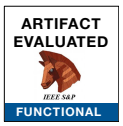

NANOTAG: Systems Support for Efficient Byte-Granular Overflow Detection on ARM MTE

Mingkai Li†, Hang Ye†, Joseph Devietti‡, Suman Jana†, Tanvir Ahmed Khan†
† *Columbia University* ‡ *University of Pennsylvania*
{*mingkai.li, hy2891*}*@columbia.edu, devietti@cis.upenn.edu, suman@cs.columbia.edu, tk3070@columbia.edu*

***Abstract*—Memory safety bugs, such as buffer overflows and use-after-frees, are the leading causes of software safety issues in production. Software-based approaches, *e.g.*, Address Sanitizer (ASAN), can detect such bugs with high precision, but with prohibitively high overhead. ARM's Memory Tagging Extension (MTE) offers a promising alternative to detect these bugs in hardware with a much lower overhead. In this paper, we perform a thorough investigation of the first production implementation of ARM MTE (Google Pixel 8) and observe that MTE can only achieve coarse precision in bug detection compared with software-based approaches such as ASAN, mainly due to its 16-byte tag granularity.**

To address this issue, we present NANOTAG[1], a system to probabilistically detect buffer overflows at byte granularity in unmodified MTE-enabled binaries with minimal changes to memory allocators, introducing an explicit detection-performance tradeoff for in-house testing. NANOTAG detects buffer overflows at byte granularity by setting up a tripwire for tag granules that may require intra-granule overflow detection. The memory access to the tripwire causes additional overflow detection in the software while using MTE's hardware to detect bugs for the rest of the accesses. We implement NANOTAG based on the Scudo Hardened Allocator, the default memory allocator on Android since Android 11. Our evaluation results across popular benchmarks and real-world case studies show that NANOTAG detects nearly as many memory safety bugs as ASAN while incurring similar run-time overhead to Scudo Hardened Allocator in MTE SYNC mode.

1. Introduction

Memory safety bugs, such as buffer overflows and use-after-frees, remain the dominant causes of software vulnerabilities—70% of Microsoft [1] and 51% of Android [2] vulnerabilities. Consequently, researchers have proposed a wide range of static and dynamic techniques over the years [3]–[15]. Among these techniques, memory safety sanitizers [8]–[15] provide high detection accuracy, as they observe the executions of applications directly.

Unfortunately, sanitizers incur significant run-time overhead. Address Sanitizer [8] and Hardware-assisted Address Sanitizer [15] impose ∼2× performance overhead [15], [16]. The overhead of binary instrumentation is even higher—3× for ASAN-Retrowrite [17], 17× for Valgrind [18], and 35× for QASAN [19]. During in-house testing, such as fuzzing, reduced throughput means sanitizers cannot run continuously. High-throughput fuzzers skip sanitization for most tests and re-run a reduced corpus with sanitization [20]. Skipping sanitization for most tests stops fuzzers from detecting silent memory safety bugs [21]. Detecting such silent bugs during in-house testing requires efficient sanitization.

MTE. Hardware vendors aim to make sanitization efficient by introducing mechanisms such as ARM's Memory Tagging Extension (MTE). MTE maintains tags for pointers and memory bytes. Matching a pointer's tag against the tag of any 16-byte memory the pointer accesses directly in hardware, MTE sanitizes memory accesses without heavy instrumentation. ARM reports only 1–2% overhead for the fastest MTE ASYNC mode [22][2]. Google Pixel 8 is the first implementation of MTE. Google also added support for MTE to Scudo Hardened Allocator [23], Android's default memory allocator, reporting no noticeable slowdown [24]. MTE has already helped catch real-world issues such as Google CVE-2024-23694 [25]. Consequently, MTE is one of the few sanitization techniques potentially usable continuously for in-house testing and fuzzing [26].

MTE's limitations. In this paper, we conduct a thorough investigation of Google Pixel 8 to show that MTE's low performance overhead comes with a significant loss in its bug detection capability. In particular, we find that MTE misses 24.32% of heap-based buffer overflows in Juliet Test Suite [27], whereas ASAN detects 98.66% of such overflows. The root cause is architectural: MTE assigns a 4-bit tag to each 16-byte memory, *tag granule*, and checks tags only at that granularity. MTE fails to detect any overflow that occurs within a single tag granule, even when it overwrites useful data [28] or padding [23], based on memory allocators. We refer to these silent cases as *intra-granule buffer overflows*.

These intra-granule buffer overflows have significant implications in testing and fuzzing [26], where the goal is to detect silent latent bugs. Such in-house testing may tolerate [20] up to 20% overhead of some MTE modes [29]. However, the loss in MTE's detection capability significantly undermines the effectiveness of in-house testing.

1. We open-source our work at https://github.com/ice-rlab/NanoTag.

2. MTE supports SYNC, ASYNC, and ASYMM modes; see §2.1.1.

Overflows that appear benign (*e.g.*, accessing padding bytes) may become exploitable under different inputs or memory layouts [30] that place critical data within the same 16-byte region. Thus, undetected intra-granule buffer overflows can directly lead to latent security vulnerabilities.

Quantitatively, in §2, we show that a significant fraction (86.57%) of memory allocations in the SPEC CPU 2017 benchmarks suite [31] allow potential intra-granule buffer overflows. Such overflows also cause real-world CVEs, including CVE-2024-12084 [32], rated critical by Google and Red Hat [33], [34]. Mitigating these issues requires byte-granular overflow detection, which MTE cannot provide.

Challenges. Byte-granular overflow detection is challenging due to significant run-time and memory overhead. Detecting such overflows in software requires additional instrumentation similar to existing sanitizers [8], [15], [17]–[19], adding at least 2× run-time overhead [16]. Moreover, additional metadata in the form of redzones [18], [35], [36] or shadow memory [18] increases memory usage by at least 10–35% [15]. In terms of memory overhead, hardware support is even more expensive as reducing MTE's granularity from 16 bytes to 1 byte would require storing 4 bits of metadata per byte, consuming one-third of physical memory.

Our Solution. In this paper, we present NANOTAG, the first system to detect memory safety bugs probabilistically at byte granularity on real ARM MTE hardware, addressing intra-granule buffer overflows in unmodified MTE-enabled binaries. During allocation, NANOTAG identifies tag granules that permit intra-granule buffer overflows, which we call *short granule*s. For these short granules, NANOTAG probabilistically enables byte-level checking in software while retaining MTE's native coarse-granular detection of buffer overflows and use-after-frees in hardware. In particular, NANOTAG's probabilistic approach introduces an explicit tradeoff between bug detection capability and performance overhead through a controllable sampling knob. Such a knob enables NANOTAG's primary use cases, *i.e.*, in-house testing (*e.g.*, fuzzing), to select the appropriate balance between detection strength and overhead. As a result, NANOTAG adds only minimal overhead on top of Scudo Hardened Allocator in MTE SYNC mode.

NANOTAG avoids high run-time overhead by relying primarily on hardware checks. It inserts additional software checks only while accessing a short granule. NANOTAG sanitizes such accesses via a *tripwire*: a tag granule tagged with a value intentionally different from the pointer's tag. NANOTAG installs tripwires probabilistically on short granules. While accessing a short granule with a tripwire, hardware raises a tag mismatch fault, triggering NANOTAG's software handler. The handler checks for buffer overflow. If the access is benign, NANOTAG resumes normal execution. For violations, NANOTAG reports detailed diagnostic information, including tag and register values.

NANOTAG avoids memory overhead by storing metadata inside unused padding bytes within each short granule and inferring access offsets from MTE SYNC mode information. Specifically, MTE SYNC mode provides the exception instruction and fault address, enabling NANOTAG to compute the exact byte offset of the access. The padding bytes inside a short granule contain its *real tag*. The tag NANOTAG assigns to the granule to install a tripwire, instead, encodes the number of valid addressable bytes. Upon a tag mismatch fault, NANOTAG extracts both tags from the granule, verifies that the real tag matches the pointer's tag, and confirms that the access is within the addressable region.

If the access is valid, NANOTAG temporarily removes the tripwire by restoring the granule's tag, sets a *trap* (e.g., hardware or software breakpoint [37]) immediately after the exception instruction, and replays the instruction without raising another fault. When the program hits the trap, NANOTAG reinstalls the tripwire and records how frequently it is triggered. Once a tripwire exceeds a threshold, NANOTAG removes it permanently to avoid unnecessary overhead.

We implement NANOTAG on top of Scudo and evaluate it systematically on Google Pixel 8. NANOTAG detects memory safety bugs at byte granularity while maintaining runtime overhead comparable to Scudo in MTE SYNC mode. On Juliet Test Suite, NANOTAG detects 97.57% heap-based overflows, closely matching ASAN (98.66%) and significantly outperforming both MTE SYNC (75.68%) and ASYNC (75.60%). On SPECrate Integer benchmarks, NANOTAG incurs a geomean overhead of 12.50%, similar to Scudo in MTE SYNC mode (11.98%). In real-world case studies, NANOTAG incurs 4.99% overhead on closed-source Geekbench 6 [38], up to 12.35% overhead on three large open-source real-world applications (`Memcached` [39], `LevelDB` [40], `RocksDB` [41]), and slows fuzzing throughput by only 15.86% across three Magma targets—one-seventh of ASAN's slowdown (111.20%).

Contributions. This paper contributes

- A thorough study of ARM MTE's hardware and software stack on Google Pixel 8 to quantitatively show that intra-granule buffer overflows expose a large surface of bypassing MTE checks (86.57%) in SPEC CPU 2017 benchmarks.
- NANOTAG, a system to efficiently detect memory safety bugs probabilistically in unmodified MTE-enabled binaries at byte granularity, addressing intra-granule buffer overflows in real hardware for the first time to help in-house testing detect such bugs with an explicit detection-performance tradeoff.
- An evaluation of NANOTAG's prototype on Juliet Test Suite and SPEC CPU 2017 benchmarks, to show that NANOTAG achieves similar bug detection capability (97.57%) as ASAN (98.66%) and incurs a run-time overhead of 12.50%, which is 7× lower than the 95.11% overhead of ASAN and comparable to the 11.98% overhead of the MTE SYNC mode.

2. Characterizing ARM's Memory Tagging Extension on Google Pixel 8

2.1. Background

We first describe ARM MTE's hardware and then provide a brief summary of the relevant software on Google

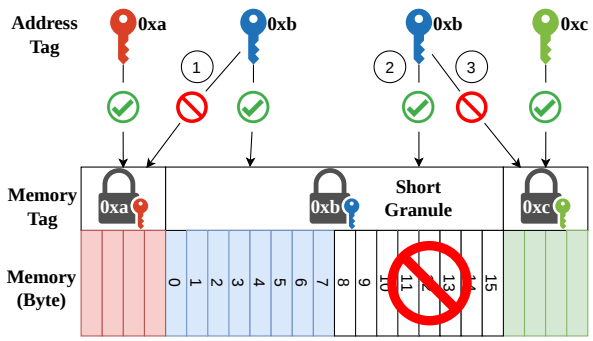

Figure 1. An overview of ARM MTE. Locks and keys represent memory and address tags, respectively. Addressable bytes are colored. The middle tag granule is a short granule. MTE detects both underflow (1) and overflow (3) across the tag granule boundary. However, it ignores the intra-granule buffer overflow (2).

Pixel 8.

