QuickSafe: Targeted Hardening Against Memory Corruption

Johannes Blaser*, Floris Gorter*, Klaus v. Gleissenthall, Herbert Bos
{*j.blaser, f.c.gorter, k.freiherrvongleissenthal, h.j.bos*}*@vu.nl*
Vrije Universiteit Amsterdam

* Equal contribution joint first authors

***Abstract*—Despite decades of research, memory safety solutions see limited adoption, as they often incur high overheads, are complex to deploy, or cover only a narrow scope of bugs.**

In this paper, we present *QuickSafe* — a targeted approach to harden programs against exploitation of known but unresolved memory errors with minimal overhead. QuickSafe yields a stopgap patch that is immediately available, while the bug awaits eventual resolution. Where most existing automatic patch generators rely on inserting runtime constraint checks in the code to stop exploits, QuickSafe instead isolates memory objects associated with a known bug from the rest of the program. Object isolation can be implemented in different ways, depending on hardware support and desired security guarantees. To assess the viability, we present two such implementations. On traditional architectures, we allocate vulnerable objects on dedicated pages flanked by inaccessible guard pages. On platforms that support Memory Tagging Extensions (MTE), we offer stronger guarantees by enforcing disjoint tag domains. To reliably identify the objects associated with a given memory error, we introduce *TagASan* — an extension of AddressSanitizer (ASan) that uses tagged pointers to trace faulting accesses back to their originating allocation sites.

As an additional contribution, we present a new dataset of 223 real-world memory errors across ten prominent projects to measure the performance of automatic patch generators. We evaluate QuickSafe on (1) this new benchmark suite, (2) the Juliet Test Suite, and (3) buggy benchmarks from SPEC CPU2006/2017. Using the guard-page-based isolation backend, QuickSafe protects against the exploitation of all evaluated bugs, incurring a geomean memory overhead of 2.46% and a geomean runtime overhead of 2.67% — with the vast majority of applications slowing down by only around 1%. We apply the MTE-based isolation strategy to a representative subset of the dataset, confirming its effectiveness and showing negligible runtime overhead of $\approx 0.12\%$.

1. Introduction

Memory errors account for approximately 70% of all reported security issues, according to major vendors [1], [2], [3], [4], [5], with Google raising this estimate to 90% for Android [6]. Worse, they underlie over 80% of exploited zero-days [7]. Unsurprisingly, both industry and academia have devoted substantial resources to detecting and mitigating them. Bug *detectors* such as AddressSanitizer (ASan) and its many alternatives [8], [9], [10], [11], [12] help developers discover bugs during testing, prior to release. We now find bugs faster than we can fix them: on average, there is a 171 day gap between a CVE's publication and its fix [13]. For instance, in the widely used ImageMagick software, bugs remain unpatched for an average of 190 days [13]. We thus need techniques that can *contain* bugs using a stopgap patch while they await eventual resolution.

While bug detectors like ASan are not suitable for deployment [14], [15], bug *mitigation* solutions aim to fill this gap. These systems limit the impact of memory safety violations, typically trading detection capability for performance. However, most state-of-the-art mitigation techniques still incur overheads of around 30% [16], [17], [18], while sacrificing compatibility [16], [17] or limiting protection to narrow classes of bugs [16], [17], [19], [18]. The few approaches that report more practical sub-5% overheads [14], tend to rely on specialized hardware [20], [21], [22], [23], making them harder to deploy in practice. Consequently, widespread use of *mitigations* remains rare.

In this paper, we present *QuickSafe*: a *practical* solution for *targeted* protection that provides effective, easy-to-deploy defences against the exploitation of *known yet unresolved* spatial or temporal memory errors. QuickSafe hardens programs by isolating memory objects associated with a given bug from the rest of the program, thereby limiting the ability of corrupted pointers to access unrelated objects. We base its design on three key insights.

Our first key insight is that memory errors are not uniformly distributed: *cold paths* — rarely executed and thus less frequently tested — tend to accumulate more bugs [24], [25], [26]. Yet most current approaches treat all code as equally suspect, incurring overheads across the entire program. This raises the question: can we apply bug mitigation techniques *selectively*, protecting only those parts of the program known to contain flaws?

Such protection against illegal memory accesses can be achieved in two ways: instrumenting the code with explicit constraints or checks, and memory isolation, enforced in hardware or software. Existing *automatic patch generators* focus mostly on checks and constraints, but achieve only modest success rates of 20% to 60% [27], [28], [29], [30].

Arguably the most advanced check-based solution today, VulShield [31], reports excellent performance ($< 2\%$ overhead), but suffers in terms of correctness and complexity. In particular, it inserts exploit-specific runtime constraint checks enforcing complex protection policies derived from sanitizer reports which may be, as we shall see, inaccurate.

Our second key insight is that the simpler design is to mitigate through *isolation*. If we know which object is responsible for a (spatial or temporal) memory error, we can isolate it such that pointers to that object can no longer trivially access other objects, thereby forgoing complex instrumentation to enforce vulnerability-specific constraints. The exact nature of the isolation mechanism is orthogonal to our approach, as long as it requires no action for pointer dereferences — excluding mechanisms requiring per-access work, such as explicit checks, MPK PKRU writes, MPX-BND checks, etc. For instance, depending on hardware and desired security guarantees, developers may opt for memory tagging [32], guard pages [33], Intel TME-MK [34], [35], or other hardware- or software-based isolation techniques. In particular, QuickSafe can use any technique that (a) constrains the pointer to the object associated with the bug, (b) requires no code changes at dereferences.

Our third key insight is that we can use pointer tagging to determine the exact allocation site corresponding to a pointer responsible for an invalid access. In particular, in a separate santitizer-supported analysis phase prior to applying our object-level isolation, we tag pointers at the allocation site, so that when the sanitizer detects the access violation, we know the corresponding allocation site.

To demonstrate QuickSafe's viability, we implement two complementary isolation strategies. The first, compatible with existing architectures, employs *guard pages* — similar to GWP-ASan [36] and KFENCE [37] — but applied in a *targeted* manner. It provides full coverage for heap, stack, and global memory, requires minimal compiler instrumentation, and leverages the Memory Management Unit (MMU) to detect violations automatically via hardware exceptions. The second strategy uses memory tagging extensions (specifically Arm MTE [32]) to enforce *disjoint tag domains*, separating protected objects from the rest of the program. We demonstrate its effectiveness for isolating both heap and stack memory, ultimately offering even stronger protection guarantees than guard pages.

While various datasets of buggy programs exist for evaluating patch efficacy, we found none that also provide *performance benchmarks* to assess the runtime cost of applied defences. To address this, we construct a large-scale dataset of 223 real-world memory errors curated from OSS-Fuzz/ARVO [38] — including vulnerable source code and proofs-of-concept to trigger the bugs — and intersect this with the Phoronix Benchmarking Suite [39]. The dataset includes popular open-source projects such as FFmpeg and GraphicsMagick, enabling a nuanced evaluation of QuickSafe's performance, accuracy, and reliability. Although standard benchmarks like SPEC CPU [40], [41] contain only four confirmed bugs, we include those, alongside the Juliet Test Suite [42], for completeness.

Our evaluation on the new dataset shows that, unlike existing techniques whose success rates range between 20% and 60%, QuickSafe successfully isolates the correct objects for *all* evaluated bugs. For our guard-page-based implementation, 60% of PoCs trigger a fault, while the remaining accesses harmlessly land within intra-page padding. For our MTE-based implementation, we conduct detailed case studies to assess its security, and show that this isolation successfully protects against heap and stack-based bugs across a representative set of real-world memory errors. Despite these protections, performance impact remains low. Targeted guard pages increase memory usage by just 38.5 MB and incur a geomean runtime overhead of 2.42%, which drops to 0.91% when excluding two *hot-path* outliers. Using MTE, QuickSafe incurs negligible slowdowns averaging 0.12%.

With the QuickSafe framework, our main contributions are:

- **QuickSafe:** A low-overhead system to harden specific spatial and temporal memory errors against exploitation by means of object-level isolation.
- **TagASan:** An ASan extension that uses tagged pointers to precisely identify faulting allocation sites.
- **Guard-Page Isolation:** A (weaker) legacy-compatible backend that isolates objects through guard pages.
- **MTE-Based Isolation:** A (stronger) backend that isolates objects through disjoint (Arm MTE) tag domains.
- **Evaluation Dataset:** A curated set of 223 real-world bugs in 10 OSS projects with performance benchmarks.
- **Availability:** The full source code and all dependencies are available at: https://github.com/vusec/QuickSafe

2. Background

Guard Pages. Guard pages are memory pages marked as *inaccessible* to detect out-of-bounds memory accesses. By surrounding memory objects with guard pages, out-of-bounds accesses that cross into these pages trigger a hardware exception. Guard pages originate from debugging tools such as Electric Fence [33] and PageHeap [43]. Since instrumenting *every* object with such barriers is prohibitively expensive for live systems, some modern implementations apply guard pages *selectively*. Specifically, LLVM's GWP-ASan [36] and Linux's KFENCE [37] probabilistically protect randomly selected memory objects at a low sampling rate, using aggregate coverage across many installations to identify bugs during deployment. While these solutions may help to *detect* bugs with minimal overhead, no solution exists to *mitigate* known bugs that are not yet patched.

Existing projects [44], [19], [45], [43] also use paging to detect or mitigate temporal memory safety violations by allocating objects on separate pages and revoking the entire page upon deallocation, typically with high overheads.

AddressSanitizer. AddressSanitizer (ASan) is a runtime analysis tool designed to catch memory errors. It instruments source code by inserting *checks* before memory accesses to verify their validity. To perform these checks efficiently, ASan places *redzones* — memory regions that act as guards

— between objects. It also maintains a *shadow memory*, a disjoint metadata structure that tracks per-object validity. Every 8 bytes of program memory corresponds to 1 byte of shadow memory, where the shadow encodes the validity state (e.g., (partially) valid, freed) of the associated memory.

Memory Tagging Extension. Processor architectures such as the SPARC M7 and more recent Arm v8.5-A and v9 provide strong in-process isolation through memory tagging. By associating *tags* with pointers and memory regions, they enforce that a dereference succeeds only when the pointer's tag matches that of the memory.

Specifically, Arm's Memory Tagging Extension (MTE) tracks a 4-bit tag for every 16 bytes of memory, using Arm's Top-Byte Ignore (TBI) [46] feature to embed the tag in the upper bits of the pointer. Compared to guard pages, MTE provides more fine-grained isolation (16 bytes versus 4–16 KB pages), while also increasing tag space diversity (16 tag values versus binary accessible/inaccessible). MTE is available on Google Pixel mobile devices [47], and Apple recently announced MTE support for the A19 SoC [48].

3. Overview

QuickSafe provides selective hardening through object isolation, protecting against the exploitation of specific spatial or temporal memory safety vulnerabilities.

Figure 1 illustrates the components and workflow of QuickSafe, showing how a buggy program is instrumented by a compile-time transformation pass to enforce selective hardening of objects identified by TagASan. At a high level, QuickSafe performs two main tasks: (i) identifying which objects to protect (Section 4), and (ii) isolating those objects at runtime (Section 5). Although sanitizer reports appear to identify the relevant allocation site, they cannot be relied upon in practice, as the reported site may be incorrect in the presence of memory corruption (Section 4). As a result, naïvely using these reports to drive isolation is ineffective.

To address this, QuickSafe includes *TagASan* — a modified version of ASan that enables robust allocation-site tracing via tagged pointers. In the example from Figure 1, compiling with TagASan and executing the PoC (`x=40`) produces a standard ASan report, augmented with the precise source of the invalid pointer — in this case, the allocation at line 17. Alternatively, if an allocation site is known to be *sensitive*, QuickSafe can proactively harden it, without requiring a PoC or confirmed vulnerability.

As shown in Figure 1, once the target allocation site is known, QuickSafe replaces the original allocation (e.g., the call to `malloc()`) with a call to `guard_alloc()`, redirecting it to one of QuickSafe's isolation strategies. To enforce runtime isolation, QuickSafe supports two complementary backends: guard-page-based isolation (Section 5.1), which surrounds the allocation with inaccessible guard pages; and MTE-based isolation (Section 5.2), which assigns a unique memory tag to the object, isolating it from all other memory in the process address space.

