CombiSan: Unifying Software Sanitizers for Comprehensive Fuzzing

Matteo Marini*
Sapienza University of Rome
m.marini@diag.uniroma1.it

Floris Gorter*
Vrije Universiteit Amsterdam
f.c.gorter@vu.nl

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

* Equal contribution joint first authors

Abstract

Modern C/C++ bug detection efforts heavily rely on fuzzing with software sanitizers. However, the most popular sanitizers have limited interoperability. As a result, developers often enable each sanitizer in isolation, if at all, requiring multiple runs. This sequential execution undermines performance and tests code in a non-uniform manner.

In this paper, we present CombiSan, a fuzzing-optimized sanitizer that simultaneously detects all the addressability, uninitialized memory, and other undefined behavior issues covered by the three most popular sanitizers: ASan, MSan, and UBSan. CombiSan features a unified shadow memory design that efficiently tracks both the addressability and the initialization state of every byte of program memory. In addition, CombiSan's instrumentation seamlessly integrates with state-of-the-art detection of other undefined behavior classes. As bugs found by different sanitizers may mask each other by terminating execution early, CombiSan defers its analysis of all aggregated issues to test case completion.

In our evaluation, CombiSan detected 81 new bugs in 10 programs tested daily by OSS-Fuzz. On average, fuzzing with CombiSan is 1.7x faster than sequentially testing with ASan+UBSan and MSan. Moreover, our results demonstrate that CombiSan has the same bug detection accuracy as these sanitizers, despite running for significantly fewer CPU hours.

1 Introduction

Memory safety violations remain a major threat to systems security. Nonetheless, developers favor unsafe languages like C and C++ for their performance, legacy integration, and low-level control. Unfortunately, these low-level features also make it easier to introduce *bugs*, which in turn may lead to vulnerabilities. In response, both academia and industry invest great effort into detecting such bugs, resulting in the development of *software sanitizers* [10, 25, 27, 61, 65] and *fuzz testing* frameworks [21, 23, 50], which have reshaped the modern software testing landscape. The OSS-Fuzz [21] continuous fuzzing framework is a case in point, having detected over 50,000 bugs while fuzzing thousands of projects with the three de facto most popular sanitizers: AddressSanitizer (ASan) [61]—to detect addressability issues—MemorySanitizer (MSan) [65]—to identify uses of uninitialized memory (UUM)—and UndefinedBehaviorSanitizer (UBSan) [10]—to capture other undefined behavior.

However, current sanitizers offer limited *interoperability*. Some sanitizers are incompatible by construction; for instance, MSan's designers explicitly decided not to integrate with ASan, based on the assumption that applying both instrumentations simultaneously would incur greater overhead than executing the tools individually [44, 65]. Additionally, both rely on shadow-memory-based metadata, with incompatible layouts. Others (e.g., MSan and UBSan) adopt compatible metadata structures, but their instrumentation conflicts, degrading detection guarantees. Lastly, even when sanitizers combine smoothly (e.g., ASan and UBSan), one sanitizer can still mask bugs from the others, even preventing fuzzing in extreme cases (Section 6). The missing interoperability of sanitizers drastically reduces overall fuzzing efficacy, because testing targets for multiple classes of bugs requires performing separate fuzzing runs for different sanitizers.

In this paper, we propose CombiSan, a new fuzzing-oriented approach to detect all the issues covered by the three most popular sanitizers in a single solution. Fundamentally, CombiSan uses a unified metadata layout that enables simultaneous detection of both use-of-uninitialized-memory (UUM) errors and addressability issues. More specifically, CombiSan features a 1-byte-to-2-bit *shadow memory* which encodes both addressability (i.e., redzone or not) and the initialization state for every byte of program memory. CombiSan opportunistically detects all *uninitialized loads*, a superset of UUM errors, and filters out uninteresting violations using a binary-based *accurate detector*. Additionally, CombiSan's ASan-like instrumentation granularity (only memory accesses and allocations) ensures smooth integration with out-of-the-box UBSan instrumentation.

Sanitizers commonly halt the execution upon detecting a bug. When using multiple sanitizers, this may result in

later (unrelated) errors going undetected. To prevent this, CombiSan suppresses termination (similar to *recovery mode* in existing sanitizers) and defers error analysis. With this method, CombiSan aggregates all the bugs triggered by a single test case and then acts upon them. This strategy ensures that bugs do not interfere with each other, transparently enabling simultaneous detection of multiple bug classes.

Other than unifying the bug detection capabilities of leading sanitizers, CombiSan also improves upon MSan's compatibility. Specifically, MSan requires instrumenting not only the target application but also all of its dependencies, including system libraries. This makes it difficult to detect UUM errors at scale. Not surprisingly, OSS-Fuzz does not use MSan by default *"due to the likelihood of false positives from uninstrumented system dependencies"* [20]. As a result, at the time of writing, only 40% of projects decided to test their code for UUM errors, compared to 77% and 57% for ASan and UBSan, respectively [21]. CombiSan overcomes this limitation with a recently proposed multi-layered design to effectively filter out false positives [47]. Unlike MSan, this approach does not require compile-time instrumentation of library code, promoting high compatibility.

We evaluated CombiSan's bug finding capabilities on a dataset of 10 OSS-Fuzz targets lacking MSan support, detecting 81 previously unknown bugs: 38 UUMs, 17 addressability issues, 13 from other types of undefined behavior, and 13 spontaneous crashes. All the bugs have been responsibly disclosed according to each project's security policy. Further, we tested CombiSan's performance on 8 common fuzzing targets, detecting a final slowdown of 4.7x, which is 1.7x times faster than the compounded slowdown of running ASan+UBSan and MSan sequentially (7.9x). On the same subjects, we also evaluated CombiSan's bug-finding capabilities, showing that it can detect the same bugs as the three mentioned sanitizers. Finally, CombiSan's logic for UUM and addressability errors incurs an average slowdown of approximately 150% on SPEC CPU2006 and 2017, which is significantly lower than the sum of the overheads of ASan and MSan.

Contributions. Summarizing, our contributions are:

- A new software sanitizer design that detects addressability issues, UUM errors, and other standard undefined behavior in a single solution, catered to fuzz testing.
- CombiSan: an open-source implementation of our design in the LLVM compiler infrastructure, with a minimal patch to the AFL++ fuzzer for optimal integration.
- A comprehensive evaluation of CombiSan that shows its performance in terms of both slowdown and memory overhead, its bug finding capabilities compared to state-of-the-art sanitizers, and a fuzzing campaign on 10 real-world targets, where CombiSan detected 81 new bugs.

Source. https://github.com/vusec/combisan

2 Background

2.1 Undefined Behavior

Programming languages do not define outcomes for all operations. This lets compilers assume that certain operations never happen, enabling optimizations. Unfortunately, such Undefined Behavior (UB) often harbors bugs. These bugs, in turn, may cause *vulnerabilities*. In the following, we discuss the most common classes of UB in C/C++.

Spatial errors. Memory objects should only be accessed within their bounds, failing to do so is UB and results in spatial memory errors. These errors (e.g., buffer overflows) enable reading from or writing to unrelated memory objects.

Temporal errors. Pointers should not be dereferenced after being invalidated. Such accesses (e.g., via dangling pointers) are UB known as temporal errors. Bugs like use-after-free and double free often cause critical memory corruption. Together with spatial errors, these are called *addressability issues*.

Use of Uninitialized Memory (UUM). Memory *used* for sensitive operations (e.g., evaluating conditional branches) has to be fully initialized, otherwise they result in UB. In contrast, *loading* uninitialized values is defined, and is used for optimizations, like load widening, by compilers [17, 47].

Other undefined behavior. The memory-related UB cases described above are generally the most critical as they often have security implications. On the other hand, there is much more behavior that is undefined, like signed integer overflows, that may still be security-relevant. From now on, we will use undefined behavior (or UB) to refer to this last category.

2.2 Sanitizers

Software sanitization is among the most used techniques to expose bugs in software [64]. It works by injecting instrumentation alongside the application code; this instrumentation then checks specific properties at runtime, raising an alert in the case of violations. The instrumentation can be inserted either at compile time or at run time (i.e., binary-only) [64].

Sanitizers are effective at detecting bugs, but they suffer from the *coverage problem*, meaning they can only detect bugs when triggered. This issue makes them most useful when used in combination with testing techniques, like fuzz testing, which provides many inputs to exercise the code differently. The combination of fuzzing and sanitizers is the de facto standard to detect software bugs and vulnerabilities [58]. In the following, we briefly introduce three prominent sanitizers.

```
int uninit;
if(uninit + 1)
  return 1;
```

Listing 1: Example conflict between MSan and UBSan, causing MSan to not report an error [33].

Addressability issues. The most common solution to detect addressability issues is AddressSanitizer (ASan) [61]. ASan discovers spatial errors by padding memory objects with redzones. It uses a *shadow memory* to keep track of redzones, and, by instrumenting loads and stores, detects accesses to them. ASan detects temporal errors by delaying reuse of freed memory with a *quarantine*, invalidating the shadow memory accordingly. Since its first release, many enhancements have been proposed to ASan, most notably by pruning [41, 69, 70, 72, 74] or accelerating [25, 46] its checks.

UUM errors. The state-of-the-art solution to detect UUM errors is MemorySanitizer (MSan) [65]. MSan uses shadow memory to track initialization of memory objects. Since loading uninitialized memory is allowed, MSan tracks the flow of uninitialized memory and defers error detection to *uses* of uninitialized data. This mechanism, known as *shadow propagation*, makes the instrumentation more complex. Additionally, to avoid *false positives*, MSan needs to track initialization of objects in library code, thus requiring recompilation of all dependencies, including standard and system libraries.

To solve MSan's compatibility issues, QMSan [47] proposes a new design based on run-time instrumentation, side-stepping the recompilation problem. Unfortunately, this comes at a significant cost, as binary-based solutions incur a high slowdown. To retain high performance, QMSan uses a multi-layered architecture: a fast *opportunistic detector* detects all loads of uninitialized memory, and a slow-path *accurate detector* filters out the loads that never end up being *used*. It then remembers known loads of uninitialized memory while fuzzing, thus limiting invocations of the slow path.

Undefined behavior. The state-of-the-art sanitizer for UB is UndefinedBehaviorSanitizer (UBSan) [10]. It performs run-time checks on operations susceptible to undefined behavior and raises an error when triggered. For example, UBSan checks arithmetic operations for integer overflow.