2.1.1. Hardware. In a nutshell, MTE detects memory safety bugs using a lock-and-key approach [42], as we show in Fig. 1. While accessing memory bytes with a pointer, the pointer carries a *key* associated with a *lock* that protects the memory bytes. MTE permits the memory access only if the pointer's key matches the memory bytes' lock. Otherwise, MTE generates an exception to report a memory safety bug.
MTE Tag Management. ARM MTE implements these locks and keys with *tags*. A tag is a 4-bit token stored either inline at the pointer or at a carved-out *tag storage* in the memory [43]. The tag that is inline at the pointer is called the *address tag* (the key), storing at the leftmost byte of the pointer. MTE does not treat it as part of the virtual address with the Top Byte Ignore (TBI) feature [44]. The tag that is at the carved-out tag storage is called the *memory tag* (the lock). MTE divides the physical address space into 16-byte chunks called *tag granule*, associating a memory tag in the tag storage with it. There is also a dedicated *tag cache* for memory tags [45]. While bringing memory bytes to a processor's cache, ARM MTE also prefetches the corresponding memory tag from tag storage to tag cache [45].
MTE Tag Check. For each memory access, MTE compares the memory tag in the tag storage (the lock) with the address tag of the pointer (the key). MTE permits the memory access only if address tag and all memory tags involved in the memory access are the same. Otherwise, the processor raises a *tag mismatch fault* either synchronously or asynchronously.
MTE Modes. MTE modes determine when MTE reports the tag mismatch fault. MTE currently supports 3 different modes [46]: (1) SYNC, (2) ASYNC, and (3) ASYMM. In MTE SYNC mode, the tag mismatch causes a synchronous exception immediately, while also recording the precise location of the *exception instruction* and the *fault address*. In MTE ASYNC mode, the tag mismatch fault causes an exception at the nearest kernel entry, *e.g.*, a system call or a timer interrupt. The processor continues execution after a tag mismatch fault until the nearest kernel entry, recording only an imprecise location near the exception instruction. ARM also introduces MTE ASYMM mode to handle read and write memory accesses in SYNC and ASYNC modes, respectively. However, Android does not allow users to manually enable MTE ASYMM mode. Instead, MTE ASYNC mode is redefined as MTE ASYMM mode in compatible devices [46].

TABLE 1. INSTRUCTIONS IN ARM MTE.

Usage	Instructions
Random tag generation	`IRG`, `GMI`
Pointer arithmetic	`ADDG`, `SUBG`, `SUBP`
Tag load and store	`LDG`, `STG`, `STZG`, `ST2G`, `STZ2G`, `STGP`
Tag load and store (privileged)	`LDGM`, `STGM`, `STZGM`
Data cache operations	`DC GVA`, `DC GZVA`, `DC IGDSW`, `DC IGDVAC`, `DC IGSW`, `DC IGVAC`

2.1.2. Instruction Set Architecture. Once both address tags and memory tags are properly set up, ARM MTE detects memory safety bugs for every load and store instruction [47] by comparing the tags in the hardware [42]. To manage tags, MTE adds several instructions that we list in Table 1. In particular, MTE provides `IRG` and `GMI` instructions to generate a random tag with a certain mask. While generating random tags, MTE excludes values in the mask. To set up the address tag, MTE provides pointer arithmetic instructions, *e.g.*, `ADDG` and `SUBG`. An instruction tags an untagged pointer either directly from a generated random tag or based on an existing tagged pointer. To set up the memory tag, MTE provides tag load and store instructions, *e.g.*, `LDG` and `STG`, to access the tag storage with the virtual address of the tag granule. When storing to the tag storage, instructions can choose to zero (*e.g.*, `STZG`) or initialize (*e.g.*, `STGP`) the corresponding tag granule. A single user-space instruction could set up the memory tag for at most 2 tag granules [42]. Setting up the memory tag for multiple tag granules (maximum 16) requires instructions that the processor can execute only in the privileged mode (*Exception Level (EL)* ≥ 1). Along with these MTE instructions that prior works have studied [29], [42], [43], [48], we find that ARM also introduces data cache operations that allow setting up the memory tag for multiple tag granules in the user space. For example, Google uses the `DC GZVA` instruction in Scudo to set up the memory tags of the whole cache line and zero its content [49].

2.1.3. Software. Google Pixel 8 has end-to-end support for MTE in software, including Instruction Set Architecture [42], operating system [50], and Scudo Hardened Allocator [23]. Briefly, the boot loader carves out around 3% of the physical memory for tag storage at the system start up [51]. The Android operating system supports configuring MTE via system calls [50] and Android Debugging Bridge [24]. Android's default memory allocator,

TABLE 2. HEAP-BASED MEMORY SAFETY BUGS IN JULIET TEST SUITE. SCUDO ONLY DETECTS 75% OF BUFFER OVERFLOW BUGS IN MTE SYNC AND ASYNC MODES, WHILE ASAN DETECTS ALMOST ALL (98.66%) OF THEM.

CWE ID	Total	ASAN	GWP-ASan	Scudo* (ASYNC)	Scudo* (SYNC)
CWE122 (heap-based buffer overflow)	3438	98.66%	23.15% ± 0.24%	75.60% ± 0.13%	75.68% ± 0.10%
CWE415 (double free)	818	100%	98.60% ± 0.23%	98.43% ± 0.14%	98.45% ± 0.19%
CWE416 (use after free)	393	100%	0%	96.54 ± 0.60%	96.94% ± 0.58%

*The variance comes from MTE's tag collision across multiple runs.

Scudo [23], uses MTE to detect memory safety bugs for memory allocations by its primary allocator [43].

2.2. How does MTE compare against purely software-based techniques?

In this section, we characterize ARM MTE's bug detection capability on Google Pixel 8 using NIST Juliet Test Suite (v1.3) [27], similar to prior work [9], [11], [29], [48], [52]. Among different Common Weakness Enumerations (CWEs) of Juliet Test Suite, out-of-bound write, out-of-bound read, and use after free are ranked 2nd, 6th, and 8th of the top 25 most dangerous software weaknesses in 2024 [53]. We select the corresponding heap-related CWEs: `CWE122` (heap-based buffer overflow), `CWE415` (double free), and `CWE416` (use after free). Each CWEs include both benign and buggy test cases. As benign test cases do not include any memory safety bug, a sound bug detector should not report false positives. The buggy test cases include one or more memory safety bugs. Similar to prior work [11], we exclude some `CWE122` test cases to avoid infinite waiting and nondeterministic results: (1) `CWE129_listen_socket` and `CWE129_connect_socket` requiring external inputs; (2) `CWE129_rand` triggering bugs randomly. Due to MTE's probabilistic nature, we run each CWE with Scudo multiple times to report the average number. For comparison, we run Juliet Test Suite with ASAN [8], GWP-ASan [54], and Scudo in MTE SYNC and ASYNC modes. For ASAN, we disable Leak Sanitizer [55] to ensure that ASAN only detects buffer overflows and use-after-frees. For GWP-ASan, we set its sampling rate to the default value (5000). Due to sampling's probabilistic nature, we also run each CWE multiple times with GWP-ASan.

Results. In our experiment, ASAN, GWP-ASan, and Scudo do not report any of the benign test cases as bugs. Therefore, we only show the results of buggy test cases in Table 2. Table 2 shows the percentages of memory safety bugs ASAN, GWP-ASan, and Scudo detect. As we show, ASAN detects almost all memory safety bugs. In comparison, GWP-ASan falls significantly short as it detects memory safety bugs in sampled memory allocations for in-production deployment.

For Scudo, it has almost the same results in MTE SYNC and ASYNC modes. We show that, although Scudo with MTE support detects most of the temporal memory safety bugs, *i.e.*, `CWE415` (double free) and `CWE416` (use after free), it detects significantly fewer spatial memory safety issues, *i.e.*, `CWE122` (heap-based buffer overflow), than ASAN.

For buffer overflows (`CWE122`), Scudo detects only 75.68% and 75.60% of heap-based buffer overflow bugs in MTE SYNC and ASYNC modes, while ASAN detects almost all (98.66%) of them. By analyzing the bugs Scudo fails to detect, we find that Scudo is less precise than ASAN, mainly due to its 16-byte tag granularity. The missed bugs in `CWE122` are mostly intra-granule buffer overflows, *i.e.*, a buffer overflow that occurs within the 16-byte tag granule. Fig. 1 shows an example of such an intra-granule buffer overflow. For access 2 in Fig. 1, it accesses the 12th byte in the tag granule, while only the first 8 bytes are addressable. Therefore, access 2 results in an intra-granule buffer overflow. However, due to MTE's coarse tag granularity, MTE fails to detect such an intra-granule buffer overflow, as MTE permits accesses to the full tag granule even though only part of it is addressable.

Observation: MTE detects only up to 75.68% heap-based buffer overflow bugs on Juliet Test Suite, while ASAN detects almost all (98.66%) of them.
Insight: *MTE ignores intra-granule buffer overflows due to 16-byte tag granularity.*

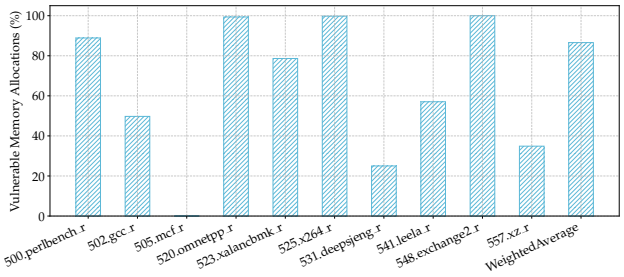

Figure 2. Percentage of vulnerable memory allocations in SPEC CPU 2017 benchmarks. 86.57% of all allocations in SPEC CPU 2017 benchmarks include short granules, leading to potential intra-granule buffer overflows. The only outlier is `505.mcf_r` (0.05%).

2.3. How frequently do programs allocate short granules?

To understand the surface of bypassing MTE checks due to missed intra-granule buffer overflows, we analyze the memory allocations in SPEC CPU 2017 benchmarks. Our experiment setup of SPEC CPU 2017 benchmarks is consistent with our evaluation, which is described in §5.1 and §5.3. A memory allocation is a *vulnerable memory allocation* if it contains a short granule. We calculate the

```
#if defined SHA512_DIGEST_LENGTH
#define MAX_DIGEST_LEN SHA512_DIGEST_LENGTH
#elif defined SHA256_DIGEST_LENGTH
#define MAX_DIGEST_LEN SHA256_DIGEST_LENGTH
#elif defined SHA_DIGEST_LENGTH
#define MAX_DIGEST_LEN SHA_DIGEST_LENGTH
#else
#define MAX_DIGEST_LEN MD5_DIGEST_LEN
#endif
```

md-defines.h, adapted from [57]

```
struct sum_buf {
  OFF_T offset;          /*offset in file of this chunk*/
  int32 len;             /*length of chunk of file*/
  uint32 sum1;           /*simple checksum*/
  int32 chain;           /*next hash-table collision*/
  short flags;           /*flag bits*/
  char sum2[SUM_LENGTH];/*checksum*/
};
```

rsync.h, adapted from [58]

```
struct sum_struct *s = new(struct sum_struct);
// . . .
s->sums = new_array(struct sum_buf, s->count);
for (i = 0; i < s->count; i++) {
    s->sums[i].sum1 = read_int(f);
    read_buf(f, s->sums[i].sum2, s->s2length);
// . . .
```

sender.c, adapted from [59]

Figure 3. CVE-2024-12084 [32], an intra-granule heap-based buffer overflow bug in *rsync* that becomes a silent data corruption in MTE, while both ASAN and NANOTAG detect it.

percentage of vulnerable memory allocations as the number of memory allocations, whose size is not a multiple of 16 bytes, divided by the total number of memory allocations in the benchmark.

Results. As shown in Fig. 2, 86.57% of all memory allocations in SPEC CPU 2017 benchmarks include short granules, leading to potential intra-granule buffer overflows. Specifically, 70% and 50% of benchmarks have more than 40% and 60% vulnerable memory allocations, respectively. We also observe that there are three benchmarks (520.omnetpp_r, 525.x264_r, and 548.exchange2_r) with more than 99% vulnerable memory allocations. This illustrates the ubiquitousness of vulnerable memory allocations in various applications. One potential approach to reduce the ubiquity of these vulnerable memory allocations would be to use a smaller tag granularity than 16 bytes. Unfortunately, such an approach would increase the size of the tag storage, carving out more physical memory, $\frac{1}{3}$ of it in the extreme case (1-byte tag granularity). For example, while studying the impact of different tag granularities, prior work [56] argued that the size of an optimal tag granularity is 16 bytes as a larger tag granularity significantly increases the memory overhead due to alignment.