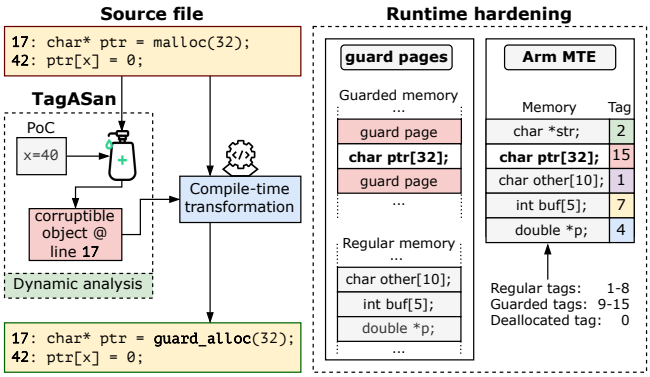

Figure 1: QuickSafe's components and workflow.

Security Guarantees. Unlike tag-based protection, guard pages provide less complete isolation, as non-contiguous out-of-bounds accesses may bypass guard zones. Still, they mitigate contiguous overflows and spatial violations with limited reach. Conversely, MTE is vulnerable to large-offset overflows, which may corrupt the tag — assuming the attacker knows the tag of the target memory. Tag collisions can also hinder MTE's protection. Neither guard pages nor MTE mitigate intra-object overflows, which a major vendor has anecdotally suggested are extremely rare issues [48].

Threat Model. We assume the presence of a known (but unpatched) memory safety bug, along with either an available PoC to trigger it or prior knowledge of the vulnerable allocation site. Memory errors not involving direct spatial or temporal violations (e.g., logic errors or certain type confusions) are out of scope, as are spatial memory errors that grant complete offset control. We further assume the absence of other software or hardware vulnerabilities.

4. Finding Faulty Allocation Sites

To selectively protect against memory corruptions caused by unresolved bugs, QuickSafe must identify *which* allocation site produced the faulty pointer. While sanitizer reports appear a pragmatic source — with tools like ASan reporting where a faulting address was allocated — they are unreliable in the presence of pointer corruption. Since ASan retrieves metadata based on the faulting address, a corrupted pointer may lead it to the wrong object.

4.1. ASan's Limitations

To illustrate ASan's limitations, Listing 1 shows two code snippets, each allocating two 32-byte heap objects. ASan inserts a 16-byte redzone between the objects. In the left example, an out-of-bounds access at index 40 triggers a redzone violation. ASan incorrectly attributes the fault to the allocation on line 2 (`ptr2`), as the invalid address lies closer to the second object than the first.

In the right example, dereferencing `ptr1+50` results in an invalid access to memory previously allocated for `ptr2`,

Listing 1 Example C snippets for misattributed ASan bugs.

```
1 char *ptr1 = malloc(32);|char *ptr1 = malloc(32);
2 char *ptr2 = malloc(32);|char *ptr2 = malloc(32);
3                         |free(ptr2);
4 ptr1[40] = 0;           |ptr1[50] = 0;
```

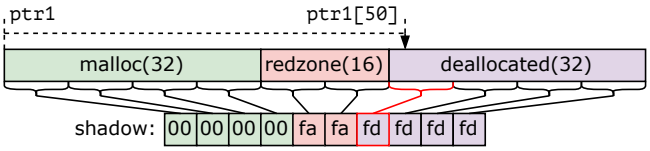

Figure 2: ASan misattributing a buffer-overflow bug.

which has since been freed. As shown in Figure 2, ASan classifies this as a use-after-free error (shadow value `fd`) and attributes it to the allocation site of `ptr2`. While technically correct, the *root cause* of the bug — and therefore, the violation we should protect against — is the buffer overflow on the object allocated via `ptr1`.

Similar misattributions can occur in faulty memory operations, such as corrupted `memcpy()` or `memset()` calls. For such range operations, ASan first checks the validity of the first and last bytes, then scans the interior using an aligned loop. For example, if a corrupted `memset()` on `ptr1` erroneously writes `0x10000` bytes, ASan checks `ptr1` (valid) and `ptr1+0x10000` (invalid), and attributes the fault to whichever object lies closest to the latter address.

Even if the redzone size is increased, ASan's approach remains fundamentally limited: attributing objects by proximity to a faulting address is inherently unreliable in the presence of pointer corruption. As a result, allocation-site information from ASan reports can be misleading and cannot serve as a sound basis for selective protection.

4.2. TagASan

To address ASan's limitations, we introduce *TagASan* — an extension of ASan that uses pointer tagging to identify allocation sites accurately. Conceptually, TagASan assigns a unique integer identifier to each allocation site and encodes this identifier into the upper bits of the corresponding pointers. When a memory error occurs, TagASan extracts the tag from the faulting pointer and uses it to recover the associated allocation site.

Figure 3 illustrates TagASan's instrumentation. At compile time, we extract source-level metadata (file and line number) for allocation sites instrumented by ASan, avoiding the inlining-related loss of information possible with standard debug info (e.g., DWARF).

We use Intel's Linear Address Masking (LAM) feature [49] to conveniently tag pointers — though other pointer tagging solutions, like Arm TBI, AMD UAI, or software-only techniques [12], [16], [17], [50], are also suitable — and extend ASan to support tagged pointers. To preserve compatibility with ASan's shadow memory model, TagASan strips the tag before performing shadow lookups, eliminating

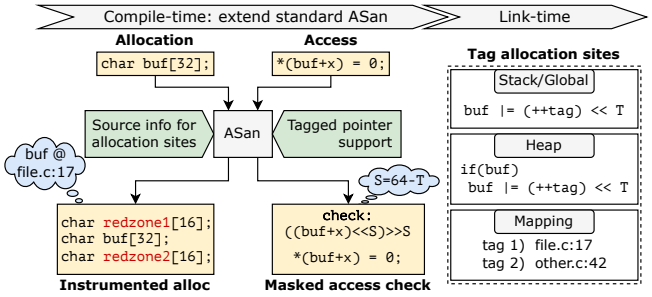

Figure 3: Overview of TagASan's instrumentation.

the need for dual (tagged and untagged) shadow entries or modifications to ASan's 48-bit address space assumption. Consequently, ASan remains unaware of tagging except when a memory violation is detected. As Intel LAM allows tagged pointers to be dereferenced directly, no untagging is needed when passing pointers to libraries or the kernel.

At link time, when all allocation sites are known, TagASan assigns tags in round-robin fashion. We also tag indirect call sites with allocator-like signatures (e.g., calls returning pointers). LAM provides a 15-bit tag space starting at bit 48, allowing for up to $2^{15} = 32{,}768$ unique identifiers. The tagging offset (`T`) is thus 48, and the masking offset (`S`) becomes 16 (see Figure 3). We apply tags by bitwise-OR'ing the allocation result with the next tag shifted by `T`. To avoid tagging `NULL` pointers, the tag is conditionally applied.

If the tag space is exhausted, identifiers wrap around, potentially causing collisions. This did not occur in our experiments. If it does, the bug report includes all candidate allocation sites sharing the tag, and re-execution with only those candidates enabled suffices to identify the correct site.

Although TagASan significantly improves attribution accuracy, it is not foolproof. If a pointer is corrupted such that its tag bits are also modified (i.e., offsets above 2^{48}), TagASan may be unable to recover the correct allocation site. While possible in theory, we did not observe such cases.

TagASan adds overhead compared to vanilla ASan, but since it is used only for a one-time re-run to pinpoint allocation sites, this overhead has no impact on QuickSafe's runtime performance. Consequently, TagASan does not affect software testing campaigns; it only needs to be enabled once after an error is detected, and is not necessary for vulnerability discovery.

5. Protecting Faulty Objects

When a memory safety violation occurs, objects may be accessed incorrectly — either accidentally or deliberately, as part of an exploit. For example, a buffer overflow might leak sensitive data or corrupt a neighbouring object. By *isolating* the offending object, such effects can be *contained*. This principle of *safety through isolation* underpins QuickSafe's core protection strategy, as illustrated in Figure 4.

QuickSafe enforces isolation at the level of the *dynamic memory allocator*, which manages heap memory objects.

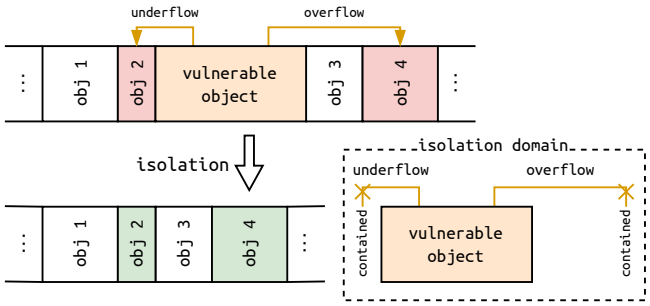

Figure 4: Overview of QuickSafe's safety-through-isolation strategy, which contains invalid memory operations.

Conventional allocators — such as those provided by the GNU C Library (`glibc`: `malloc()`, `free()`, etc.) — are optimised for performance and efficiency, often co-locating objects and reusing memory eagerly. However, as shown in Figure 4, such layouts are vulnerable to corruption and exploitation in the presence of memory bugs.

Rather than protecting the entire system allocator — which would incur prohibitive overhead — QuickSafe introduces a complementary *dynamic isolated allocator* that runs alongside the regular allocator and exposes two dedicated routines: `guard_alloc()` and `guard_dealloc()`. By selectively using this interface, QuickSafe isolates only vulnerable or sensitive objects, while other allocations continue to benefit from the default allocator's performance.

To enable this, we implement a lightweight compile-time transformation pass that rewrites specific allocation sites to use the isolated allocator instead of the original ones (e.g., replacing `malloc()` or `calloc()` with `guard_alloc()`). Only explicitly marked sites are transformed; all others remain unchanged. The pass also ensures that stack and global objects are handled appropriately.

We design the pass to be generic: it accepts plain source-location pairs (i.e., filename and line number) as input and modifies only those allocation sites. While QuickSafe typically uses TagASan to determine which sites to protect, the pass can also be used manually, allowing developers to harden specific objects even in the absence of a known vulnerability. Similarly, users may bypass the transformation step entirely and interface directly with the isolated allocator via `guard_alloc()` and `guard_dealloc()`.

This separation between identifying allocation sites, rewriting them to use isolated allocation, and enforcing runtime isolation enables *modularity*: QuickSafe can flexibly balance compatibility, performance, and security to suit the needs of a given deployment.

We provide two backend implementations for the isolated allocator. The first uses *guard pages* and is designed for broad compatibility with minimal platform assumptions (Section 5.1). The second leverages *Arm MTE*, where available, to offer stricter isolation guarantees with potentially lower overhead (Section 5.2).

5.1. Isolation through Guard Pages

QuickSafe's guard-page-based isolation strategy builds on the principles of Electric Fence [33] and PageHeap [43], incorporates the selective nature of GWP-ASan [36], and refines them for precise, object-specific hardening.

This strategy prioritises compatibility with widely deployed architectures: it does not rely on specialised hardware or elevated privileges, and can be implemented on any system that supports virtual memory with page-level access control. Our prototype targets Linux, using standard memory protection flags (i.e., `PROT_NONE`) to enforce isolation boundaries. However, there are no inherent limitations preventing this design from being ported to other platforms.

At program startup, the guard-page-based allocator reserves a contiguous virtual memory region — the *guard region* — for isolated object allocations, as shown in yellow in Figure 5. The size of this region determines the number of isolated objects that can be allocated concurrently.

This region is never released during the program's lifetime. As such, the system's default allocator can not allocate or manage objects within this region, keeping both the region and the objects it contains separate from the rest of the program. Exploiting a memory error on an isolated object — to access or corrupt a regular object — therefore requires "escaping" the guard region, which demands a high level of control over the faulty pointer. Moreover, the first and last pages of the region are always inaccessible, fully containing contiguous out-of-bounds accesses.

Initially, the guard region is unbacked by physical memory and marked inaccessible, so any access triggers a segmentation fault. On allocation, a portion is marked accessible (`PROT_READ | PROT_WRITE`) and a pointer to it is returned to the caller. Deallocation restores the region to `PROT_NONE`, ensuring subsequent accesses (e.g., use-after-free) also fault. Since access protections apply at page granularity, this strategy enforces isolation in the *virtual address space* at the level of individual pages.

Due to this page granularity, the guard region is split into two allocation pools: one for objects that fit in a *single* page, and one for objects that span *multiple* pages. They are labelled accordingly in Figure 5, and are managed independently to optimise performance and reduce fragmentation.

Single-Page Objects. Because of the page granularity, objects smaller than a page still consume a full page. To reduce memory waste, we *interlace* object and guard pages, creating the weaving layout shown in Figure 5. Since this layout is fixed, we track allocations by slot index, simplifying management. We also share guard pages: the right guard of one object serves as the left guard of the next.

