MTSan: A Feasible and Practical Memory Sanitizer for Fuzzing COTS Binaries

Xingman Chen[1], Yinghao Shi[5], Zheyu Jiang[1], Yuan Li[1], Ruoyu Wang[3], Haixin Duan[1,2], Haoyu Wang[4], Chao Zhang[1,2*]

[1]Tsinghua University [2]Zhongguancun Laboratory [3]Arizona State University
[4]Huazhong University of Science and Technology [5]Institute of Information Engineering, Chinese Academy of Sciences
{cxm16,jiangzhe19,li-y18}@mails.tsinghua.edu.cn, {duanhx,chaoz}@tsinghua.edu.cn,
shiyinghao@iie.ac.cn, fishw@asu.edu, haoyuwang@hust.edu.cn

Abstract

Fuzzing has been widely adopted for finding vulnerabilities in programs, especially when source code is not available. But the effectiveness and efficiency of binary fuzzing are curtailed by the lack of memory (safety) sanitizers. This lack of binary sanitizers is due to the information loss in compiling programs and challenges in binary instrumentation.

In this paper, we present a feasible and practical hardware-assisted memory sanitizer, MTSan, for binary fuzzing. MTSan can detect both spatial and temporal memory safety violations at runtime. It adopts a novel *progressive object recovery* scheme to recover objects in binaries, and uses a customized binary rewriting solution to instrument binaries with the memory-tagging-based memory safety sanitizing policy. Further, MTSan uses a hardware feature, ARM Memory Tagging Extension (MTE) to significantly reduce its runtime overhead. We implemented a prototype of MTSan on AArch64 and systematically evaluated its effectiveness and performance. Our evaluation results show that MTSan could detect more memory safety violations than existing binary sanitizers whiling introducing much lower runtime and memory overhead.

1 Introduction

Fuzz testing (also known as fuzzing) is a popular solution for finding bugs and security vulnerabilities in programs. Fuzzers generate a large number of test cases to test target programs and catch signals of security policy violations (such as crashes).

As a signal for bugs or vulnerabilities, crashes are insufficient. For example, an out-of-bound (OOB) memory write may not trigger any crashes if it overwrites a previously allocated memory region. Therefore, security researchers and "bug hunters" instrument programs with memory error detectors, or commonly referred to as *memory sanitizers*, before fuzzing to expose memory safety violations as soon as possible. Various sanitizers have been proposed, including AddressSanitizer (ASan [1]), ThreadSanitizer [2], UBSan [3], and MemSan [4], to help discover memory safety bugs, data race bugs, undefined behavior bugs, and uninitialized variables, respectively. These sanitizers instrument the source code of target programs with specific security checks to detect spatial and temporal memory violations at runtime.

While people have used fuzzers to find vulnerabilities in closed-source programs (which we will refer to as *binaries* in the rest of this paper), only a few sanitizers (e.g., Valgrind [5], ASan-Retrowrite [6], and QASan [7]) support instrumenting and detecting memory bugs in binaries. Unfortunately, severe limitations in these binary sanitizers prevent fuzzers from using them in vulnerability discovery. We detail these limitations below.

First, existing binary sanitizers only support detecting memory errors in heap and neglect memory errors in stack and global memory regions. This is because type information is lost when compiling the binaries, and as a result, binary sanitizers cannot recover object boundaries for objects in stack or global regions.

Second, even if object boundaries for stack and global objects are made available, existing binary sanitizers (including Memcheck in Valgrind, ASan-Retrowrite, and QASan) cannot instrument binaries with code for detecting memory errors in stack and global regions. This is because these binary sanitizers rely on redzone-based memory error detection schemes, and it is impossible to add redzones for stack and global objects without recompiling the binary or adjusting memory layouts.

Last but not least, existing binary sanitizers all introduce prohibitively high runtime and memory overhead. For example, due to the use of dynamic binary instrumentation (DBI), the runtime overhead of Memcheck is about 17.42×, and the runtime overhead of QASan is about 35.5× (on the SPEC2017 C benchmark in our experiment). While ASan-Retrowrite achieves much lower runtime overhead (2.2× lower runtime overhead than QASan with Qemu in our fuzzing experiment) by performing *static* binary instrumentation, it still introduces high memory overhead: Its average

*Corresponding author: chaoz@tsinghua.edu.cn

memory overhead is around 6.45×. High runtime and memory overhead reduces the fuzzing efficiency and limits the applicability of these binary sanitizers.

In this paper, we propose a hardware-assisted memory sanitizer for binary programs, MTSan, that addresses all three limitations. Without accessing the source, MTSan statically rewrites the target binary and enables the detection of spatial and temporal memory safety violations for heap, stack, and global objects, without changing the layout of any memory regions. Through the use of a new hardware feature on recent processors, memory tagging [8], MTSan exhibits much better runtime performance than existing binary sanitizers. These advantages make MTSan an ideal memory sanitizer for fuzzing binaries.

To overcome the challenge of missing type information in binaries, we introduce a novel approach, called *progressive object recovery*, that probabilistically recovers object boundaries using memory access information available during fuzzing. Because the inference of stack and global object boundaries is probabilistic, MTSan may incorrectly infer their boundaries. Such incorrect inferences may lead to false positive reports in fuzzing. We minimize the impact of incorrect inferences by proposing an *adaptive sanitization* strategy: MTSan intelligently determines the criticality of memory safety violation alarms and only reports the ones that are deemed *critical*. For *non-critical* reports, MTSan records them, updates the currently inferred object boundaries, without interrupting the fuzzing process. This way MTSan can focus on true positives without flooding analysts with false positive alarms.

Further, MTSan uses a new CPU feature, memory tagging, that is slated to be deployed soon on modern ARM processors [9], to significantly reduce its runtime overhead. Memory tagging is available on SPARC [10] processors, will soon be available on ARM processors via ARM Memory Tagging Extension (MTE) [11], and a subset of memory tagging (pointer tagging) will be available on Intel CPUs in the near future [12, 13]. This is the right time to study the use of memory tagging in binary sanitizers.

We implemented a prototype of MTSan on AArch64 and systematically evaluated its effectiveness on a set of popular programs with a total of 27 spatial and temporal memory errors. MTSan detected most Proof-of-Concept exploits (PoCs) in 18 vulnerabilities, which outperforms all state-of-the-art memory safety sanitizers for binaries. We further evaluated the runtime and memory overhead of MTSan on SPEC CPU 2017 [14]. The results showed that MTSan introduced average runtime overhead of 1.82× and memory overhead of 1.58×. Comparing against ASan-Retrowrite, MTSan introduced 48% lower runtime overhead and 91% lower memory overhead. Finally, we evaluated the applicability of MTSan as a memory sanitizer for fuzzing binaries. During our experiments, fuzzing with MTSan consistently led to the finding of at least three more vulnerabilities than fuzzing with other sanitizers. The evaluation results with analog instructions are also promising: MTSan yields most executions, and improves fuzzing performance by 58% when comparing to AFL++'s qemu mode. This demonstrated that MTSan can effectively detect memory vulnerabilities during fuzzing.

Contributions. In summary, we make the following contributions:

- We propose a novel *progressive object recovery* scheme for probabilistically inferring object boundaries for objects in heap, stack, and global regions.
- We introduce a novel hardware-assisted memory sanitizer, MTSan, to assist with binary fuzzing. MTSan uses ARM Memory Tagging Extension (MTE) to efficiently detect temporal and spatial memory errors.
- Because MTE is not currently available in off-the-shelf ARM processors, researchers must emulate MTE in software. To ease this process, we implement a library, *libMTE*, that simulates critical features that MTE provides on non-MTE-equipped processors.
- We implement a prototype of MTSan and systematically evaluate it regarding security, runtime and memory overhead, and effectiveness to fuzzing. The results show that MTSan outperforms state-of-the-art binary sanitizers.

In the spirit of open science, we make our code avaliable at https://github.com/vul337/mtsan-repo to help future studies.

2 Background

2.1 Memory Sanitizers

Fuzzing has been demonstrated in both academia and industry to exhibit unparalleled power in finding software bugs. One of the best fuzzing practices is combining a fuzzer with sanitizers, which find bugs sooner than crashes occur by observing incorrect behaviors for specific classes of security violations during fuzzing [17]. This is because not all bugs or security violations necessarily lead to crashes, and diagnosing root causes of crashes is not always straightforward. For example, a stack-based buffer-overflow may overwrite adjacent stack variables and alter the execution flow of the program, without crashing the process.

Most sanitizers focus on finding *memory safety violations* [17]. Memory safety violations are memory access errors caused by either dereferencing a pointer pointing outside the bounds of an intended object in memory (spatial memory safety violation), or using a pointer that is no longer valid (temporal memory safety violation).

According to their memory access checking approaches, memory sanitizers generally fall into one of the following categories [17]: (1) Location-based sanitizers, which insert invalid memory regions, e.g., *redzones*, between objects in memory and report memory safety violations when any invalid memory regions are accessed. (2) Identity-based san-

Table 1: Anatomy of existing binary sanitizers.

Binary Sanitizer	Bug-finding Techniques	Instrumentation Method	Detectable Violation Types			Object Coverage			Runtime Overhead⋆	Memory Overhead⋆
			Spatial	Temporal	Other	Heap	Stack	Global		
Undangle [15]	3	DBI	✗	∗	-	✓	✗	✗	>10×	>10×
Dr. Memory [16]	1, 2	DBI	✓	✓	-	✓	✗	✗	>10×	n/a
Memcheck [5]	1, 2, 4	DBI	✓	✓	uninit. use	✓	✗	✗	>10×	3-10×
QASan [7]	1, 2	DBI	✓	✓	-	✓	✗	✗	>10×	3-10×
ASan-Retrowrite [6]	1, 2	Binary Rewriting	✓	✓	-	✓	†	✗	1-3×	3-10×
MTSan	5, 6	Binary Rewriting	✓	✓	-	✓	✓	✓	1-3×	1-3×

1: Redzone, 2: Reuse-delay of heap obj., 3: Pointer Tracking, 4: Uninit. Value Tracking, 5: Pointer Tagging, 6: Memory Tagging;
∗: Heap Use-after-free. †: OOB access on stack canary. ⋆: Standalone execution, with no optimization applied.

itizers, which maintain metadata for memory objects, and check the intended referent for every pointer in the program.

Sanitizers for binaries. Traditionally, memory sanitizers only work on programs with source code because the memory access checking approaches (as previously mentioned) require adjusting the memory layout or manipulating pre-acquired metadata for objects. Because much information, especially types, is discarded during compiling, sanitizers cannot easily identify object sizes in a binary program, which renders the above two approaches infeasible.

As shown in Table 1, researchers have proposed several binary sanitizers in recent years. However, existing binary sanitizers suffer from some critical limitations. First, because existing binary sanitizers are all location-based, they cannot detect memory safety violations that happen in stack or global memory regions. Second, these sanitizers (except for ASan-Retrowrite) all suffer from prohibitively high runtime overhead due to their dependence on dynamic binary instrumentation (DBI) techniques. While ASan-Retrowrite (based on static binary rewriting) has low runtime overhead, it still introduces high memory overhead. High runtime and memory overhead slows down fuzzing and makes these sanitizers unsuitable for binary fuzzing. Hence, we conclude that a practical binary sanitizer must (1) be able to detect memory errors in all locations, and (2) yield low runtime and memory overhead.

2.2 Memory Tagging

Memory Tagging is a security feature that facilitates the detection of memory access violations by adding unique tags (in the form of bits) to both pointers and memory space. During runtime, it then checks these tags at every memory access to ensure that memory space is accessed with its corresponding pointer. Memory tagging can be implemented in software (in emulators) or on hardware (in processors), where the latter adds significantly less overhead. Some architectures, including lowRISC [18], SPARC [10], and ARM [19], have introduced memory tagging or its equivalent.

