list of classes consists of: 2^N with $N = \{4...18\}$, making the largest class 262144 bytes. For the heap we use a total of 76 size classes, all of them being a multiple of 16, and the smallest being the MTE tagging granularity. These are the default size classes in TCMalloc, with the exception of the first class being 16 instead of 8.

Redzone guard value on x86

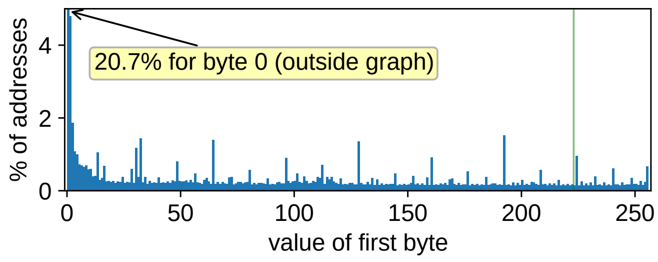

Figure 9: Byte values at dereferenced addresses in SPEC CPU2006. Bin i shows what percentage of dereferenced addresses contains value i in the first byte. Highlighted bin: 223 (default guard value).

Occurrences of the guard value in application memory outside a redzone cause a slow check when dereferenced. To achieve high efficiency, it is important to limit the number of slow checks by choosing a guard value that occurs sparsely consistently across different types of programs. To find such values, we instrumented SPEC to log the first byte at the location of each memory access, as seen in Figure 9.

- Values 0 and 1 are (unsurprisingly) the most common—as these are default initializers. This extends to a lesser degree to low values under 20. Similarly, the value 255 (used in bitmasks) is best avoided.
- Text-processing applications such as 483.xalancbmk, 400.perlbench and 401.bzip show spikes in the range of printable ASCII characters (up to 127) and in particular alphabetic characters (65-90 and 97-122).
- Powers of 2 and their multiples are prevalent.
- Some benchmarks show recurring spikes at the multiples of some application-specific number.

Webservers on x86

	Saturation connections	Throughput degradation	Latency increase 50p	75p	90p	99p
Nginx	250	7%	8%	7%	6%	5%
Apache	350	8%	9%	10%	11%	15%
Lighttpd	500	10%	14%	9%	11%	13%

TABLE 6: Web server overhead at saturation: throughput degradation and increase in 50/75/90/99 percentile latency.

We have benchmarked both our LTO-enabled TCMalloc baseline and our x86 design (configured to use explicit

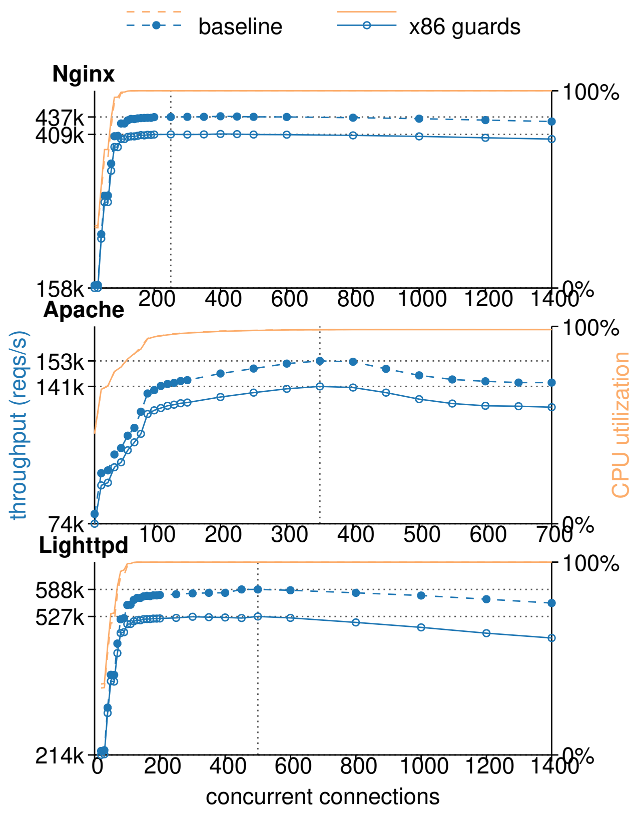

Figure 10: Web server throughput with increasing client connections. E.g., the Nginx baseline achieves its maximum throughput at saturation (100% CPU) at 250 connections, at which point the throughput degradation is 7% from our x86 persistent guards.