3 Motivation

While the ASan, MSan, and UBSan sanitizers are greatly effective in enhancing fuzzing campaigns, their level of interoperability limits their deployment. Notoriously, MSan and ASan instrumentation cannot be enabled at the same time [65], with conflicting instrumentation and memory state shadowing. Similarly, MSan and UBSan integration, while allowed by the compiler, results in accuracy loss due to the interaction between the instrumentations. For instance, Listing 1 shows a snippet of code triggering a trivial UUM error that MSan detects when run in isolation, but misses when run in combination with UBSan. Currently, the only supported sanitizer combination for fuzzing is the one between ASan and UBSan.

Motivation 1

Popular sanitizers have limited interoperability, especially due to complex UUM detection instrumentation.

Further, even when sanitizers can be combined, i.e., ASan and UBSan, their integration is not free from pitfalls, as sanitizers typically terminate the application upon triggering the first bug by the input at hand. This design choice comes with a clear drawback. With multiple sanitizers deployed at the same time, an early bug detected by one of the sanitizers prevents the others from detecting (potentially more severe) bugs later in the same execution [67]. This problem is worsened by bug reports caused by false positives or technically safe operations [12, 42, 66], or long-term unresolved bugs. For instance, in Section 6 we evaluate targets where UBSan reports bugs for *every* test case, normally prohibiting proper fuzzing.

In practice, we noted that only 57% of the projects on OSS-Fuzz enable UBSan, and that OSS-Fuzz runs ASan and UBSan in isolation [21]. As highlighted by previous research [49], some developers disregard the specifications of the standard, "using" UB under the assumption that it is, in practice, defined. As an example, while analyzing the code of a project we tested during our evaluation, we found that the developers are purposely ignoring some integer overflows, allowing them just because similar software does [19]. As a result, software harbors UB bugs and this interference further complicates testing, justifying efforts that test with UBSan separately.

Motivation 2

Fuzzing with multiple sanitizers causes interference, where one prevents another from finding bugs.

Given the many interoperability issues, the status quo to fuzz with multiple sanitizers is thus to make multiple runs, each with different sanitizers enabled. As a result, thoroughly testing a project requires significant CPU time to support the different runs. Additionally, due to the stochastic nature of fuzzing, each run exercises the software differently, i.e., they produce different inputs and potentially trigger different bugs. Consequently, the wrong sanitizer may be enabled upon triggering a bug, causing it to go undetected. Instead, we argue that a more effective and performant solution is to have a single setup with the needed sanitizers. With this methodology, the fuzzer tests all exercised code with all the sanitizers, resulting in a more uniform and comprehensive fuzzing.

Motivation 3

Deploying sanitizers sequentially, instead of simultaneously, results in slower and heterogeneous fuzzing.

In addition to inter-sanitizer compatibility issues, some sanitizers face compatibility issues of their own. ASan and UBSan have high compatibility with existing software and can generally be enabled out of the box on most targets. In contrast, MSan faces key compatibility issues that significantly limit its applicability [20, 38, 47]. Since its inner workings require instrumented libraries, recompiling them with MSan often presents additional challenges. This is especially the case with system and standard libraries as well as their dependencies. Other difficulties include altering the build system and using complex constructs (e.g., assembly code) that do not integrate well with MSan. These problems are so critical that MSan was recently disabled from all OSS-Fuzz projects when upgrading to Ubuntu 20.04. The latter does not come with pre-built MSan libraries, unlike the previous Ubuntu version (16.04) [20]. At the time of writing, MSan has only been re-enabled for 218 out of 547 C/C++ projects in OSS-Fuzz.

Motivation 4

The state-of-the-art compiler-based UUM detector faces compatibility issues hindering deployment.

The four issues above motivate our efforts to effectively combine sanitizers for comprehensive fuzzing. We note that combined sanitizers have long existed in the binary realm, most prominently Valgrind [63] and DrMemory [5], yet with slowdowns in the range of 10-20x [5] that make them impractical for fuzzing [7]. While the recent QMSan [47] alleviates both the slowdown of binary instrumentation and the compatibility issues of MSan, QMSan solely focuses on UUM detection and does not easily combine with other sanitizers. Moreover, working at the binary level can limit the accuracy of the sanitizer, notoriously confining addressability issue detection mostly to heap errors only [17, 63], and with UB errors considered nearly impossible to detect [59]. Therefore, pursuing sanitizer combinations inevitably hints at using compiler instrumentation, for both accuracy and speed.

The developers of MSan originally argued that combining it with ASan would compound their overheads, making it more efficient to run the two sequentially instead [44, 65]. Years later, maintainers of LLVM's sanitizers stated that integration between ASan and MSan was not planned due to the potential complexity and overhead of such a solution [44].

In this paper, we challenge both the assumptions above. In Section 4 we show that we can unify ASan and MSan-class bug detection into a single coherent solution in the context of fuzzing, and in Section 6 we prove that this design incurs overheads far lower than the sum of the current solutions.

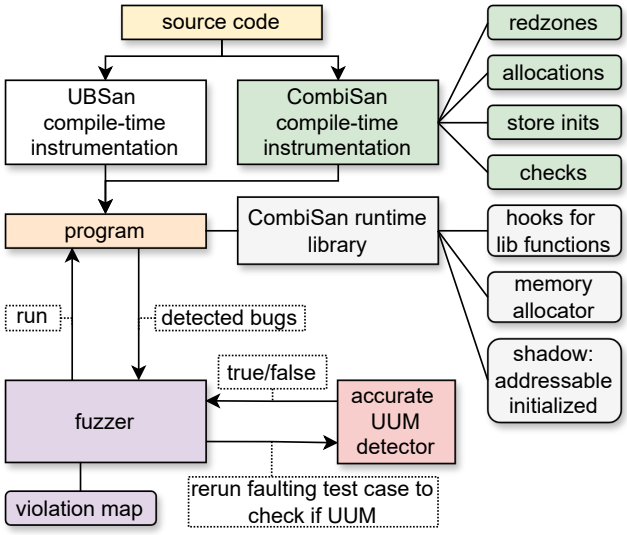

Figure 1: Architecture and workflow of CombiSan.

4 Design

This section presents CombiSan's design by first providing a general overview, and then detailing its components.

4.1 Overview

Conceptually, CombiSan aims to unify the bug detection capabilities of ASan, MSan, and UBSan into a single solution catered to fuzzing. Figure 1 displays a high-level overview of CombiSan's components and their interaction.

CombiSan uses compile-time instrumentation to transform C/C++ source code into sanitizer-enabled programs. These transformations pad stack and global objects to account for redzones, and insert code to interact with CombiSan's runtime metadata: allocations are marked uninitialized by default, store operations update this state to initialized, and all memory accesses are accompanied by sanitizer *checks* for validity. At this point, CombiSan can also apply existing UBSan instrumentation. After compiling, CombiSan links the program with a runtime library, which contains multiple features: shadow memory management, e.g., creation, a custom heap allocator (to insert redzones, mark objects as uninitialized, and quarantine them upon deallocation), and function *hooks* for interposition (e.g., to intercept `malloc`). As a result, this overarching *sanitizer* component provides CombiSan with the ability to detect addressability issues, uninitialized loads (a superset of UUMs), and UB bugs.

Next, the *fuzzer* component repeatedly executes the program with different inputs, and monitors if any bugs were aggregated during each execution. If so, it takes the action that we deem as more profitable for the bug type at hand. CombiSan treats addressability issues as fatal, resetting the

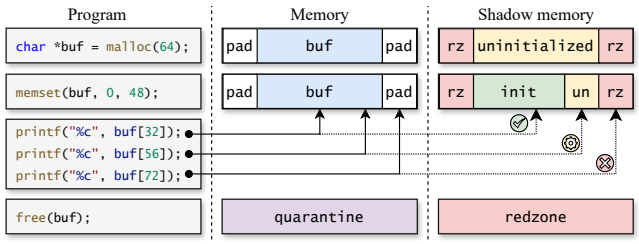

Figure 2: High-level example instrumentation of COMBISAN.

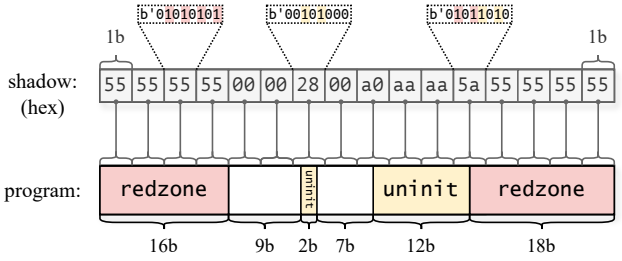

Figure 3: COMBISAN's 4-to-1 byte shadow memory mapping where each program byte corresponds to two metadata bits. The figure displays a partially initialized 30-byte object surrounded by inaccessible redzones.

(otherwise likely unstable) program state. UB bugs undergo analogous treatment when first encountered, while future occurrences (which can be frequent) are ignored to enable deeper fuzzing in the face of enduring UB allowed within codebases (Section 3). For uninitialized loads, COMBISAN invokes an accurate UUM detector to determine if this load has a subsequent *use* [65] (i.e., branch, address computation, or system call argument). Afterwards, the fuzzer updates its *violation map* to remember if the instance was not a UUM error, to avoid invoking the accurate detector for future occurrences.

Example. Figure 2 shows a simplified overview of how COMBISAN detects errors while executing a test case. Upon allocation, COMBISAN pads the example 64-byte heap allocation with unaddressable redzones, and sets the usable (non-redzone) data to uninitialized. The data only becomes initialized when the program writes to it (through `memset`).

Next, COMBISAN checks the validity of the memory accesses: when the program *reads* from memory, COMBISAN checks if the target is *poisoned*, meaning either unaddressable or uninitialized, raising an error if so; meanwhile store operations first check for addressability, and only afterwards update the target to initialized. The example program writes to memory using the `memset` operation, which in this case accesses addressable memory, and causes the memory to be marked as initialized. Next, the program performs three memory reads through the `printf` calls: the first one reads *valid* data (i.e., addressable and initialized); the second one reads addressable but uninitialized data, resulting in an error being saved; the last read operation accesses a redzone, also aggregating an error. Finally, upon deallocation, heap memory objects are set to unaddressable and added to the heap quarantine, such that future accesses raise use-after-free errors. In a fuzzing context, after the program finishes, the reported uninitialized load causes COMBISAN to invoke its accurate detector to determine if this concerns a UUM (it does, in this case).