Memory Tagging Extension on ARM. ARM first introduced Memory Tagging Extension (MTE) in ARMv8.5-A, and has started to build MTE into ARMv9-compliant CPUs, as recently announced [9]. MTE in ARM includes both *address tagging* and *memory tagging*.

Address Tagging. MTE utilizes the Top Byte Ignore (TBI) feature that was introduced in ARMv8.1 [20]. TBI allows ARM processors to ignore the top byte of each pointer; The ignored byte can then be used to store extra metadata. MTE uses four bits out of the ignore byte as the address (or pointer) tag. These tags are propagated by ARM processors with zero runtime overhead.

Memory Tagging. Like address tags, each memory tag also consists of four bits. A memory tag associates with an aligned 16-byte chunk of memory space. Memory tags are stored separately from the physical memory.

Tag Manipulation. MTE introduces additional instructions to manipulate the pointer and memory tags: The instruction `IRG` tags a register with a random 4-bit pointer tag. Instructions `LDG` and `STG` will get or set memory tags. All memory accesses to tagged memory chunks must be done via pointers with matching tags. Since each tag has four bits, there can be at most 16 unique tags. Therefore, collision may arise: A tagged pointer pointing to a different memory chunk may match a tagged memory chunk by coincidence.

3 Overview

In this section, we discussed the typical workflow of MTSan (Section 3.1). We also explained how MTSan helps find a real-world vulnerability `CVE-2017-0947` in a popular project, `libxml2`, at Section A.1 for a better understanding.

3.1 Workflow

As shown in Figure 1, MTSan has three main components: Binary Analyzer, Binary Rewriter, and MTSan Runtime Library. Binary Analyzer identifies pointers to objects, generates initial object metadata and instructions for tagging memory and pointers. Then Binary Rewriter statically instruments the target binary with instructions generated, together with the MTSan Runtime Library. MTSan Runtime Library is the core, which not only maintains an up-to-date status of object recovered, but also classifies memory violations and handles them with specific strategies.

The typical workflow of MTSan is as follows. At the beginning of a fuzzing campaign, MTSan enters the object boundary inference mode, where it performs progressive object recovery and waits for conflicts in inferred object boundaries or mismatched memory accesses to arise. In either case, MTSan discovers a potential memory safety violation, en-

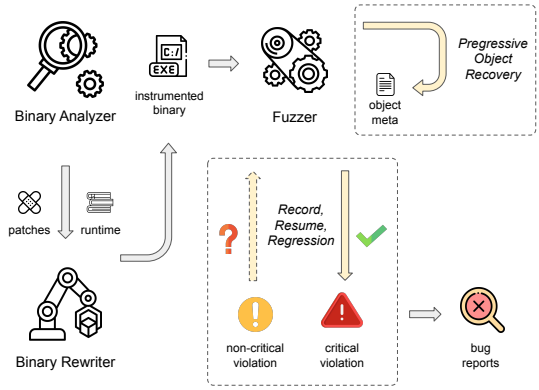


Figure 1: Overview of the MTSan pipeline.

ters adaptive sanitization, and determines its severity by categorizing it into two severity levels. If the memory safety violation is deemed *critical* (i.e., we are certain that a true-positive memory safety bug is found), MTSan will immediately report an error and terminates the execution of the current process (but not the entire fuzzing run). If the violation is deemed *non-critical*, MTSan will record the violation and the violating input, resume fuzzing, and perform a regression fuzzing of the recorded input at a later time. We refer to it as a *Record-Resume-Regression* (RRR) scheme.

4 Progressive Object Recovery

To detect spatial and temporal memory safety violations in binaries, a fundamental challenge that MTSan must address is the recovery of object properties: Boundaries and lifetime of objects. While type inference in binaries remains an open research problem, MTSan only requires inferring object sizes instead of accurately inferring variable types.

Inspired by pioneering research [21, 22] that uses runtime data for variable type inference, we design *progressive object recovery* for MTSan. During runtime, MTSan first identifies all pointers pointing to objects in heap, stack, and global regions (Section 4.1). Then MTSan infers the boundary and lifetime for each object during individual executions (Section 4.2). Because fuzzing involves a huge number of executions, MTSan further unifies inference results from different executions during fuzzing and *progressively* refines all inferred object properties (Section 4.3).

4.1 Object Pointer Identification

Because heap objects are always explicitly allocated and deallocated (e.g., allocated by calling `malloc`), identifying heap object pointers is trivial. We focus on identifying pointers pointing to objects in stack and global regions. MTSan captures (1) raw object pointers, including values from stack pointer register (`SP`), values directly derived out of `SP`, and memory addresses in global memory regions, and (2) pointers that are derived from raw object pointers via pointer arithmetic.

MTSan also captures allocation and deallocation sites for each pointer. For a stack object, its allocation site is the instruction that allocates the stack frame, and its deallocation site is the instruction that releases the stack frame. For a global object, we regard the process initialization as its allocation site.

4.2 Object Property Inference

MTSan infers object properties, i.e., the boundary and lifetime of an object, by tracking how its pointers are used at runtime. We briefly discuss how MTSan infers boundaries and lifetime of objects during a single execution.

Inferring object boundaries. Boundaries of heap objects can be determined by observing the `size` argument of allocators. Thus, our discussion focuses on stack and global variables. We assume all objects must be accessed either via its raw pointer (which points to the beginning of the object) or a derived pointer of the raw pointer (which points to inside the object). To ease explanations in the rest of this paper, we define an operator `deref(addr, size)` to describe "loading `size` bytes from address `addr`." For example, suppose an address `A` points to the beginning of an object `alpha`, then the operations `deref(A, 8)` and `deref(A+24, 8)` mean that the memory space from `A` to `A+32` (including the gap) belongs to `alpha`. MTSan uses intra-procedural value-set analysis (VSA) to statically recover the range of each object. This allows MTSan to initialize as many metadata slots (of objects) as possible, which reduces the number of metadata updates at runtime.

Note that the assumption that all accesses to an object must be derived out of the same pointer is too strict and does not always hold for stack or global objects. We will discuss how MTSan handles such cases in Section 5.2.

Determining object lifetime. The lifetime of the heap object starts when it is allocated and terminates when it is deallocated. The lifetime of a global object starts and ends with the process. Finally, the lifetime of a stack object starts when its corresponding stack frame is allocated and ends when the frame is deallocated.

Object properties may be either *deterministic* or *presumptive*. Deterministic properties are always correct once they are inferred by MTSan, while presumptive properties can be incorrectly inferred. For example, lifetime of an object is deterministic because it is solely determined by the allocation and deallocation sites of the object. Likewise, the boundary of a heap object is deterministic because it is determined during allocation. The boundaries for most stack and global objects are presumptive: For example, we cannot infer the real size of a 24-byte char array if only the first 10 bytes are ever used during runtime. In addition to intra-procedural VSA, MTSan also utilizes known program properties (e.g., saved frame pointers, stack canaries, saved return addresses, and saved function call arguments) to identify as many deterministic object boundaries as possible.

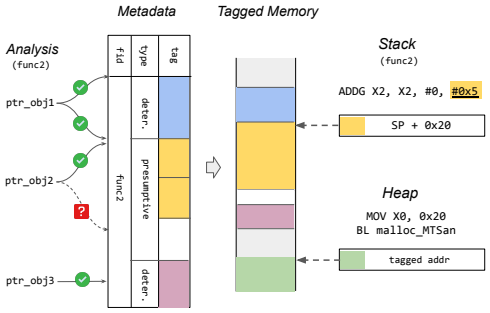


Figure 2: An example of the metadata region and instrumented code for pointer and memory tagging. Different colors represent memory areas with different tags.

4.3 Progressive Recovery of Object Properties

As fuzzing progresses, more unique executions are found, which provides an opportunity for MTSan to progressively refine presumptive object properties. For example, suppose MTSan has inferred the size of object `alpha` as $4 \times$ `sizeof(int)` during one execution, then it observes a memory access `deref`$(A+7\times$`sizeof(int)`$,$`sizeof(int)`$)$ during another execution. MTSan will update the upper bound of `alpha` from $A+4\times$`sizeof(int)` to $A+8\times$`sizeof(int)` by updating the metadata. A conflict is created if the new upper bound overlaps with an existing object, which is a sign of potential memory safety violations. We will discuss how to handle conflicts in Sections 5.1 and 5.2.2.

5 Adaptive Sanitization

With inferred object properties, MTSan rewrites the binary, injects memory sanitization logic, and sends it to a fuzzer. Due to the existence of presumptive object boundaries, some memory errors that MTSan reports are inevitably false positives. We deem binary rewriting as an engineering challenge and will discuss it in Section 6. In this section, we will present our sanitization method and conflict-resolving strategies with a focus on reducing false positive alarms.

5.1 Sanitization Approach

Hardware-assisted memory tagging allows MTSan to detect spatial and temporal memory access violations with extremely low overhead: A pointer access is only valid if the pointer and the memory location it points to have the same tag. When a tagged pointer is used to access a memory location with a different (unmatched) tag, ARM MTE will generate a segmentation fault signal. Key steps include *tag generation*, *tag assignment*, and dealing with *memory safety violation reports*.

5.1.1 Tag Generation

MTSan assigns every object in the target binary a random tag, which will be used to tag its corresponding pointers and the memory space. MTSan ensures that adjacent objects have different tags: When assigning a tag to an object, MTSan generates a tag that differs from the tags of its neighboring objects. For heap objects, MTSan does not store tags anywhere in memory because they are directly used in the generated instrumentation code for tag assignment.

Unlike tags for heap objects, stack and global object tags must be stored in an allocated memory region, which we will refer to as the *metadata region*. This is because MTSan will update the metadata and generate new tags when it discovers new objects during progressive object recovery. Figure 2 shows an example of the metadata region. To minimize runtime overhead, each function and each global object in the target binary has a unique slot at fixed offsets in the metadata region. During tag generation, tags for stack and global objects will be stored in the metadata region.

Compound objects. MTE supports tagging memory at a granularity of 16 bytes. MTSan stays consistent with this granularity, which means it can only describe objects whose sizes are greater than or equal to 16 bytes. To address this challenge, we introduced the *compound object*, which bundles adjacent small objects into one to create objects that are large enough. Overflows between sub-objects within a compound object cannot be detected by MTSan and are an unsolved problem. They are a source of false negatives, and we will discuss them in Section 8.

5.1.2 Tag Assignment and Propagation

For heap objects, MTSan will tag their pointers and their memory spaces with the same tags (as previously generated) immediately upon the allocation of objects. This is done by inserting MTE instructions into the binary at allocation sites. Tagging memory for stack and global objects is more complicated. MTSan inserts at their allocation sites code snippets that will find object tags in corresponding slots in the metadata region, and then tag pointers and memory locations accordingly. Finally, MTSan retags the memory space that objects reside at their deallocation sites with a different random tag to prevent UAF.

Pointers are frequently used in comparisons and subtractions, which means blindly tagging pointers may break program logic. For example, the comparison result of two pointers may be used as an exit-loop condition, and tagging these pointers may incorrectly impact the comparison. To address this problem, MTSan checks the operands of certain instructions at runtime and replaces them with top-byte ignoring instructions (`SUBP`, `SUBPS`, and `CMPP`) when necessary.

Thanks to MTE, during runtime, object tags are propagated with the pointer at no extra cost. When an object is deallocated, MTSan re-tags its memory location with a different tag, which means the original tagged pointer can no longer be used (without triggering a segmentation fault).

5.1.3 Memory Safety Violations

Once a tagged pointer is used to access a memory location with an unmatched tag, MTE will generate a segmentation fault signal indicating a memory error. Benefiting from

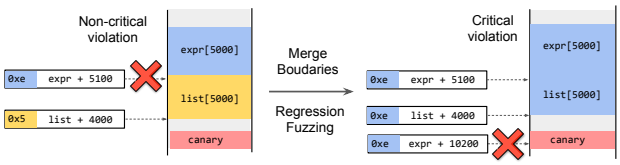


Figure 3: An example of Record, Resume, and Regression. Different colors represent memory areas with different tags.