Observation: 86.57% of all memory allocations in SPEC CPU 2017 benchmarks include short granules.
Insight: *Short granules, leading to intra-granule buffer overflows MTE ignores, are ubiquitous in allocations.*

2.4. How do intra-granule buffer overflows lead to undetected real-world vulnerabilities?

In Fig. 3, we show the code snippets of a real-world vulnerability, CVE-2024-12084 [32]. This vulnerability is due to a heap-based buffer overflow bug in *rsync* [60]. In particular, the buffer overflow occurs at line 6 of sender.c, while filling up s->sums[i].sum2 with client-provided s->s2length bytes [33]. As shown in rsync.h, sum2 is an array of SUM_LENGTH bytes, which is a field of a 40-byte structure, sum_buf. The attacker exploits this vulnerability by controlling the value of s->s2length, whose maximum value MAX_DIGEST_LEN is typically larger than SUM_LENGTH, as defined in md-defined.h. As the size of sum_buf (40 bytes) is not a multiple of 16 bytes, the last tag granule in the memory allocation has 8 non-addressable bytes. Therefore, MTE ignores any overflow into these 8 non-addressable bytes in the last tag granule.

We implement a simple Proof-of-Concept (PoC) of CVE-2024-12084 by setting s->s2length to 24, initiating an intra-granule buffer overflow in the last tag granule of sum_buf. As we evaluate this PoC with MTE-enabled Scudo, we observe that MTE fails to detect the overflow in both SYNC and ASYNC modes due to its 16-byte tag granularity. On the other hand, both ASAN and NANOTAG prevent this PoC by detecting intra-granule buffer overflows.

Observation: Ignoring intra-granule buffer overflows, MTE fails to detect a PoC of CVE-2024-12084.
Insight: *Detecting intra-granule buffer overflows helps uncover real-world vulnerabilities.*

3. Design

In Fig. 4, we show NANOTAG's workflow to detect memory safety bugs in unmodified MTE-enabled binaries at byte granularity. With an overview of NANOTAG's design goals in §3.1, we describe its components in §3.2-§3.5.

3.1. Design Goals

NANOTAG aims to detect memory safety bugs in unmodified binaries during in-house testing. If the application binary contains memory safety bugs, *i.e.*, buffer overflows or use-after-frees, NANOTAG's goal is to detect these bugs: **(D1)** at byte granularity, **(D2)** without instrumenting the source code or the binary, and **(D3)** with run-time overheads similar to MTE-enabled Scudo. With these design goals, NANOTAG ensures that in-house testing, *e.g.*, fuzzing campaigns, (1) does not ignore potential in-production vulnerability because of overflows to padding bytes, (2) detects as many bugs as possible, and (3) runs as fast as possible. NANOTAG achieves these design goals with two layers of sanitizations. For the first layer of sanitization, NANOTAG detects buffer overflows at 16-byte granularity and use-after-frees for all memory allocations with ARM MTE's hardware checks. For the second layer of sanitization, NANOTAG

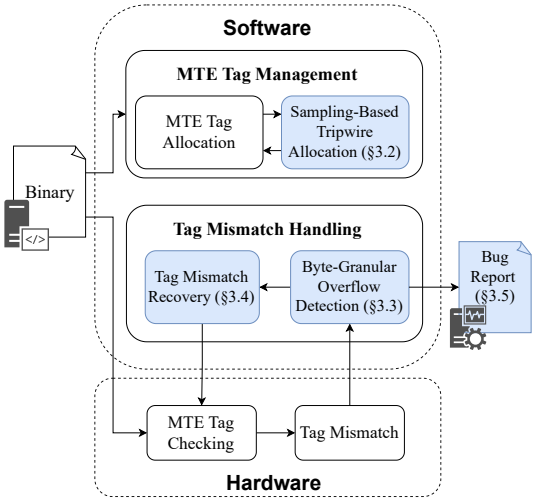

Figure 4. NANOTAG's workflow. Blue blocks specify additional components in NANOTAG, while white blocks denote unmodified components in MTE's hardware and software.

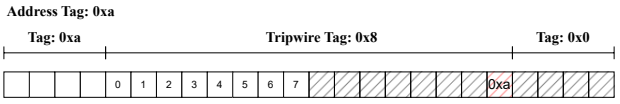

Figure 5. An example of how NANOTAG assigns tags to short granules where only a fraction of 16 bytes is addressable (the middle one). Blocks in hatches indicate non-addressable bytes. We highlight the last byte of the short granule in red.

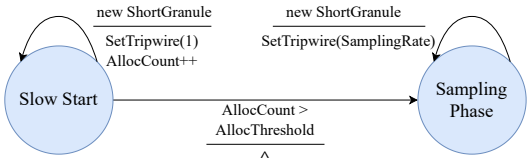

Figure 6. NANOTAG's sampling-based tripwire allocation.

detects buffer overflows at byte granularity, including intra-granule buffer overflows, for sampled memory allocations (§3.2) with NANOTAG's additional software checks (§3.3, §3.4). If NANOTAG detects memory safety bugs based on software and hardware checks, it reports them to application developers. The developer then analyzes the bug report (§3.5) to fix these bugs before deploying the application in production. We assume micro-architectural attacks against MTE [29], [61]–[63] and adversarial attacks to bypass byte-level checks are outside the scope of NANOTAG's primary use case, in-house testing.

3.2. Sampling-Based Tripwire Allocation

NANOTAG avoids instrumenting the source code or the binary (D2) by setting up tripwires that invoke additional software checks. NANOTAG sets up these tripwires via sampling, amortizing the overhead of additional software checks (D3).

Tripwire. As we show in Fig. 1, a *short granule* is a tag granule where only part of its 16 bytes is addressable, *e.g.*, the tag granule in the middle of Fig. 1. To detect intra-granule buffer overflows in the short granule, NANOTAG needs to differentiate memory accesses to the short granule from other granules. Therefore, NANOTAG probabilistically sets up a *tripwire* for such short granules, *i.e.*, NANOTAG allocates the short granule a special *tripwire tag*, which is different from other tags in the same memory allocation. Similar to HWASAN [15], NANOTAG sets the tripwire tag as the number of addressable bytes in the short granule, which is `0x8` for the short granule (the middle one) in Fig. 5. Consequently, when a pointer accesses the tripwire, it always causes a tag mismatch fault, either from a true memory safety bug or only from a safe program access to the short granule's addressable bytes, which NANOTAG intercepts in its tag mismatch handler (§3.3). To identify the root cause of the tag mismatch fault, NANOTAG stores the legit address tag of the memory allocation, which is `0xa` in Fig. 5, in the last 4 bits of the short granule since the last byte (highlighted in red) of the short granule is always unused. For tag granules other than the short granules, their memory tags are randomly-generated and consistent with the pointer's address tag (*e.g.*, address tag and memory tag `0xa` in Fig. 5). While resizing and freeing the allocation, NANOTAG needs to propagate and clear out the last 4 bits of the short granule.

Sampling. Similar to MTE's probabilistic detection of memory safety bugs, NANOTAG also detects intra-granule buffer overflows probabilistically via sampling, amortizing the overhead of additional software checks. As we show in Fig. 6, while setting up tripwires, NANOTAG employs sampling in two phases: slow start and sampling phase. In the slow start phase, for every new short granule allocation (`new ShortGranule`), NANOTAG sets a tripwire (`SetTripwire(1)`), while also incrementing the *AllocCount*. As *AllocCount* reaches above a threshold (*AllocThreshold*), NANOTAG transitions from the slow start phase to the sampling phase. In the sampling phase, NANOTAG sets the tripwire only for the sampled ones (`SetTripwire(SamplingRate)`). NANOTAG uses a sampling algorithm similar to GWP-ASan [54]. NANOTAG generates a random number (`rand`) from `[1, 2 * SamplingRate]`, and allocates a tripwire after every `rand` short granule allocations. After setting up a tripwire, NANOTAG generates the random number `rand` again for the subsequent tripwire. By default, NANOTAG sets both *SamplingRate* and *AllocThreshold* to 1000, which we empirically find to yield high bug detection capability (97.57%, §5.5). Both of these parameters are configurable.

Implications of Sampling. For NANOTAG's primary use case, *i.e.*, in-house testing, as stronger detection capability incurs higher overhead, NANOTAG makes this tradeoff explicit with a dynamically configurable interface. By configuring this interface dynamically, NANOTAG's probabilistic

approach makes in-house testing techniques (*e.g.*, fuzzing) more effective. For example, as fuzzers observe interesting cases (*e.g.*, new path), they can increase NANOTAG's bug detection capability. For uninteresting cases, fuzzers can instead prioritize performance. As fuzzing typically executes thousands of executions per second, many of which result in similar program behavior, the fuzzer is effective as long as it can detect the bug at one of these executions.

3.3. Byte-Granular Overflow Detection

NANOTAG implements a byte-granular overflow detection algorithm (Algorithm 1) in the tag mismatch handler.

Information Extraction. In MTE SYNC mode, NANOTAG gets necessary information (*i.e.*, exception information) from the tag mismatch fault including: the exception instruction's precise location (*i.e.*, program counter, `pc`), fault address of the memory, values of general-purpose registers (`x0` ~ `x30`) and the stack pointer register (`sp`). NANOTAG uses this exception information to infer all inputs in Algorithm 1. For fault address (`f`), NANOTAG directly obtains it from the exception information. For the starting address (`start`) and size (`size`) of the memory access, NANOTAG first reads the memory bytes corresponding to `pc` and then decodes them to get the exception instruction. Combining this decoded instruction with values of registers (`x0` ~ `x30`, `sp`), NANOTAG obtains `start` and `size`. In ARM MTE, only memory access instructions, *i.e.*, ARM load and store instructions [47], can cause an MTE tag mismatch fault. Based on the ISA specification of these instructions [47], NANOTAG extracts the size of the memory access (`size`) from the opcode, and decodes memory-access-related fields, *e.g.*, the base register index and the memory access offset, from the instruction encoding. NANOTAG calculates the starting address of the memory access (`start`) with values of registers based on the addressing mode and memory-access-related fields of the instruction. NANOTAG infers memory tag (`memtag`) and the last 4 bits (`metadata`) of the short granule from the fault address (`f`). Assuming the fault address is inside the short granule, NANOTAG uses the normal load instruction (`LDRB`) to obtain the last 4 bits of the short granule (`metadata`), and uses MTE tag load instruction (`LDG`) to obtain the memory tag of the short granule (`memtag`). NANOTAG infers the address tag (`addrtag`) with both the exception instruction and values of registers. Similar to `start` and `size`, NANOTAG decodes the exception instruction to get the base register index and the offset. The offset can be either an immediate value in the instruction or a value in a register. If the offset is an immediate value, NANOTAG takes the topmost byte of the base register as the address tag (`addrtag`). Otherwise, NANOTAG calculates the address tag (`addrtag`) with both the topmost byte of the base register and the topmost byte of the offset register.

Overflow Detection. NANOTAG determines whether a memory access is benign or a memory safety bug based on a set of addressability properties. In NANOTAG, a byte's addressability on the heap follows the following properties:

Algorithm 1 NANOTAG's byte-granular overflow detection.

Input: fault address (`f`), starting address (`start`) and size (`size`) of the memory access, address tag (`addrtag`), memory tag (`memtag`), and the last 4 bits (`metadata`) of the short granule.

Output: **true** if the memory access is benign, **false** if it is a memory safety bug.

```
1:  if memtag = 0 or addrtag = 0 then
2:      return false
3:  if addrtag ≠ metadata then
4:      return false
5:  else
6:      shortgranule ← f & ¬(16 − 1) ▷ starting address of the short granule
7:      permitted ← shortgranule + memtag ▷ end address of addressable bytes in the short granule
8:      attempted ← start + size ▷ end address of the memory access
9:      if attempted ≤ permitted then
10:         return true
11:     else
12:         return false
```

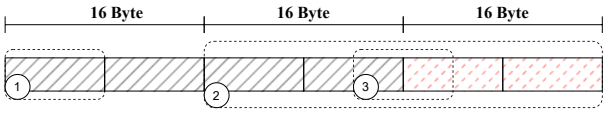

Figure 7. Examples of memory access patterns of ARM load and store instructions. The red block represents the tripwire. The memory access can be unaligned, *e.g.*, access 3, and may access multiple tag granules, *e.g.*, access 2 and 3.