4.2 Sanitizing with COMBISAN

In this section, we detail the *sanitizer* part of COMBISAN, more specifically how we create a *unified* shadow memory mapping to detect multiple classes of bugs simultaneously.

4.2.1 Shadow Memory Layout

COMBISAN's key intuition lies in its *unified* shadow memory model, which tracks both the initialization and addressability statuses of each byte of program memory. More precisely, each byte of memory maps to two bits of shadow memory, as visualized in Figure 3. The first bit represents addressability: zero means valid, one (set) means invalid (i.e., redzone or quarantined). The second bit indicates if the corresponding program byte is initialized or not. By embedding both statuses in the same shadow memory, COMBISAN can efficiently check for validity by evaluating whether the joint 2-bit shadow value is zero (valid) or non-zero (invalid), with just one metadata lookup and comparison. Another benefit is that, as in ASan, large regions of valid (or unused) memory remain zero-initialized, promoting zero-page deduplication.

Since COMBISAN maps 8 bits of memory to 2 bits of shadow memory, a 4-to-1 byte relation, it incurs 25% memory overhead by default. This is double of ASan's shadow memory (1 byte of shadow per 8 program bytes, so 8-to-1), but less than MSan's which requires a bit-to-bit mapping (i.e., 1-to-1, 100% overhead) to support shadow propagation. Considering that COMBISAN combines ASan's and MSan's bug finding capabilities, the shadow overhead is modest. Furthermore, just like ASan and other tools that detect use-after-free errors, COMBISAN's final memory overhead is ultimately dominated by the heap quarantine [25, 27], making shadow memory less pronounced in the final memory overhead.

4.2.2 Memory Allocations and Accesses

COMBISAN interacts with its shadow memory by instrumenting memory allocations and accesses. Upon allocation, COMBISAN pads objects to accommodate for inaccessible redzones, and then updates the shadow memory to reflect this by marking them as unaddressable. Additionally, COMBISAN sets the shadow memory corresponding to the object itself to uninitialized. When objects are deallocated, the shadow memory for the entire object becomes unaddressable.

Next, COMBISAN accompanies memory accesses with

```
1 u16* shadow_ptr = (ptr >> 2) + BASE;
2 u16 shadow = *shadow_ptr;
3 if(shadow & 0x5555 != 0) // mask init bits
4   error!
5 *shadow_ptr = 0; // set init
```

Listing 2: CombiSan's instrumentation for an 8-byte memory store to `ptr`. For memory loads, the mask operation on line 3 and the shadow update on line 5 are omitted.

```
1 u8 shadow = *((ptr >> 2) + BASE);
2 if(shadow != 0) {
3   bit_offset = (ptr & 0x3) << 1;
4   four_bits = (shadow >> bit_offset) & 0xF;
5   if(four_bits != 0)
6     error!
7 }
```

Listing 3: CombiSan's instrumentation for a 2-byte memory load to `ptr`. The *slow-path check* is only required for one and two byte accesses. For {1,2}-byte stores we initially mask the shadow value to exclude initialization bits from the check, and then set the sub-byte shadow value to zero at the end.

checks that inspect their validity. For memory *load* operations, CombiSan evaluates whether the corresponding shadow memory is completely zero, because any set bit indicates either unaddressable or uninitialized memory. In contrast, for *store* operations, CombiSan first checks if the memory is addressable by loading the corresponding shadow memory, masking off the initialization bits, and then comparing the masked shadow value to zero. If the memory is unaddressable, CombiSan registers an error. Afterwards, CombiSan updates the shadow memory to zero, because it is now initialized, and guaranteed to be addressable (unless a bug occurred). The store instrumentation highlights the necessity of two distinct metadata bits to capture validity; sharing a single bit would create an undecidable choice between resetting the bit (initialization) and triggering a fault (redzone hit).

Thanks to the (joint) binary encoding of validity in its shadow memory, CombiSan can conveniently fetch and update metadata. In particular, CombiSan performs highly efficient checks for 4-byte aligned loads/stores that access four or more bytes, where it operates on complete bytes of shadow memory. For example, Listing 2 shows how CombiSan first checks for addressability and then updates the initialization state for an 8-byte store, both by operating directly on 2 bytes of shadow memory. CombiSan's shadow address computation is nearly identical to ASan's: divide the pointer by the shadow granularity (divide by four, so shift by two), and add this on top of the static shadow base address (line 1).

For memory accesses that span only one or two bytes, CombiSan needs to interact with its shadow memory at sub-byte granularity. Listing 3 shows how we achieve this for an example 2-byte load. CombiSan first employs a *fast check* that optimistically compares the shadow *byte* to zero (line 2), and if it is non-zero, enters a slow-path check that extracts and compares the appropriate bits. The *bit offset* (line 3) denotes the number of bits to skip over depending on the alignment of the pointer. After shifting the shadow value by the bit offset, we either extract two or four bits using an `AND 0x3` or `AND 0xF` mask for 1 or 2 byte accesses (line 4), respectively. Since access sizes are known at compile time, CombiSan does not emit slow-path checks for accesses larger than two bytes.

We point out that ASan *always* needs to insert a twofold check. More specifically, as in CombiSan, ASan first optimistically checks for a non-zero shadow value, but then always requires a fallback slow-path check to support partially accessible shadow memory granules [61]. Furthermore, for accesses that cross the shadow granule alignment, CombiSan follows a similar approach as ASan. For loads, we check the first and last byte for validity (like ASan). For stores, we check all accessed bytes individually, and initialize them as we go.

With this overall instrumentation scheme, CombiSan acts as an opportunistic detector [47] of loads of uninitialized memory. This allows CombiSan to detect an overapproximation of UUM errors: while not all *loads* of uninitialized memory are followed by a *use* of uninitialized memory, the opposite holds, i.e., UUM errors are always preceded by a load of uninitialized memory. Thus, CombiSan's instrumentation detects all UUM errors without loss of sensitivity, but it may produce false positives. We overcome this loss of precision by consulting an *accurate* UUM detector during fuzzing (Section 4.3). We remark that opportunistic detection only requires CombiSan to match the byte granularity of memory accesses, rather than the bit-precise metadata required by MSan's shadow propagation [47]. This conveniently matches the precision needed to detect addressability issues, ultimately enabling a compact and unified metadata layout.

4.2.3 Optimizations

We introduce multiple compile-time optimizations that reduce false-positive reports of uninitialized loads by proving they do not result in UUM errors. We achieve this in two ways: (i) by marking memory as initialized whenever possible, e.g., for constructor methods, and (ii) by omitting checks for uninitialized loads if we statically know the loaded value does not propagate to a *use* sink.

Constructors. We identify two cases where stack structures are initialized through (external) constructor calls, and therefore do not have to be marked as uninitialized. In particular, we scan for `va_start` intrinsics, which initialize the corresponding `va_list` variable for variadic functions. CombiSan considers these corresponding stack allocations as initialized. Additionally, we find C++ constructor calls for which we then identify the corresponding *implicit object*

(`this`). After the external constructor initializes the object, we mark the corresponding shadow memory as initialized.

External allocations. Like ASan and MSan, COMBISAN *interposes* on dynamic allocation functions like `malloc` for pervasive instrumentation. However, since COMBISAN does not require external code to be instrumented, initialization of external heap allocations may not be visible. To avoid loads from such memory to cause false-positive reports, we distinguish between *visible* (i.e., in the target module) and *external* heap allocations. We instrument visible allocations at compile-time, while external allocations go through interposition. Both types receive redzones, however only visible allocations are marked as uninitialized. We also support visible indirect calls by prepending a runtime *handler* that compares the call target to known heap allocator functions.

This approach can, in principle, lead to false negatives if libraries fail to fully initialize an object. However, we consider this trade-off favorable, as reducing false positives decreases the number of invocations of the slow path, thereby improving throughput. Moreover, we argue that sensitive dependencies should undergo independent testing campaigns. We observed no false negatives in our extensive evaluation (Section 6).

Static taint analysis. While MSan deploys full-scale dynamic shadow propagation (i.e., *dynamic taint analysis*) to identify *uses* of uninitialized loads, we can statically approximate such analysis to filter out provably safe (uninitialized) loads. More specifically, for every load instruction, COMBISAN transitively follows the use-def chains of the loaded value, and determines if it can end up in MSan-class *uses* (e.g., conditional branches), but conservatively bails out if the loaded value *escapes*. Such escapes occur primarily if the value is stored to memory, but also for example when the value is returned. Currently, the analysis is not inter-procedural, hence if the loaded value ends up as a function call argument, the analysis also bails out. In all other cases, we statically excluded the possibility that the loaded value ends up being *used*, and hence the associated load does not need to be checked for being uninitialized. While this analysis leaves room for extensions, our preliminary performance results showed that the current strategy suffices.

4.2.4 Runtime Management

While COMBISAN mostly instruments the target program at compile-time, certain operations require runtime instrumentation. More specifically, COMBISAN interposes on heap allocations to manage their redzones and validity, and certain library calls to either check for validity or propagate metadata.

Heap. For heap allocations, COMBISAN pads the objects with redzones and marks them as unaddressable. Unless this concerns an external allocation (see Section 4.2.3), COMBISAN marks the usable data of each object as uninitialized. When the program deallocates an object, COMBISAN updates the entire memory chunk to unaddressable, causing future accesses (i.e., *use-after-free*) to register a fault. To prevent the memory from quickly being reused, which could lead to dangling pointer accesses going undetected, like ASan we employ a *heap quarantine*, which delays the availability of the memory for future allocations.

Library calls. Certain standard library calls are of particular interest not only for detecting addressability issues, but also for ensuring accurate detection of uninitialized loads. A prime example is `memcpy`, where COMBISAN verifies that both the source and destination locations are addressable, and additionally *propagates* the initialization state from the source to the destination (like MSan), because the program is implicitly copying the data. This combined operation of checking for addressability and copying the initialization state is implemented in a single iteration, where the copied shadow values are inspected for the presence of set addressability bits. Overall, COMBISAN includes such *interceptors* for the union of both ASan's and MSan's interceptors.