MTE, MTSan can also detect memory safety violations that happen in library code when vulnerable objects are passed as arguments without having to instrument any library code. Upon process starting, MTSan Runtime Library registers a signal handler that catches such signals and performs further sanitization checks (see Section 5.2).

5.2 Adaptive Sanitization

Object pointer identification may fail due to the complexity in binary code. Compilers may emit multiple pointers to access the same object at different offsets. For example, two members of a struct on the stack can be directly accessed by two distinct stack pointers. MTSan may incorrectly recognize them as two raw object pointers that point to two objects (instead of one), causing false buffer overflow alarms (and other memory errors). If MTSan raises false alarms at vital program locations in a binary (e.g., at the beginning of the `main` function), simply terminating the process and reporting the alarm to fuzzers can completely stall fuzzing. Therefore, MTSan categorizes memory safety violations into two severity levels and adopts different strategies for each level.

5.2.1 Severity of Memory Safety Violations

Based on the likelihood of false positives, we categorized memory safety violations that MTSan reports into two severity categories. We deem a violation as **critical** if it only relies on checks of deterministic properties. All critical violations are true positive memory errors. Typical examples include: (1) A pointer is used to access another object with a deterministic boundary (e.g., stack canary), (2) An object pointer is used after the object's lifetime ends (e.g., use-after-free of a heap object), and (3) Any other exceptions or signals that Linux kernel raises. When a critical violation occurs, MTSan will terminate the process and notify the fuzzer.

We classify a memory safety violation as **non-critical** if it relies on checks of presumptive properties. For example, a pointer is used to access beyond its object's presumptive boundary. It is possible that this non-critical violation is a false positive caused by incorrectly inferred object properties. Instead of terminating the process, MTSan will use the RRR strategy as discussed next.

5.2.2 Record, Resume, and Regression

Non-critical violations, i.e., violations to presumptive boundaries that were recovered during progressive object recovery, are fed into the Record, Resume, and Regression (RRR) strategy. The intuition behind RRR is that given enough time, fuzzers will likely expose true positives and filter away false positives.

Step 1: Record. When a non-critical violation occurs, MTSan records the input and makes a copy of the metadata region.

Step 2: Resume. MTSan assumes that the report is a false positive (due to mistakes in object property inference) and correspondingly updates presumptive properties (i.e., boundaries of stack and global objects) by merging all involved objects into one (and updating the metadata region). Hence, future executions will no longer trigger the same violation. Then MTSan resumes the fuzzing process.

Step 3: Regression (fuzzing). MTSan will put the recorded input into the fuzzing queue by the time of violation to initiate a regression fuzzing, hoping to trigger a critical violation. This way, MTSan keeps around test cases that are likely satisfying the constraints of triggering vulnerabilities, even if these test cases do not increase code coverage. These extra test cases will increase the likelihood for the fuzzer to generate input that traverses the vulnerable path, which in turn increases the chance of exposing critical violations.

6 Implementation

Binary Analyzer. We built the Binary Analyzer component on top of angr [23] and IDA Pro [24]. We implemented object recovery using angr's forced execution and constant propagation, and allocation site analysis based on IDA Pro's function identification.

Binary Rewriter. We implemented the binary rewriter using capstone [25] and keystone [26]. Instead of generating multiple rewritten binaries (e.g., T-Fuzz [27] and StochFuzz [28]), MTSan statically instruments the target binary, puts all necessary code snippets in a shared memory region, and updates instructions at runtime when necessary. This enables the use of fork-server mode and eliminates performance overhead caused by repeatedly loading binary variants. For now, our Binary Rewriter only supports dynamic-linked AArch64 C binaries.

MTSan Runtime Library. MTSan Runtime Library implements progressive object recovery and adaptive sanitization. It also maintains and updates the metadata region. This library is loaded into the target process when it starts.

Fuzzer. We patched the fuzzer (AFL++ [29]) to support non-critical violations. Our patch records non-critical violations and adds the input to the queue. The patch can be easily ported to any greybox fuzzer, which means MTSan will support other fuzzers with more engineering effort.

6.1 Optimizations

Two-stage fuzzing. High-quality seed input is essential for fuzzing. By default, MTSan treats non-critical errors that are triggered by these seeds as benign. Before fuzzing, a fuzzer usually performs a dry run on all seeds, which MT-

San regards as the first stage and initializes the metadata region. Trusting input seeds and ignoring non-critical errors triggered by them allows MTSan to reduce the number of potential false positive alarms.

Tag reserving. MTSan reserves two tags for known safe memory areas to reduce the number of metadata look-ups: Tag `0x0` for unused memory, and tag `0xf` for safe objects. The remaining tags, tag `0x1` to `0xe` are used for tagging stack and global objects.

Safe stack and global objects. MTSan considers stack objects that are only accessed using constant pointers (`SP` + constant offset) *safe*, i.e., cannot be vulnerable objects of spatial or temporal memory safety violations. Similarly, MTSan deems certain global objects safe. This way MTSan can reduce the number of instrumented code snippets without compromising its ability of memory error detection.

Trusting accesses before the first read. MTSan treats all memory accesses as benign before the first read of the program, because as a design choice, MTSan only reports memory safety violations that are exposed by problematic input.

7 Evaluation

7.1 Experimental Setup

7.1.1 Dataset

We built a dataset of vulnerable programs by collecting reproducible vulnerabilities and projects from recently published papers [30–33] as well as the Linuxflaw project [34]. We then selected vulnerabilities that satisfy the following requirements: (1) The target program must be successfully compiled on AArch64, (2) The target vulnerability must be reproducible using at least one public Proof-of-Concept exploit (PoC), (3) The target program can be harnessed and fuzzed without excessive human effort, and (4) All binary sanitizers in our evaluation fully support the target program. We also ensured that various vulnerability types (spatial and temporal memory errors in stack, heap, and global memory regions) are covered. Further, we filtered out null-pointer-dereference, stack(-recursive)-overflow, divide-by-zero, and floating-point-exception vulnerabilities because they are out of scope for MTSan and are detectable without using sanitizers. Eventually, our dataset comprises 27 vulnerabilities. Table A.7 in the appendix details these vulnerabilities and target programs. We also used the `Juliet` test suite v1.3 [35] and evaluated MTSan on CWEs that are related to memory corruption. Table 2 details these test suites.

7.1.2 Comparison Targets

We carefully chose state-of-the-art binary sanitizers against which to compare MTSan, including Memcheck (in Valgrind) [5], QASan [7], and ASan-Retrowrite [6].

While all three sanitizers are available on AArch64, we had to make minor changes to fix issues in them. Binary sanitizers may not offer any readily available configurations for binary fuzzing. For example, ASan-Retrowrite supports coverage instrumentation and sanitization instrumentation on AArch64 PIE binaries. However, they are not configured to be enabled at the same time by default. We fixed several bugs and force-enabled two features simultaneously during the fuzzing experiments. We will upstream our patches.

7.1.3 Instruction Analogs and libMTE

Although there is growing community interest and support for ARM MTE, unfortunately, no hardware is available at the time of writing. Following the evaluation methodology of HAKC [36] and PARTS [37], we ensure correct functionality using software emulation and measure runtime and memory overhead using instruction analogs. We use the same instruction analogs (See Figure A.9 in the appendix) that HAKC uses to accurately reflect the runtime overhead of MTSan.

As part of our research, we implemented **libMTE**, a library that enables MTSan on commodity hardware without MTE support. LibMTE maintains a shadow memory for memory-tagging-enabled pages. libMTE instruments every memory access and adds an additional check to see if the pointer tag and the memory tag match. Like in real MTE, libMTE raises segmentation fault signals upon any failed tag checks. Together with MTSan, libMTE allows us to evaluate the performance of MTSan using pure software emulation, and to perform a large-scale binary fuzzing experiment to measure the effectiveness and efficiency of MTSan.

7.1.4 Evaluation Environment

We first ensure that MTSan functions correctly using emulation. Luckily, the community offers a well-supported software stack for AArch64 MTE. For example, Linux kernel has offered official support for MTE since 5.10 [38]. Qemu [39] and ARM's official emulator, Fixed Virtual Platforms (FVP) [40], both support MTE. For our effectiveness experiments, we use Qemu 6.0.0 with Linux kernel 5.15.0.

However, neither Qemu nor FVP is cycle-accurate. To get accurate performance estimates on real hardware, we ran all other experiments on a PC equipped with two HUAWEI Kunpeng 920 [41] and 378 GB RAM, running Ubuntu 21.04 and Linux kernel version 5.13.0. We conducted all experiments on the same machine.

7.2 Effectiveness of Memory Safety Violation Detection

7.2.1 Setup

In this experiment, we evaluated MTSan and other binary sanitizers using (1) the NIST `Juliet` test suite [35], and (2) PoCs of the 27 real-world vulnerabilities, to see how effectively these sanitizers can detect memory errors.

While we used PoCs to reproduce each vulnerability, the number of PoCs for each vulnerability is usually limited; in most cases, only one PoC is available. For a more realistic simulation of the fuzzing process, where vulnerabilities may be triggered by different input cases, we seeded ASan-

Table 2: Security evaluation results of Juliet test suite. In each test set (of a specific vulnerability type), there are multiple *good* and *bad* test cases, as presented in the *Cases* column. For MTSan, we calculate FP and FN rates using critical bug reports as the standard for TPs, and separately list TPs that were additionally detected with non-critical violations (*Non-C. Report*). To minimize noises caused by random input sources where certain Juliet test cases read, we take the best result among ten trials.

CWE-ID	Description	Cases	Valgrind		QASan		ASan-Retrowrite		MTSan			MTSan-no-rsv			MTSan-no-rec	
			FN	FP	FN	FP	FN	FP	FN	FP	Non-C. Report	FN	FP	Non-C. Report	FN	FP
121	Stack-based Buffer Overflow	3100 + 3100	54.29%	0	100.00%	0	90.19%	0	44.65%	0	19.29%	44.65%	0	19.29%	44.65%	0
122	Heap-based Buffer Overflow	3870 + 3870	3.72%	0	8.68%	0	5.25%	0	19.69%	0	0.00%	19.69%	0	0.00%	19.69%	0
124	Buffer Under-write	1168 + 1168	27.31%	0	27.74%	0	26.97%	0	1.46%	0	0.43%	1.46%	0	0.43%	1.46%	0
126	Buffer Over-read	870 + 870	60.69%	0	66.90%	0	66.90%	0	53.10%	0	13.79%	53.10%	0	13.79%	53.10%	0
127	Buffer Under-read	1168 +1168	54.88%	0	38.36%	0	38.36%	0	9.85%	0	0.43%	9.85%	0	0.43%	9.85%	0
415	Double Free	818 + 818	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0.00%	0	0.00%	0.00%	0
416	Use-After-Free	393 + 393	5.09%	0	15.78%	0	29.01%	0	3.05%	0	0.00%	3.05%	0	0.00%	3.05%	0
Total		**11387 + 11387**	**29.29%**	**0**	**42.61%**	**0**	**39.15%**	**0**	**24.17%**	**0**	**6.39%**	**24.17%**	**0**	**6.39%**	**24.17%**	**0**

instrumented programs with the original PoC, and fuzzed them for one hour under the crash exploration mode of AFL++ [29] to generate more PoCs. Then we used these newly generated test cases (many of which will trigger the intended vulnerabilities), together with benign input cases, to test the effectiveness of binary sanitizers.