Initially, slots are allocated linearly. Freed slots are marked inaccessible but not immediately reused. Once all slots have been used at least once, we adopt a reuse strategy inspired by KFENCE [37] and GWP-ASan [36], maintaining a disjoint *free-slot list* of deallocated indices. New allocations draw from this list at random, introducing

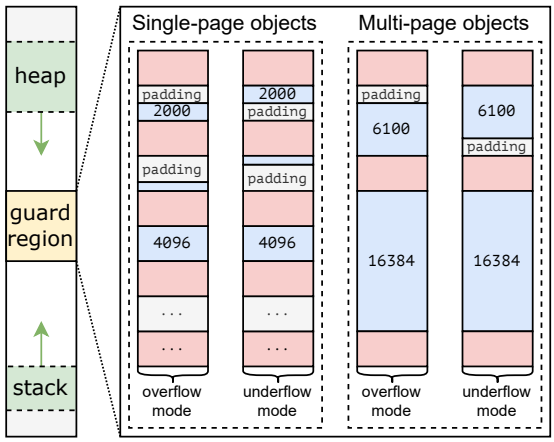

Figure 5: Memory layout of isolation through guard pages. Single-page objects (e.g., size 2000 and 4096 bytes) use a pre-reserved paging pattern. Multi-page objects (e.g., size 6100 and 16384 bytes) are allocated on demand. Under- and overflow modes adjust the alignment of the objects.

a lightweight, probabilistic *pseudo-quarantine* that delays reuse and hinders temporal exploitation.

Multi-Page Objects. Unlike single-page objects, the layout of large allocations varies with size and position, so guard page placement cannot be predetermined. To manage these dynamically, we use a page-granular *buddy allocator*. The first page of the multi-page pool is inaccessible, ensuring separation from the single-page object allocation pool.

To allocate a multi-page object, we compute the required number of pages, add one extra page to serve as a right-sided guard page, and request the total from the buddy allocator. Only the object pages are made accessible; the guard page remains inaccessible and cannot be reused, ensuring separation between objects. It also acts as the left-side guard for the next allocation, reducing overhead. The permanently inaccessible first page of this pool acts as the left-sided guard page for the first allocation.

Because the layout is dynamic, we must track object sizes to allow correct deallocation. Since routines like `free()` do not receive the size of the object, our `guard_dealloc()` function does not either. We therefore maintain disjoint metadata mapping each live allocation's base page to its size (in pages). On deallocation, we consult this mapping to determine which pages to reclaim.

Quarantining. While single-page objects benefit from implicit pseudo-quarantining via randomised reuse, this approach does not apply to multi-page objects. To harden multi-page objects against exploitation through temporal errors, we provide a configurable ring-based quarantine — a well-established strategy in memory safety tools [8], [19], [51], [52]. Recently freed objects remain in quarantine for a configurable duration before returning to the buddy allocator, delaying reuse and improving detection and protection.

Targeted Alignment. Objects that do not fill their entire page leave padding that may absorb out-of-bounds accesses. For example, a 1024-byte object placed at the start of a 4096-byte page leaves 3072 bytes of padding before the guard page. While this does not impact safety — as no other objects may occupy these bytes — it reduces the likelihood that smaller out-of-bounds accesses are detected.

To address this, we support two intra-page alignment modes: *underflow* and *overflow* (see Figure 5). Underflow mode places objects at the beginning of the page and is the default, as it incurs less overhead and tolerates small overruns. This is useful, as aborting on minor faults such as off-by-ones may harm usability. In contrast, overflow mode aligns objects near the end of the page. We may need padding to satisfy alignment constraints (e.g., 16 bytes), but this configuration improves detection of buffer overruns. Minor overflows into trailing padding may (harmlessly) evade detection, but the overall detection probability is increased.

Stack & Global Variables. While we can handle heap objects with `guard_alloc()` directly, stack and global objects require additional instrumentation. For the stack, we promote the object to a heap allocation (similar to prior work [21]) and insert a `guard_dealloc()` call at the end of its lifetime to preserve scoping semantics.

For global variables, we inflate the original allocation by adding padding and flanking the object with guard pages. At program startup, we mark these pages inaccessible.

Precision Mode. While alignment modes suffice for deployment, a thorough evaluation must distinguish between accesses that remain fully unprotected — and thus exploitable — and those that are effectively isolated but do not trigger faults due to intra-page padding. To support this, we provide an optional *precision mode*, which enforces isolation with *byte granularity*. Not intended for production environments, it allows us to confirm whether QuickSafe would have rendered a memory safety violation benign — even in the absence of a fault under either of the alignment modes.

This mode maintains disjoint metadata tracking the exact size (in bytes) of all isolated objects. When enabled, the compile-time transformation pass instruments all memory accesses — including library calls — with bounds checks against this metadata. Any invalid accesses are flagged, allowing us to confirm protection.

Deallocations. While the transformation pass ensures that only selected allocation sites use QuickSafe's isolated allocator, deallocation sites are not necessarily available in the same way, and thus cannot reliably be replaced with direct calls to our deallocation routine (`guard_dealloc()`). Furthermore, a deallocation site may serve as a sink for multiple allocation sites. However, if a pointer to an object in the guard region is passed to the *default* deallocator (e.g., `free()`), the system allocator will not recognise it and likely abort execution — reporting invalid deallocations.

To avoid this, the transformation pass replaces *all* deallocation sites with a wrapper that checks whether the pointer

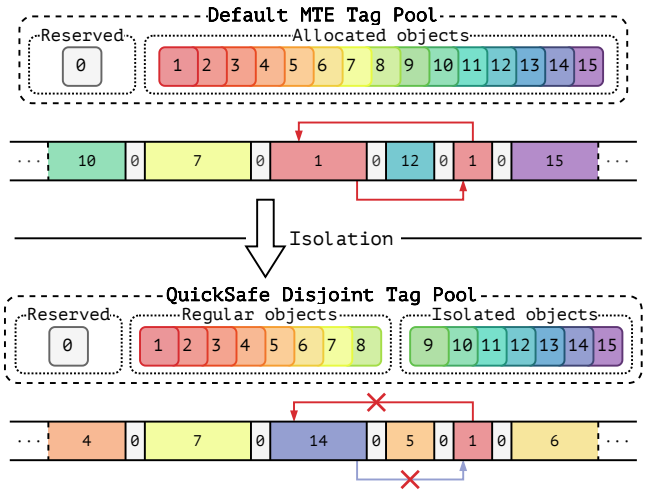

Figure 6: Tagging layout with isolation through MTE. The target object gets tag 14, isolating it from the regular objects.

falls within QuickSafe's guard region. If so, it delegates to `guard_dealloc()`; otherwise, it uses the original deallocator. To minimise overhead, we implement this check using a single pointer subtraction and constant comparison, as in Linux's KFENCE [53].

5.2. Isolation through MTE

While guard-page-based isolation prioritises compatibility with existing systems, modern hardware offers opportunities for stronger guarantees and reduced overheads. To illustrate QuickSafe's modularity in its choice of isolation backend, we implement an alternative backend using Arm's Memory Tagging Extension (MTE) [32]. Like guard pages, it isolates only the bug-related objects, but it offers stronger guarantees with potentially lower runtime costs.

Although MTE's runtime checks are handled in hardware, the allocator is responsible for assigning tags. On allocation, the allocator assigns a tag to the memory region and returns a pointer with the same tag. On deallocation, it clears the tag from memory; any dangling pointer with the now-invalid tag trigger segmentation faults when dereferenced.

Integration with glibc. While the MTE-based isolation could enhance any tagging scheme (e.g., Sticky Tags [21]), we prototype it as an extension to MTE-enabled `glibc`. As in `glibc`, we reserve the zero tag exclusively for internal metadata and deallocated memory.

As shown in the top half of Figure 6, `glibc` stores metadata in-band, ensuring that contiguous buffer overflows cross into metadata and trigger a fault. Similarly, zero-tagging deallocated memory causes use-after-free errors to trigger faults. Live allocations are assigned random tags (1–15). This default scheme probabilistically protects against temporal and spatial memory safety violations.

Disjoint Tag Domains. To enable *targeted* protection, we partition the non-zero tags into two disjoint domains: one for regular allocations and one exclusively for isolated objects. Tags in both domains are drawn at random, but no tag is shared between domains.

Figure 6 illustrates how this separation ensures that any access via a pointer to an isolated object — out of bounds or otherwise misused — cannot match the tag of any regular object. Such violations are therefore reliably detected. The same protection applies in reverse: if a memory error targets an isolated object, the tag mismatch is likewise caught.

Safety Guarantees. While isolated and non-isolated objects are mutually protected, violations between isolated objects may go undetected if they share the same tag. We reduce this risk by assigning tags to isolated objects at random from a dedicated *pool* (Figure 6). Undetected collisions thus require both objects to originate from the same domain *and* be assigned the same tag.

As in `glibc`'s default scheme, zero-tagging freed memory ensures that accessing such memory triggers a fault. Reallocated memory receives a fresh random tag, making it unlikely that a dangling pointer will match — providing probabilistic temporal safety.

Customisation and Extensibility. QuickSafe's default configuration splits the non-zero tags roughly evenly between regular and isolated objects. This is not fundamental: one could, for example, reserve a single tag (e.g., tag 15) for all isolated objects — useful when isolating a single object or handling purely spatial bugs. This preserves more tags for regular allocations, retaining `glibc`'s default detection guarantees while still enforcing isolation.

Alternatively, finer-grained schemes could assign tags per allocation site, allowing specific objects to be isolated not only from the rest of the program but also from one another. Such schemes allow multiple bugs to be isolated independently. While QuickSafe's default configuration supports arbitrarily many target bugs with reasonable security guarantees, these alternatives could offer stronger interdependent isolation or enable application-specific tuning.

Stack & Global Variables. As with guard pages, the compile-time transformation pass promotes stack variables to heap objects, allowing them to benefit from the same isolation guarantees. Promoted variables draw from the same tag pool as other isolated objects. Most tagging strategies exclude regular stack memory, often due to compatibility concerns or the high cost of tagging variables that frequently enter and leave scope [21], [32].

However, even without tagging the stack directly, QuickSafe can still enforce isolation by mapping the stack with `PROT_MTE`. This flag causes the hardware to perform tag checks on stack accesses, even if the memory is not explicitly tagged. Since the stack defaults to the zero-tag and all isolated objects use non-zero tags, this ensures separation from regular stack memory.

We exclude globals from MTE-based isolation due to compatibility concerns surrounding tagged global pointers. Nonetheless, standalone instrumentation tools such as

LLVM's MemTagSanitizer [54] could extend coverage to global objects. QuickSafe is designed to integrate seamlessly with such tools. While LLVM's support for stack and global tagging is currently limited to Android, no changes to QuickSafe are required to benefit from this support if it becomes more broadly available.

6. Implementation

TagASan. We implement *TagASan* by modifying ASan in-tree in LLVM 16.0.6, extending its compile-time instrumentation (using Intermediate Representation (IR)) and its runtime library. We integrate a new IR-level pass to apply pointer tags into the Link Time Optimisation (LTO) pipeline.

Transformation Pass. To ensure that only targeted allocation sites use QuickSafe's isolated allocator, we implement a compile-time transformation pass as an out-of-tree LLVM LTO pass that runs during the final LTO stage to avoid interference with upstream optimisation passes. Implemented using LLVM's new pass manager [55], [56], it integrates cleanly into the standard optimisation pipeline and can be used via Clang and `ld.lld` using `-Wl,--load-pass`, without requiring manual application via `opt`.

Guard-Page-Based Dynamic Allocator. We implement the guard-page-based dynamic isolated allocator as a C++ shared library, linked against the target program. This library exposes two runtime functions: `guard_alloc()` and `guard_dealloc()`, which allocate and deallocate isolated objects within the guard region.

To support standard memory behaviour (e.g., reallocation), we implement dedicated *handler functions* for common library routines. Some programs use custom memory allocators (i.e., wrappers around `malloc()`, `calloc()`, etc.). Without explicit handlers, our transformation pass may instrument allocation sites within these wrappers, inadvertently isolating all their objects. Since handlers are typically concise (one or two lines), we provide additional handlers for the custom allocators encountered during evaluation.

MTE-Based Dynamic Allocator. We implement the MTE-based isolated allocator as an extension to `glibc 2.39`. To constrain tag usage for regular allocations, we modify MTE's `prctl()` tag-generation mask to restrict the set of assignable tags. We then introduce a new internal allocation routine that assigns tags in accordance with our isolation strategy (Section 5.2) by offsetting the now-constrained random tags generated by MTE. The public interface `guard_alloc()` internally invokes this allocator to ensure that isolated objects are correctly tagged.