(P1): The byte is non-addressable if its memory tag is 0.

(P2): The byte is non-addressable for a pointer with address tag 0.

(P3): The byte is addressable when its memory tag matches the address tag of the pointer.

(P4): The byte is addressable when it is the addressable part of a short granule, and the last 4 bits of the short granule match the address tag of the pointer.

As we show in Algorithm 1, NANOTAG rejects a memory access if its address tag or memory tag is zero based on P1 and P2 (lines 1~2). As ARM MTE protects memory regions by tagging them with a randomly selected memory tag which is always not zero [43], a tag granule with memory tag being zero means that it is either not allocated or not protected by MTE. Therefore, P1 and P2 prevent buffer overflows that are across heap, stack, or code segments. A tag mismatch fault occurs when P3 is violated. NANOTAG implements Algorithm 1 in the tag mismatch handler to analyze whether the memory access that causes the tag mismatch fault is benign or a memory safety bug based on P4. It rejects the memory access if the last 4 bits of the tag granule is not the same as the address tag of the pointer (line 3~4), which indicates that the tag mismatch fault is not caused by a tripwire.

As we show in Fig. 7, ARM load and store instructions

can be unaligned, accessing multiple tag granules. For example, a store instruction (access 2) can write two 16-byte values from two registers into two tag granules in the memory. Similarly, a load instruction (access 3) can read memory starting from any address, which may cross the tag granule's boundary. Therefore, NANOTAG determines whether the instruction accesses short granule's addressable bytes by calculating both the end address of addressable bytes in the short granule (`permitted`, line 6˜7) and the end address of the bytes the instruction accesses (`attempted`, line 8). As we show in Algorithm 1, NANOTAG only permits memory accesses if the `attempted` last byte of the memory access is not larger than the `permitted` last byte of the short granule (line 9˜12). As long as the instruction is accessing at least one of the non-addressable bytes in the short granule, the end address of the memory access (`attempted`) is always going to be bigger than the end address of addressable bytes in the short granule (`permitted`). Consequently, NANOTAG will flag such an access as a memory safety bug.

3.4. Tag Mismatch Recovery

In the tag mismatch handler, NANOTAG resumes the program execution if the memory access is benign. However, resuming the program execution after a tag mismatch fault is not trivial. If NANOTAG resumes the program without removing the tripwire, the program will access the tripwire, causing a tag mismatch fault again. On the other hand, removing the tripwire before resuming the program avoids the tag mismatch fault but disables the byte-granular overflow detection in the tag granule ever after. NANOTAG addresses this issue by presenting a *delegation-escalation-revocation* approach. Overall, NANOTAG achieves the following goals:
(G1). The processor executes all instructions in the program order, no matter whether a tag mismatch fault occurs or not.
(G2). NANOTAG revokes the permission as soon as possible.
(G3). After NANOTAG revokes the permission, accessing the tripwire will trigger a tag mismatch fault.

Delegation. NANOTAG resolves the tag mismatch fault by manipulating the tags involved in the memory access. In particular, as we show in Fig. 8(A), NANOTAG removes the tripwire by setting the tripwire tag to be the same as the address tag of the pointer. As NANOTAG needs to reset the tripwire tag during *revocation*, NANOTAG swaps the memory tag and the last 4 bits of the corresponding short granule. The program behavior stays the same as if the tag mismatch fault had not occurred, since the program executes the same instruction (G1). NANOTAG does not modify the address tag of the pointer, since a single instruction can access multiple tag granules that do not share the same memory tag, as we show for accesses 2 and 3 in Fig. 7. Modifying the address tag of the pointer to match one of the memory tags may cause a tag mismatch fault in another tag granule. NANOTAG also avoids emulating the exception instruction as, in software, it is challenging to precisely emulate instructions that require atomicity (*e.g.*, `RCWCAS`) or exclusive access (*e.g.*, `STTXR`), causing the program to have a different behavior.

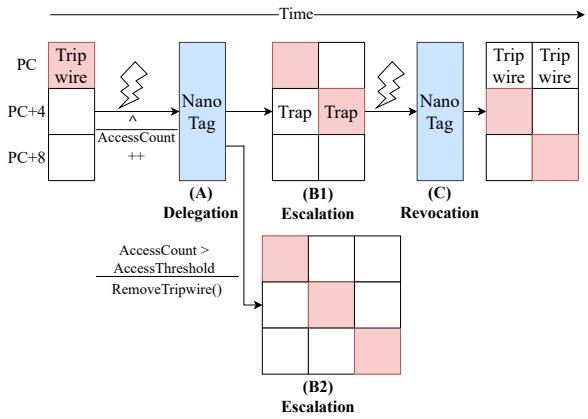

Figure 8. Delegation-escalation-revocation in NANOTAG. A red block denotes the program instruction processor currently executes, blue blocks represent NANOTAG's byte-granular overflow detection and tag mismatch recovery, and lightning symbols denote tag mismatch fault and trap.

Escalation. After *delegation*, the program can use the pointer to access every byte in the delegated short granule. If NANOTAG does not revoke the permission promptly, the program will run in an escalated state for a long time, violating (G2). Therefore, NANOTAG sets a *trap* right after the exception instruction to revoke the permission, as we show in Fig. 8(B1). A trap is an instruction that causes an exception during execution, such as instructions ARM hardware breakpoints [37], [64] observe, software breakpoint instructions (*e.g.*, `BRK`), or illegal instructions. After setting the trap, NANOTAG resumes the program execution from the exception instruction. NANOTAG's escalation has limitations (*e.g.*, exception instruction at the end of a function) that we describe in §6.

Revocation. Once the program's execution resumes, the exception instruction accesses the memory without causing a tag mismatch fault. The program then executes the next instruction that contains a trap. This trap causes an exception, which NANOTAG intercepts to revoke the permission for the delegated short granule. In particular, NANOTAG sets the memory tag of the short granule back to the original tripwire tag, by swapping the memory tag and the last 4 bits of the corresponding short granule again, as we show in Fig. 8(C). After *revocation*, accessing the short granule will again trigger a tag mismatch fault (G3). NANOTAG also removes the trap during *revocation*, resuming the program from the current instruction.

Tripwire Access Control. As NANOTAG recovers after a tag mismatch fault, accessing the same tripwire repeatedly may introduce significant performance slowdown. NANOTAG avoids the potential overhead by adopting an access control mechanism as we show in Fig. 8. In particular, NANOTAG maintains a counter, *AccessCount*, in the padded bytes of the short granule, varying from 1 to 15 bytes. In case of a single-byte padding, NANOTAG leverages the most

significant 4 bits of the padding for *AccessCount*, as the padding's least significant 4 bits already contain the valid address tag. In case of a multi-byte padding, NANOTAG could utilize at least 12 bits for *AccessCount*. In both cases, NANOTAG increments a short granule's *AccessCount* every time the corresponding tripwire causes a tag mismatch fault. When a short granule's *AccessCount* reaches the largest number the counter could hold (*e.g.*, 15 for a 4-bit counter), NANOTAG removes the short granule's tripwire. NANOTAG supports configuring another parameter, *AccessThreshold*. As a short granule's *AccessCount* reaches *AccessThreshold*, NANOTAG also removes the tripwire, as we show in Fig. 8(B2). By default, NANOTAG sets *AccessThreshold* to 64, which we empirically find to provide high bug detection capability (97.57%, §5.5).

3.5. NANOTAG Bug Report

Once NANOTAG detects a memory safety bug, it provides a detailed report to help developers pinpoint the bug's root cause. NANOTAG reports program's `pc`, fault address, values of registers, address tag, and memory tag. If the memory safety bug is an intra-granule buffer overflow, NANOTAG also reports the number of addressable bytes in the short granule, while also listing the number of bytes the exception instruction accesses in the short granule to cause the bug. To reliably reproduce bugs, NANOTAG supports increasing its bug detection capability by adjusting its configurable parameters, *e.g.*, by setting *AllocThreshold* as `INT_MAX`, assuming no tag collision for MTE.

4. Implementation

We implement NANOTAG using a standalone user-mode signal handler [65] and Scudo Hardened Allocator [23]. In particular, NANOTAG's signal handler consists of 893 lines of code. We implement NANOTAG's configurable parameters using environment variables [66]. We open-source NANOTAG at https://github.com/ice-rlab/NanoTag.

Sampling-Based Tripwire Allocation. We implement NANOTAG's sampling-based tripwire allocation by modifying 227 lines of code in Scudo. We implement NANOTAG's sampling-based tripwire allocation by modifying Scudo's primary allocator. We compile both Scudo and NANOTAG with the `O2` optimization level.

Overflow Detection. MTE raises a tag mismatch fault as a segmentation fault, *i.e.*, a `SIGSEGV` signal with codes 8 and 9 for MTE ASYNC and SYNC mode, respectively. Consequently, we implement NANOTAG's tag mismatch handler as a user-space signal handler using `sigaction` with `SA_SIGINFO` and `SA_RESTART` flags. Our prototype supports a variety of load and store instructions, including regular, atomic, and SIMD instructions. We find that some functions in `glibc` [67] introduce out-of-bound read accesses in their assembly implementations. For example, the `strcpy`'s ARM assembly implementation [68] uses SIMD load instructions to iteratively read from the source string and then checks if it has reached the null terminator. While there exist prior ARM efforts [69] to make `strcpy` compatible with MTE, it will still require additional engineering efforts to make `glibc` compatible with byte-granular overflow detection. Consequently, to avoid rewriting `glibc`, NANOTAG's tag mismatch handler skips the tag mismatch faults these instructions cause.

Tag Mismatch Recovery. We implement NANOTAG's tag mismatch recovery mechanism by setting up ARM `BRK` instructions, as we could not set hardware breakpoints on Google Pixel 8. When the processor executes this `BRK` instruction, it generates a `SIGTRAP` signal, which NANOTAG catches with another signal handler. To differentiate from other `SIGTRAP` signals, NANOTAG uses a special immediate value while setting the `BRK` instruction.

Bug Report. NANOTAG's signal handler parses the `ucontext` argument [70], extracting information, *e.g.*, `pc`, fault address, and values of registers, to generate the bug report, similar to what we describe in §3.3.

5. Evaluation

In this section, we experimentally evaluate NANOTAG to answer the following research questions:

RQ1. How effectively does NANOTAG detect memory safety bugs at byte granularity?

RQ2. How efficiently does NANOTAG detect memory safety bugs at byte granularity in terms of run-time overhead?

RQ3. How does NANOTAG generalize to real-world applications in terms of run-time efficiency and fuzzing throughput?

RQ4. How do NANOTAG's parameters affect its bug detection capability and run-time overhead?

5.1. Experimental Methodology

Similar to prior work [29], [71], we perform all experiments on a rooted Google Pixel 8 Pro with Termux [72]. We first describe our hardware and software setup. Then, we provide a brief overview of our benchmarks and performance metrics.

Hardware. The Google Pixel 8 Pro is one of the first handsets with MTE support [24]. This device has four "big" Cortex-A715 cores, four "little" Cortex-A510 cores, and one "prime" Cortex-X3 processor core, with a frequency range of 402-2367 MHz, 324-1704 MHz, and 500-2914 MHz, respectively. The device has 12 GB LPDDR5 RAM.

Software. The Google Pixel 8 Pro uses the Android 15. On this system, we first install Termux [72], an Android terminal emulator that provides a Linux development environment on Android OS. As Termux launches programs directly by calling the `execve()` system call, it incurs minimal overhead compared to a virtualized environment [73]. We also leverage `chroot` [74], a Linux command, to change the system's root directory to a specific path, the Ubuntu 22.04 root file system (`rootfs`), to emulate a Linux development environment. In particular, we use the Linux kernel version of 5.15.148. As compiler, we use LLVM's front-end compiler clang 14 [75].

TABLE 3. MEMORY SAFETY BUGS DETECTED IN JULIET TEST SUITE. NANOTAG DETECTS A SIMILAR NUMBER (97.57%) OF HEAP-BASED BUFFER OVERFLOW BUGS AS ASAN (98.66%).