We used GCC-10.3.0 to compile each target program with both their default compiler flags and -O3. For a fair comparison, we disabled memory leak and uninitialized memory checks, and filtered out any `SIGABRT`, `SIGSEGV`, `SIGBUS` and `SIGTERM` signals. Because *RRR* is only applicable during fuzzing, we disabled it in this experiment, and other components are enabled by default. Meanwhile, to understand the impact of each component, we also evaluated MTSan with varying configurations: Disabling object recovery (*-no-rec*); disabling the two-stage fuzzing (*-no-stg*); and no tag-reserving for special memory areas (*-no-rsv*).

Note that different PoCs for the same bug may trigger different types of memory violations. For example, a PoC for CVE-2017-9047 (see Appendix A.1) may only overwrite an adjacent object on the stack (which may be non-critical) while another PoC for the same CVE may overwrite the stored return address (critical). In our experiment, *critical* indicates that MTSan reports at least one critical memory safety violation for a bug, and *non-critical* indicates that only non-critical memory violations are reported for this bug.

7.2.2 Results: Juliet Test Suite

The `Juliet` test suite provides two variants for each test program: A *good* variant that is not vulnerable, and a *bad* variant that exhibits memory corruptions. A *false positive* (FP) is when a sanitizer reports a *good* program as vulnerable, and a *false negative* (FN) is when a sanitizer fails to report a *bad* program as vulnerable. Table 2 shows the results.

MTSan is more effective than comparison targets in detecting memory corruptions. Overall, MTSan exhibits an FN rate of 24.17%, which outperforms other sanitizers. When compared to ASan-Retrowrite, MTSan shows a 38.3% reduction in FN counts. Table 2 shows that MTSan obtains the lowest FN rate across five CWEs (121, 124, 126, 127, and 416). Besides, all sanitizers perform equally well identifying true negatives and have zero false positives.

MTSan is less effective in detecting off-by-one overflows. In Table 2, MTSan performs the worst among all sanitizers with the highest FN rate of 19.69% for CWE-122 (Heap-based Buffer Overflow). We investigated these FNs and found that they are mostly off-by-one overflows. We will further explain and discuss the limitations in Section 7.2.4.

Object recovery leads to more detected violations. We separately counted the cases where the target bug was only triggered with non-critical violations. These violations existed only when object recovery was enabled. MTSan detected non-critical violations for several CWEs (121, 124, 126, and 127). This is most eminent for CWE-121 (Stack-based Buffer Overflow) where 19.29% of TPs were only found with non-critical violations. Additionally, by comparing results between MTSan-no-rsv and MTSan, we conclude that reserving tags for special data types has no impact on the result for the Juliet test suite.

7.2.3 Results: Real-world Vulnerabilities

MTSan is more effective than existing binary sanitizers in detecting real-world memory bugs. According to Table 3, MTSan achieved the best results at most vulnerabilities. MTSan detected most Proof-of-Concept exploits (PoCs) in 18 vulnerabilities, which outperforms all state-of-the-art memory safety sanitizers for binaries. For CVE-2017-9047 (Listing A.1), MTSan reported 40 PoCs as critical and 449 PoCs as non-critical among 489 PoCs. Meanwhile, Valgrind, ASan-Retrowrite, and QASan failed to detect any of them.

MTSan detected most heap memory violations among all sanitizers. MTSan detected the highest number of PoCs among 5 out of 12 heap vulnerabilities, without any false positives. Valgrind and QASan also achieved high success rates in detecting heap memory errors, but failed in some cases (which we will discuss in Section 7.2.4).

MTSan detected most stack and global memory safety violations with a low false negative rate. Among all stack and global vulnerabilities, MTSan successfully detected 13 out of 15, while other binary sanitizers did not detect any. This shows that MTSan has a unique advantage in detecting memory safety violations that happen at stack and global objects. Furthermore, among the 12 detected vulnerabilities,

Table 3: Security evaluation results. Numbers indicate the number of PoCs. Yellow cells indicate that MTSan detected more PoCs than comparison targets, while blue cells indicate that MTSan is worse than at least one comparison target. SOF = stack-buffer-overflow, GOF = global-buffer-overflow, HOF = heap-buffer-overflow, UAF = heap-use-after-free.

Vulnerability ID	Type	Total	Valgrind	QASan	ASan-Retro write	MTSan			MTSan-no-rec		MTSan-no-rsv		MTSan-no-stg	
						Total	Critical	Non-cri.	Critical	Non-cri.	Critical	Non-cri.	Critical	Non-cri.
CVE-2017-14408	SOF	38	0	0	0	19	19	0	0	0	19	0	19	0
CVE-2017-14409	GOF	114	0	0	0	84	49	35	0	0	49	34	49	22
Bug #2065	GOF	400	0	0	0	400	0	400	0	0	0	400	0	400
CVE-2017-8786	HOF	469	469	469	469	469	469	0	469	0	469	0	469	0
CVE-2017-7245	SOF	646	0	0	0	248	248	0	0	0	248	0	248	0
CVE-2017-7246	SOF	627	0	0	0	262	262	0	0	0	262	0	262	0
Bug #2056	SOF	102	0	0	0	102	0	102	0	0	0	102	0	102
CVE-2017-9047	SOF	489	0	0	0	489	40	449	0	0	40	449	40	449
CVE-2017-8363	HOF	26	26	26	22	26	26	0	26	0	26	0	26	0
CVE-2017-8361	GOF	13	0	0	0	0	0	0	0	0	0	0	0	0
CVE-2017-8365	GOF	2	0	0	0	2	2	0	0	0	2	0	2	0
CVE-2016-10270	HOF	89	89	89	89	89	89	0	89	0	89	0	89	0
CVE-2016-10271	HOF	235	235	231	200	235	235	0	235	0	235	0	235	0
CVE-2009-2285	HOF	32	31	0	0	32	32	0	32	0	32	0	32	0
CVE-2013-4243	HOF	4	4	4	4	4	4	0	4	0	4	0	4	0
CVE-2015-8668	HOF	23	20	23	23	23	23	0	23	0	23	0	23	0
CVE-2017-12858	UAF	35	35	35	35	34	34	0	34	0	34	0	34	0
Ubuntu #1775776 [42, 43]	UAF	1	1	1	1	1	1	0	1	0	1	0	1	0
Ubuntu #1775776 [44]	HOF	1	0	1	1	1	1	0	1	0	1	0	1	0
CVE-2020-21676	SOF	10	0	0	0	0	0	0	0	0	0	0	0	0
CVE-2020-21675	SOF	20	0	0	0	8	8	0	0	0	8	0	8	0
CVE-2018-17294	SOF	2	0	0	0	2	0	2	0	0	0	2	0	2
CVE-2020-21050	SOF	16	0	0	0	16	10	6	0	0	10	6	10	6
Issue #73	HOF	12	12	12	12	12	12	0	12	0	12	0	12	0
CVE-2018-20004	SOF	10	0	0	0	8	8	0	0	0	8	0	8	0
CVE-2018-20005	UAF	19	19	19	19	19	19	0	19	0	19	0	19	0
CVE-2021-20294	SOF	5	0	0	0	4	4	0	0	0	4	0	4	0
Total	27	3440	941	910	875	2589	1595	994	945	0	1595	993	1595	981

MTSan successfully detected all PoCs for 8 of them. Compared to other binary sanitizers, MTSan has the lowest false negative rate.

Object recovery is necessary for stack and global BOFs. Comparing to MTSan-no-rec, MTSan achieved higher detection numbers for PoCs that exploit stack or global buffer overflow vulnerabilities, which means object recovery is critical for detecting these types of vulnerabilities. We also notice that without object recovery, MTSan-no-rec detected the same number of heap-based violations, which is expected.

Performance optimizations do not impact MTSan's detection capabilities. By comparing with MTSan-no-rsv and MTSan-no-stg, we conclude that our optimizations do not negatively impact the detection capability of MTSan. Moreover, two-stage fuzzing improves MTSan's detection capability: For example, according to Table A.8 in Appendix, MTSan-no-stg could not detect any violations for CVE-2017-8361 in the O3 version of `sndfile-convert`. This is because when the target bug was triggered, adjacent victim objects were never used during the same execution, thus non-critical violations did not happen.

Compiler optimizations has a limited effect on MTSan's effectiveness. As shown in Table 3 and Table A.8 (in Appendix), MTSan still keeps a good detection capability on O3 versions of programs. For three vulnerabilities (CVE-2017-14409, CVE-2017-8361, CVE-2018-17291), the detection result changed under O3. This is because different optimization levels usually lead to different object layouts and alignments. For example, MTSan detected fewer non-critical PoCs for CVE-2017-14409. We manually analyzed `MP3Gain` and found that the vulnerable global object, `ispow`, has eight more bytes for padding in O3. MTSan cannot detect buffer overflows that only clobber these padding bytes. Section 7.2.4 will analyze more false negative cases.

7.2.4 False Negative Analysis

In general, binary sanitizers are not able to detect all memory violations, which can lead to vulnerabilities spared during fuzzing. We analyzed the reasons and will discuss them regarding both design and implementation. We first discuss the false negatives of MTSan and then the others.

Low granularity of MTE. MTE provides a memory tag for every `0x10` aligned bytes. Due to this hardware limitation, overflows within the MTE's granularity cannot be detected. We examined FNs in the `Juliet` test suite and found that these overflows are mostly off-by-one vulnerabilities that do not overflow to the next 0x10-aligned memory address. Listing A.3 in the appendix shows such an example in CWE-122.

Compound objects. Compound objects are introduced for presenting objects which contain multiple neighbouring but fused objects. Neighbouring objects, of which the boundaries are not aligned to `0x10`, are merged as compound objects. However, MTSan cannot detect overflows within (compound) objects. This is the root cause of five FNs: CVE-2017-7245, CVE-2017-7246, Bug #2065, CVE-2017-14408 and CVE-2017-14409. For interested readers, we pro-

Table 4: Average execution per second of each configuration during three fuzzing campaigns. Greater numbers indicate better runtime performance. We calculate the percentages for the average number of executions of MTSan-no-⋆ by comparing them to MTSan (libMTE) (marked with ⋆), while calculating other percentages by comparing them to AFL++ Qemu.

Binary	AFL++ Qemu	QASan	ASan-Retrowrite	MTSan (analog)	MTSan (libMTE)	MTSan-no-rec	MTSan-no-rrr	MTSan-no-stg	MTSan-no-rsv
bc	56.3	34.67	115.54	323.8	94.1	80.49	100.31	97.46	83.45
bmp2tiff	8.38	21.5	156.1	245.336	169.6	224.73	164.05	150.94	148.13
fig2dev	213.47	224.51	170.91	183.816	101.76	225.05	134.31	134.9	107.76
gif2tiff	6.71	5.74	222.46	133.76	152.25	331.74	159.2	150.86	149.77
lou_translate	2.27	0.61	1.86	2.864	2.42	2.81	1.88	1.76	1.28
img2sixel	15.3	15.29	34.77	79.12	13.99	31.31	38.26	37.85	30.2
xml_read_memory_fuzzer	183.94	67.18	82.64	225.792	61.25	141.14	95.14	93.93	87.42
ziptool	134.28	61.68	174.14	353.944	111.18	243.98	111.59	124.72	155.67
mp3gain	23.97	9.42	162.41	134.688	80.46	169.5	95.86	95.47	161.75
mxmldoc	222.61	89.87	159.28	301.896	116.79	162.46	100.99	105.45	106.84
testmxml	180.92	151.75	177.47	193.352	115.35	194.91	106.2	106.57	136.34
pcretest	42.31	2.24	70.88	91.192	37.49	88.43	43.61	41.38	38.34
pcre2test	40.78	19.16	64.24	173.072	29.12	62.15	20.82	14.36	15.7
readelf	355.48	181.63	67.2	383.576	80.92	231.32	97.08	95.48	96.76
sndfile-convert	235.61	149.97	185.08	153.888	179.48	150.32	140.91	142.89	142.97
tiff2ps	307.7	15.94	191.48	373.832	214.89	237.65	151.66	126.6	132.46
tiffcp	249.37	38.67	236.66	307.2	214.42	222.49	159.04	164.51	143.1
tiffcrop	231.48	48.65	226.14	307.808	214.01	254.62	218.47	181.47	167.58
Average	139.49	63.25 (-54.66%)	138.85 (-0.46%)	220.50 (+58.07%)	110.53 (-20.77%)	169.73 (+53.56%)⋆	107.74 (-2.52%)⋆	103.70 (-6.18%)⋆	105.86 (-4.22%)⋆

vide a detailed analysis in the appendix (Section A.2).