7. Benchmarking Dataset

As QuickSafe selectively protects objects associated with a known memory error, the overhead it incurs depends on how frequently such objects get allocated. To validate QuickSafe's assumption that bugs appear on cold paths, we must therefore include a diverse set of real-world bugs. Synthetic test cases or small benchmarks (e.g., the Juliet Test Suite [42]) are insufficient.

Project	Bugs (#)	Bug type S	Bug type T	Object type H	Object type S	Object type G	Eval use Sec	Eval use Perf
aom	12	11	1	11	1	-	11	9
c-blosc2	42	38	4	42	-	-	34	8
dav1d	5	4	1	5	-	-	5	5
espeak-ng	16	13	3	6	7	3	15	12
ffmpeg	80	77	3	72	4	4	62	57
flac	7	5	2	5	2	-	3	2
graphicsmagick	43	38	5	37	5	1	32	32
libraw	18	18	-	9	4	1	14	14
libxml2	56	27	29	35	1	20	38	38
zstd	9	9	-	7	1	1	9	3
Total	**288**	**240**	**48**	**229**	**25**	**30**	**223**	**180**

TABLE 1: Breakdown of bugs included in our benchmark suite with number of ARVO bugs per project ("#"), the type of bug (Spatial, or Temporal), the object type (Heap, Stack, or Global). The last two columns indicate how many of the bugs are used in the security and performance evaluations after filtering out unreproducible and unbenchmarkable bugs.

We contribute a new benchmark suite to support research on vulnerability hardening and patch generation. Existing datasets typically focus on either buggy source code or performance, but not both. However, to assess real-world effectiveness *and* performance, we need both known memory errors *and* realistic benchmark workloads.

Data Sources and Construction. We start from the memory safety bugs in the Atlas of Reproducible Vulnerabilities for Open Source Software (ARVO) [38]. ARVO provides vulnerable source code, PoCs, and detailed metadata (e.g., bug type and severity) for reproduced OSS-Fuzz [57] bugs. For performance benchmarks, we intersect the corresponding projects with the Phoronix Benchmarking Suite [39], discarding all benchmarks unrelated to software performance.

This intersection yields 339 memory errors across 24 projects that are both (i) included in ARVO and (ii) covered by at least one Phoronix benchmark. However, the distribution is skewed: 85% of these 339 bugs appear in only ten of the 24 projects. As merging these sources requires a large manual effort, we focus on these ten projects — each supported with dedicated and automated build- and run-scripts. This subset, comprising 288 memory errors spanning disclosures from 2017 to 2024, is shown in Table 1.

We point out that we included all usable spatial and temporal memory errors, but OSS-Fuzz bugs are biased towards ASan-detectable issues, which excludes intra-object overflows. Moreover, ASan also cannot reliably attribute large (arbitrary) offset faults as spatial errors, since they often trigger segmentation faults (unmapped memory is not marked as invalid in shadow memory).

Benchmark Configuration. Integrating the Phoronix benchmarks into the ARVO test cases posed several challenges. In many cases, ARVO provided only a minimal build system to reproduce the bug, which did not always compile the same binaries used by Phoronix. Since the bugs span

$\approx$ 7 years, we had to resolve extensive version drift and dependency breakage. We moved away from ARVO's minimal PoC-oriented build scripts, where any (old) toolchain and pre-built compiler sufficed. We manually synchronised the build configurations for each project to ensure that both the buggy test case and the benchmarking workload ran on a consistent codebase and build tree. As a result, we provide uniform, modern, and complete builds, enabling drop-in evaluation of our design and supporting future work.

Benchmarks can be executed in two distinct modes: (i) triggering the bug using ARVO's PoCs, and (ii) measuring performance using the Phoronix workloads. For performance measurements, we selected the most common configuration variant for each benchmark, preserving key behaviours such as multithreading and workload scaling from the default Phoronix settings.

In all but one project, Phoronix benchmarks used constant inputs and measured variable runtimes. For GraphicsMagick, however, the benchmark used a fixed runtime with variable inputs. To maintain consistency across our dataset, we modified the benchmark flags to use a fixed input instead, allowing uniform measurement of runtime and memory usage across all benchmarks.

Filtering and Final Dataset. From the 288 bugs, we exclude 65 that were unsuitable for our use case. Most failed to reproduce on our platform (e.g., due to architecture mismatches, different compiler versions, or outdated dependencies), or were misattributed (e.g., located in subprojects or external libraries). This leaves 223 valid bugs, which we use for our security evaluation. OSS-Fuzz assigned 100 of them *high*, 101 *medium*, and 22 *unknown* severity ratings.

However, not all of them are suitable for evaluating *performance*. In some cases, the memory error occurs in a project used as a library, but the associated objects are allocated outside the library — typically in the fuzzer harness used by the ARVO PoCs. Isolating these objects at the allocation site affects only the harness, not the binary under test, leading to artificially low overhead.

To ensure fairness and realism, we exclude such cases from the performance evaluation, yielding 180 bugs across ten real-world projects. These reflect a diverse and realistic range of bug types, object lifetimes, and allocation patterns across popular and representative codebases. Table 1 shows a breakdown by project, bug type, object type, and usage (and Appendix A lists the exact bugs).

8. Evaluation

We evaluated QuickSafe's performance and security for both the guard-page-based and MTE-based backends. For these experiments, we used our new dataset of real-world memory bugs, along with the buggy SPEC CPU benchmarks and the Juliet Test Suite.

8.1. Security through Isolation

We evaluated QuickSafe's ability to isolate the correct object using our guard-page backend and our custom dataset of real-world bugs on a system with an Intel Ultra 9 285K CPU (supporting LAM) and 128 GB RAM, running Ubuntu 24.04. First, we used TagASan to identify the allocation sites associated with the tested bugs. We did not encounter any ambiguous or duplicate allocation-site candidates. Next, we passed these allocation sites to QuickSafe's instrumentation pass, enabling *precision mode* guard pages to track accesses with byte granularity. We then executed the buggy program to observe whether QuickSafe correctly identified the faulting access (i.e., whether the reported backtrace matched the original (Tag)ASan report). As precision mode also detects accesses into intra-object padding, it allows us to accurately verify whether the correct object was protected, even when no segmentation fault is triggered by the MMU.

Dataset Bug Detection. During this evaluation, we observed that TagASan and standard ASan often disagreed on the faulty allocation site. We identified three causes for these discrepancies: attribution of corrupted pointers to the wrong object, inlining eliminating debug information, and ASan reporting multiple candidate allocations for global variables.

Out of the 223 evaluated bugs, QuickSafe successfully protected the correct object and detected the faulting access in 219 cases (98%). Of these, 60% were caught via guard pages, and 39% via precision checks for intra-page accesses.

For the remaining four bugs, we manually confirmed that the faulting access was not reached in the QuickSafe-instrumented binary. In two cases, compiler optimisations elided the bug altogether [58]. For example, a function containing a vulnerable stack buffer was inlined with QuickSafe but not with ASan, likely because ASan's instrumentation inflated the function's size beyond the inlining threshold. This change prevented the faulting access from occurring. In two other cases, bugs were never triggered because internal consistency checks aborted execution early.

In summary, QuickSafe correctly isolated the relevant allocation sites and the targeted guard pages successfully mitigated the faulting accesses for all evaluated bugs.

Juliet Test Suite. We evaluated the different guard page *modes* of QuickSafe using the Juliet Test Suite [42] to assess detection versus protection properties. We selected all relevant categories in the test suite and filtered out cases that do not deterministically incur memory safety violations. To prevent compiler optimisations from removing bugs, we compiled all test cases with `-O0`. We then used TagASan to identify the relevant allocation sites and applied QuickSafe's instrumentation in each configuration.

Table 2 summarises the detection results for each mode across spatial and temporal errors on the stack and heap. In underflow mode, QuickSafe detected all buffer underflow cases but missed overflows; in overflow mode, the reverse held. In both configurations, undetected faults accessed only intra-object padding — meaning that violations were *mitigated* but not *detected*. Some overflows also went undetected in overflow mode when the object size was not a multiple of the required alignment (16 bytes) and the overflow offset was too small to reach the guard page. In precision mode,

CWE	bad (#)	QS-U	QS-O	QS-dbg	QS-MTE
121 Stack BOF	2736	0%	82%	100%	82%
122 Heap BOF	3214	0%	82%	100%	82%
124 BUW	978	100%	0%	100%	100%
126 BOR	642	0%	93%	100%	93%
127 BUR	978	100%	0%	100%	100%
415 DF	784	100%	100%	100%	100%
416 UAF	375	100%	100%	100%	100%

TABLE 2: Juliet Test Suite bug detection results. QS-U and QS-O use guard pages with under- and overflow alignment. QS-dbg inserts precision checks, QS-MTE uses Arm MTE.

QuickSafe detected *all* evaluated violations. With MTE enabled (Section 8.3), QuickSafe detected the same set of bugs as the union of underflow and overflow modes. The only missed cases involved small overflows that failed to cause tag mismatches — the same corner cases noted earlier.

SPEC CPU. We also ran the SPEC CPU2006 and CPU2017 suites with TagASan (and ASan as a control) to enumerate known memory safety violations, revealing four global buffer overflows: in `400.perlbench`, `464.h264ref`, `602.gcc`, and `625.x264`. We applied QuickSafe to the allocation sites identified by TagASan and instrumented the programs using precision mode. In all cases, QuickSafe successfully detected the bug and protected the corresponding memory objects against the faulty access.

Finally, we measured the number of *allocation sites* in each SPEC CPU benchmark to evaluate whether the 2^{15} tag limit imposed by Intel LAM is sufficient in practice. The largest program (`502.gcc`) contained 30,501 allocation sites. The geometric mean was 935 for SPEC CPU2006 and 990 for SPEC CPU2017 — suggesting that TagASan's identifier space is likely sufficient for most applications. Alternatively, with Arm TBI or AMD UAI providing 8 pointer tag bits, pinpointing the allocation site may require multiple PoC runs, reducing the candidate set by 2^8 per run.

8.2. Guard Pages: Performance

We evaluated the runtime and memory overhead introduced by QuickSafe when selectively isolating vulnerable objects using guard pages. Our evaluation covered 180 bugs from our custom dataset with valid benchmarking workloads (Section 7), along with the four known buggy SPEC CPU benchmarks. While ASan is not designed for deployment, we include it as a *reference point* for full instrumentation.

We compiled all programs using Clang 16.0.6 and LLD 16.0.6 with the same compiler and linker flags (excluding project-specific requirements), using `-O3` and full LTO. All experiments ran on an AMD Ryzen 9 9950X (32-core, 5.75 GHz) with 128 GB RAM, running Ubuntu 24.04 (Linux 6.8.0-55). We executed each benchmark ten times, calculated the median runtime across these runs, and present aggregate overheads as the geometric mean of the medians. Across all benchmarks, runtime and memory variance remained low (coefficient of variation $<0.6\%$).

As QuickSafe applies targeted instrumentation, we need to confirm that the evaluated benchmarks actually exercised

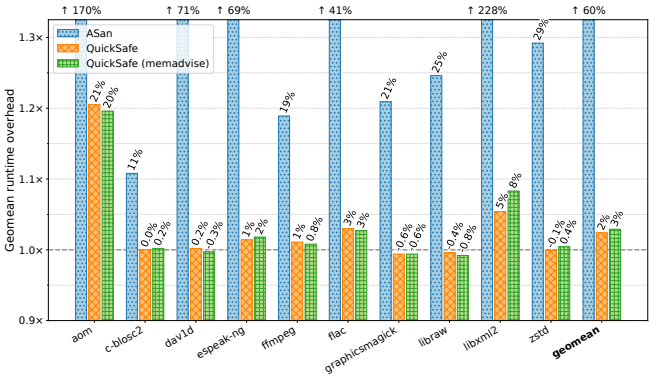

Figure 7: Geomean runtime overhead of QuickSafe (and ASan for reference) per project, and across all benchmarks.

protected code paths. We manually verified that QuickSafe redirected *at least one allocation* to the isolated allocator in over 90% of benchmarks. This confirms that our reported overheads reflect genuine runtime activity. In the remaining cases, the relevant allocation site was not exercised, suggesting that the bug and its associated objects lie on a fully unexecuted (*cold*) path. Similarly, the consistently low overheads we observed indicate that, even when isolation occurred, the protected objects were allocated infrequently — confirming QuickSafe's core assumption that memory bugs and their associated objects reside on cold paths, which we further verify with a separate profiling run (showing that only $\approx 1.3\%$ of all allocations are actually guarded).

We used QuickSafe's guard page strategy in *underflow mode*. Two `espeak-ng` benchmarks were run in *overflow mode* to avoid triggering the targeted bug. To evaluate memory reclamation behaviour, we also tested a variant where the isolated allocator explicitly invoked `madvise(DONTNEED)` on deallocation. While not required for correctness, this configuration helps quantify the impact of more aggressive memory reclamation strategies.