4.2.5 Undefined Behavior

We designed COMBISAN to support seamless integration with existing UBSan instrumentation. In contrast, combining UBSan with MSan leads to conflicts [33], because both instrument similar operations, such as loaded values and arithmetic instructions. UBSan inserts checks, for example to detect integer overflows, while MSan instruments the same operations to propagate shadow state from source to destination. The conflicts appear to arise from UBSan marking its instrumentation as not requiring sanitization, thereby breaking MSan's shadow propagation. Unlike MSan, COMBISAN's instrumentation granularity matches that of ASan: it targets only memory accesses and allocations. As a result, COMBISAN sidesteps the above conflicts, and integrates smoothly with UBSan. All the provisions discussed in Section 4.2 enable COMBISAN to unify the bug detection capabilities of ASan, MSan, and UBSan into one solution (*Motivation 1*).

4.3 Fuzzing with COMBISAN

COMBISAN is designed to work in the context of fuzzing, with compact metadata enabling the accurate detection of addressability issues and opportunistic detection of UUM errors. While fuzzing, COMBISAN tests every input for multiple bugs, as opposed to performing multiple runs with different sanitizers. This ensures efficient and uniform fuzzing as all the tested code will be simultaneously exercised under the same sanitizers (*Motivation 2*).

Still, combining the detection power of multiple sanitizers comes with new challenges in the context of fuzzing, above

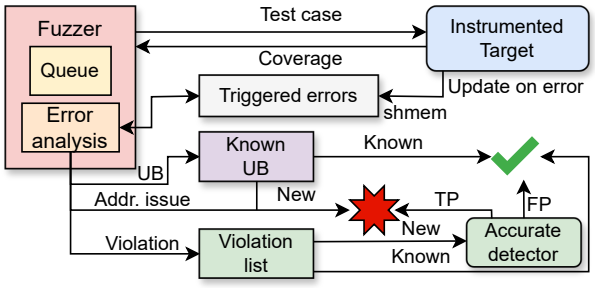

Figure 4: Fuzzing overview. When CombiSan detects an error, it communicates the error to the fuzzer through a shared map. When the execution is over, the fuzzer acts on the bugs.

all the ability to retain high accuracy when detecting multiple classes of bugs at the same time (*Motivation 3*). Figure 4 visualizes CombiSan's architecture while fuzzing.

4.3.1 Bugs Detection and Management

Typically, sanitizers immediately crash the application when triggering a bug; the crash is then detected and analyzed by the fuzzer. While this allows for convenient integration between fuzzers and sanitizers, it is not ideal in a multi-sanitizer scenario, as early exits induced by one sanitizer may mask subsequent bugs triggered by other sanitizers. To address this issue, CombiSan handles errors by merely recording the necessary details and continuing the execution. Only once the test case terminates, CombiSan informs the fuzzer, which then analyzes all the aggregated errors.

The fuzzer can then act on bugs according to different policies, depending on their class. By default, CombiSan consistently treats inputs triggering addressability and UUM bugs (detailed later) as crashing inputs, but opts for a more nuanced approach for UB bugs. Specifically, since software may contain many UB bugs (Section 3), CombiSan treats a test case as a crashing input only if it triggers a previously unseen UB error, while allowing those with already known UB(s) to complete successfully. This allows the fuzzing backend to process new bugs (e.g., by applying deduplication), but prevents further instances of the same bug from stopping deeper exploration of the programs. As Section 6 will show, we found targets triggering undefined behavior for every test case, hindering progress when fuzzing with UBSan. Instead, CombiSan can overcome this issue by reporting the error once, and then allowing for the fuzzer to continue.

4.3.2 UUM Errors

During each execution, CombiSan detects all the loads of uninitialized memory (*violations* from now on). As detailed before (Section 4.2.2), all UUM errors follow a violation, while the opposite does not hold. In practice, detecting violations allows for detecting a superset of UUM errors. Unfortunately, the number of false positives (i.e., violations that do not end in a UUM error) is non-negligible, so a mechanism to filter them out is necessary not to lose accuracy [47].

To address this issue, CombiSan draws from the multi-layered approach proposed by QMSan [47]: during each execution, it collects information about all the triggered violations, without raising any error, as in the case of addressability issues. Then, when the execution is over, the violations are analyzed alongside all the other errors, employing an accurate detector to check if any violation results in a UUM error. To avoid unnecessary invocations of the accurate detector, CombiSan maintains a *violation map* at runtime, and only uses the accurate detector when it detects a new violation.

When adding violations to the map, CombiSan follows QMSan's approach and identifies them with three properties. The first one is the address of the load instruction that triggered it, indicating where in the code it is raised. Unfortunately, identifying violations only by the offending address may result in false negatives if an instruction may or may not read uninitialized memory based on context, as could happen, for instance, in a memory load from an auxiliary function that can read arbitrary memory. So, to account for context, violation detection is augmented with spatial locality and temporal locality. Spatial locality is defined by the calling context when the violation is triggered, while temporal locality is detected through the combination of the identifiers of the current and last violations. Our evaluation (Section 6) shows that this approach is effective in practice, as we found no evidence of false negatives compared to MSan.

Compatibility. The major limitation of MSan is the requirement for instrumented dependencies (*Motivation 4*), which severely limits compatibility. CombiSan can overcome this limitation, i.e., it can still perform UUM detection with uninstrumented libraries. First, it considers heap allocations coming from library code as safe, and treats them as initialized (Section 4.2.3). Then, its design is robust to untracked initializations of memory objects from library code (due to uninstrumented store operations) since violations are checked by the accurate detector, which discards false positives.

Just like ASan, CombiSan can miss bugs (not only UUM errors) in uninstrumented libraries. This is reasonable, as relevant dependencies should undergo a separate testing process, and incidentally finding bugs in them is not the focus of a fuzzing campaign on the main target. Still, if a function is known to potentially trigger specific errors, CombiSan can use an interceptor to model its behavior and check for errors (Section 4.2.4), like many state-of-the-art sanitizers do.

Finally, CombiSan is designed to use binary-based solutions as its accurate detector, as these solutions are as accurate as their compiler-based counterparts [47], but do not require instrumenting dependencies at compile time. In practice, unlike MSan, CombiSan incurs no false positives in UUM error detection, even in face of uninstrumented dependencies.

5 Implementation

We implemented CombiSan in LLVM 20.1.0 and AFL++ 4.32c. CombiSan piggybacks on ASan's instrumentation, modifying it where necessary: the LLVM IR pass, shadow memory model, runtime allocator, function hooks, etc. UBSan can be enabled out of the box with its sanitizer flags. Further, our patch to AFL++ is minimal and likely generic enough to port to other general-purpose fuzzers with little effort.

To remember past (safe) violations, CombiSan uses a violation map, created during fuzzing startup and then maintained by the run-time module. When a violation is detected, hashes of the offending address, as well as spatial and temporal localities, are computed, and the map is updated accordingly. This mechanism is similar to how coverage collection is implemented in modern fuzzers. Regarding spatial locality, CombiSan only considers the first 3 entries of the callstack, which is enough to characterize an offending site [37]. Using a higher value is possible, but in our early tests this only resulted in more invocations of the accurate detector due to the increased sensitivity, without detecting new bugs.

We opt for Valgrind as the accurate detector for UUM errors, as it is well-established and actively maintained. Since CombiSan queries it in a black-box fashion, equivalent tools like DrMemory could also be used. Since Valgrind also uses shadow memory, it cannot easily analyze binaries that also use one, including programs instrumented by CombiSan. Therefore, to replay executions, we provide it with a version of the target compiled without sanitizer instrumentation.

6 Evaluation

In this section we present a set of experiments we designed to evaluate CombiSan. To show its bug-finding capabilities, we assembled a dataset of real-world latest-version software to test. Next, we evaluated CombiSan's accuracy in finding bugs using common sanitizer and fuzzing benchmarks, in comparison with ASan, MSan, and UBSan. Finally, we measured CombiSan's performance in terms of slowdown, memory overhead, and fuzzing throughput. We again compare these results with those of the three state-of-the-art sanitizers.

All the experiments were performed on a machine with Ubuntu 24.04 LTS equipped with an AMD Ryzen Threadripper PRO 7995WX CPU with 96 physical Cores and 500 GB of RAM. Each experiment has been performed with minimal background activity on the machine, and the fuzzing runs have been executed in isolation using Docker 27.5.1, binding each run to a physical core while keeping the relative logical core on idle. Following AFL++'s documentation [1], we used *persistent mode* fuzzing, which is preferred for compile-time instrumentation over the more binary-oriented *fork mode*, and is standard in production fuzzing frameworks like OSS-Fuzz. Unless stated otherwise, we configured UBSan to detect the same bug categories as in OSS-Fuzz [56].

Project	Crash	A	UUM	UB	Total
libredwg	3	3	5	3	14
assimp	0	0	0	2	2
ghostscript	0	0	4	3	7
libdwarf	0	0	0	0	0
libucl	2	9	3	0	14
gpac	1	0	11	2	14
serenity	0	0	0	0	0
opensc	0	0	6	0	6
inchi	7	5	8	3	23
libheif	0	0	1	0	1
Total	**13**	**17**	**38**	**13**	**81**

Table 1: Dataset and results of the bug finding experiment. The table divides the bugs we found in addressability issues (A), UUM errors, undefined behavior (UB), and bugs that trigger a crash but are not related to sanitizers, like segmentation faults (*Crash*). Table 6 (in the Appendix) lists additional information like software version and fuzzing harness used for our tests.

6.1 Finding New Bugs

To test CombiSan's bug-finding capabilities, we assembled a dataset of software tested daily on OSS-Fuzz using the same selection criterion as in QMSan [47]. We ranked all the OSS-Fuzz projects by the number of security-related bugs found in OSS-Fuzz from 2024 to the time of testing, and selected the top 10 entries without MSan support. For projects with more than one fuzzing harness, we ranked them according to how many bugs were found using each, and selected the top one. Ranking projects and harnesses according to the number of security bugs found in OSS-Fuzz is a proxy to choose well-tested software, while choosing targets without MSan support is helpful in showcasing CombiSan's increased compatibility. Table 1 displays the targets of our final dataset.

For each subject, we performed 5 fuzzing runs of 72 hours each. As starting seeds, we used the public fuzzing queues OSS-Fuzz maintains for each subject, allowing CombiSan to efficiently start bug detection from saturated queues. At the end of the runs, we collected the crashes, if any, and deduplicated them based on the type of error (e.g., UUM error, buffer overflow, etc.) and the location of the error. We confirmed our findings through manual inspection, to further remove duplicates missed by automatic techniques, and to confirm the bugs are true positives. All the bugs described in this section have been confirmed by the developers.