Other limitations. We detail other limitations of MTSan in Section 8.

Among the comparison targets, Valgrind, QASan and ASan-Retrowrite are all location-based sanitizers and share similar mechanisms and pitfalls. Besides implementation issues, we summarize other reasons for FNs in three categories.

Limitation of location-based sanitizing scheme. QASan and Valgrind cannot insert redzones at stack and global region. PoCs that triggered memory errors in stack and global memory regions are all neglected. ASan-Retrowrite utilizes the stack canary as redzone. However, in our evaluation, this design did not provide benefits beyond the canary's own functionality. As an identity-based sanitizer, MTSan supports detecting stack and global memory violations, by recovering objects and checking accesses to them.

The pitfall of redzones. Redzones are not silver bullets. An out-of-bound memory access may go beyond the upper bound of a redzone and access another valid object, without being caught by the redzone. Besides, to detect UAF violations, a redzone-based sanitizer usually turns a freed chunk as a redzone; When this freed chunk is later reallocated, the sanitizer loses the redzone, which may result in FNs.

Limitation of inspection location. Lack of library support may bring false negatives to ASan-Retrowrite. For example, the PoCs for CVE-2009-2285 trigger memory safety violations in the library function `LZWDecodeCompat`. ASan-Retrowrite failed to detect them as the library code was not instrumented. However, MTSan benefits from MTE's hardware-assisted memory access checking. Objects allocated from binaries are still being sanitized even when passed to library functions.

7.3 Performance

We evaluated the runtime and memory overhead of MTSan on SPEC CPU 2017 [14]. In summary, MTSan has lower runtime and memory overhead than other sanitizers in comparison. Specifically, MTSan has 47.8% lower runtime overhead and 90.8% lower memory overhead than ASan-Retrowrite. Interested readers can find details and results of our experiments in the appendix (Section A.3).

7.4 Fuzzing Efficiency

7.4.1 Experiment setup

Environment. To assess the fuzzing efficiency of MTSan, we ran AFL++ [29] to fuzz the target programs, with existing binary sanitizers separately enabled. We used AFL++ 3.15a together with our patch for supporting non-critical reports. We used `libMTE` during the fuzzing evaluation, which allowed us to evaluate the complete workflow of MTSan. We also evaluated the analog mode to provide us with a reference of the performance of MTSan with hardware support. We have also implemented an `afl-gcc` style instrumentation for MTSan to provide the necessary coverage feedback support.

Following the common practice of recent fuzzing research, we collected input cases from actively maintained seed pools [45, 46] as the initial seeds. We also followed AFL++'s best practice guidance [47] and filtered out seeds that caused timeouts.

Configuration. We carefully set up comparison targets for the fuzzing evaluation. For QASan, we used qemuafl [48] shipped with AFL++ and enabled `AFL_USE_QASAN` for each fuzzing campaign. Valgrind is not designed for fuzzing and lacks the necessary features to work with AFL++ (e.g., sending signals when violations are detected). So we did not use Valgrind in the fuzzing evaluation.

To better understand the contribution of each component, we conducted an ablation study where we compared the performance of MTSan under different configurations: Disabling object recovery (*-no-rec*); disabling two-stage fuzzing (*-no-stg*); disabling RRR (*-no-rrr*); and reserving no tags for

special memory areas (*-no-rsv*).

7.4.2 Overall Results

Fuzzing speed. We run 3×24-hour trials per benchmark for each binary sanitizer selected. Table 4 shows the average number of executions with different sanitizers. MTSan with analog instructions yields the highest number of executions, which is 58.07% higher than AFL++ Qemu and approximately twice as many as MTSan with libMTE. ASan-Retrowrite also yields good results on fuzzing speed, which is only 0.46% lower than AFL++ Qemu. QASan has the worst performance: The average number of executions for QASan is 54.66% fewer than AFL++ Qemu.

We compare the results of different configurations of MTSan and draw the following conclusions. First, progressive object recovery (which is necessary for sanitizing stack and global objects) adds 53.56% runtime overhead to MTSan. Second, *-no-rsv* introduces 4.22% runtime overhead to MTSan, which means that reserved tag improves the fuzzing performance. Third, since MTSan-no-rrr yields similar execution numbers to MTSan, *RRR* only incurs runtime overhead of less than 3%. However, MTSan with *-no-stg* exhibits runtime overhead of 6.18%, which suggests that the more FPs may increase the runtime overhead that *RRR* introduces.

Bugs found. The immediate goal of binary fuzzing is to find bugs. Table 5 shows that MTSan performed the best and reported 20 bugs during fuzzing. Among the listed vulnerabilities, four of them were only detected during fuzzing with MTSan. Table 5 also shows that 10 vulnerabilities were triggered with at least one non-critical violation. However, due to the intrinsic randomness in fuzzing, this does not mean RRR was in effect in these cases. We need more analysis to show the impact of RRR, which will be detailed in Section 7.4.3. Note that the results for both MTSan-no-rsv and MTSan-no-stg show small decreases in bug counts. This indicates that our optimizations have positive effects on bug finding.

7.4.3 Internal Statistics

Finally, we measured the statistics of MTSan to understand its inner workings during our evaluation.

Progressive object recovery. First, we study the accuracy of progressive object recovery. We compiled all binaries with debug information enabled (by specifying `-g`) and used debug information as the ground truth, which provides variable boundaries and how each pointer was derived. We classify all objects that MTSan identified into five categories:

- *full-match*: An identified object matches both the boundary and all pointers of a ground-truth variable.
- *merged-match*: an identified object shares boundaries with at least one adjacent object, and the merged object matches the boundary and pointers of a merged ground-truth variable (which is merged from multiple ground-truth variables).

Table 5: Bugs found during the fuzzing evaluation.

Vulnerability ID	QASan	Asan-Retro.	MTSan		MTSan-no-rec	MTSan-no-rrr	MTSan-no-rsv	MTSan-no-stg
			Cri.	Non-C.				
CVE-2017-14408			✓			✓	✓	✓
CVE-2017-14409	✓	✓	✓	✓	✓	✓	✓	✓
Bug #2065 [49]	✓	✓	✓		✓	✓	✓	✓
CVE-2017-9047			✓	✓				
CVE-2017-8361	✓				✓			
CVE-2016-10270	✓		✓		✓	✓	✓	✓
CVE-2016-10271	✓		✓		✓	✓	✓	✓
CVE-2013-4243	✓	✓	✓		✓	✓	✓	✓
CVE-2015-8668	✓	✓	✓		✓	✓	✓	✓
CVE-2017-12858	✓		✓		✓	✓	✓	
CVE-2020-21675	✓	✓	✓			✓	✓	✓
CVE-2020-21050	✓	✓	✓		✓	✓	✓	✓
CVE-2018-20005					✓			
CVE-2018-20592⋆	✓	✓	✓			✓	✓	✓
Issue #237 [50]⋆	✓	✓	✓		✓	✓		✓
Issue #5 [51]⋆	✓	✓					✓	
CVE-2016-5321⋆	✓		✓	✓	✓	✓	✓	✓
CVE-2017-7244⋆		✓	✓	✓			✓	✓
CVE-2016-5102⋆	✓	✓	✓	✓	✓	✓	✓	✓
CVE-2020-21533⋆	✓	✓	✓	✓	✓	✓	✓	✓
CVE-2020-21534⋆	✓	✓		✓	✓			
CVE-2020-21676⋆		✓	✓	✓	✓	✓	✓	✓
CVE-2017-14410⋆			✓	✓			✓	✓
Issue #40 [52]⋆			✓	✓			✓	✓
Total	**17**	**14**	**20**	**10**	**16**	**16**	**19**	**18**

⋆: a vulnerability not initially included in our dataset, but triggered during the fuzzing evaluation.

- *weak-match*: an identified object only matches all pointers (but not the boundary) of a ground-truth variable.
- *sub-match*: an identified object is a sub-object of a ground-truth variable.
- *bad-match*: an identified object does not match any ground-truth variables.

As shown in Table 6, *merged* object exists in all programs, which illustrates the necessity of the design of compound object. However, compound objects may cause FNs, and improving the granularity of memory tagging will reduce the number of FNs. The average percentage of objects with deterministic boundaries (see *def-bound*) is 36.42%, and improved static analyses will increase this percentage. The result also shows that *sub-match*, *weak-match*, and *bad-match* objects represent a relatively small percentage of all objects (5.08% on average). This means that RRR was not frequently invoked during fuzzing.

While MTSan is *fault-tolerant* (it can detect memory safety violations even when results from the analysis phases are not fully accurate), we examined the error cases to better understand their causes. *Weak-match* cases are mainly caused by pointer arithmetic inaccuracies in VSA. The main reason behind *sub-match* and *bad-match* cases is that MTSan failed to identify different pointers that are created for the same object. For example, variable `filter` in function `is_format_lzma` actually consists of two fields, `id` and `options` (See Listing A.4). The compiler created two pointers (`SP+0x48` and `SP+0x50`) for these two fields. MTSan recognized the two fields as two *sub-match* objects and assigned two different tags.

Convergence of object recovery and false positives. We examined (1) *updating* of object boundaries and (2) *merging* of objects (which corresponds to non-critical false positives)

Table 6: Internal statistics on progressive object recovery in MTSan. The percentages refer to the ratio of all objects of each category while absolute numbers indicate the exact amount of objects in each category.

Binary	Total Objects	Full-match	Merged-match	Sub-match	Weak-match	Bad-match	Def-bound
mp3gain	357	36.41%	53.50%	15	14	7	31.65%
pcre2test	1314	65.45%	26.86%	57	35	9	20.09%
pcretest	1193	29.25%	62.36%	39	40	21	23.97%
libxml2_read	6695	50.31%	48.05%	44	49	17	39.34%
sndfile-convert	2602	42.24%	51.65%	73	67	19	44.16%
tiffcp	1709	45.17%	52.31%	19	21	3	39.26%
tiffcrop	2065	40.87%	55.59%	23	45	6	34.87%
tiff2ps	106	35.85%	64.15%	0	0	0	34.91%
gif2tiff	1605	46.23%	50.78%	20	24	4	39.88%
bmp2tiff	1579	45.92%	51.36%	19	21	3	39.58%
ziptool	864	53.01%	41.20%	24	14	12	40.51%
bc	279	55.56%	41.94%	3	7	3	48.03%
fig2dev	2575	40.43%	49.24%	124	105	37	35.38%
lou_translate	1010	35.05%	61.78%	10	20	5	26.73%
img2sixel	1051	38.25%	46.72%	101	39	18	36.44%
mxmldoc	369	47.15%	48.51%	4	11	1	38.75%
testmxml	172	58.72%	36.63%	4	3	1	49.42%
readelf	2793	39.03%	53.81%	111	46	43	32.55%
Average	1574	44.72%	49.80%	38	31	11	36.42%

to see if they converged over time as more state space of each program was explored during fuzzing.

Because the initial corpus may affect the convergence process, we prepared three types of corpora:

- *full*: including all test cases initially collected.
- *mini*: only including test cases shipped by each project [1].
- *zero*: only including a file with the string "aaaaaaaaaa".