CWE ID	Total	ASAN	GWP-ASan	Scudo (ASYNC)*	Scudo (SYNC)*	NANOTAG
CWE122 (heap-based buffer overflow)	3438	98.66%	23.15% ± 0.24%	75.60% ± 0.13%	75.68% ± 0.10%	97.57% ± 0.15%
CWE415 (double free)	818	100%	98.60% ± 0.23%	98.43% ± 0.14%	98.45% ± 0.19%	98.37% ± 0.12%
CWE416 (use after free)	393	100%	0%	96.54 ± 0.60%	96.94% ± 0.58%	96.69% ± 0.44%

*The variance comes from MTE's tag collision across multiple runs.

Benchmarks. We evaluate NANOTAG's bug detection capability (§5.2) and performance efficiency (§5.3) using Juliet Test Suite [27] and SPEC CPU 2017 benchmarks [31], respectively. We evaluate NANOTAG's generality to real-world applications using Geekbench 6 [38], `Memcached` [39], `LevelDB` [40], `RocksDB` [41], and Magma [76] (§5.4).
Metrics. For performance efficiency, we report overhead numbers as the percentage increase in benchmark execution time with a sanitized configuration (*e.g.*, ASAN, NANOTAG, etc.) in comparison to the baseline configuration (*i.e.*, MTE-disabled Scudo). We run each benchmark multiple times to report both arithmetic and geometric means.

5.2. Bug Detection Capability

We study NANOTAG's memory safety bug detection capability using Juliet Test Suite [27], comparing the results with both Scudo in MTE SYNC and ASYNC modes, GWP-ASan, and ASAN.
Settings. While using Juliet Test Suite to evaluate NANOTAG, we use the same experimental setting as our analysis (§2). For example, we evaluate NANOTAG for both benign and buggy versions of each CWEs. We run Juliet Test Suite with Scudo in MTE SYNC and ASYNC modes, GWP-ASan, ASAN, and NANOTAG. We compile all test cases with `O0` so that compiler can not optimize them to avoid memory safety bugs. Due to MTE's probabilistic nature, for Scudo and NANOTAG, we run Juliet Test Suite for ten iterations and report the average number. We run Juliet Test Suite directly on Android 15 using Termux.
Results. As we discussed in §2.2, ASAN, GWP-ASan, and Scudo do not label any of the benign test cases in the Juliet Test Suite as memory safety bugs. Similarly, NANOTAG flags none of the benign test cases as a memory safety bug in any iteration of the experiment. Therefore, we only show the results for the buggy test cases of Juliet Test Suite in Table 3. In particular, Table 3 shows the percentage of bugs ASAN, GWP-ASan, Scudo, and NANOTAG detect for different CWEs in Juliet Test Suite. As we also show in §2.2, ASAN detects 22.98% of more heap-based buffer overflow bugs (`CWE122`) than Scudo in MTE SYNC or ASYNC mode, primarily due to intra-granule buffer overflows. In contrast, NANOTAG identifies these intra-granule buffer overflows successfully, detecting 97.57% of all heap-based buffer overflow bugs, similar to 98.66% of bugs ASAN detects. The 1.09% gap between NANOTAG and ASAN in `CWE122` is primarily due to MTE's probabilistic nature. As we mention in §1 and §2, MTE uses only 4-bit tags due to constraints on tag storage and TBI. Consequently, two random tags may collide with each other with a probability of 6.25% ($\frac{1}{16}$ as $2^4 = 16$). Prior work [77] aims to avoid such tag collisions by utilizing a multi-granular memory tagging in hardware. For double free (`CWE415`) and use-after-free (`CWE416`), NANOTAG performs similarly to Scudo, detecting 98.37% and 96.69% double free and use-after-free bugs, respectively. Similar to heap-based buffer overflow bugs, MTE's probabilistic nature prevents NANOTAG from detecting the rest of the temporal bugs.

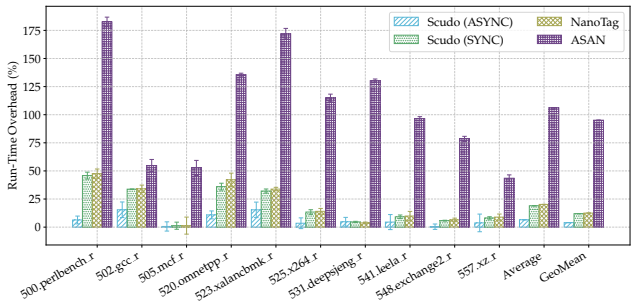

Figure 9. The run-time overhead of MTE-enabled Scudo, NANOTAG, and ASAN for SPEC CPU 2017 benchmarks compared to a Scudo baseline with MTE disabled. In terms of geometric mean, NANOTAG incurs an overhead of 12.50% similar to Scudo's 11.98% overhead in MTE SYNC mode.

Takeaway: Detecting intra-granule buffer overflows, NANOTAG performs similar to ASAN in Juliet Test Suite.

5.3. Run-Time Overhead

We study NANOTAG's run-time overhead using SPEC CPU 2017 benchmarks [31]. Specifically, we use SPECrate Integer benchmarks to measure the performance overhead NANOTAG, MTE-enabled Scudo, and ASAN incur over a Scudo baseline with MTE disabled. We show the results in Fig. 9. Reusing padding bytes to store metadata, NANOTAG avoids incurring any memory overhead.
Settings. Using SPEC CPU 2017 benchmarks version (1.1.9), we evaluate NANOTAG for all 10 SPECrate Integer benchmarks, including the ones written in Fortran. We only use SPECrate Integer benchmarks similar to prior

TABLE 4. RUN-TIME OVERHEAD OF SCUDO WITH MTE ASYNC AND SYNC MODES, NANOTAG, AND VALGRIND [18] ON GEEKBENCH 6 [38] COMPARED TO THE SCUDO BASELINE WITH MTE DISABLED. ON THE CLOSED-SOURCE GEEKBENCH 6 BENCHMARKS, NANOTAG ADDS ONLY 1.23% OVERHEAD OVER THE MTE SYNC MODE.

	Scudo (ASYNC)	Scudo (SYNC)	NANOTAG	Valgrind Memcheck
Geekbench 6	1.96%	3.76%	4.99%	1348.60%

TABLE 5. LARGE REAL-WORLD APPLICATIONS, THEIR VERSIONS, BENCHMARKS, AND WORKLOADS WE STUDY.

Applications	Versions	Benchmarks	Workloads
`Memcached`	1.6.14	`mc-benchmark` [85]	`GET`, `SET`
`LevelDB`	1.22	`db_bench` [86]	`fillseq`, `readseq`, `fillrandom`, `readrandom`, and `readreverse`
`RocksDB`	10.10.1	`db_bench` [87]	

TABLE 6. RUN-TIME OVERHEAD OF NANOTAG AND ASAN ON THREE LARGE REAL-WORLD APPLICATIONS.

Target	NANOTAG Slowdown		ASAN Slowdown
	SamplingRate=10	*SamplingRate*=100	
`Memcached`	6.07%	1.30%	22.53%
`LevelDB`	16.93%	12.56%	95.26%
`RocksDB`	18.35%	17.75%	1698.31%
GeoMean	12.35%	6.13%	153.90%

work [29] as Google Pixel 8's 12 GB physical memory is not sufficient to run SPECspeed Integer benchmarks that require 16 GB of physical memory. We cross-compile all benchmarks on an ARM server with Cavium ThunderX2 CPUs [78] provided by Chameleon Cloud [79], and run them on Google Pixel 8 using `chroot` Ubuntu 22.04. While compiling all benchmarks with `clang`, we use the `O3` optimization level. We use this same set of binaries to evaluate Scudo and NANOTAG via `LD_PRELOAD`, following the SPEC tutorial [80]. As ASAN requires additional compilation flags, we evaluate ASAN using a separate set of binaries compiled with the required flags (`-fsanitize=address -fsanitize-recover=address`).

Results. Fig. 9 shows the run-time overhead of Scudo, NANOTAG, and ASAN for SPEC CPU 2017 benchmarks. In terms of geometric mean, NANOTAG incurs a performance overhead of 12.50%, significantly outperforming ASAN (95.11%). Even in the worst case, NANOTAG slows down `500.perlbench_r` by 47.60%, which is only around $\frac{1}{4}$ of ASAN's 182.59% overhead. Compared to the 11.98% overhead of Scudo in MTE SYNC mode, NANOTAG incurs only an additional 0.52% overhead. Its largest gap to Scudo in MTE SYNC mode occurs in `520.omnetpp_r`, which is 6.13%. For Scudo, we also observe a noticeable gap between MTE SYNC and ASYNC modes. In terms of geometric mean, Scudo only incurs a run-time overhead of 4.00% in MTE ASYNC mode, which is 7.98% lower than the overhead of Scudo in MTE SYNC mode. The largest gap between MTE SYNC and ASYNC modes also occurs for `500.perlbench_r`, which is 39.49%. Reducing MTE SYNC mode's overhead will also make NANOTAG more efficient, which we plan as future work.

Takeaway: In terms of geometric mean, NANOTAG incurs a run-time overhead of 12.50% for SPEC CPU 2017 benchmarks.

5.4. Real-World Case Studies

5.4.1. Closed-Source Application. We study NANOTAG's run-time overhead on real-world closed-source applications using Geekbench 6 [38], one of the most popular benchmarking apps with 1 million downloads on Google Play Store.

Settings. We use the Geekbench 6 Linux/AArch64 Preview version [81], which runs the same set of benchmarks as the Android or iOS version of Geekbench 6 [82], [83]. The closed-source binary consists of 16 benchmarks that always run sequentially, one benchmark after another. As the binary does not provide an option to run a single benchmark separately, while evaluating NANOTAG, we set *AllocThreshold* to 100,000, much higher than its default value (1000), so that we can measure NANOTAG's overhead for all benchmarks. Similar to SPEC CPU 2017 benchmarks, we run Geekbench 6 on chroot Ubuntu 22.04 with a single core and measure the absolute run time. For baseline, we use Scudo with MTE disabled and compare its performance against NANOTAG and Scudo with MTE ASYNC and SYNC modes. ASAN-Retrowrite [17], the state-of-the-art static binary instrumentation tool to sanitize closed-source binaries, does not support the C++ exception handling mechanism Geekbench 6 requires. Consequently, for Geekbench 6, we compare NANOTAG's performance against Valgrind [18]'s Memcheck tool [84].

Results. As we show in Table 4, NANOTAG incurs a run-time overhead of only 4.99% on Geekbench 6, which is close to the overheads Scudo ASYNC (1.96%) and SYNC (3.76%) modes incur, and much lower than the overhead of Valgrind (1348.60%). For all configurations, Table 4 shows the overheads of the first 14 out of 16 benchmarks of Geekbench 6. While running the 15th benchmark (`ray-tracer`) with NANOTAG, we observe an unexpected segmentation fault. The root cause of the segmentation fault is a software breakpoint (`BRK` instruction) NANOTAG sets up during *escalation* to restore a tripwire. For some specific instruction addresses of `ray-tracer`, software breakpoints cause a segmentation fault instead of causing an exception. While testing the benchmark with `GDB` manually, we also could not set up breakpoints for those addresses. In the future, we will explore setting up ARM hardware breakpoints [37], [64], currently not supported on Google Pixel 8, to solve this problem.

5.4.2. Large Real-World Applications. We evaluate NANOTAG's run-time overhead on three large real-world applications: `Memcached` [39], `LevelDB` [40], and `RocksDB` [41].

Settings. We set up real-world applications similar to prior work [88]. For these applications, we summarize their ver-

TABLE 7. FUZZING THROUGHPUT SLOWDOWN FOR NANOTAG AND ASAN ON MAGMA BENCHMARKS COMPARED TO THE SCUDO BASELINE. NANOTAG SLOWS DOWN FUZZING THROUGHPUT BY 15.86%, $\frac{1}{7}$ OF ASAN'S SLOWDOWN.