Table 1 shows the results of this experiment. CombiSan exposed a total of 81 new bugs. Almost half (38) are UUM errors: an expected result since we tested subjects that did not enable MSan support in OSS-Fuzz. We then counted 17 addressability issues and 13 for undefined behavior bugs. If provided with the test cases, ASan and UBSan detect these addressability and UB bugs, respectively, and also MSan would detect the found UUM errors, assuming that enabling MSan for the subject at hand is feasible in terms of compatibility.

CombiSan incidentally exposed another 13 bugs unrelated to any sanitizers, like segmentation faults as a result of dereferencing a wild pointer. We confirmed that these bugs were not introduced by our instrumentation by re-executing them with an uninstrumented version of the program. OSS-Fuzz should be able to detect these bugs in any configuration, and we speculate it would have eventually found them.

For peculiar cases, with `ghostscript` we noticed how every input was triggering undefined behavior due to a minor difference in the type of a function pointer compared to the function's signature. This issue was preventing fuzzing with UBSan, as every input causes a crash. Instead, thanks to how CombiSan handles UB (Section 4.3.1), fuzzing could go on, eventually detecting also other undefined behavior. In the end, CombiSan detected three new UB bugs for this project.

For `OpenSC`, a set of tools for managing smart card tools, we initially discovered 6 UUM bugs. OSS-Fuzz currently tests it with ASan and UBSan, and we detected no new bugs for these classes. Due to its complexity, this project has 12 harnesses. Insights from interacting with its maintainers on the semantics of its components led us to test other 6 harnesses mentioned in recent security advisories [55]: using the same fuzzing setup, we uncovered another 15 UUM bugs. We believe this test highlights how many bugs (especially UUMs) these projects harbor, urging for an efficient solution to detect them.

6.2 Detection Accuracy

We conducted multiple experiments to evaluate whether comprehensive runs of CombiSan can detect the same bugs as the existing state-of-the-art sanitizers can individually.

Juliet. We evaluated CombiSan's accuracy in detecting bugs using the NIST Juliet Test Suite [36], a collection of buggy test cases. Each test comes with a *bad* version, which contains the bug, and a *good* version, where the bug is fixed. To test CombiSan, we selected a total of 14 relevant CWE categories. As in related work [17, 27, 47], we removed test cases that do not (deterministically) contain a bug on 64-bit systems. For each category, we compared the results with the relevant sanitizer (ASan, MSan, or UBSan). The detailed results of this experiment are shown in Table 7 in the Appendix.

To summarize, CombiSan successfully detected the same bugs as ASan, MSan, and UBSan for the *bad* (i.e., buggy) test cases. For CWE 758, CombiSan reported more bugs in *bad* tests than UBSan due to the presence of uninitialized loads in these test cases. Similarly, for CWEs 122, 457, and 476, CombiSan raised errors for certain *good* tests, which are supposed to be bug-free, suggesting a false positive: however, these tests contain (safe) uninitialized loads, which CombiSan's accurate detector can discard. Finally, CombiSan raised an error for all *good* CWE 843 cases, since they all contain stack use-after-scope errors (true positives).

	ASan	MSan	UBSan	CombiSan		
Subject				**A**	**M**	**UB**
c-ares	1	0	0	1	0	0
guetzli	0	0	0	0	0	0
json	0	0	0	0	0	0
libxml2	4	3	1	3	2	1
openssl	1	3	0	1	3	0
pcre2	6	7	5	6	7	11
re2	1	2	1	1	2	1
woff2	1	0	0	1	0	0

Table 2: Number of bugs found while fuzzing Google's FTS subjects. The three rightmost columns show addressability issues (A), UUM errors (M), and undefined behavior (UB).

Fuzzing. To further evaluate CombiSan's accuracy, we assembled a dataset of 8 common fuzzing benchmarks from Google's Fuzzer Test Suite (FTS), as is common in related fuzzing and sanitizer work [3, 17, 35, 47, 57]. We selected the same subjects as evaluated for QASan [17] and QMSan [47]. Then, we compiled each subject with multiple sanitizers: ASan, MSan, UBSan, ASan and UBSan in combination, and CombiSan both with and without UB detection.

For each subject and configuration, we performed 11 fuzzing runs of 24 hours using the seeds provided by Google's FTS or an empty seed otherwise. For `guetzli`, two seeds are available in the test suite but cause throughput issues (discussed in Section 6.3), hence we used an empty seed.

For two subjects, code that is executed in every run triggers a UB bug (creation of a null pointer for `openssl` and indirect call through a pointer of a different type for both `openssl` and `libxml2`). To support fuzzing with UBSan, we had to disable detection of these errors for these subjects. We highlight how CombiSan overcomes this issue by ignoring the UB after the first time it is triggered, while UBSan triggers a crash every time, preventing fuzzing. For fairness, we disabled those checks for CombiSan as well.

After the fuzzing runs, we aggregated all the crashes found by each configuration, and clustered them by the crashing callstack to remove obvious duplicates. Then, we performed root cause analysis to identify the unique bugs found by each configuration. Table 2 shows the result of this experiment.

In all but one subject, CombiSan was able to find the same bugs as ASan, MSan, and UBSan. We point out that CombiSan finds these bugs with significantly less CPU time, as ASan, MSan, and UBSan use a combined total of 72 hours.

For the remaining subject, `libxml2`, CombiSan failed to find one addressability issue and one UUM error. Upon further inspection, these two bugs are the manifestation of the same issue: a heap object is deallocated and its content is later accessed (and used). ASan detects this as a use-after-free issue, while MSan detects it as UUM error. We manually confirmed that CombiSan correctly detects the addressability issue when the input is provided; further, we observed that executing this input produces two new loads of uninitialized

memory that CombiSan had not detected in any of the 11 runs. We thus conclude that CombiSan can detect this bug, but an input able to trigger it was not produced by the fuzzer.

On a different note, we observe that for `pcre2` CombiSan found 11 UB bugs, whereas UBSan only found 5. By looking at the fuzzing metrics of each run, we noticed that, for this subject, the runs with UBSan had a lower stability (∼93% on average) than those with the other sanitizers (100% for most of the runs, >99% for the others). Since this is the project with the highest number of bugs in the experiment, we believe that not sanitizing for memory safety errors made the fuzzing unstable, resulting in suboptimal testing and fewer bugs found. To test this hypothesis, we performed a run on this subject with both ASan and UBSan enabled at the same time; in this setup, UBSan detected 9 bugs—only 2 missing from those detected by CombiSan—with a stability on par with the other configurations. Since UBSan reproduces the two missing bugs, we conclude that they were not found due to fuzzing entropy (i.e., the fuzzer did not produce such test cases). We reproduced this test with ASan and UBSan in combination for every subject, finding the same bugs found by ASan and UBSan in isolation, further confirming our hypothesis.

Replaying test cases. Since fuzzing is non-deterministic by nature, we designed an experiment to compare the accuracy of CombiSan to that of the other sanitizers on the same set of inputs. For each of the above subjects, we performed a 24-hour fuzzing run using the same setup explained above, without any sanitizers. Then, from the resulting saturated queues, we started a new fuzzing run and stored the first 100,000 produced test cases. We *replayed* the executions of these test cases with each sanitizer and counted the unique crashes they detected, deduplicating based on the top three entries of the callstack when the error occurs. Since CombiSan needs an accurate detector to confirm violations, we reused AFL++'s startup phase to simulate a fuzzing run on these inputs, enabling it to use the accurate detector when needed.

Table 8 (in the Appendix) presents the results of this experiment. CombiSan detected the same unique crashes as ASan and UBSan for all subjects, without any false positives. For UUM errors, CombiSan detected 8 fewer than MSan: 1 in `openssl`, and 7 in `pcre2`. After manual inspection, we identified them as addressability issues, which MSan incidentally detects when the accessed values are also uninitialized and later used. CombiSan identifies and reports these bugs as addressability issues from accessing unaddressable memory.

CVEs and issues. We further assessed CombiSan's accuracy by confirming it detects the spatial and temporal memory errors of 15 reproducible CVEs evaluated in previous sanitizer work [25, 27, 72, 74]. For completeness, we added three use-of-uninitialized-memory bugs by selecting the latest OSS-Fuzz issues from ARVO [48] that we could reproduce on our Ubuntu environment. CombiSan successfully detected the

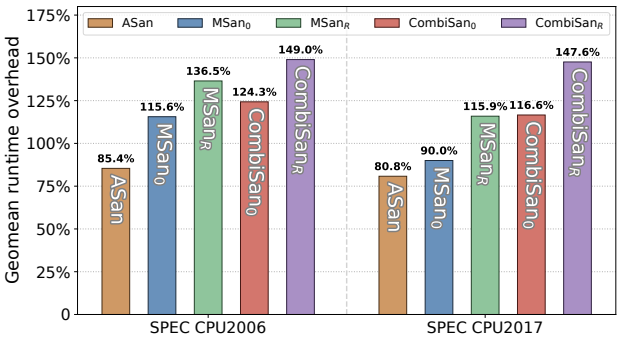

Figure 5: SPEC CPU runtime overhead comparison.

15 spatial and temporal memory errors, as well as the uninitialized load corresponding to the three UUM errors. Table 9 displays the list of CVEs and issues along with their bug type.

6.3 Performance

Aside from the ability to detect bugs, CombiSan's performance characteristics are a crucial factor to consider in software testing usecases. In particular, we are interested in measuring three properties: slowdown on a single execution, memory overhead, and the slowdown during fuzzing. In this section, we compare these results with the state-of-the-art sanitizers. Additionally, we evaluate the benefit of our compile-time optimizations for the number of reported uninitialized loads.

SPEC CPU. We measured CombiSan's overhead on the SPEC CPU benchmarking suites by comparing its runtime and memory usage to an uninstrumented baseline. We also measured ASan's and MSan's overhead for direct comparisons. We excluded UBSan due to the large number of bug reports it generates, and the fact that its instrumentation is orthogonal to both ASan and CombiSan.