Interested readers may refer to Figure A.8 in the appendix for the convergence diagram of each fuzzed program. We briefly present our findings below. First, vast majority of *merges* and *updates* occur within the first hour, while a few sporadically happen after. Second, more *merges* and *updates* occur when better test cases are used. Finally, the numbers of *updates* and *merges* increase over time when fuzzing with the *mini* and *zero* corpora, but rarely exceeded the numbers when *full* corpora ware used.

We manually analyzed false positive cases and found that they correspond to *sub-match* and *bad-match* objects. For example, for the code snippet shown in Listing A.4, MTSan assigned different tags to fields `id` and `options`. As fuzzing proceeded, `is_format_lzma` invoked function `lzma_properties_decode` and passed as an argument the pointer of `filter`, which will be used for accessing the entire object. Then, `lzma_properties_decode` accessed `options` using the pointer that was tagged by `id`, which raised a false positive.

The effectiveness of RRR. We evaluated how RRR helped escalate non-critical violations into critical violations. Time-to-discovery (TTD) (of bugs) is a metric which directly reflects fuzzing effectiveness. To this end, we first recorded the TTD of vulnerabilities that were triggered during the fuzzing evaluation, including both non-critical and critical errors. However, chronological order does not indicate any causal relationship between events. So we tagged all queued test cases during RRR and recorded when they were selected for further mutation. This way we could ensure whether a critical violation event was related to RRR or not.

[1] If a project does not provide any test cases, we instead used the smallest test input in corpus.

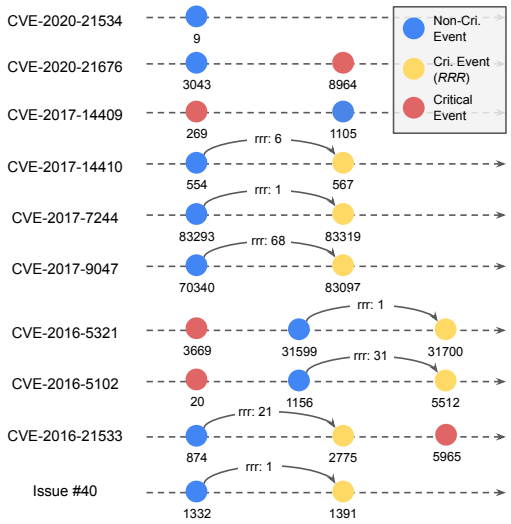


Figure 4: Time-to-Discovery of vulnerabilities (in seconds) detected during the fuzzing evaluation. Vulnerabilities that were never detected with a non-critical violation are excluded here. The numbers on the cutter indicates how many times an RRR case was selected. When a vulnerability was triggered in multiple trials, we first select non-critical events based on the median of TTD, and then their related critical events.

Figure 4 shows that RRR escalated seven non-critical violations to critical violations. Four critical violations were only derived during RRR, and one critical violation was detected sooner because of RRR. However, in three cases, critical violations were triggered before non-critical violations, which means that RRR does not always decrease TTD.

8 Discussions and Limitations

Limited number of tags. MTE allows 4-bit tags to be assigned to each memory allocation and address. Due to the length of the tag, the probability of tag collision is 6.25%. Although MTSan avoids re-using tags in neighbouring objects, the possibility of different objects sharing the same tag still exists. However, as fuzzing is a highly repetitive procedure, a vulnerability is not likely triggered only once during the fuzzing. Overall, the longer the fuzzing time, the less likely that a vulnerability missed due to tag collision.

Sub-object overflow. MTSan cannot detect overflows in sub-objects. However, supporting sub-objects is still an open research problem for both source-level and binary sanitizers. Similarly, MTSan does not support detecting memory violations in objects within a single heap chunk (e.g., an object array allocated with `malloc(N*sizeof(object))`).

Coverage limit. Recall that fuzzing is a process of progres-

sive exploration of programs. The coverage limit may cause an incomplete recovery of objects. However, as fuzzing gets longer, the coverage limit could be improved.

Custom memory allocators. The prototype of MTSan hooks `malloc`, `calloc`, `realloc`, `reallocarray` and `mmap`. However, certain programs may use custom memory allocators (CMA), and currently MTSan infers CMA-allocated objects as belonging to a single heap object. Existing researches [53, 54] about heap abstractions and modeling may help MTSan to support more binaries.

Padding bytes. Recovering the accurate bounds for objects remains an open challenge. Currently, MTSan does not support detecting overflows at padding bytes.

9 Related Work

9.1 Binary Sanitizers

Valgrind [5] is a well-known dynamic binary instrumentation (DBI) framework that includes a sanitizer called "memcheck". Memcheck [55] is capable of detecting spatial and temporal violations for heap object, but does not support memory violation detection for stack or global objects. Furthermore, memcheck is costly, costing anywhere from 2× to 300× overhead, rendering it unsuitable for usage with frequent executions during testing, particularly fuzzing, where increased throughput directly correlates to higher bug-finding probability. In addition to memcheck, Undangle [15] also utilizes DBI architecture to implement a binary sanitizer. However, Undangle only targets UAF bugs. Dr. Memory [16] is a memory monitoring tool capable of identifying memory-related programming errors, such as double frees, memory leaks and accesses to invalid memory including unaddressable or freed memory. However, Dr. Memory also suffers from a 20.4× of performance overhead. QASan [7] is another sanitizer that has been proposed recently. It is solely concerned about memory violations in heap objects, though. Furthermore, because QASan is designed to work with Qemu, any execution must account for the performance overhead of Qemu's TCG as well as sanitizing. ASan-Retrowrite [6] is a sanitizing binaries implementation built on top of Retrowrite, a state-of-the-art static binary rewriter for COTS binaries. The design goal of ASan-Retrowrite is to develop a binary version of Address Sanitizer. ASan-Retrowrite can achieve a better performance through binary rewriting. However, in order to scale to real-world software, ASan-Retrowrite sacrifices some precision, resulting in just a fraction of vulnerability types being sanitized.

9.2 Hardware Expansions

HWAsan [56] tags each pointer with a random value that is associated with a specific object and stores memory tags in a shadow memory. MemtagSanitizer [57] uses a similar technique, but using MTE's tag storage (shadow). However they work when the source code is available. NO-FAT [58] designed a novel architecture that encodes the object size and the base address in the pointer value itself for spatial safety while also tagging the upper 16-bits of data pointers on 64-bit platforms with a random value for temporal safety. IN-FAT [59] indexes metadata with a 16-bit pointer tag and ensures spatial memory safety at the sub-object granularity. CHERI [60] (Capacity Hardware Enhanced RISC Instructions) use 128-bit fat pointers/capabilities to restrict the range of memory that each pointer is permitted to access. Work [61] has been proposed recently takes advantage of CHERI's hardware capability to guarantee total spatial safety. These solutions provide solid security guarantees as well as excellent performance, however they are not yet accessible on binaries.

9.3 Variable and Type Recovery

Angr [23] is a state-of-the-art open-source binary analysis infrastructure, which leverages an advanced concolic execution engine for variable recovery. IDA Pro [24] is one of the most widely-used commercial decompilation toolkits. Ghidra [62] is another binary decompiler by NSA, which leverages a register-based data-flow analysis for recovery. Osprey [63] proposed a novel probabilistic technique for variable and structure recovery and achieved a precision rate of 90.18%. However, they all suffer from the issue of accuracy. This not only cause false negatives and false positives, but also introduce fuzzing-blockers, which prevents them from being used in binary sanitizing. Rewards [21] and Howard [22] are based on dynamic analysis. Rewards employed data flow tracking, and Howard improves rewards using heuristics to resolve conflicts. But they cannot be directly used to binary fuzzing since benign and malicious inputs are intermingled during fuzzing.

10 Conclusion

Fuzzing is a popular solution for finding vulnerabilities, but has many limitations on sanitizing binaries. One obvious reason was the lack of memory sanitizers for binaries. We present in this paper a novel solution, MTSan, that addresses these issues. It applies a novel progressive object recovery scheme to infer object properties in binaries, including stack and global objects, and uses ARM MTE to perform memory-tagging-based sanitizing and detect spatial and temporal memory safety violations during fuzzing. Our evaluation shows that MTSan is both effective and efficient, and can greatly improve binary fuzzing.

Acknowledgements

We would like to sincerely thank all the anonymous reviewers and our shepherd for their valuable feedback that greatly helped us to improve this paper. This work was supported in part by the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (61972224), and Beijing National Research Center for Information Science

and Technology (BNRist) under Grant BNR2022RC01006. This work was also partially funded by the Defense Advanced Research Projects Agency (DARPA) under Grant No. FA875019C0003 and N6600120C4020.

References

[1] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference*, pp. 309–318.

[2] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009, p. 62–71.

[3] "Undefinedbehaviorsanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[4] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in C++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization)*, pp. 46–55.

[5] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[6] 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)*. IEEE, 2020, pp. 1497–1511.

[7] A. Fioraldi, D. C. DElia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *2020 IEEE Secure Development (SecDev)*. IEEE, 2020, pp. 23–30.

[8] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," *arXiv preprint arXiv:1802.09517*, 2018.

[9] S. Newsroom, "Samsung introduces game changing exynos 2200 processor with xclipse gpu powered by amd rdna 2 architecture." https://news.samsung.com/global/samsung-introduces-game-changing-exynos-2200-processor-with-xclipse-gpu-powered-by-amd-rdna-2-architecture, 2022.

[10] "Application data integrity (ADI)," https://www.kernel.org/doc/Documentation/sparc/adi.rst.

[11] S. Bannister., "Memory tagging extension: Enhancing memory safety through architecture." https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety, 2019.

[12] "Intel® architecture instruction set extensions and future features," https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html.

[13] "Intel preparing linear address masking support (LAM)," https://www.phoronix.com/scan.php?page=news_item&px=Intel-LAM-Glibc.

[14] "Standard performance evaluation corporation, spec cpu 2017," https://www.spec.org/cpu2017/.

[15] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 133–143.

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

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

[18] "Tagged memory and minion cores in the lowrisc soc," https://lwn.net/Articles/627095/.

[19] A. Ltd, "Armv8.5-a memory tagging extension. whitepaper," 2019.

[20] "Virtual address tagging," https://developer.arm.com/documentation/den0024/a/ch12s05s01.

[21] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.

[22] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *NDSS*, 2011.

[23] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen *et al.*, "SoK:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157.

[24] "IDA Pro, a powerful disassembler and a versatile debugger," https://www.hex-rays.com/ida-pro/.

[25] N. A. Quynh, "Capstone: Next-gen disassembly framework," *Black Hat USA*, vol. 5, no. 2, pp. 3–8, 2014.

[26] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[27] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[28] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 659–676.

[29] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[30] 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 21)*, 2021, pp. 479–494.

[31] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *31th USENIX Security Symposium (USENIX Security 22)*, 2022.

[32] Z. Jiang, S. Gan, A. Herrera, F. Toffalini, L. Romerio, C. Tang, M. Egele, C. Zhang, and M. Payer, "Evocatio: Conjuring bug capabilities from a single poc," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1599–1613.

[33] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu, "Fot: A versatile, configurable, extensible fuzzing framework," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 867–870.

[34] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 919–936.

[35] "Juliet test suite for C/C++," https://samate.nist.gov/SRD/testsuites/juliet/Juliet_Test_Suite_v1.3_for_C_Cpp.zip, 2017.

[36] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakc," in *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, vol. 22, 2022, pp. 1–17.

[37] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 177–194.

[38] "Memory tagging extension (MTE) in aarch64 linux," https://www.kernel.org/doc/html/latest/arm64/memory-tagging-extension.html.

[39] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. Califor-nia, USA, 2005, pp. 10–5555.

[40] "Fixed virtual platforms," https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms, 2020.