Target	`.text` Size (Baseline)	Baseline (exec/s)	NANOTAG Slowdown	ASAN Slowdown
libpng	277.3 KB	28516.9	17.83%	35.70%
libxml2	1.1 MB	9243.23	7.26%	113.49%
poppler	4.4 MB	356.57	30.85%	339.32%
GeoMean			15.86%	111.20%

sions, benchmarks, and workloads in Table 5. As the current prototype of NANOTAG's slow start phase supports only a limited number of system calls, we disable this phase for these system-call-heavy applications [88]. As we only evaluate NANOTAG's sampling phase for these applications, we set the sampling rate (*SamplingRate*) to 10 and 100 instead of 1000 to compensate for the lack of the slow start phase. For both sampling rates, we compare NANOTAG against ASAN in terms of run-time overhead over the baseline Scudo with MTE disabled.

Results. As we show in Table 6, NANOTAG incurs a geometric mean run-time overhead of up to 12.35% on these three large real-world applications even with a *SamplingRate* of 10. For a *SamplingRate* of 100, NANOTAG slows down these applications by only 6.13%. In contrast, ASAN incurs a run-time overhead of 153.9% for these applications.

5.4.3. Fuzzing Throughput. We evaluate NANOTAG's generality on fuzzing using the Magma fuzzing benchmark [76].

Settings. Using the Magma version 1.2.1 [89], we fuzz the `libpng`, `libxml2`, and `poppler` targets with AFL++ [90] in the persistent mode [91] on Google Pixel 8, in the absence of accessible MTE-enabled servers [92], [93]. Following the prior work [94], we run fuzzers for 5 minutes to report the average throughput of 5 fuzzing campaigns. We measure slowdowns NANOTAG and ASAN incur over the Scudo baseline.

Results. Table 7 shows the fuzzing throughput slowdowns of NANOTAG and ASAN. `libpng`, `libxml2`, and `poppler` represent diverse fuzzing targets, from small to larger sizes, as we show the `.text` section size and baseline fuzzing throughput in Table 7. In terms of geometric mean, NANOTAG slows down the fuzzing throughput by only 15.86%, which is around $\frac{1}{7}$ of ASAN's slowdown (111.20%). NANOTAG's highest slowdown (30.85%) occurs in `poppler`, which is $\frac{1}{10}$ of ASAN's slowdown.

> **Takeaway:** NANOTAG incurs a negligible run-time overhead of only 4.99% on Geekbench 6, a real-world closed-source application, incurs up to 12.35% run-time overhead on three large real-world applications: `Memcached`, `LevelDB`, and `RocksDB`, and slows down fuzzing throughput by only 15.86% on Magma.

5.5. Sensitivity Study

As we describe in §3, NANOTAG includes three configurable parameters: *AccessThreshold*, *AllocThreshold*, and *SamplingRate*. We now study how these parameters affect NANOTAG's performance overhead and bug detection capabilities. As we vary the values of these parameters, we investigate NANOTAG's performance overhead using the `500.perlbench_r` benchmark, which suffers the highest run-time overhead (47.6%) among SPEC CPU 2017 benchmarks, while studying NANOTAG's bug detection capabilities using `CWE122` (heap-based buffer overflow) in Juliet Test Suite.

Settings. The default values of *AccessThreshold*, *AllocThreshold*, and *SamplingRate* are 64, 1000, and 1000, respectively. While studying the sensitivity of one parameter, we use the default values for the remaining parameters.

Results. We show the results in Table 8. In Table 8 (a), as we increase *AccessThreshold* from 32 to 256, NANOTAG's run-time overhead increases from 47% to 59%, without affecting its bug detection capability that remains around 97%. Similarly, as we increase *AllocThreshold* from 100 to 10,000 in Table 8 (b), and decrease *SamplingRate* from 1000 to 10 in Table 8 (c), NANOTAG's run-time overhead increases from 47% to 58%, and from 47% to 100%, respectively, while its bug detection is unaffected. When we increase *AccessThreshold* beyond 256, increase *AllocThreshold* beyond 10,000 (*e.g.*, 100,000), or decrease the *SamplingRate* below 10 (*e.g.*, 1), NANOTAG behaves similarly to a scenario without any tripwire access control or sampling, which we discuss in §6. NANOTAG's bug detection capability drops only when *AccessThreshold* is less than 32 (*e.g.*, 93% for an *AccessThreshold* of 4). As we discuss Juliet Test Suite's limitations in §6, NANOTAG's bug detection capability remains ∼97% even with a larger *SamplingRate*.

In general, while a lower *SamplingRate* has a significant impact on NANOTAG's run-time overhead, its performance is stable, *i.e.*, between 40% and 60% as we vary the values of *AccessThreshold*, *AllocThreshold*, and *SamplingRate*.

> **Takeaway:** NANOTAG achieves high bug detection capability with low run-time overhead by using reasonable values for its parameters (*e.g.*, *SamplingRate* $\geq$100, *AccessThreshold* $\leq$256).

6. Limitations

Tripwire Access Control and Sampling. As we describe in §3, NANOTAG relies on tripwire access control and sampling to amortize the overhead of additional software checks. Without tripwire access control and sampling, some applications would suffer from significant run-time overhead, *e.g.*, up to 5× and 15× on `perlbench_r`, respectively.

Adjusting NANOTAG's Parameters for Fuzzing. NANOTAG supports configuring its parameters dynamically to balance between detection strength and performance overhead (§3). Automatic adjustment of this detection-performance tradeoff across many executions of fuzzing would be valuable future work.

False Negatives. NANOTAG may suffer from false negatives due to the end of a function and tag collisions. In case

TABLE 8. BUG DETECTION CAPABILITY (%) AND RUN-TIME OVERHEAD (%) FOR NANOTAG'S DIFFERENT CONFIGURATIONS.

(a) *AccessThreshold*			(b) *AllocThreshold*			(c) *SamplingRate*		
Value	**Bug Detection Capability (%)**	**Run-Time Overhead (%)**	**Value**	**Bug Detection Capability (%)**	**Run-Time Overhead (%)**	**Value**	**Bug Detection Capability (%)**	**Run-Time Overhead (%)**
32	97.38	47.08	100	97.49	47.36	10	97.52	100.95
64	97.49	47.60	1000	97.41	47.60	100	97.20	61.23
128	97.50	59.00	10000	97.45	58.02	1000	97.44	47.60
256	97.32	59.32						

of a tag mismatch fault at the end of a function, *i.e.*, the exception instruction is followed by a `RET` instruction, NANOTAG could not set the trap to the `RET` instruction. Consequently, as NANOTAG does not set the tripwire back to the short granule, it may lead to some potential false negatives. Also, NANOTAG may increase the possibility of MTE's tag collision. For example, if a pointer's address tag matches with the last 4 bits of a tag granule, NANOTAG still permits the memory access even if the access was a true memory safety bug. Treating such tag granule as a short granule, NANOTAG still limits the number of addressable bytes to this tag granule.

Juliet Test Suite. Most of the benchmarks in the Juliet Test Suite are micro-benchmarks with only one or two memory allocations. Due to such a few memory allocations, NANOTAG does not use sampling from the start of the program. For example, if NANOTAG enables sampling from the beginning, omitting tripwires for one out of two short granules, NANOTAG may miss the short granule causing the memory safety bug. Consequently, while sampling from the beginning, NANOTAG fails to detect 10% of heap-based buffer overflow bugs in Juliet Test Suite that allocate only a few short granules. Updating Juliet Test Suite to represent realistic memory allocations [43], [95] would be a valuable future work.

Reducing NANOTAG's Overhead Even Further. NANOTAG incurs 12.50% overhead primarily due to MTE 's $\sim$ 12% overhead. Reducing MTE 's overhead even further would require micro-architectural and kernel-level modifications [93], [96].

Side-Channel Attacks. Recent works [29], [61] demonstrate that the production implementation of ARM MTE (*e.g.*, Google Pixel 8) is vulnerable to speculative execution attacks. Consequently, preventing speculative execution attacks on MTE and MTE-enabled systems [29], [48], [71], [97] using persistent tags [29] or memory integrity enforcement [98] would be valuable future work.

7. Related Work

We classify existing related work between software (§7.1) and hardware (§7.2) techniques and compare them against NANOTAG in Table 9.

7.1. Software-Based Techniques

In-house testing techniques. Software-based memory safety sanitizers are widely-adopted to analyze memory safety bugs during in-house testing. Such memory safety sanitizers could be location-based [8], [9], [11], [12], [15], [17]–[19], [99] or pointer-based [13], [14], [52], [100]–[103]. Among them, ASAN [8], one of the most popular location-based sanitizers, uses redzones and a quarantine list to detect buffer overflows and use-after-frees, respectively. Several techniques [9], [11], [12], [15], [42] improve upon ASAN [16] by avoiding redundant queries [9], [12], segment folding [11], or hardware-assisted tagging [15], [42]. Overall, these software-based techniques are easy to deploy in commodity hardware. They also detect memory safety bugs at byte granularity. Unfortunately, these software-based techniques typically incur high overhead, slowing down in-house testing, such as fuzzing campaigns. As fuzzing is computationally intensive, developers avoid these software-based techniques for many campaigns [20]. Instead, these campaigns rely on crashes to identify malicious inputs [104], ignoring salient bugs [21]. NANOTAG addresses this key limitation by enabling low-overhead byte-granular overflow detection.

TABLE 9. COMPARISON OF NANOTAG AGAINST PRIOR WORK.

Proposal	**Granularity**	**Commodity Deployable**	**Overhead**
Software-Based In-House Techniques	Byte-Level	Easy	Usually High
Software-Based In-Production Techniques	Page-Level	Easy	Low
Hardware-Based In-Production Techniques	Varies	Relatively Difficult	Usually Low
NANOTAG	**Byte-Level**	**Easy**	**Low**

In-production deployment techniques. Prior work [54], [99] also aim to deploy memory safety sanitizers in production via sampling, notably GWP-ASan [54], which Google uses for its various products. Although NANOTAG uses a similar sampling algorithm to GWP-ASan, their design principles are very different. While GWP-ASan targets in-production use cases, NANOTAG's primary use case is in-house testing. Therefore, while GWP-ASan detects overflows for sampled memory allocations at page granularity, NANOTAG enables byte-granular overflow detection for sampled memory allocations. For unsampled memory allocations, NANOTAG still detects overflows at 16-byte granularity with MTE, while GWP-ASan disables any detection. In general, these sampling-based memory sanitizers target

in-production deployment, thus trading their bug detection capability, including granularity, for better performance.

Finally, recent works leverage ARM MTE for sanitization [29], [48], [105]–[108] and isolation [71], [97], [109]–[113]. NANOTAG improves the effectiveness of these techniques by addressing MTE's coarse precision due to the 16-byte tag granularity.

7.2. Hardware-Based Techniques

Prior work [42], [114]–[127] proposes many hardware designs to detect memory safety bugs for in-production deployment, including capability-based architectures [114]–[117], tagged architectures [42], [118], [118], [120]–[122], hardware bounds checking [119], [124], [125], cache-line metadata [123], [127], and ECC-based techniques [126]. While some of them [114], [115], [119], [123]–[125] can detect memory safety bugs at byte granularity, they remain academic prototypes or have been deprecated [125]. For the few [42], [122], [128] that have commodity implementations, they only have page [128], cache-line [122] or 16-byte [42] granularity. Altogether, while these hardware-based techniques offer low run-time overhead, they either suffer from imprecise granularity or are challenging to deploy with commodity hardware. In contrast, NANOTAG enables byte-granular detection of memory safety bugs in unmodified binaries directly on a commodity implementation of ARM MTE.

Recent hardware proposals [43], [77], [105], [129] improve MTE's tag collision rate [77], tag storage overhead [43], performance [129], and granularity [105]. Specifically, DMTI [105] improves MTE's tag granularity by assigning every memory byte a 4-bit MTE tag in hardware, introducing large tag storage overhead. On the other hand, NANOTAG avoids such tag storage overhead by storing the metadata inside unused padding bytes in the short granule, while still detecting memory safety bugs at byte granularity.