While public patches exist for the addressability issues in SPEC CPU (which we applied), the benchmarks contain many unresolved UUM errors, hindering a comparison between MSan and CombiSan. To overcome this issue, we ran them in two modes. The *zero* mode, with MSan$_0$ and CombiSan$_0$, treats all memory as always initialized by making every shadow update write zero for the initialization bits. In this way, neither tool reports UUM errors, thereby establishing a convenient lower bound for the memory access instrumentation overhead (and CombiSan remaining fully functional for addressability issues). The *recovery* mode, with MSan$_R$ and CombiSan$_R$, operates the shadow memory normally and continues execution upon errors, with their reporting turned off to avoid inflating the overhead (e.g., for stacktrace symbolization). This mode may incur higher overhead in the face of frequent transitions to suppress errors, and is also the closest configuration to CombiSan while fuzzing.

	ASan	MSan$_0$	MSan$_R$	CombiSan
CPU'06	240%	107%	111%	292%
CPU'17	200%	107%	112%	247%

Table 3: SPEC CPU memory overhead comparison. For clarity: 100% corresponds to a 2x increase over the baseline.

Figure 5 displays the median runtime slowdown for five runs of SPEC CPU2006 and SPECrate2017 (with one parallel copy). Per-program overheads can be found in Table 10. The two modes of CombiSan show a geometric mean (geomean) runtime overhead of 124.3% and 149.0% on CPU2006, and 116.6% and 147.6% on SPECrate2017. Considering the more realistic recovery mode, CombiSan's overall slowdown of ~150% is significantly less than the compounded ASan+MSan overhead of 222% (CPU2006) and 197% (SPECrate2017). As a result, using CombiSan provides higher performance than running the two existing sanitizers sequentially.

To put CombiSan's overhead into perspective against ASan's and MSan's individual overheads: CombiSan is 19.8%–36.9% slower compared to ASan, depending on the configuration; comparing with MSan yields stable results across the two configurations: CombiSan is only 4% and 5% slower on CPU2006 for the *zero* and *recovery* modes, respectively, and 14% and 15% slower on SPECrate2017.

Regarding memory consumption, Table 3 shows the geomean memory overhead for the different sanitizers. We observe that the overhead is mostly dominated by the heap quarantine, rather than by the shadow memory. Indeed, MSan, which does not have a quarantine, has the lowest memory overhead (between 107% and 112% across all combinations) despite having the worst memory-to-shadow ratio. On the other hand, CombiSan only uses approximately 15% more memory than ASan, regardless of the evaluation *mode*, which we mostly attribute to the extra 1/8th of shadow memory.

Fuzzing. CombiSan is optimized to work with fuzzing: to estimate the overhead it adds in this scenario, we used the same Google FTS subjects as before. We compiled each subject in multiple configurations: one without sanitizers serving as the baseline, one for each state-of-the-art sanitizer (ASan, MSan, and UBSan), one with the combination of ASan and UBSan, and two with CombiSan (with and without UBSan). For UBSan, we used the same configuration from Section 6.2.

We performed fuzzing runs using the seeds provided by Google FTS, or an empty seed otherwise. For one subject (`guetzli`), we noticed that, when using the provided seeds, the two fastest configurations (i.e., baseline and UBSan) experienced low throughput as a result of diverging coverage, as already discussed in the literature by previous work [18]. We instead used an empty seed for this subject, resulting in the fastest configurations having the expected throughput and coverage. In total, we performed 11 fuzzing runs for each configuration, and report the median of the throughput.

Table 4 shows the results of this experiment. Overall, for CombiSan we measured a geomean slowdown of 3.72x and 4.68x compared to the baseline, without and with UBSan enabled, respectively. Compared to ASan, CombiSan has added a slowdown of 14% without UB detection, and 43% with UB detection. These results align with our findings on the SPEC CPU benchmarks, where the slowdown relative to ASan was between 19.8% and 36.9% depending on the configurations. During fuzzing, the infrastructure of the fuzzer itself adds a slowdown common to all the configurations (e.g., producing the test cases, analyzing the coverage, etc.), which explains why the relative slowdown is slightly reduced compared to that experiment. Compared to MSan, CombiSan is in fact *faster* when not enabling detection of undefined behavior, and only 14% slower when included.

Since CombiSan provides bug-finding capabilities that match those of ASan, MSan, and UBSan combined, we reason over the *compounded overhead* to contextualize CombiSan's performance. We consider a scenario where we test a target with `N` inputs for each sanitizer category, i.e., we require `N` inputs to be tested for ASan-class bugs, `N` for MSan-class bugs, and `N` for UBSan-class bugs. This means that the total time required to test the software with ASan+UBSan and MSan is the *sum* of their execution times, because we have to run `N` cases with ASan+UBSan, followed by another `N` cases with MSan, totaling `2N`. In contrast, CombiSan achieves the same by only executing a total of `N` cases. Now, we can quantify the performance of the two setups by comparing the slowdown ratios. ASan+UBSan and MSan incur a slowdown of 4.11x and 3.78x over the baseline, respectively. Running them sequentially results in a total slowdown of 7.89x. In comparison, CombiSan (with UB detection) incurs a slowdown of only 4.68x. From this we conclude that CombiSan is 1.7x *faster* $(7.89/4.68)$ than the existing sanitizers. Similarly, running CombiSan without UB detection is *twice* as fast as running ASan and MSan sequentially $((3.27+4.11)/3.72)$.

Impact of optimizations. To understand the benefits of our compile-time optimizations designed to reduce the number of uninteresting uninitialized loads, we measured their impact on the SPEC CPU2006 suite. More specifically, we measured the total number of reported violations, as well as the number of unique violation sites (distinguished by the instruction address). Out of the 19 programs, five had zero violations to begin with, five did not see significant improvements (at most a 2% reduction of violation sites), and the remaining nine are shown in Table 5. We cumulatively enabled each optimization, and report both the absolute and relative changes to contextualize the magnitude of the reductions.

Two programs (`perlbench` and `omnetpp`) highlight the difficulty of static reasoning over execution frequency: reducing the number of unique violation sites by 13% and 24% only impacted the total violation count by 1% or less. Clearly, these

	AFL++	ASan	MSan	UBSan	A+UB	CombiSan no UB			CombiSan with UB		
Subject	exec/sec	vs. AFL	vs. AFL	vs. AFL	vs. AFL	vs. AFL	vs. A	vs. M	vs. AFL	vs. A	vs. M
c-ares	37888	1.34x	1.07x	1.08x	1.46x	1.43x	1.07x	1.34x	1.50x	1.12x	1.41x
guetzli	18509	3.68x	2.87x	1.67x	3.76x	3.38x	0.92x	1.18x	6.23x	1.69x	2.17x
json	22984	2.81x	1.92x	1.11x	3.06x	3.24x	1.15x	1.69x	3.67x	1.30x	1.91x
libxml2	14123	8.67x	6.34x	1.26x	9.29x	9.69x	1.12x	1.53x	11.98x	1.38x	1.89x
openssl	18593	4.92x	21.36x	1.02x	5.20x	5.20x	1.06x	0.24x	5.23x	1.06x	0.24x
pcre2	18996	1.52x	4.57x	2.91x	1.67x	1.59x	1.05x	0.35x	1.89x	1.25x	0.41x
re2	18193	5.77x	1.70x	2.26x	11.69x	6.72x	1.16x	3.95x	8.13x	1.41x	4.78x
woff2	7990	2.49x	13.13x	1.06x	2.63x	4.29x	1.73x	0.33x	6.93x	2.78x	0.53x
geomean		**3.27x**	**4.11x**	**1.44x**	**3.78x**	**3.72x**	**1.14x**	**0.90x**	**4.68x**	**1.43x**	**1.14x**

Table 4: Dataset and results of fuzzing performance evaluation. The table presents the non-sanitized baseline throughput (AFL++ in the table) in terms of executions per second; then, it presents the relative slowdown of every tested configuration. For space constraints, AFL++, ASan, MSan, and UBSan are indicated with AFL, A, M, and UB, respectively.

Program	B	+A	+T	+C	rel%	abs (%)
perlbench	1437	1437	1286	1244	13%	140M (1%)
gcc	5361	1190	1161	977	82%	759M (88%)
mcf	14	0	0	0	100%	55M (100%)
milc	13	0	0	0	100%	5M (100%)
gobmk	325	325	321	160	51%	14M (71%)
libquantum	1	0	0	0	100%	12M (100%)
h264ref	947	42	42	42	96%	700M (99%)
omnetpp	17	17	17	13	24%	18 (0%)
sphinx3	83	1	1	1	99%	351M (99%)

Table 5: Absolute and relative reduction of unique uninitialized load violation sites per optimization (cumulatively). The last column represents the *total* reduction count (non-unique). Table headers indicate the optimizations: external [A]llocs, use [T]aint analysis, [C]onstructors, and the [B]aseline.

violations occur in relatively *cold* parts of the code.

In contrast, the other programs showcase highly successful reductions, for example for `gcc`, where we decreased the absolute violations by 88% (759 million hits), spanning 82% of the unique violation sites (more than 4000 sites removed).

The results also indicate that each optimization has its own benefits. First, marking external allocations as initialized works very well for e.g., `gcc`, `h264ref`, `sphinx3`, etc. Second, marking constructor-operated memory as initialized after invocation is beneficial for `gcc` and `gobmk`. Third, the static taint analysis shows limited efficacy, but remains useful for `gcc` and `perlbench`. Overall, we find that our optimizations successfully reduce the violation count.

7 Limitations

Non-fuzzing context. CombiSan's metadata model is optimized to synergize with fuzzing. Outside of fuzzing, CombiSan works as a standalone sanitizer for addressability issues, uninitialized loads, and UBSan-class bugs. In general, CombiSan functions smoothly for programs that do not contain any (or few) uninitialized loads, e.g., in SPEC CPU2006 there are five programs without any uninitialized loads by default, and three more after applying our optimizations. However, for programs that result in many (either true or false positive) uninitialized loads, CombiSan needs to invoke its slow-path accurate detector to filter out the uninteresting uninitialized loads and thereby avoid generating an impractical number of reports.

Thread safety. By using an 8-to-2 bits mapping for the shadow memory, CombiSan's accesses to the shadow memory are currently not thread-safe. Since multiple bytes of memory map to the same shadow byte, updating the initialization status of multiple bytes that share a shadow byte may result in race conditions. However, we believe this limitation to be acceptable, considering that fuzzing harnesses are usually single-threaded, since fuzzing concurrent applications is challenging [6]. Moreover, increasing the size of the shadow memory to have a 1-to-1-bit mapping (like MSan) would address this issue, at the cost of increased memory overhead.