Custom Dataset. Figure 7 shows the geometric mean runtime overhead per project, as well as the overall geometric mean across all benchmarks. We compare three configurations: ASan, QuickSafe, and QuickSafe with explicit memory reclamation (`memadvise`). With a geomean slowdown of 2.42% in its default configuration, QuickSafe's runtime overhead is modest. The memory-reclaiming variant introduces slightly higher overheads (2.86%), but arguably still low enough for practical deployment.

The highest project geomean overhead (21%) occurs in `aom`, driven by a single systematic outlier (OSS-Fuzz #11517). The targeted object — involved in a stack buffer overflow — is allocated on a hot path, causing QuickSafe's instrumentation to trigger frequently. The resulting volume of protection-related system calls leads to a 4.71× slowdown. This overhead could be reduced by *hoisting* the stack object higher in the call graph, outside hot-path loops. The next highest geomean overhead (5.43%) arises in `libxml2`, again due to a single outlier (OSS-Fuzz #61337), a use-after-

Project	ASan		QuickSafe		QuickSafe (memadvise)	
aom	1.77x	(866.1 MB)	1.02x	(14.5 MB)	1.02x	(14.5 MB)
c-blosc2	1.66x	(277.3 MB)	1.16x	(72.8 MB)	1.08x	(32.4 MB)
dav1d	4.56x	(573.6 MB)	1.09x	(14.8 MB)	1.09x	(14.7 MB)
espeak-ng	3.33x	(456.0 MB)	1.12x	(23.6 MB)	1.07x	(14.6 MB)
ffmpeg	1.12x	(312.9 MB)	1.02x	(43.6 MB)	1.01x	(35.6 MB)
flac	3.08x	(10.1 MB)	4.03x	(14.7 MB)	4.01x	(14.7 MB)
graphicsmagick	1.13x	(107.8 MB)	1.02x	(13.0 MB)	1.02x	(13.0 MB)
libraw	1.42x	(151.7 MB)	1.03x	(9.6 MB)	1.03x	(12.8 MB)
libxml2	28.1x	(378.6 MB)	3.09x	(71.0 MB)	2.64x	(14.6 MB)
zstd	1.50x	(466.2 MB)	1.04x	(31.8 MB)	1.02x	(15.1 MB)
Mean	2.67x	(319.8 MB)	1.33x	(38.5 MB)	1.27x	(21.6 MB)

TABLE 3: Relative and absolute memory overhead of QuickSafe and ASan per project. Mean aggregate overhead is used to account for (rare) negative absolute values.

free bug where the targeted object is repeatedly allocated and deallocated, resulting in an overall 3.16× slowdown.

These two cases account for just two of the 180 tested benchmarks (1.11%), confirming that while QuickSafe may incur higher overheads when targeting hot-path allocations, such cases are rare in practice. Omitting these two outliers, QuickSafe's average slowdown drops to just 0.91% (1.01% with explicit memory reclamation).

Table 3 shows that QuickSafe's average memory overhead is 38.5 MB (33%) in its base configuration, and 21.6 MB (27%) with explicit reclamation. Notably, `madvise()` nearly halves the memory overhead in absolute terms. Again, one outlier (#61337 in `libxml2`) significantly skews the average — the same hot-path bug responsible for elevated runtime overhead. Rapid allocation and deallocation caused the OS to retain memory pages, even though QuickSafe marked them inaccessible. Enabling explicit memory reclamation drops the overhead to expected levels (14 MB). Moreover, the negligible additional runtime overhead of explicit reclamation shows that doing so by default is also feasible for real-world applications.

Scalability. To assess the scalability of QuickSafe, we conducted an experiment where we harden against multiple bugs from our data set at the same time. Specifically, to account for version drift causing bugs and allocation sites to move or disappear, we targeted all the bugs that live in the same software source tree. This selection yields four projects (`espeak-ng`, `ffmpeg`, `graphicsmagick`, `libraw`) with 3, 4, 5, and 5 concurrent unique targets. The measured overhead of these projects remains practically equivalent to when just a single bug is targeted, regardless of how many concurrent bugs we enabled. See Figure 8 in the Appendix for more details. Moreover, our MTE-based strategy naturally scales well, as non-targets are typically also tagged, keeping tagging overhead unchanged.

SPEC CPU. Because QuickSafe explicitly requires a known vulnerability to target, applying it to benchmarks without such bugs would leave all allocations unchanged — no instrumentation would execute, and the results would be uninstructive. Accordingly, we restricted our evaluation to the four SPEC programs with known bugs; the remaining benchmarks contain no known bugs.

Benchmark	Runtime overhead		Memory overhead	
	ASan	QuickSafe	ASan	QuickSafe
400.perlbench	6.45x	1.00x	4.45x	1.04x
464.h264ref	2.58x	1.01x	6.46x	1.39x
602.gcc_s	3.26x	1.00x	1.69x	1.00x
625.x264_s	5.43x	1.00x	1.31x	1.13x
Geomean	4.15x	1.00x	2.83x	1.13x

TABLE 4: Runtime and memory overhead of QuickSafe and ASan on SPEC CPU programs known to contain bugs.

For those four programs — though a small set, they are representative of performance-sensitive environments — none exhibited unusual memory behaviour, so we report only the base configuration results in Table 4. Across all four, QuickSafe introduces no measurable runtime overhead and a modest 13% memory overhead. This increase stems from linking against the C++ standard library (used by QuickSafe), whereas the evaluated SPEC programs are otherwise purely C-based.

We also evaluated the impact of replacing all deallocation sites with a wrapper function (to ensure guarded allocations are not passed to the default deallocator) on the complete SPEC CPU2006 and CPU2017 suites. On SPEC CPU2006, this incurred a negligible 0.6% geomean runtime overhead; on SPEC CPU2017, it incurred 2.1%. In the latter case, `508.namd` skews the overhead—experiencing a 36% slowdown (possibly due to code alignment changes) — while the other programs show negligible slowdown.

8.3. Memory Tagging: Case Studies

To evaluate QuickSafe with memory tagging isolation, we used two devices: a Google Pixel 8 Pro with MTE support hosting an Ubuntu (24.04.2) chroot environment (to measure performance), and a MacBook M2 Pro hosting an AArch64 QEMU image with emulated MTE support (to assess security). Due to the complexity of running these experiments (limited resources and overheating on the Pixel, and CPU-emulation for MTE being slow), we evaluate effectiveness and performance using a representative subset from our large dataset, using the most recent (i.e., highest-ID) bug from each program. For `espeak-ng`, we chose the last *reproducible* case, as several failed to reproduce on Arm (with QuickSafe and ASan). As all ten selected bugs are heap-related, we included the most recent *stack* memory violation in the dataset for completeness (`libraw` #60728).

Security. For each bug, we present a brief case-study describing the flaw and how QuickSafe protected against the memory error. Guarded objects in these experiments received a memory tag between `9` and `15`. For each bug, we manually confirmed that the MTE violation backtrace matched the ASan report, ensuring they are the same bug.

aom #67216 (heap buffer overflow) Pointer `bptr` (Listing 2) was assigned tag `10` by QuickSafe's guarded allocator. On dereference, MTE raised a violation because

the pointer referenced memory with tag 0. Subtracting one from the pointer returned it to the correctly tagged region. Since tag 0 is reserved for glibc metadata header chunks, this indicated an off-by-one overflow just past the original heap allocation. The memory originated from the fuzzing harness input buffer.

c-blosc2 #50991 (heap buffer overflow) This bug arose from a memset (Listing 3) whose size argument evaluated to nearly 300 trillion bytes. The target pointer op bore tag 10. The associated object, allocated at blosc2.c:1810 via c-blosc2's my_malloc, was correctly instrumented by QuickSafe. As the object was only 640 bytes in size, the operation overflowed into memory tagged 0.

dav1d #57927 (heap use-after-free) The memcpy operation in Listing 4 dereferenced memory that had previously been deallocated. The pointer carried tag 13, assigned by QuickSafe's allocator, but the accessed memory had tag 0, indicating glibc had revoked the tag upon free. As the memory had not yet been reused, its tag remained zero. The original allocation was via an instrumented realloc at obu.c:1497.

espeak-ng #37410 (heap use-after-free) This use-after-free involved pointer tr (Listing 5), which originally received tag 10 from QuickSafe's allocator (instrumented at tr_languages.c:241). At the point of dereference, the memory held tag 12, indicating it had already been reallocated to a different guarded object. The resulting mismatch reliably detected the temporal violation.

ffmpeg #62089 (heap buffer overflow) Here, QuickSafe assigned tag 10 to a 262148-byte calloc allocation referenced by buf (Listing 6). A derived pointer, level2_codecounts, was offset 131072 bytes into the object. A loop controlled by alphabet_size = 0x10000 wrote past the object's end. MTE raised a tag mismatch violation 12 bytes beyond the object boundary — consistent with its 16-byte tagging granularity.

flac #61292 (heap use-after-free) Pointer op2 (Listing 7) was dereferenced after its memory had been freed via the call to realloc at options.c:744. Although QuickSafe had assigned tag 10, the memory reverted to tag 0, indicating it was no longer valid. The mismatch reliably flagged the use-after-free.

graphicsmagick #58661 (heap buffer overflow) This bug involved an out-of-bounds access through the filename field (Listing 8), a member of a clone_info object allocated by GraphicsMagick's custom allocator at image.c:1328 (instrumented by QuickSafe). At the time of the access, *i held the value −2286 (0xfffffffffffff712), exceeding the 2053-byte filename buffer and underflowing the object by one byte. The pointer carried tag 14, while the accessed memory had tag 0, triggering a mismatch.

libraw #60728 (stack buffer overflow) This case demonstrated QuickSafe's ability to protect stack variables under MTE. The bug targeted a 2D array named color (Listing 9), which QuickSafe promoted to a guarded heap allocation. The pointer was tagged 10. Since the buffer's size was a multiple of MTE's 16-byte granularity, QuickSafe detected the off-by-one access caused by h = 3 (d = 0), which attempted to access a glibc chunk header (tag 0).

Listing 2 Bug location of aom #67216 (heap buffer overflow).

```
dif ^= (od_ec_window)bptr[0] << s; // entdec.c:94
```

Listing 3 Bug location of c-blosc2 #50991 (heap buffer overflow).

```
memset(op,0,blockshape[0]*blockshape[1]); // ndlz4x4.c:540
```

Listing 4 Bug location of dav1d #57927 (heap use-after-free).

```
free(itut_t35_ctx->itut_t35); // picture.c:133
...
memcpy(p->itut_t35, itut_t35, n_itut_t35 *
↪  sizeof(*itut_t35)); // picture.c:268
```

Listing 5 Bug location of espeak-ng #37410 (heap use-after-free).

```
free(tr); // translate.c:284
...
n = tr->groups2_count[c]; // dictionary.c:2237
```

Listing 6 Bug location of ffmpeg #62089 (heap buffer overflow).

```
buf = av_calloc(1, 262148); // jpegxl_parser.c:732
level2_codecounts = (uint32_t *)(buf + 131072); // 741
for (int i = 1; i < dist->alphabet_size + 1; i++) // 783
  level2_codecounts[i] += level2_codecounts[i-1]; // 784
```

Listing 7 Bug location of flac #61292 (heap use-after-free).

```
op->argument.import_cuesheet_from.add_seekpoint_link =
↪  &(op2->argument.add_seekpoint); // options.c:256
```

Listing 8 Bug location of GM #58661 (heap buffer overflow).

```
if(clone_info->filename[*i]=='.') // topol.c:340
```

Listing 9 Bug location of libraw #60728 (stack buffer overflow).

```
int color[3][8]; // xtrans_demosaic.cpp:213
// xtrans_demosaic.cpp:275
color[h][d] = g + rix[i << c][h] + rix[-i << c][h];
```

Listing 10 Bug location of libraw #60983 (heap buffer overflow).

```
memmove(ptr, buf + streampos, to_read); //
↪  libraw_datastream.cpp:367
```

Listing 11 Bug location of libxml2 #67670 (heap use-after-free).

```
xmlFree(cur); // tree.c:3636
...
while (cur->parent) // api.c:603
```

Listing 12 Bug location of zstd #44239 (heap buffer overflow).

```
// zstd_compress_literals.c:49
ostart[0] = (BYTE)((U32)set_basic + (srcSize<<3));
```

libraw #60983 (heap buffer overflow) This bug involved a memmove (Listing 10) exceeding the bounds of its destination. Pointer ptr referenced a 40960-byte region guarded by QuickSafe with tag 10. The operation attempted to transfer 61381 bytes, resulting in an overflow beyond the allocation. MTE raised a tag mismatch just after the object boundary (40960 is a multiple of 16). The object was allocated at unpack.cpp:237. ASan misattributed the fault, whereas TagASan was successful.

libxml2 #67670 (heap use-after-free) Pointer cur (Listing 11) was dereferenced after its memory had been freed at tree.c:3636. The pointer bore tag 13; the memory held tag 0, triggering a mismatch. The object had originally

been allocated via `xmlMalloc` at `tree.c:2275`, which QuickSafe correctly instrumented. ASan attributed the fault to the incorrect object; TagASan again succeeded.

zstd #44239 (heap buffer overflow) This subtle bug involved an out-of-bounds write to `ostart[0]` (Listing 12), associated with a 28593-byte object tagged `10`. The object's size was 15 bytes short of a 16-byte multiple, so the allocator padded the end. The off-by-two write landed in this padding, avoiding a tag mismatch. While no violation occurred, the access remained safe due to the padding, demonstrating mitigation even in the absence of a tag violation.