[41] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun, "Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services," *IEEE Micro*, vol. 41, no. 5, pp. 67–75, 2021.

[42] "Launchpad, ubuntu, gnu bc crashes on some inputs," https://bugs.launchpad.net/ubuntu/+source/bc/+bug/1775776.

[43] "Github, fot-the-fuzzer/pocs/bc," https://github.com/fot-the-fuzzer/pocs/blob/master/bc/1.07.1/crashes/hbo_storage.c:274_1.gdb.txt.

[44] "Github, fot-the-fuzzer/pocs/bc," https://github.com/fot-the-fuzzer/pocs/blob/master/bc/1.07.1/crashes/hbo_execute:150_1.gdb.txt.

[45] "Github, mozillasecurity/fuzzdata," https://github.com/MozillaSecurity/fuzzdata.

[46] "Github, strongcourage/fuzzing-corpus," https://github.com/strongcourage/fuzzing-corpus.

[47] "Github, aflplusplus, fuzzing with afl++," https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing_in_depth.md#a-collecting-inputs.

[48] "Github, aflplusplus/qemuafl," https://github.com/AFLplusplus/qemuafl.

[49] "[pcre-dev] [bug 2065] new: global buffer overflow write in decode_modifiers (pcre2test.c)," https://www.mail-archive.com/pcre-dev@exim.org/msg04890.html.

[50] "Github, mxml, issue #237," https://github.com/michaelrsweet/mxml/issues/237.

[51] "Github, codedoc, issue #5," https://github.com/michaelrsweet/codedoc/issues/5.

[52] "Sourceforge, mp3gain, bugs, two crash bugs on mp3gain," https://sourceforge.net/p/mp3gain/bugs/40/.

[53] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *ACM Comput. Surv.*, vol. 49, no. 2, jun 2016. [Online]. Available: https://doi.org/10.1145/2931098

[54] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis, "Dynpta: Combining static and dynamic analysis for practical selective data protection," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1919–1937.

[55] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.

[56] "Hardware-assisted addresssanitizer design documentation," https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[57] "Memtagsanitizer," https://llvm.org/docs/MemTagSanitizer.html.

[58] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-fat: Architectural support for low overhead memory safety checks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 916–929.

[59] S. Xu, W. Huang, and D. Lie, "In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 224–240.

[60] R. N. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson *et al.*, "Capability hardware enhanced risc instructions: CHERI instruction-set architecture (version 5)," University of Cambridge, Computer Laboratory, Tech. Rep., 2016.

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

[62] "Ghidra, a software reverse engineering (sre) suite of tools developed by nsa's research directorate in support of the cybersecurity mission," https://ghidra-sre.org/.

[63] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 813–832.

[64] G. J. Duck and R. H. 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*, 2018, pp. 181–195.

[65] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018, pp. 81–116.

[66] "Github, mozillasecurity/fuzzdata," https://github.com/MozillaSecurity/fuzzdata.

[67] "Github, strongcourage/fuzzing-corpus," https://github.com/strongcourage/fuzzing-corpus.

[68] K. Serebryany, "Arm memory tagging extension and how it improves C/C++ memory safety." *login Usenix Mag.*, vol. 44, no. 2, 2019.

[69] "[pcre-dev] [bug 2056] new: stack-based buffer overflow in read_capture_name32 (pcretest.c)," https://lists.exim.org/lurker/message/20170224.153825.15d70558.da.html.

[70] "Dynamorio, dynamic instrumentation tool platform," https://dynamorio.org.

[71] "Armv8.3 pointer authentication," https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf.

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

[73] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1153–1166.

[74] P. Nasahl, R. Schilling, M. Werner, J. Hoogerbrugge, M. Medwed, and S. Mangard, "Cryptag: thwarting physical and logical memory vulnerabilities using cryptographically colored memory," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 200–212.

[75] "Github, libsixel, issue #73," https://github.com/saitoha/libsixel/issues/73.

A Appendix

A.1 Running Example: CVE-2017-9047

CVE-2017-9047 is a vulnerability located in `libxml2`, a popular library for parsing XML input. Listing A.1 shows the vulnerable function `xmlSnprintfElementContent`, which recursively dumps data from `content` to a char array `buf` of length `size`. Initially, `len` is set to `strlen(buf)`. Under certain conditions, `content->prefix` is appended to `buf` (Line 11), and `content->name` is also appended to `buf` (Line 19). However, the check at Line 14 used the initial value of buffer length (`len`) instead of the updated buffer length, leading to a buffer overflow. `xmlSnprintfElementContent` is invoked at Line 29 which

Listing A.1: Motivating Example: CVE-2017-9047.

```
void xmlSnprintfElementContent(char *buf, int size,
    xmlElementContentPtr content, int englob) {
    /* ... */
    len = strlen(buf);
        /* ... */
        if (content->prefix != NULL) {
            if (size - len < xmlStrlen(content->prefix) + 10) {
                strcat(buf, " ...");
                return;
            }
            strcat(buf, (char *) content->prefix);
            strcat(buf, ":");
        }
        if (size - len < xmlStrlen(content->name) + 10) {
            strcat(buf, " ...");
            return;
        }
        if (content->name != NULL)
            strcat(buf, (char *) content->name);
        /* ... */
}
int xmlValidateElementContent(xmlValidCtxtPtr ctxt, xmlNodePtr
    child, xmlElementPtr elemDecl, int warn, xmlNodePtr parent){
    /* ... */
    if (ctxt != NULL) {
        char expr[5000];  // vulnerable buffer
        char list[5000];  // victim buffer
        expr[0] = 0;
        xmlSnprintfElementContent(&expr[0], 5000, cont, 1);
    /* ... */
}
```

sets the argument `buf` to a stack buffer. Existing binary sanitizers cannot detect stack buffer overflow. Worse, because the victim buffer `list` residing after the overflowed buffer is of length 5000, this buffer overflow does not cause crashes until it overflows at least 5000 bytes, which can be difficult to achieve during fuzzing.

When analyzing the binary code of `libxml2`, the sizes (or boundaries) for char arrays `expr` and `list` are unavailable because all type information has been discarded during compiling. Hence, MTSan must infer object boundaries in stack and global regions during runtime in a manner that is similar to existing work about dynamic type inference in binary code [22]. A unique challenge in MTSan is that during fuzzing, benign and bug-triggering input co-exist, which may cause conflicts in inferred object boundaries. Our insight is that *conflicts among inferred object boundaries—caused by inferencing from both benign and bug-triggering input—are indicators for memory errors*.

We discuss how MTSan resolves conflicts and reports memory errors in two scenarios: (1) MTSan spots the benign input first during fuzzing, and (2) MTSan spots the bug-triggering input first during fuzzing.

MTSan spots the benign input first. In this case, some or all bytes in char arrays `expr` and `list` are updated during executions. MTSan infers object boundaries by observing how `expr` and `list` are accessed. Because they are always accessed via different pointers, MTSan recognizes them as distinct objects on the stack. Then when an overflow-triggering input arrives during fuzzing, `list` will be accessed using a pointer that is derived from the address of `expr`. In this case,

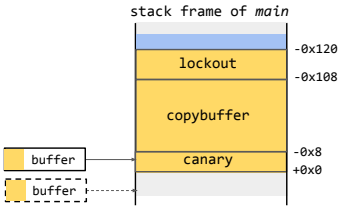


Figure A.5: Stack memory layout of CVE-2017-7245. The colors represent the memory and pointer with different tags.

Listing A.2: Code snippets of CVE-2017-7245 and 7246.

```
int main(int argc, char **argv){
    /* ... */
    char copybuffer[256];
    PCRE_COPY_SUBSTRING(rc, bptr, use_offsets, count, i,
        copybuffer, sizeof(copybuffer));
    /* ... */
}
int pcre32_copy_substring(PCRE_SPTR32 subject, int *ovector,
  int stringcount, int stringnumber, PCRE_UCHAR32 *buffer,
  int size){
    /* ... */
    yield = ovector[stringnumber+1] - ovector[stringnumber];
    /* ... */
    memcpy(buffer, subject + ovector[stringnumber], \
        IN_UCHARS(yield));    // CVE-2017-7246
    buffer[yield] = 0;        // CVE-2017-7245
    /* ... */
}
```

MTSan immediately spots a conflict with the previously inferred boundary between `expr` and `list`. This conflict indicates a buffer overflow vulnerability.

MTSan spots the bug-triggering input first. In this case, MTSan will infer boundaries for `expr` and `list` like in the prior case. The overflow-triggering input will cause MTSan to make an incorrect inference: In MTSan's eyes, `expr` is larger than 5,000 bytes and will overlap with `list`. This is temporary, because as soon as MTSan encounters benign input (that executes the same path), it will infer the boundary between `expr` and `list` once more, at which time a conflict will arise. Again, this conflict indicates a buffer overflow vulnerability.

A.2 FN Analysis of CVE-2017-7245 and 7246

We use CVE-2017-7245 and 7246 to further explain this case. In CVE-2017-7245, the vulnerable object is `copybuffer` in the `main` function. This object occupies `SP+0x2448` to `SP+0x2548`. Its adjacent objects are `lockout` and `canary`. As shown in Figure A.5, `lockout`, `copybuffer` and `canary` are merged into a compound object because of their are not aligned to `0x10`. The out-of-bound (OOB) access happens at `pcre32_copy_substring`, line 16 in Listing A.2. When the OOB access happens at `canary`, MTSan cannot detect the violation (which will be later detected by `_stack_chk_fail` in Glibc). When the OOB access overwrites a higher address outside the current stack frame, MTSan will detect this critical violation. CVE-2017-7246 shares the same vulnerable object with CVE-2017-7245 and the above analysis still applies.

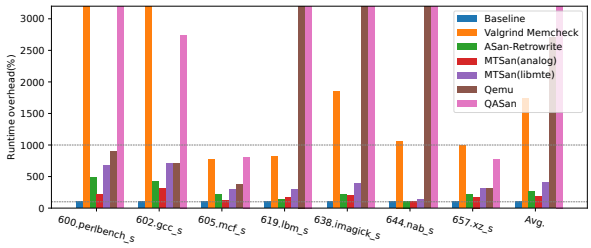


Figure A.6: Runtime overhead of MTSan on SPEC CPU 2017 C benchmarks, compared with ASan-Retrowrite, QASan, and Valgrind.

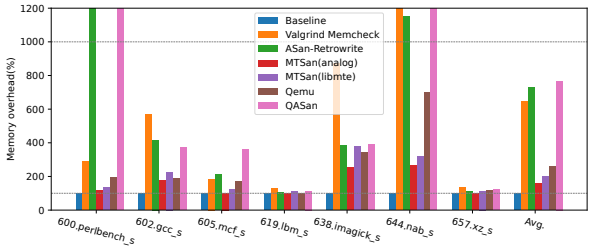


Figure A.7: Memory overhead of MTSan on SPEC CPU 2017 C benchmarks, compared with ASan-Retrowrite, QASan, and Valgrind.

A.3 Performance Evaluation

Evaluation setup. We used C benchmarks in SPEC CPU 2017 [14] to evaluate the performance overheads of MTSan. Evaluating performance overhead in the absence of available hardware is difficult. However, we employed instruction analog to provide us with a reference for worst-case performance, and implemented libMTE to evaluate the performance with pure software simulation.

It is worth noting that we were unable to get every benchmark program to be executed successfully on all binary sanitizers even with our best efforts. Valgrind and ASan-Retrowrite failed to run a complete execution on `625.x264_s`. C++ benchmarks are also excluded since Retrowrite and MTSan do not officially support them. Finally we selected all C benchmarks (except `625.x264_s`) in SPEC CPU 2017.