Kernels. While CombiSan's instrumentation supports kernel targets (e.g., Linux), our current implementation relies on Valgrind, which cannot run on a kernel. However, since kernels typically are *self-contained*, i.e., without external dependencies, CombiSan could use KMSan [9] as an accurate UUM detector without introducing compatibility issues, though with potentially higher costs for re-running executions.

8 Related Work

Bug detectors. There exists a large amount of prior work on software bug detectors, covering various scopes of error (sub)classes. Such sanitizers often detect *spatial* errors [2, 13, 15, 32, 40, 52, 54, 71], *temporal* errors [4, 11, 16, 28, 53, 68], or *both* [25, 27, 31, 46, 51, 61]. Some tools have a narrow scope, for example heap-only [24, 45], or only buffer *overflows* [26, 39]. Highly performant solutions are typically hardware-assisted [8, 29, 43, 60, 73], limiting deployment. Other bug classes also typically have their own sanitizers,

like type confusion [30,34], UUM errors [47,65], or race conditions [59,62]. With CombiSan, we detect a union of bug classes and scopes: we cover heap, stack, and global memory, and detect spatial errors (bounded by redzone size), temporal errors (bounded by quarantine size), UUM errors, and UB errors, in a single solution.

Sanitizer optimization for fuzzing. FuZZan [35], ReZZan [3], and SAND [38] explore alternative sanitizer setups to optimize the performance of *fork-mode* fuzzing. Through optimizations in sanitizer metadata design and management, these efforts manage to reduce the impact of the sanitizer runtime on the (dominant) overhead of forking. Fork-mode fuzzing differs in key architectural aspects from *persistent-mode* fuzzing, which is much faster. We evaluated the latter in the paper as it is the recommended approach when source code is available (i.e., with compiler-based sanitizers) [1], and also represents the default choice for OSS-Fuzz.

Combined sanitizers. Prior research has explored other techniques to combine bug detectors. Some notable designs are at the binary level, like Valgrind [63] and Dr.Memory [5], however, they have limited performance and accuracy [59]. For example, bounds checking is mostly heap-only, and detecting most UB is assumed infeasible [17, 59].

SoftBoundCETS [51] proposes a compiler-based combination of spatial and temporal bug detection, but does not *unify* the metadata, and instead operates independently. CombiSan in contrast retrieves all metadata of an address in one lookup. EffectiveSan [14] mainly detects type confusion bugs, but, as a side effect of having type information, also detects *some* spatial and temporal errors. In contrast, CombiSan detects multiple bug classes without sacrificing accuracy.

Bunshin [70] leverages multi-variant execution to composite incompatible sanitizers: instead of unifying, it synchronizes parallel executions with different sanitizers. This side-steps the interoperability issues between sanitizers, but inherits the compatibility limitations of MSan and those of multi-variant execution itself. For example, multi-variant execution struggles with nondeterministic behavior, common in persistent fuzzing [18]. Moreover, solutions that parallelize inputs across `N` disjoint sanitizers fundamentally either increase the total run time by a factor ~`N` or require ~`N` extra cores—which both cause significant performance overhead in modern core-saturated fuzzing campaigns.

QMSan. We conclude the discussion of related work with QMSan [47], acknowledging the influence its multi-layered design had on CombiSan. Both systems address MSan's compatibility issues. To this end, QMSan trades performance for compatibility by employing dynamic binary instrumentation. CombiSan optimizes both aspects, while also achieving unification with addressability and UB sanitization.

The authors of QMSan [47] mention that their design is compatible with the addressability checks of QASan [17], and they propose future work on optimized concurrent detection of both classes of errors (i.e., UUM and addressability) in binary-level fuzzing. In practice, it is possible to run QMSan alongside the original QASan release [17], which uses a custom DSO. However, instrumentation and shadow memory are added independently, resulting in significant performance and memory overhead. Later, compatibility was lost in the QASan integration in AFL++, which uses its unified DSO.

9 Conclusion

We presented CombiSan, a fuzzing-oriented software sanitizer that combines the bug detection capabilities of ASan, MSan, and UBSan. CombiSan uses a new *unified* shadow memory model that enables simultaneous detection of addressability and initialization issues, while ensuring smooth integration with UBSan. While fuzzing, CombiSan prevents interference in bug detection by deferring bug analysis to test case completion, instead of the common approach of terminating early. Finally, CombiSan also addresses MSan's compatibility issues, thereby unlocking easy-to-deploy fuzzing for UUM bugs. We hope that CombiSan will see adoption in production fuzzing campaigns, where it can improve efficiency both in terms of throughput and detected bugs.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This work was supported in part by the Italian MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU through project SERICS (PE00000014), by Intel Corporation through the "Allocamelus" project, by NWO through project "Theseus" and the Dutch Prize for ICT research, and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

Ethical Considerations

While developing CombiSan, we have carefully considered the ethical implications of our research, which aims to make software more secure by identifying software bugs before they can be exploited. Still, finding bugs in software can represent a security risk that must be handled properly. We thus put in place safeguards to protect the relevant stakeholders.

Stakeholders and impact. The stakeholders involved in our project are the developers and the users of the software we tested for bugs with CombiSan and other sanitizers, including users of software that use our targets as dependencies.

Our research positively impacts the stakeholders, who benefit from our findings by using more secure software.

As any other sanitizer, CombiSan is designed to find bugs in software with the goal of mitigating the risk of bugs being exploited by malicious actors. Still, these solutions can also be used by adversarial entities to find unpatched vulnerabilities. We discourage any such use of CombiSan. We highlight how this dual-use scenario should not discourage the public release of bug-detecting solutions: *security by obscurity* is not, and should never be, an option. Instead, we stress even more the importance of software testing in the development cycle, which can only benefit from solutions like CombiSan to promptly detect and fix bugs before software is released.

Responsible disclosure. After finding a bug with any sanitizer, we first confirmed if the bug had already been reported. In case it was not, we reported it to the project maintainers according to their security policy, and by contacting them privately if such a policy is missing. All reports were made privately (e.g., private bug trackers or security advisories on GitHub), except those for *assimp* and *inchi*, where the maintainers explicitly request bug reports to be done through public GitHub issues. In our reports, we provided as many details as possible about each bug, including a PoC, and we offered our expertise to ease the bug-fixing process. For private reports, we proposed a standard time window of 90 days before reporting any of our findings to the public; in all cases, this time window was sufficient for the developers to take all the actions they deemed necessary.

Open Science

We comply with the open science policy by releasing the CombiSan prototype as open source and partaking in Artifact Evaluation. The prototype consists of a modified LLVM 20.1.0 framework, a modified AFL++ 4.32c project, and an infrastructure to test CombiSan. The artifact can be found at: https://zenodo.org/records/16949365

References

[1] AFl++. AFL++ Documentation - Persistent mode. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md. Accessed: 2025-08-26.

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

[3] Jinsheng Ba, Gregory J. Duck, and Abhik Roychoudhury. Efficient Greybox Fuzzing to Detect Memory Errors. In *37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davit. xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 502–519. IEEE, 2022.

[5] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 213–223. IEEE, 2011.

[6] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342, 2020.

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

[8] Clang/LLVM. Hardware-assisted AddressSanitizer Design Documentation. Online. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[9] Clang/LLVM. Kernel Memory Sanitizer (KMSAN). https://docs.kernel.org/next/dev-tools/kmsan.html. Accessed: 2025-07-03.

[10] Clang/LLVM. UBSan: Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed: 2025-07-03.

[11] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 815–832, 2017.

[12] Dominik Klemba Dominik Czarnota. Understanding AddressSanitizer: Better memory safety for your code. https://blog.trailofbits.com/2024/05/16/understanding-addresssanitizer-better-memory-safety-for-your-code. Accessed: 2025-08-13.

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

[14] Gregory J Duck and Roland HC Yap. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.

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

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

[17] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *2020 IEEE Secure Development (SecDev)*, pages 23–30. IEEE, 2020.

[18] Elia Geretto, Andrea Jemmett, Cristiano Giuffrida, and Herbert Bos. LIBAFLGO: Evaluating and Advancing Directed Greybox Fuzzing.

[19] ghostscript. ghostpdl source code. https://github.com/ArtifexSoftware/ghostpdl/blob/master/pdf/pdf_int.c#L124. Accessed: 2025-08-26.

[20] Google. MSan disabled by default. https://github.com/google/oss-fuzz/issues/6294. Accessed: 2025-07-03.

[21] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed: 2025-07-03.

[22] Google. OSS-Fuzz, sanitizers. https://google.github.io/oss-fuzz/getting-started/new-project-guide/#sanitizers. Accessed: 2025-08-22.

[23] Google. syzkaller - kernel fuzzer, 2016.

[24] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers. In *OSDI*, 2023.

[25] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik Van Der Kouwe, Cristiano Giuffrida, and Herbert Bos. FloatZone: Accelerating Memory Error Detection using the Floating Point Unit. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 805–822, 2023.

[26] Floris Gorter and Cristiano Giuffrida. Canon Pointers: Conditional Delta Tagging for Compatible Buffer Overflow Protection. In *PLAS*, October 2025.

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

[28] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In *2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1307–1322, 2022.

[29] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags. In *IEEE S&P*, 2024.

[30] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. TypeSan: Practical Type Confusion Detection. In *2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 517–528, 2016.

[31] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing Memory Error Detection for Large-scale Applications and Fuzz Testing. In *NDSS*, 2018.

[32] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight Bounds Checking. In *Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.

[33] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik Van Der Kouwe, and Klaus von Gleissenthall. Don't look UB: Exposing Sanitizer-eliding Compiler Optimizations. *ACM on Programming Languages*, 7(PLDI):907–927, 2023.

[34] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient Detection of Type Confusion Errors for C++. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387, 2017.

[35] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 249–263, 2020.

[36] Frederick Boland Jr. and Paul Black. Juliet 1.1 C/C++ and Java Test Suite. In *IEEE Computer*, 2012.

[37] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[38] Ziqiao Kong, Shaohua Li, Heqing Huang, and Zhendong Su. SAND: Decoupling Sanitization from Fuzzing for

Low Overhead . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 255–267. IEEE Computer Society, 2025.

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

[40] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Twelfth European Conference on Computer Systems*, pages 205–221, 2017.

[41] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. PartiSan: Fast and Flexible Sanitization via Run-time Partitioning. In *RAID*, 2018.

[42] Shaohua Li and Zhendong Su. UBFuzz: Finding Bugs in Sanitizer Implementations. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 435–449, 2024.

[43] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *ACM CCS*, 2022.