Performance. QuickSafe reuses the original `glibc` allocator's tag generation mechanism, and only adds *six additional instructions*, which are non-branching and do not affect registers or cache lines, and should therefore incur minimal overhead. We confirm this by building and running the benchmarks for the case studies on the Google Pixel 8 Pro, which supports native MTE. To minimize noise from thermal throttling and cache/storage effects on the Pixel, we placed the phone on a cooling pad with fans directed at its chassis. Prior to measurements, each workload was executed in warm-up runs to allow the CPU, cache, and storage subsystems to reach a steady state. During our attempts to build ffmpeg, the Pixel 8 Pro repeatedly exceeded its thermal limits — triggering emergency shutdowns. We thus excluded the benchmark. Under these conditions, QuickSafe's allocator incurred an average overhead of 0.12%, which lies within the observed run-to-run fluctuation ($\pm 1.05\%$) of the benchmarks (Figure 9). Consequently, even on thermally constrained mobile hardware, QuickSafe's additional instructions indeed impose no meaningful performance penalty.

9. Discussion

QuickSafe responds to memory safety violations by logging the fault and aborting execution. While this is appropriate for preventing exploitation, it may be undesirable in environments that prioritise availability. In such cases, QuickSafe could raise an alert while allowing execution to continue. For guard pages, this means removing the page protection (`PROT_NONE`) after the first access; for MTE, this involves removing `PROT_MTE` from the affected page.

Exploitability. QuickSafe correctly identifies and protects the relevant objects for all bugs in our dataset, based on the provided PoCs. However, these PoCs may not reflect the full exploitability of each bug. For example, a PoC might trigger a benign off-by-one access, while the underlying vulnerability permits arbitrary memory corruption. As such, QuickSafe cannot guarantee comprehensive protection against all possible exploit variants. A more complete characterisation could be achieved by extending our dataset with *PoC farming* [59], [60], which generates diverse triggering inputs for a single bug. We leave this to future work.

QuickSafe's guard page strategy inherits the limitations of redzones: large spatial violations may bypass isolation undetected. While the MTE-based strategy provides stronger guarantees, it is constrained by current MTE implementations. First, the limited tag space (16 tags) introduces a non-zero chance that two isolated or regular objects share a tag, allowing undetected intra-domain violations. Our default configuration partitions the tag space into disjoint domains, ensuring inter-domain isolation but reducing tag diversity within each domain compared to `glibc`'s default. Second, sufficiently large pointer offsets may still corrupt a tag, assuming the attacker knows the tag of the target memory.

Ultimately, QuickSafe is not intended as permanent defence. Rather, it serves as a low-cost, easily deployable stopgap to bridge the window between disclosure and patching.

C++ Container Support. Our prototype does not support C++ standard library container types (e.g., `std::vector`) as isolated objects. These types typically act as handles to internal dynamic allocations rather than storing data inline, which complicates finding relevant allocations. Supporting them would require deeper integration with the C++ runtime or extensions to the compile-time pass. As this limitation is not fundamental, we leave it to future work.

10. Related Work

Bug Mitigations. Many systems have been proposed to mitigate memory errors in deployment. These target either spatial bugs [61], [62], [63], [64], [65], [17], [21], temporal bugs [44], [19], [66], [67], [50], [68], [69], [70], or both [71], [72], [22], [73]. Low-overhead solutions typically support only a limited class of bugs — for instance, heap-only errors [17], [18] or buffer *overflows* but not underflows [16].

More recently, tools offering *selective* protection have gained traction due to their flexibility. Examples include MiraclePtr [74], libc++ hardening [75], CheckedC [76], and LLVM Bounds Safety [77], which support *incremental adoption* by isolating instrumentation to selected code regions. Similarly, GWP-ASan [36] and KFENCE [37] apply low-rate sampling to detect bugs in production with minimal overhead. QuickSafe complements these efforts by offering precise, low-overhead isolation for both spatial and temporal memory errors through selective instrumentation.

Memory Tagging. Memory tagging has seen significant research interest in recent years, driving advances in sanitizer techniques [78], [79], [54], x86-based alternatives [80], and runtime mitigation strategies [21], [81], [82]. Modern memory allocators increasingly adopt tagging schemes: for example, KASAN's SLUB allocator [83], glibc [84], Scudo [85], and PartitionAlloc [86] each implement distinct policies based on the shared fundamental principle of assigning unique tags to memory objects. QuickSafe integrates seamlessly with such systems, extending them with disjoint tag domains to enforce *stronger* isolation for specific objects. For example, QuickSafe can *enhance* the split tag domains introduced by Color My World [82] by dedicating a tag to target unsafe allocation sites, preventing those objects from corrupting other unsafe sites, strengthening the protection.

Targeted Hardening. VulShield [31], a concurrent effort, derives runtime security policies from sanitizer reports and enforces them via a Linux kernel module. Policies are defined as Boolean conditions, but the authors report that only 56.6% of patches can be expressed this way. Although VulShield hardens binaries, it still requires source code for analysis and depends on a custom kernel module — a notable deployment barrier in end-user and constrained environments (e.g., cloud or containerised systems).

In contrast, QuickSafe requires no kernel modifications or elevated privileges, and supports targeted protection for all evaluated memory errors. While recompilation is needed, no special system changes are required. VulShield claims comparable overheads, but its evaluation is limited: beyond synthetic tests, it includes only a single real-world case — a 2013 Nginx bug tested over loopback with low traffic and short timeframes, making results hard to generalise.

PET [87] also derives protection policies from sanitizer reports, enforcing them via kernel instrumentation and eBPF. While our prototype targets userspace, QuickSafe's design generalises to kernel code without fundamental changes.

Unlike VulShield and PET, QuickSafe does not rely on sanitizer reports for correctness — which, as shown, can misattribute faults under pointer corruption. VulShield additionally depends on SVF [88] for object size inference, which may be incomplete or unscalable in large programs. QuickSafe only requires the allocation site — which can be identified reliably with TagASan or even manually.

Other targeted hardening approaches include InstaGuard [89] (hardware watchpoints on Android); Talos [90] and RVM [91] (which avoid vulnerable functions); Phoenix [92] (syscall sequence filtering for containerised platforms); kernel patching methods [93], [94], [95]; and automatic patch generation [27], [28], [29], [30]. While these may overlap with QuickSafe in some scenarios, they pursue fundamentally different goals.

11. Conclusion

Despite extensive research, existing mitigations against memory bugs often remain impractical due to compatibility issues or performance overhead. This paper proposed a new approach: by focusing on known, unresolved bugs — which typically reside on *cold paths* — we enable targeted hardening with minimal cost. For this, we introduced *TagASan*, an extension of ASan that accurately attributes memory errors to their originating allocation sites. We then presented *QuickSafe*, a selective protection system with two example implementations: one that isolates vulnerable objects using guard pages (for compatibility) and one with disjoint tag domains (via Arm MTE). To evaluate our design, we built a dataset of 223 real-world memory errors with realistic performance benchmarks. QuickSafe successfully hardens all evaluated bugs against exploitation, with practical overhead.

Acknowledgements

We thank the reviewers for their feedback. This project was co-funded by the European Union under ERC Advanced Grant no. 101141972 ("Ghostbuster"), and ERC Starting Grant no. 101115046 ("SecuStack"). It was additionally funded by NWO under Gravitation grant no. 024.006.037 ("CiCS") and grant no. NWA.1215.18.006 ("Theseus"), and the Dutch Ministry of Economic Affairs and Climate under the AVR "VeriPatch" project. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union, the European Research Council or any of the funding agencies. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] P. Kehrer. (2019) Memory Unsafety in Apple's Operating Systems. Blogpost. Langui.sh. [Online]. Available: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/

[2] G. T. C. Projects. (2020) Memory safety. Blogpost. Google - The Chromium Projects. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/

[3] G. Thomas and S. Fernandez. (2019) A proactive approach to more secure code. Blogpost. Microsoft - Security Research & Defense. [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[4] R. Levick and S. Fernandez. (2019) We need a safer systems programming language. Blogpost. Microsoft - Security Research & Defense. [Online]. Available: https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/

[5] D. Hosfelt. (2019) Implications of Rewriting a Browser Component in Rust. Blogpost. Mozilla Hacks. [Online]. Available: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/

[6] J. van der Stoep and C. Zhang. (2019) Queue the Hardening Enhancements. Blogpost. Google - Security Blog. [Online]. Available: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html

[7] M. Stone. (2022) The More You Know, The More You Know You Don't Know. Blogpost. Google - Project Zero. [Online]. Available: https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html

[8] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *International Symposium on Code Generation and Optimization (CGO)*, 2011, pp. 213–223.

[9] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation," in *ACM SIGPLAN Notices*, 2007.

[10] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight Bounds Checking," in *Tenth International Symposium on Code Generation and Optimization (CGO)*, 2012, pp. 135–144.

[11] 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 *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 805–822.

[12] F. Gorter and C. Giuffrida, "RangeSanitizer: Detecting Memory Errors with Efficient Range Checks," in *USENIX Security*, 2025.

[13] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021.

[14] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for Security," in *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019, pp. 1275–1295.

[15] S. Nagy, "Address Sanitizer local root," 2016. [Online]. Available: https://www.openwall.com/lists/oss-security/2016/02/17/9

[16] 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.

[17] 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 *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 535–552.

[18] Z. Yu, G. Yang, and X. Xing, "ShadowBound: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 7177–7193.

[19] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, "DangZero: Efficient Use-After-Free Detection via Direct Page Table Access," in *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022, pp. 1307–1322.

[20] K. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "Top of the Heap: Efficient Memory Error Protection of Safe Heap Objects," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1330–1344.

[21] F. Gorter, T. Kroes, H. Bos, and C. Giuffrida, "Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags ," in *2024 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2024, pp. 4239–4257.

[22] 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 *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[23] A. Richardson, "Complete spatial safety for C and C++ using CHERI capabilities," University of Cambridge, Tech. Rep., 2020.

[24] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High System-Code Security with Low Overhead," in *IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 866–879.

[25] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed Data-Intensive systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 249–265.

[26] T. Kellogg, "Cold Paths," 2021. [Online]. Available: https://timkellogg.me/blog/2021/01/29/cold-paths

[27] A. Alhefdhi, H. K. Dam, T. Le-Cong, B. Le, and A. Ghose, "Adversarial patch generation for automated program repair," *Software Quality Journal*, vol. 33, no. 1, p. 12, 2025.

[28] J. Kim and S. Kim, "Automatic patch generation with context-based change application," *Empirical Software Engineering*, 2019.

[29] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, 2016.

[30] Y. Nong, H. Yang, L. Cheng, H. Hu, and H. Cai, "Automated software vulnerability patching using large language models," *arXiv*, 2024.

[31] Y. Li, C. Zhang, J. Zhu, P. Li, C. Li, S. Yang, and W. Tan, "VulShield: Protecting Vulnerable Code Before Deploying Patches," in *NDSS*, 2025.

[32] Arm. (2021) Memory Tagging Extension. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

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

[34] Intel, "Intel® architecture memory encryption technologies," 2021.

[35] D. Schrammel, M. Unterguggenberger, L. Lamster, S. Sultana, K. Grewal, M. LeMay, D. M. Durham, and S. Mangard, "Memory Tagging using Cryptographic Integrity on Commodity x86 CPUs," in *IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, 2024.

[36] K. Serebryany, C. Kennelly, M. Phillips, M. Denton, M. Elver, A. Potapenko, M. Morehouse, V. Tsyrklevich, C. Holler, J. Lettner *et al.*, "Gwp-asan: Sampling-based detection of memory-safety bugs in production," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, 2024.