Runtime overhead. Figure A.6 shows that the average runtime overheads for MTSan is 1.82×, which is the lowest among the binary sanitizers. Among them, Valgrind and QASan have an average overhead of 17.4× and 35.5×. ASan-Retrowrite has a runtime overhead of 2.57×. We also evaluated the runtime overhead of MTSan with libMTE. The runtime overhead of MTSan (libMTE) is 4.01×, which is slightly higher than MTSan and ASan-Retrowrite.

Note that we evaluated all sanitizers using standalone executions of benchmark programs, which means the overhead of Qemu was included in the overall overhead for QASan. According to Figure A.6, the runtime overhead of Qemu is 26.7×. This may be unfair to QASan because it was specially designed to work with AFL's Qemu-mode. Its runtime overhead may be amortized from the fork server mechanism by sharing the TCG cache across multiple runs. For a fair comparison, we also evaluated the binary fuzzing performance and report the results in Section 7.4.

Memory overhead. As is shown in Figure A.7, the extra memory consumption of MTSan is 1.58×, which is significantly lower than that of Valgrind (6.45×), QASan (7.67×) and ASan-Retrowrite (7.30×). The memory overhead of MTSan (libMTE) is 2.1×, which is slightly higher than MTSan, but still lower than all comparison targets.

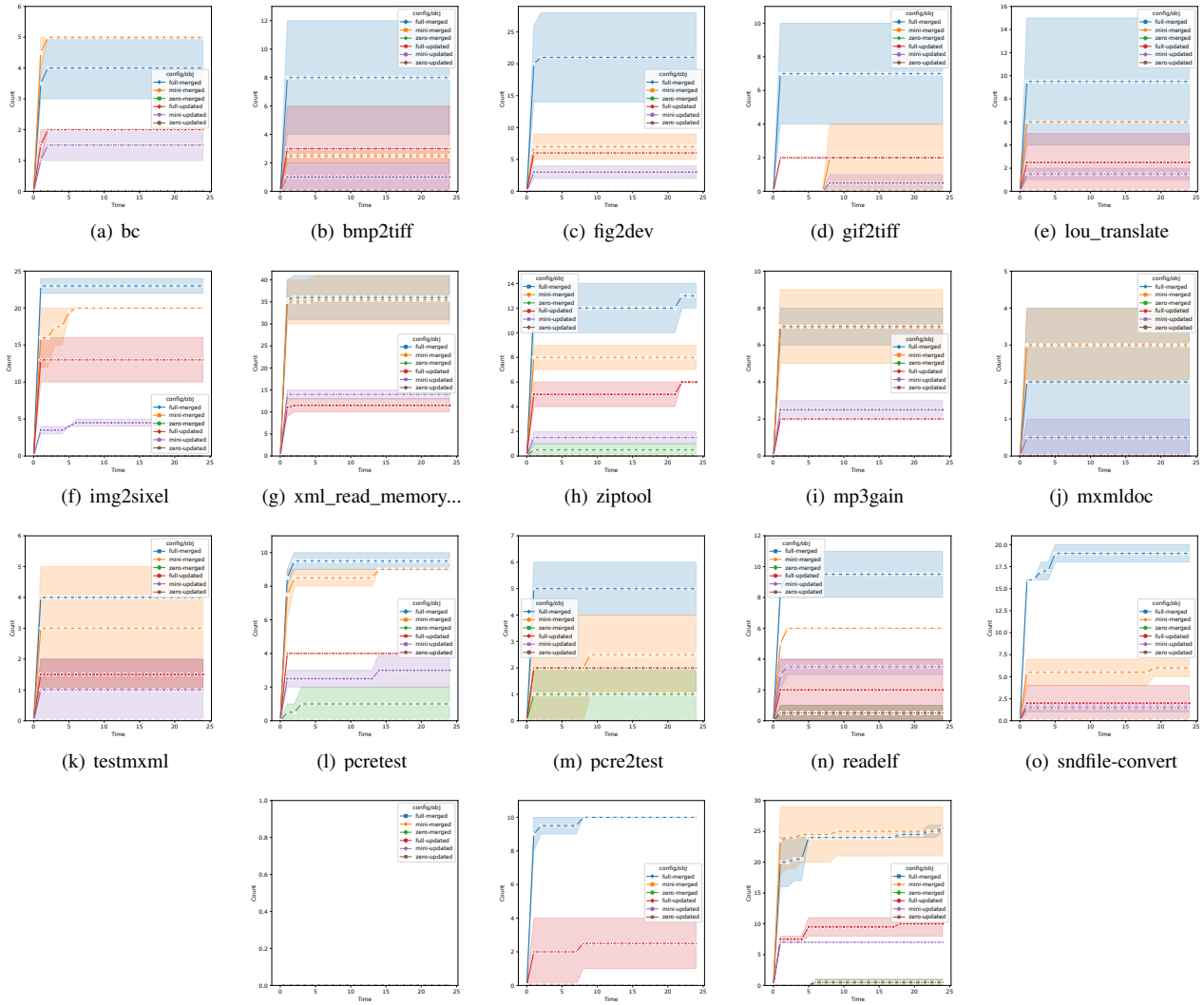
Summary. Overall, MTSan has lower runtime overhead and memory overhead than the comparison targets. Specifically, MTSan has reduced 47.8% of runtime overhead and 90.8% of memory overhead than ASan-Retrowrite.

Table A.7: Vulnerabilities and programs used in our evaluation.

Vulnerability ID	Project	Version	Harness Program	Command
CVE-2017-14408	mp3gain	1.5.2-rc2	mp3gain	@@
CVE-2017-14409	mp3gain	1.5.2-rc2	mp3gain	@@
Bug #2065 [49]	pcre2	10.22	pcre2test	-d -i -8 @@
CVE-2017-8786	pcre2	10.22	pcre2test	-d -i -32 @@
CVE-2017-7245	pcre	8.40	pcretest	-32 -d @@
CVE-2017-7246	pcre	8.40	pcretest	-32 -d @@
Bug #2056 [69]	pcre	8.40	pcretest	-32 -d @@
CVE-2017-9047	libxml2	ec6e3e	xml_read_memory_fuzzer	@@
CVE-2017-8363	libsndfile	c2be6f	sndfile-convert	@@ $TMP.wav
CVE-2017-8361	libsndfile	c2be6f	sndfile-convert	@@ $TMP.wav
CVE-2017-8365	libsndfile	c2be6f	sndfile-convert	@@ $TMP.wav
CVE-2016-10270	libtiff	4.0.1	tiffcp	-i @@ /dev/null
CVE-2016-10271	libtiff	4.0.1	tiffcrop	-i @@ /dev/null
CVE-2009-2285	libtiff	3.8.2	tiff2ps	@@ -O /dev/null
CVE-2013-4243	libtiff	4.0.1	gif2tiff	@@ /dev/null
CVE-2015-8668	libtiff	4.0.1	bmp2tiff	@@ /dev/null
CVE-2017-12858	libzip	1.2.0	ziptool	@@ cat ./index
Ubuntu #1775776 [42,43]	gnu-bc	1.07.1	bc	@@
Ubuntu #1775776 [44]	gnu-bc	1.07.1	bc	@@
CVE-2020-21676	fig2dev	3.2.7b	fig2dev	-L pstricks @@
CVE-2020-21675	fig2dev	3.2.7b	fig2dev	-L ptk @@
CVE-2018-17294	liblouis	81fe3c	lou_translate	./zhcn-g2.ctb –
CVE-2020-21050	libsixel	2df643	img2sixel	@@ -o /dev/null
Issue #73 [75]	libsixel	2df643	img2sixel	@@ -o /dev/null
CVE-2018-20004	mxml	2.12	testmxml	@@ /dev/null
CVE-2018-20005	mxml	53c75b	mxmldoc	@@
CVE-2021-20294	binutils	c56374	readelf	–dyn-syms @@

Table A.8: Security evaluation results of MTSan on O3 version of programs. Numbers indicate the number of PoCs.

Vulnerability ID	MTSan (O3)			MTSan-no-rec (O3)		MTSan-no-rsv (O3)		MTSan-no-stg (O3)	
	Total	Critical	Non-cri.	Critical	Non-cri.	Critical	Non-cri.	Critical	Non-cri.
CVE-2017-14408	19	19	0	0	0	19	0	19	0
CVE-2017-14409	68	49	19	0	0	49	18	49	14
Bug #2065	400	0	400	0	0	0	400	0	400
CVE-2017-8786	469	469	0	469	0	469	0	469	0
CVE-2017-7245	248	248	0	0	0	248	0	248	0
CVE-2017-7246	262	262	0	0	0	262	0	262	0
Bug #2056	102	0	102	0	0	0	102	0	102
CVE-2017-9047	489	40	449	0	0	40	449	40	449
CVE-2017-8363	26	26	0	26	0	26	0	26	0
CVE-2017-8361	13	0	13	0	0	0	13	0	0
CVE-2017-8365	2	2	0	0	0	2	0	2	0
CVE-2016-10270	89	89	0	89	0	89	0	89	0
CVE-2016-10271	235	235	0	235	0	235	0	235	0
CVE-2009-2285	32	32	0	32	0	32	0	32	0
CVE-2013-4243	4	4	0	4	0	4	0	4	0
CVE-2015-8668	23	23	0	23	0	23	0	23	0
CVE-2017-12858	34	34	0	34	0	34	0	34	0
Ubuntu #1775776	1	1	0	1	0	1	0	1	0
Ubuntu #1775776	1	1	0	1	0	1	0	1	0
CVE-2020-21676	0	0	0	0	0	0	0	0	0
CVE-2020-21675	8	8	0	0	0	8	0	8	0
CVE-2018-17294	0	0	0	0	0	0	0	0	0
CVE-2020-21050	16	10	6	0	0	10	6	10	6
Issue #73	12	12	0	12	0	12	0	12	0
CVE-2018-20004	8	8	0	0	0	8	0	8	0
CVE-2018-20005	19	19	0	19	0	19	0	19	0
CVE-2021-20294	4	4	0	0	0	4	0	4	0
Total	2584	1595	989	945	0	1595	988	1595	971



(a) bc (b) bmp2tiff (c) fig2dev (d) gif2tiff (e) lou_translate

(f) img2sixel (g) xml_read_memory... (h) ziptool (i) mp3gain (j) mxmldoc

(k) testmxml (l) pcretest (m) pcre2test (n) readelf (o) sndfile-convert

(p) tiff2ps (q) tiffcp (r) tiffcrop

Figure A.8: Convergence of object recovery and false positives during fuzzing evaluation. The x-axis shows the time (in hours) since the fuzzer launched, and the y-axis shows the count.

Listing A.3: A FN case in the `Juliet` test suite (CWE-122).

```
#define SRC_STRING "AAAAAAAAAA"
void
CWE122_Heap_Based_Buffer_Overflow__c_CWE193_char_memcpy_15_bad(){
    char source[10+1] = SRC_STRING;
    data = (char *)malloc(10*sizeof(char));
    /* ... */
    /* POTENTIAL FLAW: no enough space for data to hold source */
    memcpy(data, source, (strlen(source) + 1) * sizeof(char));
    printLine(data);
    free(data);
}
```

Listing A.4: Code snippets of `is_format_lzma` in `libxml2`.

```
static int is_format_lzma(xz_statep state){
    /* ... */
    lzma_filter filter;
    /* ... */
    filter.id = LZMA_FILTER_LZMA1;
    if (lzma_properties_decode(&filter, NULL, state->in, 5) \
        != LZMA_OK)
        return 0;
    opt = filter.options;
    /* ... */
}
```

```
[stg  xT, xN, imm]
    ldr    x16, =TAG_MEM
    mov    x17, xT
    lsr    x17, x17, #49
    str    xT, [x16]
```

```
[ldg  xT, xN]
    ldr    x16, [xN]
    mov    x17, #0xF0
    lsl    x17, x17, #49
    and    xT,  x17, x17
```

Figure A.9: Implementation of MTE instruction analogs.