[44] LifeIsStrange. How to use simultaneously Asan and Msan? https://github.com/google/sanitizers/issues/1039. Accessed: 2025-07-03.

[45] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. CAMP: Compiler and Allocator-based Heap Memory Protection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4015–4032, 2024.

[46] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. GIANTSAN: Efficient Memory Sanitization with Segment Folding. In *ASPLOS*, 2024.

[47] Matteo Marini, Daniele Cono D'Elia, Mathias Payer, Leonardo Querzoni, et al. QMSan: Efficiently Detecting Uninitialized Memory Errors During Fuzzing. In *Network and Distributed System Security (NDSS) Symposium 2025*, 2025.

[48] Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Hammond Pearce, Brendan Dolan-Gavitt, et al. Arvo: Atlas of Reproducible Vulnerabilities for Open Source Software. *arXiv preprint arXiv:2408.02153*, 2024.

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

[50] Microsoft. Project OneFuzz, 2020.

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

[52] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[53] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *2010 international symposium on Memory management*, 2010.

[54] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *ACM on Measurement and Analysis of Computing Systems*, 2(2):1–30, 2018.

[55] OpenSC. OpenSC, security advisories. https://github.com/OpenSC/OpenSC/security/advisories?state=published. Accessed: 2025-07-03.

[56] OSS-Fuzz. UBSan flags. https://github.com/google/oss-fuzz/blob/master/infra/base-images/base-builder/Dockerfile. Accessed: 2025-08-20.

[57] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.

[58] Mathias Payer. The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes. *IEEE Security & Privacy*, 17(1):78–82, 2019.

[59] Joschua Schilling, Andreas Wendler, Philipp Görz, Nils Bars, Moritz Schloegel, and Thorsten Holz. A Binary-level Thread Sanitizer or Why Sanitizing on the Binary Level is Hard. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1903–1920, 2024.

[60] David Schrammel, Martin Unterguggenberger, Lukas Lamster, Salmin Sultana, Karanvir Grewal, Michael LeMay, David M Durham, and Stefan Mangard. Memory Tagging using Cryptographic Integrity on Commodity x86 CPUs. In *2024 IEEE 9th European Symposium*

on Security and Privacy (EuroS&P), pages 311–326. IEEE, 2024.

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

[62] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *workshop on binary instrumentation and applications*, pages 62–71, 2009.

[63] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[64] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[65] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.

[66] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[67] Vincent Ulitzsch, Deniz Scholz, and Dominik Maier. ASanity: On Bug Shadowing by Early ASan Exits. In *2023 IEEE Security and Privacy Workshops (SPW)*, pages 364–370. IEEE, 2023.

[68] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable Use-After-Free Detection. In *Twelfth European Conference on Computer Systems*, pages 405–419, 2017.

[69] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High System-Code Security with Low Overhead. In *2015 IEEE Symposium on Security and Privacy (S&P)*, pages 866–879. IEEE, 2015.

[70] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: Compositing Security Mechanisms through Diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 271–283, 2017.

[71] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: an Efficient Pointer Arithmetic Checker for C Programs. In *5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[72] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs. In *OSDI*, 2021.

[73] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *ASPLOS*, 2019.

[74] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating Address Sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4345–4363, 2022.

A Appendix

Table 6 contains the details of the tested OSS-Fuzz targets. Table 7 displays the bug detection results of ASan, MSan, UBSan, and CombiSan on the Juliet Test Suite. The detection rate for the UBSan-class categories is relatively low, because many test cases require a specific input to trigger the bug. We ran the test cases with one equalized input for simplicity, and looked for any deviations between standalone UBSan and CombiSan enabling UBSan. Table 8 presents the results of the replaying experiment. Table 9 contains various ASan-class CVEs evaluated in prior work, along with the type of memory involved in the bug. Table 10 displays the individual runtime overhead of the evaluated SPEC CPU programs.

Project	Harness	Commit	San
libredwg	llvmfuzz	4b6048d	-
assimp	assimp_fuzzer	b3a47a6	AU
ghostscript	device_psdcmyk_fuzzer	3f12145	A
libdwarf	fuzz_srcfiles	91ab209	-
libucl	ucl_add_string_fuzzer	3e7f023	-
gpac	fuzz_probe_analyze	9d6c670	-
serenity	FuzzICOLoader	bbdbdab	-
opensc	fuzz_pkcs15init	bd7de44	-
inchi	inchi_input_fuzzer	1e9becb	A
libheif	file_fuzzer	6dcbad3	-

Table 6: Harnesses and versions (commit hashes) used in our tests for OSS-Fuzz subjects. The last column indicates which sanitizers each project explicitly requested from the framework between ASan (A), and UBSan (U), or if they requested none (-). Currently, OSS-Fuzz enables ASan and UBSan by default for projects not requesting any sanitizer [22].

Sanitizer	CWE	#cases	#good	#bad
ASan	CWE121	2634	2634	2634
	CWE122	3178	3178	3178
	CWE124	932	932	932
	CWE126	612	612	612
	CWE127	932	932	932
	CWE415	765	765	765
	CWE416	374	374	374
MSan	CWE457	882	882	882
UBSan	CWE190	2299	2262	627
	CWE191	1714	1714	447
	CWE194	540	540	540
	CWE476	285	285	268
	CWE758	494	494	204
	CWE843	74	74	74
CombiSan	CWE121	2634	2634	2634
	CWE122	3178	**3177**	3178
	CWE124	932	932	932
	CWE126	612	612	612
	CWE127	932	932	932
	CWE415	765	765	765
	CWE416	374	374	374
	CWE457	882	**866**	882
	CWE190	2299	2262	627
	CWE191	1714	1714	447
	CWE194	540	540	540
	CWE476	285	**279**	268
	CWE758	494	494	**422**
	CWE843	74	**0**	74

Table 7: Results of CombiSan and other sanitizers on the Juliet Test Suite. Diverging values are marked bold.

	ASan	MSan	UBSan	CombiSan		
Subject				**A**	**M**	**UB**
c-ares	1	0	0	1	0	0
guetzli	0	0	0	0	0	0
json	0	0	0	0	0	0
libxml2	2	16	1	2	16	1
openssl	1	3	0	1	2	0
pcre2	44	14	2	44	7	2
re2	1	2	1	1	2	1
woff2	0	0	0	0	0	0

Table 8: Number of unique crashes found by each sanitizer on the Google FTS subjects by replaying an equalized input. The columns follow the same naming as Table 2.

ID	Type	*San	CombiSan
CVE-2009-1759	stack-buffer-overflow	ASan	✓
CVE-2009-2285	heap-buffer-overflow	ASan	✓
CVE-2013-4243	heap-buffer-overflow	ASan	✓
CVE-2015-8668	heap-buffer-overflow	ASan	✓
CVE-2015-9101	heap-buffer-overflow	ASan	✓
CVE-2016-10095	stack-buffer-overflow	ASan	✓
CVE-2016-10269	heap-buffer-overflow	ASan	✓
CVE-2016-10270	heap-buffer-overflow	ASan	✓
CVE-2017-12858	heap-use-after-free	ASan	✓
CVE-2017-12937	heap-buffer-overflow	ASan	✓
CVE-2017-14407	stack-buffer-overflow	ASan	✓
CVE-2017-14408	stack-buffer-overflow	ASan	✓
CVE-2017-14409	global-buffer-overflow	ASan	✓
CVE-2017-5976	heap-buffer-overflow	ASan	✓
CVE-2017-5977	heap-buffer-overflow	ASan	✓
CVE-2017-7263	heap-buffer-overflow	ASan	✓
OSS-Fuzz #67552	stack-UUM	MSan	✓
OSS-Fuzz #66510	heap-UUM	MSan	✓
OSS-Fuzz #65120	heap-UUM	MSan	✓

Table 9: CombiSan's bug detection results on known CVEs and issues along with the sanitizer used for ground truth.

Program	A	M_0	M_R	C_0	C_R
400.perlbench	4.23	2.41	2.76	5.04	5.63
401.bzip2	1.55	1.88	2.17	1.71	1.81
403.gcc	2.73	2.78	2.93	3.33	3.47
429.mcf	1.22	1.99	2.06	1.41	1.45
433.milc	1.11	1.87	2.03	1.58	1.63
444.namd	1.51	1.90	2.15	1.51	1.96
445.gobmk	1.77	1.77	1.89	1.90	2.11
447.dealII	2.12	2.14	2.37	2.35	2.80
450.soplex	1.57	2.26	2.50	1.76	1.97
453.povray	2.84	2.48	2.62	3.69	4.44
456.hmmer	2.69	2.65	3.25	2.83	3.26
458.sjeng	2.10	2.61	2.78	2.22	2.58
462.libquantum	1.21	2.12	2.65	1.47	1.57
464.h264ref	1.90	2.58	2.77	3.60	3.83
470.lbm	1.18	1.85	1.89	1.93	1.97
471.omnetpp	2.14	2.16	2.33	2.44	2.62
473.astar	1.40	1.53	1.65	1.49	1.60
482.sphinx3	1.65	2.08	2.22	1.70	2.21
483.xalancbmk	2.65	2.39	2.53	4.17	4.34
geomean	**1.853**	**2.156**	**2.365**	**2.243**	**2.490**
500.perlbench_r	2.44	2.60	2.76	2.82	3.39
502.gcc_r	2.97	2.50	3.01	3.56	3.89
505.mcf_r	1.16	1.55	1.64	1.29	1.38
508.namd_r	1.93	2.24	2.92	2.12	2.52
510.parest_r	2.45	2.32	2.75	2.47	3.17
519.lbm_r	1.03	1.66	1.73	1.43	1.90
520.omnetpp_r	3.50	2.85	3.03	3.92	4.20
523.xalancbmk_r	1.82	1.77	1.83	2.76	2.89
525.x264_r	2.02	2.05	2.10	3.31	3.46
531.deepsjeng_r	1.72	1.94	2.02	1.85	2.19
538.imagick_r	1.50	1.60	1.99	2.27	2.53
541.leela_r	1.86	1.79	1.94	1.90	2.54
544.nab_r	1.17	0.87	1.36	1.19	1.27
557.xz_r	1.36	1.85	2.04	1.53	1.57
geomean	**1.808**	**1.900**	**2.159**	**2.166**	**2.476**

Table 10: Individual SPEC CPU runtime overhead results for ASan (A), MSan (M), and CombiSan (C).