[37] Linux, "Kernel Electric-Fence (KFENCE)," 2021. [Online]. Available: https://docs.kernel.org/dev-tools/kfence.html

[38] X. Mei, P. S. Singaria, J. Del Castillo, H. Xi, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, H. Pearce, B. Dolan-Gavitt *et al.*, "ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software," *arXiv preprint arXiv:2408.02153*, 2024.

[39] M. Larabel. (2024) Phoronix Test Suite. Version 10.8.4. [Online]. Available: https://www.phoronix-test-suite.com/

[40] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006. [Online]. Available: https://doi.org/10.1145/1186736.1186737

[41] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

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

[43] Microsoft, "GFlags and PageHeap," 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap

[44] 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.

[45] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 2006, pp. 269–280.

[46] Arm, "Arm Cortex-A Series: Programmer's Guide for ARMv8-A," 2015, chapter 12.5.1: Virtual Address tagging.

[47] A. Developers, "Arm Memory Tagging Extension (MTE)," Online, 2025, https://developer.android.com/ndk/guides/arm-mte.

[48] A. S. Engineering and A. (SEAR), "Memory Integrity Enforcement: A complete vision for memory safety in Apple devices," Online, 2025, https://security.apple.com/blog/memory-integrity-enforcement/.

[49] Intel, "Intel Architecture Instruction Set Extensions and Future Features," 2020, chapter: Linear Address Masking.

[50] L. Bernhard, M. Rodler, T. Holz, and L. Davit, "xTag: Mitigating use-after-free vulnerabilities via software-based pointer tagging on Intel X86-64," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 502–519.

[51] 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.

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

[53] Linux, "is_kfence_address," 2021. [Online]. Available: https://elixir.bootlin.com/linux/v6.13.7/source/include/linux/kfence.h#L51

[54] LLVM, "MemTagSanitizer," Online, 2020, https://llvm.org/docs/MemTagSanitizer.html.

[55] LLVM. (2025) Using the New Pass Manager. [Online]. Available: https://llvm.org/docs/NewPassManager.html

[56] LLVM. (2021) The New Pass Manager. [Online]. Available: https://blog.llvm.org/posts/2021-03-26-the-new-pass-manager/

[57] Google, "OSS-Fuzz," Online, 2021, https://google.github.io/oss-fuzz/.

[58] R. Isemann, C. Giuffrida, H. Bos, E. Van Der Kouwe, and K. v. Gleissenthall, "Don't look UB: Exposing sanitizer-eliding compiler optimizations," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 907–927, 2023.

[59] Z. Jiang, S. Gan, A. Herrera, F. Toffalini, L. Romerio, C. Tang, M. Egele, C. Zhang, and M. Payer, "Evocatio: Conjuring bug capabilities from a single poc," in *Proceedings of the 2022 ACM SIGSAC conference on computer and communications security*, 2022.

[60] Google, "AFL crash exploration mode," Online, 2024, https://github.com/google/AFL\#10-crash-triage.

[61] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 245–258.

[62] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–30, 2018.

[63] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "PAriCheck: an efficient pointer arithmetic checker for C programs," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010, pp. 145–156.

[64] 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*, 2009.

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

[66] R. M. Farkhani, M. Ahmadi, and L. Lu, "PTAuth: temporal memory safety via robust points-to authentication," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1037–1054.

[67] H. Cho, J. Park, A. Oest, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[68] 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.

[69] S. Ainsworth and T. M. Jones, "MarkUs: Drop-in use-after-free prevention for low-level languages," in *2020 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020, pp. 578–591.

[70] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *2010 International Symposium on Memory Management (ISMM)*, 2010, pp. 31–40.

[71] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Everything you want to know about pointer-based checking," in *1st Summit on Advances in Programming Languages (SNAPL)*, 2015.

[72] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CUP: Comprehensive User-Space Protection for C/C++," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018.

[73] T. Zhang, D. Lee, and C. Jung, "BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free," in *ASPLOS*, 2019.

[74] Google, "raw_ptr<T> (aka. MiraclePtr, aka. BackupRefPtr, aka. BRP)," Online, 2022, https://chromium.googlesource.com/chromium/src/+/main/base/memory/raw_ptr.md.

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

[76] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked C: Making C safe by extension," in *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 2018, pp. 53–60.

[77] Clang/LLVM, "-fbounds-safety: Enforcing bounds safety for C," Online, 2023, https://clang.llvm.org/docs/BoundsSafety.html.

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

[79] 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 *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 841–858.

[80] D. Schrammel, M. Unterguggenberger, L. Lamster, S. Sultana, K. Grewal, M. LeMay, D. M. Durham, and S. Mangard, "Memory Tagging using Cryptographic Integrity on Commodity x86 CPUs," in *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, 2024.

[81] Google, "Retrofitting Temporal Memory Safety on C++," 2022. [Online]. Available: https://security.googleblog.com/2022/05/retrofitting-temporal-memory-safety-on-c.html

[82] 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.

[83] Linux, "The Kernel Address Sanitizer (KASAN)," Online, 2025, https://docs.kernel.org/dev-tools/kasan.html.

[84] GNU glibc, "38.7 Memory Related Tunables," Online, 2025, https://www.gnu.org/software/libc/manual/html_node/Memory-Related-Tunables.html.

[85] pcc, "scudo: Add initial memory tagging support," Online, 2019, https://reviews.llvm.org/D70762.

[86] R. Townsend, "feat: basic MTE support for the partition allocator," Online, 2021, https://chromium-review.googlesource.com/c/chromium/src/+/2695355.

[87] Z. Wang, Y. Chen, and Q. Zeng, "PET: Prevent discovered errors from being triggered in the linux kernel," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4193–4210.

[88] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 265–266.

[89] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wang, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[90] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response ," in *2016 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[91] Z. Huang and G. Tan, "Rapid Vulnerability Mitigation with Security Workarounds," in *Proceedings of the 2nd NDSS Workshop on Binary Analysis Research, ser. BAR*, vol. 19, 2019.

[92] H. Kermabon-Bobinnec, Y. Jarraya, L. Wang, S. Majumdar, and M. Pourzandi, "Phoenix: Surviving unpatched vulnerabilities via accurate and efficient filtering of syscall sequences," in *Proceedings of the 2024 Network and Distributed System Security Symposium*. Internet Society San Diego, CA, USA, 2024.

[93] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, "An investigation of the android kernel patch ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3649–3666.

[94] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in Linux (experience paper)," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 633–645.

[95] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 386–396.

Appendix

Guard Pages Scalability

Figure 8 displays the runtime overhead of incrementally targeting multiple bugs at the same time.

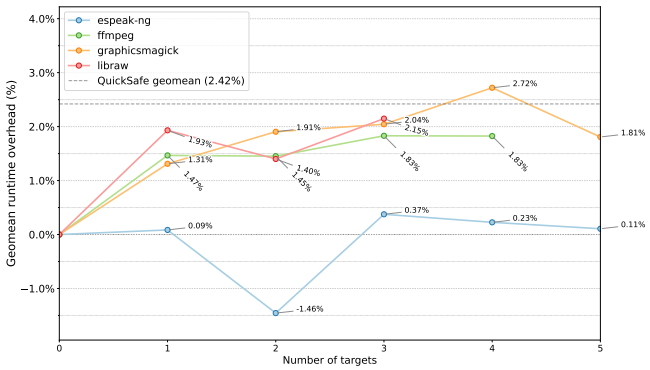

Figure 8: Runtime overhead of QuickSafe's guard page-based isolation for multiple concurrent targets.

MTE Performance

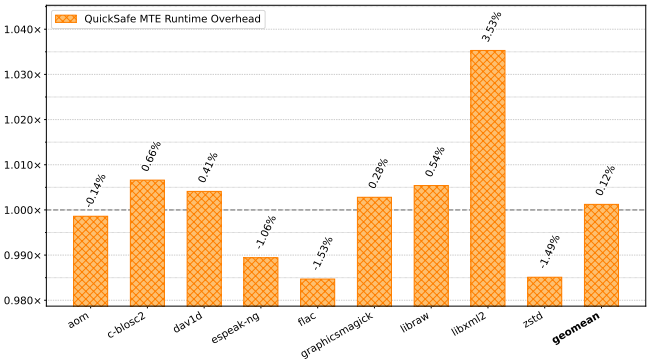

Figure 9: Runtime performance overhead of QuickSafe's MTE-based isolation strategy versus the default glibc MTE implementation, running on a Google Pixel 8 Pro with MTE enabled.

Figure 9 shows the slowdown of the performance evaluation run on the Google Pixel 8 Pro discussed in Section 8.3.

Dataset

Table 5 (below) lists the 223 memory errors included in our dataset (curated from OSS-Fuzz/ARVO). Each of the ten programs is paired with a corresponding performance benchmark from the Phoronix Benchmarking Suite. For each bug, the table provides the OSS-Fuzz ID and the allocation site (file and line number) as identified by TagASan.