persistent spatial guard of 64 bytes) on three major web servers: Nginx 1.17.4, Apache 2.4.41 and Lighttpd 1.4.54. We instrumented loadable modules and non-system libraries (including APR and APR-Util) for all servers. We used two Intel Xeon Silver 4110 machines—server and client—each with 8 hyper-threaded cores at 2.10 GHz and 32 GB of memory, connected by a dedicated 100 Gbit/s network link. We used a Linux 4.15 kernel with `sendfile` enabled and used a large number of keepalive connections with a short timeout. We ran our experiments with 16 worker processes, requesting 64-byte pages for 30 seconds using the `wrk` benchmark with an increasing number of concurrent connections. We repeated each experiment 11 times and report the medians here. All standard deviations are less than 1% except for 99-percentile latency for which it goes up to 2.6%. Figure 10 illustrates how we determined saturation points, i.e., the number of connections with the highest throughput at 100% CPU utilization. Table 6 details the throughput and latency impact on x86 for all servers: at saturation, throughput degrades by only 7-10% and 99-percentile latency increases by 5-15%.

Benchmark	ST-heap	ST-stack	ST-both	MTS	Scudo	Scudo+MTS
400.perlbench	1.02	1.09	1.11	1.36	1.29	1.54
401.bzip2	1.03	1.03	1.07	1.07	1.00	1.08
403.gcc	1.08	1.02	1.10	1.12	1.26	1.36
429.mcf	0.99	1.00	1.00	1.00	1.00	1.01
433.milc	1.01	1.01	1.01	1.02	0.99	1.01
444.namd	1.00	0.98	0.99	1.01	1.01	1.01
445.gobmk	1.00	1.04	1.05	1.25	1.00	1.25
447.dealII	1.06	0.97	1.06	1.02	1.09	1.09
450.soplex	1.01	1.01	1.00	1.00	1.01	1.03
453.povray	1.05	1.04	1.11	1.41	1.04	1.46
456.hmmer	1.01	1.01	0.99	1.01	1.00	1.01
458.sjeng	1.01	1.01	1.02	3.67	1.00	3.69
462.libquantum	1.03	1.00	1.00	1.02	1.01	1.00
464.h264ref	1.01	1.00	1.01	1.01	1.00	1.01
470.lbm	1.02	1.00	1.01	1.01	1.00	1.01
471.omnetpp	1.17	1.01	1.14	1.02	1.20	1.23
473.astar	1.03	0.99	1.02	1.01	1.03	1.03
482.sphinx3	1.02	1.00	1.01	1.00	1.00	1.01
483.xalancbmk	1.06	1.03	1.08	1.23	1.26	1.43
geomean	1.031	1.012	1.040	1.152	1.058	1.202

TABLE 7: Google Pixel 8 Pro SPEC CPU2006 MTE results. MTS=MemTagSanitizer, ST=StickyTags, both=heap+stack.