8. Conclusion

Memory safety bugs remain one of the leading causes of software vulnerabilities. With hardware support, MTE accelerates detecting such bugs, but fails to detect intra-granule buffer overflows due to its 16-byte tag granularity. We propose NANOTAG to detect memory safety bugs probabilistically in unmodified MTE-enabled binaries at byte granularity, addressing intra-granule buffer overflows in real hardware for the first time.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This work was supported in part by the Center for Ubiquitous Connectivity (CUbiC), sponsored by Semiconductor Research Corporation (SRC) and Defense Advanced Research Projects Agency (DARPA) under the JUMP 2.0 program. This work was partially supported by the Google Cyber NYC Institutional program. We thank the National Science Foundation's Chameleon Cloud [79] for providing compute nodes on which we run experiments to obtain some results presented in this paper. We also thank Kostya Serebryany from Tesla, Evgenii Stepanov from Google, and Simha Sethumadhavan from Columbia University for helpful discussions.

Ethics Considerations

None.

References

[1] M. Miller. (2019) Trends, challenge, and shifts in software vulnerability mitigation. [Online]. Available: https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[2] "Memory safety — android open source project," https://source.android.com/docs/security/test/memory-safety, [Online; accessed 1-April-2025].

[3] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with KINT," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, C. Thekkath and A. Vahdat, Eds. USENIX Association, 2012, pp. 163–177. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wang

[4] G. Li, H. Zhang, J. Zhou, W. Shen, Y. Sui, and Z. Qian, "A hybrid alias analysis and its application to global variable protection in the linux kernel," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 4211–4228. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/li-guoren

[5] H. Zhang, J. Kim, C. Yuan, Z. Qian, and T. Kim, "Statically discover cross-entry use-after-free vulnerabilities in the linux kernel," in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/statically-discover-cross-entry-use-after-free-vulnerabilities-in-the-linux-kernel/

[6] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds. ACM, 2021, pp. 84–99. [Online]. Available: https://doi.org/10.1145/3477132.3483570

[7] facebook/infer: A static analyzer for java, c, c++, and objective-c. [Online]. Available: https://github.com/facebook/infer

[8] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[9] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 4345–4363. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen

[10] G. J. Duck and R. H. C. Yap, "Effectivesan: type and memory error detection using dynamically typed C/C++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds. ACM, 2018, pp. 181–195. [Online]. Available: https://doi.org/10.1145/3192366.3192388

[11] H. Ling, H. Huang, C. Wang, Y. Cai, and C. Zhang, "GIANTSAN: efficient memory sanitization with segment folding," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds. ACM, 2024, pp. 433–449. [Online]. Available: https://doi.org/10.1145/3620665.3640391

[12] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "SANRAZOR: reducing redundant sanitizer checks in C/C++ programs," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 479–494. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/zhang

[13] 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 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/yu-zheng

[14] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: highly compatible and complete spatial memory safety for c," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, M. Hind and A. Diwan, Eds. ACM, 2009, pp. 245–258. [Online]. Available: https://doi.org/10.1145/1542476.1542504

[15] "Hardware-assisted addresssanitizer — android open source project," https://source.android.com/docs/security/test/hwasan, [Online; accessed 1-April-2025].

[16] "Addresssanitizer — android open source project," https://source.android.com/docs/security/test/asan, [Online; accessed 1-April-2025].

[17] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1497–1511. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00009

[18] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," pp. 89–100, 2007. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[19] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with qasan," in *IEEE Secure Development, SecDev 2020, Atlanta, GA, USA, September 28-30, 2020*. IEEE, 2020, pp. 23–30. [Online]. Available: https://doi.org/10.1109/SecDev45635.2020.00019

[20] Z. Kong, S. Li, H. Huang, and Z. Su, "Sand: Decoupling sanitization from fuzzing for low overhead," in *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2025, pp. 255–267. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00187

[21] Y. Jeon, W. Han, N. Burow, and M. Payer, "FuZZan: Efficient sanitizer metadata design for fuzzing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 249–263. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/jeon

[22] "Enhanced security through memory tagging extension," https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhanced-security-through-mte, [Online; accessed 1-April-2025].

[23] "Scudo — android open source project," https://source.android.com/docs/security/test/scudo, [Online; accessed 1-April-2025].

[24] "Project zero: First handset with mte on the market," https://googleprojectzero.blogspot.com/2023/11/first-handset-with-mte-on-market.html, [Online; accessed 1-April-2025].

[25] Memory corruption bug uncovered by mte fixed by google after grapheneos report. [Online]. Available: https://discuss.grapheneos.org/d/12628-memory-corruption-bug-uncovered-by-mte-fixed-by-google-after-grapheneos-report

[26] Synchronous mode (sync). [Online]. Available: https://source.android.com/docs/security/test/memory-safety/arm-mte#sync-mode

[27] "Juliet c/c++ 1.3 - nist software assurance reference dataset," https://samate.nist.gov/SARD/test-suites/112, [Online; accessed 1-April-2025].

[28] "3.2.2 the gnu allocator," https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html, [Online; accessed 1-April-2025].

[29] F. Gorter, T. Kroes, H. Bos, and C. Giuffrida, "Sticky tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags," in *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, pp. 4239–4257. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00263

[30] T. A. Khan, Y. Zhao, G. Pokam, B. Mozafari, and B. Kasikci, "Huron: hybrid false sharing detection and repair," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 453–468.

[31] Spec cpu® 2017 benchmark. [Online]. Available: https://www.spec.org/cpu2017/

[32] "Nvd - cve-2024-12084," https://nvd.nist.gov/vuln/detail/CVE-2024-12084, [Online; accessed 1-April-2025].

[33] Rsync: Heap buffer overflow, info leak, server leaks, path traversal and safe links bypass. [Online]. Available: https://github.com/google/security-research/security/advisories/GHSA-p5pg-x43v-mvqj

[34] "Cve-2024-12084 - red hat customer portal," https://access.redhat.com/security/cve/CVE-2024-12084, [Online; accessed 1-April-2025].

[35] R. Hastings and B. Joyce, "Purify: A tool for detecting memory leaks and access errors in c and c++ programs," in *Proceedings of the Winter 1992 USENIX Conference*, pp. 125–138.

[36] B. Perens. Electric fence. [Online]. Available: https://elinux.org/Electric_Fence

[37] Overview: Breakpoints and watchpoints. [Online]. Available: https://developer.arm.com/documentation/dui0446/z/controlling-target-execution/overview--breakpoints-and-watchpoints

[38] "Geekbench 6 - cross-platform benchmark," https://www.geekbench.com/, [Online; accessed 1-April-2025].

[39] memcached - a distributed memory object caching system. [Online]. Available: https://memcached.org/

[40] google/leveldb. [Online]. Available: https://github.com/google/leveldb

[41] facebook/rocksdb. [Online]. Available: https://github.com/facebook/rocksdb

[42] "Armv8.5-a memory tagging extension white paper," https://developer.arm.com/documentation/102925/latest/, [Online; accessed 1-April-2025].

[43] A. Partap and D. Boneh, "Memory tagging: A memory efficient design," *CoRR*, vol. abs/2209.00307, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2209.00307

[44] "Tagged pointers — android open source project," https://source.android.com/docs/security/test/tagged-pointers, [Online; accessed 1-April-2025].

[45] "Where is the mte tag stored and checked?" https://developer.arm.com/documentation/ka005620/1-0/?lang=en, [Online; accessed 1-April-2025].

[46] "Mte modes," https://developer.arm.com/documentation/108035/0100/How-does-MTE-work-/MTE-modes, [Online; accessed 1-April-2025].

[47] "Arm a-profile a64 instruction set architecture — loads and stores," https://developer.arm.com/documentation/ddi0602/2024-09/Index-by-Encoding/Loads-and-Stores?lang=en, [Online; accessed 1-April-2025].

[48] 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 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 841–858. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/chen-xingman

[49] P. Collingbourne, "scudo: Use dc gzva instruction in storetags()," https://github.com/llvm/llvm-project/commit/46c59d91dc7a39cc98be7a68d6dc60f3e8a35df0, [Online; accessed 1-April-2025].

[50] "Memory tagging extension (mte) in aarch64 linux," http://docs.kernel.org/arch/arm64/memory-tagging-extension.html#memory-tagging-extension-mte-in-aarch64-linux, [Online; accessed 1-April-2025].

[51] "Mte bootloader support," https://source.android.com/docs/security/test/memory-safety/bootloader-support, [Online; accessed 1-April-2025].

[52] 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 *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 1901–1915. [Online]. Available: https://doi.org/10.1145/3548606.3560598

[53] "Cwe - 2024 cwe top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html, [Online; accessed 1-April-2025].

[54] K. Serebryany, C. Kennelly, M. Phillips, M. Denton, M. Elver, A. Potapenko, M. Morehouse, V. Tsyrklevich, C. Holler, J. Lettner, D. Kilzer, and L. Brandt, "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, ICSE-SEIP 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 168–177. [Online]. Available: https://doi.org/10.1145/3639477.3640328

[55] "Leaksanitizer - clang 21.0.0git documentation," https://clang.llvm.org/docs/LeakSanitizer.html, [Online; accessed 1-April-2025].

[56] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," *CoRR*, vol. abs/1802.09517, 2018. [Online]. Available: http://arxiv.org/abs/1802.09517

[57] "rsync/lib/md-defines.h," https://github.com/RsyncProject/rsync/blob/9615a2492bbf96bc145e738ebff55bbb91e0bbee/lib/md-defines.h#L11-L21, [Online; accessed 1-April-2025].

[58] "rsync/rsync.c," https://github.com/RsyncProject/rsync/blob/9615a2492bbf96bc145e738ebff55bbb91e0bbee/rsync.h#L955-L962, [Online; accessed 1-April-2025].

[59] "rsync/sender.c," https://github.com/RsyncProject/rsync/blob/9615a2492bbf96bc145e738ebff55bbb91e0bbee/sender.c#L96-L100, [Online; accessed 1-April-2025].

[60] "rsync," https://rsync.samba.org/, [Online; accessed 1-April-2025].

[61] J. Kim, J. Park, S. Roh, J. Chung, Y. Lee, T. Kim, and B. Lee, "Tiktag: Breaking arm's memory tagging extension with speculative execution," in *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, M. Blanton, W. Enck, and C. Nita-Rotaru, Eds. IEEE, 2025, pp. 4063–4081. [Online]. Available: https://doi.org/10.1109/SP61157.2025.00039

[62] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[63] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom *et al.*, "Meltdown: Reading kernel memory from user space," *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020.

[64] About the debug register interface. [Online]. Available: https://developer.arm.com/documentation/ddi0379/a/Debug-Register-Interfaces/About-the-Debug-Register-Interface?lang=en

[65] "Signal handling (the gnu c library)," https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html, [Online; accessed 1-April-2025].

[66] Environmentvariables - community help wiki. [Online]. Available: https://help.ubuntu.com/community/EnvironmentVariables

[67] "The gnu c library - gnu project - free software foundation," https://www.gnu.org/software/libc/, [Online; accessed 1-April-2025].

[68] "glibc/sysdeps/aarch64/strcpy.s," https://github.com/bminor/glibc/blob/ce2f26a22e6b6f5c108d156afd9b43a452bb024c/sysdeps/aarch64/strcpy.S, [Online; accessed 1-April-2025].

[69] "aarch64: Mte compatible strcpy," https://github.com/bminor/glibc/commit/bb2c12aecbd26a8d29f63b51b80b7c84e65d1818, [Online; accessed 1-April-2025].

[70] System v contexts (the gnu c library). [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/System-V-contexts.html

[71] M. Momeu, A. J. Gaidis, J. v. d. Heidt, and V. P. Kemerlis, "IUBIK: isolating user bytes in commodity operating system kernels via memory tagging extensions," in *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, M. Blanton, W. Enck, and C. Nita-Rotaru, Eds. IEEE, 2025, pp. 867–885. [Online]. Available: https://doi.org/10.1109/SP61157.2025.00135

[72] "Termux — the main termux site and help pages." https://termux.dev/en/, [Online; accessed 1-April-2025].

[73] "Getting started - termux wiki," https://wiki.termux.com/wiki/Getting_started, [Online; accessed 1-April-2025].

[74] "chroot(2) - linux manual page," https://man7.org/linux/man-pages/man2/chroot.2.html, [Online; accessed 1-April-2025].