project	id	allocation site (file)	line
aom	10200	aom/av1/common/frame_buffers.c	60
aom	10252	aom/av1/common/thread_common.c	499
aom	11517	aom/av1/common/reconintra.c	1138
aom	11523	aom/av1/av1_dx_iface.c	377
aom	12549	aom/av1/common/frame_buffers.c	76
aom	33480	aom/av1/common/alloccommon.c	199
aom	33511	aom/av1/common/alloccommon.c	199
aom	44862	aom/av1/decoder/decodeframe.c	2496
aom	49652	aom/av1/common/av1_common_int.h	1236
aom	67184	libfuzzer/FuzzerLoop.cpp*	596
aom	67216	libfuzzer/FuzzerLoop.cpp*	596
c-blosc2	24837	libfuzzer/FuzzerLoop.cpp*	544
c-blosc2	25118	libfuzzer/FuzzerLoop.cpp*	544
c-blosc2	26442	c-blosc2/tests/fuzz/fuzz_compress.c*	37
c-blosc2	27818	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	29369	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	29705	[...]/decompress/zstd_ddict.c	152
c-blosc2	29816	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	29976	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	30193	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	30772	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	30868	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	31120	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	31189	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	31535	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	31556	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	31585	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	31586	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	31710	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	32076	[...]/decompress/zstd_ddict.c	152
c-blosc2	33071	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	33251	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	33264	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	35086	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	35537	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	36103	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	41111	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	42131	c-blosc2/blosc/blosc2.c	1810
c-blosc2	42537	c-blosc2/blosc/blosc2.c	1810
c-blosc2	42562	c-blosc2/blosc/blosc2.c	1810
c-blosc2	43513	libfuzzer/FuzzerLoop.cpp*	584
c-blosc2	43519	c-blosc2/blosc/blosc2.c	1810
c-blosc2	46460	libfuzzer/FuzzerLoop.cpp*	596
c-blosc2	50433	c-blosc2/blosc/schunk.c	1024
c-blosc2	50991	c-blosc2/blosc/blosc2.c	1810
dav1d	38152	dav1d/decode.c	3090
dav1d	38369	dav1d/decode.c	3127
dav1d	38439	dav1d/decode.c	3127
dav1d	40661	dav1d/decode.c	3069
dav1d	57927	dav1d/obu.c	1497
espeak-ng	34285	espeak-ng/libespeak-ng/translate.c	2564
espeak-ng	34298	espeak-ng/libespeak-ng/tr_languages.c	241
espeak-ng	34299	espeak-ng/tests/ssml-fuzzer.c*	61
espeak-ng	34332	espeak-ng/libespeak-ng/translate.c	2026
espeak-ng	34494	espeak-ng/libespeak-ng/numbers.c	482
espeak-ng	34533	espeak-ng/tests/ssml-fuzzer.c*	61
espeak-ng	34718	espeak-ng/libespeak-ng/translate.c	98
espeak-ng	34747	espeak-ng/libespeak-ng/translate.c	88
espeak-ng	37238	espeak-ng/libespeak-ng/tr_languages.c	241
espeak-ng	37410	espeak-ng/libespeak-ng/tr_languages.c	241
espeak-ng	39042	espeak-ng/libespeak-ng/synthdata.c	986
espeak-ng	41437	espeak-ng/libespeak-ng/translate.c	2564
espeak-ng	41504	espeak-ng/libespeak-ng/translate.c	2026
espeak-ng	43037	espeak-ng/libespeak-ng/synthdata.c	986
espeak-ng	48589	espeak-ng/tests/ssml-fuzzer.c*	61
ffmpeg	1345	ffmpeg/libavcodec/dfa.c	49
ffmpeg	1348	ffmpeg/libavutil/buffer.c	85
ffmpeg	1354	ffmpeg/libavutil/bprint.c	50
ffmpeg	1378	ffmpeg/libavutil/buffer.c	85
ffmpeg	1427	ffmpeg/./libavutil/mem_internal.h	38
ffmpeg	1434	ffmpeg/libavutil/buffer.c	85
ffmpeg	1466	ffmpeg/libavcodec/xpmdec.c	234
ffmpeg	1478	ffmpeg/libavutil/buffer.c	85
ffmpeg	9215	ffmpeg/tools/target_dec_fuzzer.c*	96
ffmpeg	9350	ffmpeg/libavcodec/dvdsub_parser.c	57
ffmpeg	13552	ffmpeg/libavutil/buffer.c	85
ffmpeg	13851	ffmpeg/libavutil/buffer.c	85
ffmpeg	13997	ffmpeg/libavutil/buffer.c	177
ffmpeg	14168	ffmpeg/./libavutil/mem_internal.h	38
ffmpeg	15270	ffmpeg/libavutil/buffer.c	85
ffmpeg	16135	ffmpeg/libavutil/buffer.c	85
ffmpeg	17814	ffmpeg/libavcodec/hcom.c	60
ffmpeg	20494	ffmpeg/libavutil/buffer.c	177
ffmpeg	20742	ffmpeg/libavutil/buffer.c	178
ffmpeg	24636	ffmpeg/libavcodec/jpeg2000dec.c	948
ffmpeg	24653	ffmpeg/libavcodec/cfhd.c	319
ffmpeg	25425	ffmpeg/libavutil/buffer.c	85
ffmpeg	26490	ffmpeg/libavformat/vividas.c	478
ffmpeg	26821	ffmpeg/libavutil/buffer.c	72
ffmpeg	30135	ffmpeg/libavcodec/parser.c	259
ffmpeg	30744	ffmpeg/libavutil/buffer.c	178
ffmpeg	33402	ffmpeg/libavcodec/options.c	143
ffmpeg	36329	ffmpeg/tools/target_dec_fuzzer.c*	318
ffmpeg	36340	ffmpeg/tools/target_dec_fuzzer.c*	318
ffmpeg	36355	ffmpeg/tools/target_dec_fuzzer.c*	318
ffmpeg	36356	ffmpeg/tools/target_dec_fuzzer.c*	318
ffmpeg	38076	ffmpeg/libavcodec/decode.c	1369
ffmpeg	40109	ffmpeg/libavutil/fifo.c	51
ffmpeg	42821	ffmpeg/libavutil/buffer.c	82
ffmpeg	42938	ffmpeg/libavutil/buffer.c	95
ffmpeg	42939	ffmpeg/libavutil/buffer.c	95
ffmpeg	42940	ffmpeg/libavutil/buffer.c	95
ffmpeg	42944	ffmpeg/libavutil/buffer.c	95
ffmpeg	45722	ffmpeg/libavutil/buffer.c	192
ffmpeg	45846	ffmpeg/libavutil/buffer.c	82
ffmpeg	46194	ffmpeg/libavcodec/wmalosslessdec.c	341
ffmpeg	47871	ffmpeg/libavutil/buffer.c	82
ffmpeg	47877	ffmpeg/libavutil/buffer.c	82
ffmpeg	47899	ffmpeg/libavformat/mov.c	7518
ffmpeg	47911	ffmpeg/libavutil/buffer.c	82
ffmpeg	48281	ffmpeg/libavutil/buffer.c	82
ffmpeg	48429	ffmpeg/libavutil/buffer.c	82
ffmpeg	48799	ffmpeg/libavutil/buffer.c	82
ffmpeg	49271	ffmpeg/libavutil/buffer.c	82
ffmpeg	50014	ffmpeg/libavutil/buffer.c	82
ffmpeg	50970	ffmpeg/libavutil/buffer.c	82
ffmpeg	51648	ffmpeg/libavcodec/options.c	129
ffmpeg	53623	ffmpeg/libavcodec/vqcdec.c	82
ffmpeg	55926	ffmpeg/libavcodec/options.c	133
ffmpeg	57829	ffmpeg/libavutil/buffer.c	82
ffmpeg	58134	ffmpeg/libavcodec/hevc_ps.c	1698
ffmpeg	58185	ffmpeg/libavcodec/mjpegdec.c	767
ffmpeg	59640	ffmpeg/libavformat/jpegxl_anim_dec.c	141
ffmpeg	59673	ffmpeg/libavutil/buffer.c	82
ffmpeg	59828	ffmpeg/libavformat/jpegxl_anim_dec.c	141
ffmpeg	61991	ffmpeg/libavutil/buffer.c	192
ffmpeg	62089	ffmpeg/libavcodec/jpegxl_parser.c	732
flac	17069	flac/libFLAC/bitreader.c	268
flac	53454	flac/oss-fuzz/fuzzer_reencoder.cc*	129
flac	61292	flac/metaflac/options.c	744
graphicsmagick	7935	graphicsmagick/magick/pixel_cache.c	3345
graphicsmagick	7951	graphicsmagick/magick/omp_data_view.c	150
graphicsmagick	8600	graphicsmagick/coders/mat.c	587
graphicsmagick	8615	graphicsmagick/coders/png.c	4222
graphicsmagick	8743	graphicsmagick/coders/png.c	1109
graphicsmagick	8763	graphicsmagick/coders/png.c	1109
graphicsmagick	8834	graphicsmagick/coders/png.c	4222
graphicsmagick	9357	graphicsmagick/Magick++/lib/BlobRef.cpp	32
graphicsmagick	10055	graphicsmagick/magick/utility.c	6307
graphicsmagick	10096	graphicsmagick/magick/render.c	881
graphicsmagick	10400	graphicsmagick/coders/png.c	4196
graphicsmagick	10653	graphicsmagick/magick/render.c	2262
graphicsmagick	13160	graphicsmagick/magick/render.c	325
graphicsmagick	15707	graphicsmagick/magick/render.c	7067
graphicsmagick	20048	graphicsmagick/coders/pict.c	891
graphicsmagick	20053	graphicsmagick/coders/pict.c	891
graphicsmagick	20271	graphicsmagick/coders/pict.c	899
graphicsmagick	20272	graphicsmagick/coders/pict.c	899
graphicsmagick	20318	graphicsmagick/magick/pixel_cache.c	2497
graphicsmagick	21956	graphicsmagick/magick/pixel_cache.c	2497
graphicsmagick	22046	graphicsmagick/magick/pixel_cache.c	2497
graphicsmagick	23077	graphicsmagick/magick/pixel_cache.c	2497
graphicsmagick	32263	graphicsmagick/magick/image.c	336
graphicsmagick	40680	graphicsmagick/coders/msl.c	1948
graphicsmagick	43511	graphicsmagick/coders/pict.c	899
graphicsmagick	46843	graphicsmagick/magick/image.c	336
graphicsmagick	52084	graphicsmagick/magick/colormap.c	76
graphicsmagick	52133	graphicsmagick/Magick++/lib/BlobRef.cpp	32
graphicsmagick	52917	graphicsmagick/coders/txt.c	309
graphicsmagick	54716	graphicsmagick/coders/sun.c	586
graphicsmagick	54810	graphicsmagick/coders/sun.c	571
graphicsmagick	58661	graphicsmagick/magick/image.c	1328
libraw	22978	libraw/metadata/misc_parsers.cpp	294
libraw	22979	libraw/metadata/misc_parsers.cpp	294
libraw	23383	libraw/utils/utils_libraw.cpp	245
libraw	30324	libraw/decoders/fp_dng.cpp	602
libraw	42108	libraw/metadata/identify.cpp	704
libraw	52579	libraw/utils/thumb_utils.cpp	57
libraw	52588	libraw/postprocessing/postprocessing_aux.cpp	48
libraw	52672	libraw/utils/thumb_utils.cpp	57
libraw	52694	libraw/decoders/unpack.cpp	350
libraw	52802	libraw/utils/thumb_utils.cpp	57
libraw	52901	libraw/preprocessing/raw2image.cpp	85
libraw	60728	libraw/demosaic/xtrans_demosaic.cpp	213
libraw	60983	libraw/decoders/unpack.cpp	237
libraw	65027	libraw/tables/wblists.cpp	20
libxml2	1972	libxml2/valid.c	5441
libxml2	10304	libxml2/buf.c	172
libxml2	12419	libxml2/tree.c	1878
libxml2	17737	libxml2/entities.c	159
libxml2	23120	libxml2/tree.c	1880
libxml2	23765	libxml2/buf.c	137
libxml2	24055	libxml2/buf.c	137
libxml2	24925	libxml2/tree.c	2283
libxml2	25014	libxml2/tree.c	757
libxml2	25210	libxml2/tree.c	2179
libxml2	34461	libxml2/tree.c	4234
libxml2	46323	libxml2/tree.c	4247
libxml2	52554	libxml2/tree.c	2302
libxml2	55980	libxml2/buf.c	138
libxml2	56964	libxml2/parserInternals.c	319
libxml2	57073	libxml2/parserInternals.c	308
libxml2	57077	libxml2/parserInternals.c	289
libxml2	57080	libxml2/parserInternals.c	289
libxml2	57084	libxml2/parserInternals.c	289
libxml2	57110	libxml2/parserInternals.c	289
libxml2	57151	libxml2/parserInternals.c	289
libxml2	57224	libxml2/parserInternals.c	289
libxml2	57225	libxml2/parserInternals.c	289
libxml2	57284	libxml2/parserInternals.c	289
libxml2	57288	libxml2/parserInternals.c	289
libxml2	57294	libxml2/parserInternals.c	289
libxml2	57469	libxml2/parserInternals.c	289
libxml2	61337	libxml2/HTMLparser.c	383
libxml2	62911	libxml2/dict.c	160
libxml2	62996	libxml2/tree.c	749
libxml2	63086	libxml2/parser.c	4955
libxml2	63127	libxml2/parser.c	4955
libxml2	65363	libxml2/tree.c	718
libxml2	66446	libxml2/valid.c	2562
libxml2	66502	libxml2/valid.c	2562
libxml2	67560	libxml2/tree.c	719
libxml2	67586	libxml2/tree.c	719
libxml2	67670	libxml2/tree.c	2275
zstd	3522	zstd/lib/compress/zstd_compress.c	886
zstd	14368	libfuzzer/FuzzerLoop.cpp*	539
zstd	16445	zstd/lib/legacy/zstd_v03.c	2953
zstd	17451	libfuzzer/FuzzerLoop.cpp*	539
zstd	35209	zstd/tests/fuzz/simple_compress.c*	45
zstd	38553	zstd/tests/fuzz/simple_compress.c*	45
zstd	43365	zstd/lib/common/zstd_internal.h	195
zstd	44122	zstd/tests/fuzz/sequence_compression_api.c*	292
zstd	44239	zstd/tests/fuzz/sequence_compression_api.c*	300

TABLE 5: Memory errors dataset details. Bugs originate from the OSS-Fuzz [57] project, and were subsequently reproduced by ARVO [38]. Allocation sites marked with "$\star$" are excluded from performance benchmarks.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 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 presents a framework designed to mitigate known vulnerabilities until proper patches are available. It focuses on vulnerabilities in cold (infrequently executed) code paths and selectively applies memory-safety hardening techniques to vulnerable heap-based allocations, isolating them to contain cross-object memory violations. The authors enhance an existing memory-safety sanitizer to improve its accuracy in identifying the original allocation site for precise, targeted mitigations. The framework is evaluated on a curated dataset of vulnerable performance benchmarks reproduced from OSS-Fuzz/ARVO.

A.2. Scientific Contributions

- Provides a New Data Set For Public Use
- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. Although the techniques used in the paper are not particularly new, the authors effectively combine existing approaches into a system for **targeted, temporary vulnerability mitigation** with high precision and low overhead. Reviewers were positive towards targeted mitigations as promising solution to addressing the long-known issue of high performance overheads hindering widespread adoption of run-time memory-safety techniques.
2) The paper creates a new tool to enable future science. Reviewers highlighted improvements in **TagASan over ASan** as a significant contribution on its own.
3) The paper provides a new data set for public use. The authors have committed to open sourcing their **curated, functional, and reproducible OSS-Fuzz/ARVO benchmarks**. The reviewers appreciated the thorough evaluation which demonstrates the correctness of the targeted mitigations, supports the hypothesis that vulnerabilities often reside on cold paths, and measures the performance overhead thanks to being grounded in the Phoronix benchmarking suite.

A.4. Noteworthy Concerns

1) The current design and implementation do not address intra-object memory-safety violations nor arbitrary-offset spatial unsafety. These limitations can leave even protected objects exploitable.