	MTE Analogs				*MTE Hardware*			
Benchmark	**MTS**	**MTS+heap**	**ST-stack**	**ST-both**	**MTS**	**MTS+heap**	**ST-stack**	**ST-both**
400.perlbench	1.22	1.20	1.07	1.09	1.14	1.17	1.07	1.10
401.bzip2	1.10	1.10	1.03	1.04	1.17	1.17	1.03	1.04
403.gcc	1.07	1.17	1.00	1.07	1.11	1.41	1.00	1.08
429.mcf	1.03	1.03	0.99	1.00	1.00	1.01	0.99	1.01
433.milc	1.01	1.01	1.00	1.03	0.99	0.99	0.99	1.01
444.namd	0.98	0.98	0.97	0.97	1.01	1.01	0.97	0.97
445.gobmk	1.48	1.48	1.03	1.03	1.78	1.73	1.03	1.03
447.dealII	0.85	0.99	0.99	1.00	1.00	0.93	0.99	0.99
450.soplex	1.00	1.00	1.00	1.00	1.00	1.03	1.00	1.01
453.povray	1.14	1.14	1.05	1.05	1.39	1.30	1.05	1.05
456.hmmer	0.99	1.00	1.01	1.02	0.99	1.01	1.01	1.02
458.sjeng	4.19	4.20	1.04	1.04	7.33	6.01	1.04	1.04
462.libquantum	1.06	1.07	1.04	1.03	1.10	1.08	1.02	1.02
464.h264ref	1.01	1.01	1.00	1.00	1.02	1.02	1.01	1.01
470.lbm	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04
471.omnetpp	0.97	1.01	1.00	1.03	1.01	1.17	1.01	1.04
473.astar	0.99	1.00	1.00	1.01	1.00	1.01	1.00	1.00
482.sphinx3	1.00	1.01	1.00	1.00	0.99	1.01	1.00	1.00
483.xalancbmk	1.28	1.37	1.02	1.07	1.59	2.23	1.02	1.07
geomean	1.140	1.161	1.014	1.027	1.228	1.257	1.014	1.027

TABLE 8: Samsung Galaxy S22 SPEC CPU2006 MTE results. MTS=MemTagSanitizer, ST=StickyTags, both=heap+stack.

Benchmark	MTS	TC-NP	MTS+TC	ST-stack	ST-heap	ST-both
600.perlbench_s	1.06	1.01	1.07	1.04	1.00	1.05
602.gcc_s	1.05	1.06	1.12	1.01	1.02	1.03
605.mcf_s	1.00	1.00	1.00	1.00	1.00	1.00
619.lbm_s	1.02	1.04	1.06	1.03	1.03	1.06
620.omnetpp_s	1.22	1.05	1.27	1.00	1.02	1.03
623.xalancbmk_s	1.26	1.00	1.26	1.01	1.00	1.01
625.x264_s	1.41	0.99	1.41	1.01	1.00	1.01
631.deepsjeng_s	1.04	1.01	1.05	1.02	1.00	1.02
638.imagick_s	1.00	1.00	1.00	1.00	1.00	1.00
641.leela_s	1.00	1.00	1.00	1.00	1.00	1.00
644.nab_s	1.00	1.00	1.00	1.00	1.00	1.00
657.xz_s	-	-	-	-	-	-
geomean	1.088	1.014	1.104	1.010	1.007	1.019

TABLE 9: MacBook M2 SPEC CPU2017 analogs results. MTS=MemTagSanitizer, ST=StickyTags, TC=TCMalloc, NP=non-persistent.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper details a speculative attack to leak MTE tags on real hardware. To mitigate the attack, the authors propose a reorganization of heap and unsafe stack objects that provides deterministic, bounded spatial memory safety.

A.2. Scientific Contributions

- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field
- Identifies an Impactful Vulnerability

A.3. Reasons for Acceptance

1) The authors detail a new MTE tag leak side-channel attack.
2) The authors implement a new heap layout to provide deterministic, bounded spatial safety that mitigates the new side-channel attack.
3) The authors provide evaluation on performance and security benefits.

A.4. Noteworthy Concerns

1) The memory overhead of 15% is non-trivial for many real world scenarios.
2) Exclusive use of StickyTags as a protection mechanism without an additional Use-After-Free (UaF) mitigation makes exploiting UaF easier, and makes detecting UaF exploits difficult. This is due to the reuse of object classes key to the design of StickyTags. However, the authors note that UaF protections can be deployed, and show this in an evaluation on UaF Juliet testcases.