[75] Clang 14.0.0 documentation. [Online]. Available: https://releases.llvm.org/14.0.0/tools/clang/docs/index.html

[76] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, pp. 49:1–49:29, 2020. [Online]. Available: https://doi.org/10.1145/3428334

[77] M. Unterguggenberger, D. Schrammel, P. Nasahl, R. Schilling, L. Lamster, and S. Mangard, "Multi-tag: A hardware-software co-design for memory safety based on multi-granular memory tagging," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*, J. K. Liu, Y. Xiang, S. Nepal, and G. Tsudik, Eds. ACM, 2023, pp. 177–189. [Online]. Available: https://doi.org/10.1145/3579856.3590331

[78] Thunderx2 - cavium - wikichip. [Online]. Available: https://en.wikichip.org/wiki/cavium/thunderx2

[79] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbah, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 219–233. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/keahey

[80] Avoiding runcpu - cpu 2017. [Online]. Available: https://www.spec.org/cpu2017/Docs/runcpu-avoidance.html

[81] "Preview versions - geekbench," https://www.geekbench.com/preview/, [Online; accessed 1-April-2025].

[82] "Geekbench 6 - apps on google play," https://play.google.com/store/apps/details?id=com.primatelabs.geekbench6&hl=en_US, [Online; accessed 1-April-2025].

[83] "Geekbench 6 on the app store," https://apps.apple.com/us/app/geekbench-6/id1565728895, [Online; accessed 1-April-2025].

[84] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 17–30. [Online]. Available: http://www.usenix.org/events/usenix05/tech/general/seward.html

[85] antirez/mc-benchmark. [Online]. Available: https://github.com/antirez/mc-benchmark

[86] leveldb/benchmarks/db_bench.cc. [Online]. Available: https://github.com/google/leveldb/blob/main/benchmarks/db_bench.cc

[87] rocksdb/tools/db_bench_tool.cc. [Online]. Available: https://github.com/facebook/rocksdb/blob/main/tools/db_bench_tool.cc

[88] M. Ugur, C. Jiang, A. Erf, T. A. Khan, and B. Kasikci, "One profile fits all: Profile-guided linux kernel optimizations for data center applications," *ACM SIGOPS Operating Systems Review*, vol. 56, no. 1, pp. 26–33, Jun. 2022.

[89] Release v1.2.1, hexhive/magma. [Online]. Available: https://github.com/HexHive/magma/tree/v1.2.1

[90] A. Fioraldi, D. C. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Y. Yarom and S. Zennou, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[91] llvm_mode persistent mode. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md

[92] Ampereone product brief. [Online]. Available: https://amperecomputing.com/briefs/ampereone-family-product-brief

[93] S. Kaushik, M. Madhav, N. Aboulenein, J. Bessette, S. Brahmadathan, B. Chaffin, M. Erler, S. Jourdan, T. Maciukenas, R. Masti *et al.*, "Optimized memory tagging on ampereone processors," *arXiv preprint arXiv:2511.17773*, 2025.

[94] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2313–2328. [Online]. Available: https://doi.org/10.1145/3133956.3134046

[95] Z. Zhou, V. Gogte, N. Vaish, C. Kennelly, P. Xia, S. Kanev, T. Moseley, C. Delimitrou, and P. Ranganathan, "Characterizing a memory allocator at warehouse scale," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds. ACM, 2024, pp. 192–206. [Online]. Available: https://doi.org/10.1145/3620666.3651350

[96] T. Noh, Y. Wang, T. Garfinkel, M. Madhav, D. Moghimi, M. Erez, and S. Narayan, "Arm mte performance in practice," in *Usenix Security Symposium*, 2026.

[97] J. You, J. Seo, K. Lee, Y. Cho, and Y. Paek, "Bastag: Byte-level access control on shared memory using arm memory tagging extension," in *Proceedings of the 32nd ACM Conference on Computer and Communications Security (CCS '25)*. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://junseungyou.github.io/assets/bastag.pdf

[98] Memory integrity enforcement: A complete vision for memory safety in apple devices. [Online]. Available: https://security.apple.com/blog/memory-integrity-enforcement/

[99] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 866–879. [Online]. Available: https://doi.org/10.1109/SP.2015.58

[100] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, J. Vitek and D. Lea, Eds. ACM, 2010, pp. 31–40. [Online]. Available: https://doi.org/10.1145/1806651.1806657

[101] G. J. Duck and R. H. C. Yap, "Heap bounds protection with low fat pointers," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, A. Zaks and M. V. Hermenegildo, Eds. ACM, 2016, pp. 132–142. [Online]. Available: https://doi.org/10.1145/2892208.2892212

[102] G. J. Duck, R. H. C. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/stack-object-protection-low-fat-pointers/

[103] R. M. Farkhani, M. Ahmadi, and L. Lu, "Ptauth: Temporal memory safety via robust points-to authentication," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. D. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 1037–1054. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade

[104] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, "Ivysyn: Automated vulnerability discovery in deep learning frameworks," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 2383–2400. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/christou

[105] A. Hager-Clukas and K. Hohentanner, "DMTI: accelerating memory error detection in precompiled C/C++ binaries with ARM memory tagging extension," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*, J. Zhou, T. Q. S. Quek, D. Gao, and A. A. Cárdenas, Eds. ACM, 2024. [Online]. Available: https://doi.org/10.1145/3634737.3637655

[106] Y. Chen and S. Li, "Hemate: Enhancing heap security through isolating primitive types with arm memory tagging extension," in *Proceedings of the 19th International Conference on Availability, Reliability and Security, ARES 2024, Vienna, Austria, 30 July 2024 - 2 August 2024*. ACM, 2024, pp. 30:1–30:11. [Online]. Available: https://doi.org/10.1145/3664476.3664492

[107] H. Liljestrand, C. C. Perez, R. Denis-Courmont, J. Ekberg, and N. Asokan, "Color my world: Deterministic tagging for memory safety," *CoRR*, vol. abs/2204.03781, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.03781

[108] "Memtagsanitizer - llvm 21.0.0git documentation," https://llvm.org/docs/MemTagSanitizer.html, [Online; accessed 1-April-2025].

[109] J. Kim, J. Park, Y. Lee, C. Song, T. Kim, and B. Lee, "Petal: Ensuring access control integrity against data-only attacks on linux," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, Eds. ACM, 2024, pp. 2919–2933. [Online]. Available: https://doi.org/10.1145/3658644.3690184

[110] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakcs," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-257/

[111] J. Seo, J. You, Y. Cho, Y. Cho, D. Kwon, and Y. Paek, "Sfitag: Efficient software fault isolation with memory tagging for ARM kernel extensions," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*, J. K. Liu, Y. Xiang, S. Nepal, and G. Tsudik, Eds. ACM, 2023, pp. 469–480. [Online]. Available: https://doi.org/10.1145/3579856.3590341

[112] S. Lim, T. Prasad, X. Han, and T. Pasquier, "Safebpf: Hardware-assisted defense-in-depth for ebpf kernel extensions," in *Proceedings of the 2024 on Cloud Computing Security Workshop, CCSW 2024, Salt Lake City, UT, USA, October 14-18, 2024*, A. P. Fournaris and P. Palmieri, Eds. ACM, 2024, pp. 80–94. [Online]. Available: https://doi.org/10.1145/3689938.3694781

[113] K. D. Duy, K. Cho, T. Noh, and H. Lee, "Capacity: Cryptographically-enforced in-process capabilities for modern ARM architectures," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 874–888. [Online]. Available: https://doi.org/10.1145/3576915.3623079

[114] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 20–37. [Online]. Available: https://doi.org/10.1109/SP.2015.9

[115] J. Z. Yu, C. Watt, A. Badole, T. E. Carlson, and P. Saxena, "Capstone: A capability-based foundation for trustless secure memory access," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 787–804. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason

[116] J. Z. Yu, M. Li, A. Badole, T. E. Carlson, M. Swift, and P. Saxena, "Caplification: Bridging capability-aware and capability-oblivious software," in *Proceedings of the 30th ACM Symposium on Access Control Models and Technologies*, 2025, pp. 33–44.

[117] N. W. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, A. T. Markettos, A. Mazzinghi, R. M. Norton, M. Roe, P. Sewell, S. D. Son, T. M. Jones, S. W. Moore, P. G. Neumann, and R. N. M. Watson, "Cornucopia: Temporal safety for CHERI heaps," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 608–625. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00098

[118] Z. Liu, Y. Rong, C. Li, W. Tan, Y. Li, X. Han, S. Yang, and C. Zhang, "CCTAG: configurable and combinable tagged architecture," in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/cctag-configurable-and-combinable-tagged-architecture/

[119] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the C programming language," pp. 103–114, 2008. [Online]. Available: https://doi.org/10.1145/1346281.1346295

[120] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 225–240. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/zeldovich/zeldovich.pdf

[121] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A. Sadeghi, "TIMBER-V: tag-isolated memory bringing fine-grained enclaves to RISC-V," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/timber-v-tag-isolated-memory-bringing-fine-grained-enclaves-to-risc-v/

[122] "Application data integrity (adi)," https://docs.kernel.org/arch/sparc/adi.html, [Online; accessed 1-April-2025].

[123] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using califorms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 558–571. [Online]. Available: https://doi.org/10.1145/3352460.3358299

[124] T. Zhang, D. Lee, and C. Jung, "BOGO: buy spatial memory safety, get temporal memory safety (almost) free," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 631–644. [Online]. Available: https://doi.org/10.1145/3297858.3304017

[125] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the intel MPX system stack," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, pp. 28:1–28:30, 2018. [Online]. Available: https://doi.org/10.1145/3224423

[126] F. Qin, S. Lu, and Y. Zhou, "Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs," in *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005), 12-16 February 2005, San Francisco, CA, USA*. IEEE Computer Society, 2005, pp. 291–302. [Online]. Available: https://doi.org/10.1109/HPCA.2005.29

[127] K. Sinha and S. Sethumadhavan, "Practical memory safety with REST," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, M. Annavaram, T. M. Pinkston, and B. Falsafi, Eds. IEEE Computer Society, 2018, pp. 600–611. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00056

[128] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon

[129] W. Song, D. Xie, Z. Xue, and P. Liu, "A parallel tag cache for hardware managed tagged memory in multicore processors," *IEEE Transactions on Computers*, vol. 73, no. 11, pp. 2488–2503, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10633901

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

The paper presents NANOTAG, a system that improves the bug-detection capability of ARM's Memory Tagging Extension (MTE) by enabling byte-granular detection of heap memory safety bugs in unmodified binaries. The authors first conduct an empirical study on real MTE hardware (Google Pixel 8) and show that MTE's fixed 16-byte tag granularity causes it to miss a substantial fraction of heap-based buffer overflows, particularly intra-granule overflows. They demonstrate that such cases are common in real programs and can lead to silent failures even for real-world vulnerabilities. To address this limitation, NANOTAG combines MTE's low-overhead hardware checks with selective software-based checking using a sampling-based "tripwire" mechanism for short (partially used) tag granules.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Provides a Valuable Step Forward in an Established Field.

A.3. Reasons for Acceptance

1) This paper provides a valuable step forward in an established field. Overall, the paper provides a practical engineering improvement that increases detection granularity while maintaining relatively low overhead, advancing the feasibility of fine-grained memory error detection on real hardware and motivating further research on hybrid hardware/software memory safety techniques. The paper shows that ARM MTE's coarse granularity can cause a meaningful portion of heap overflows to go unnoticed, making a good case for exploring byte-level detection. The proposed approach can increase the detection granularity while maintaining manageable overheads.
2) The paper creates a new tool to enable future science. A real prototype of the proposed idea has been implemented and evaluated.

A.4. Noteworthy Concerns

1) The paper does not have a threat model or a discussion of the security implications of sampling. This makes it unclear if NanoTag would be applicable to production systems. For example attackers in an adversarial setting could wait for the end of the slow start to attack.
2) The paper does not discuss the trade-offs of the probabilistic approach in a fuzzing environment, where a probabilistic technique will make it hard to reproduce bugs.