Efficient Greybox Fuzzing to Detect Memory Errors

Jinsheng Ba
National University of Singapore
Singapore
jinsheng@comp.nus.edu.sg

Gregory J. Duck
National University of Singapore
Singapore
gregory@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Greybox fuzzing is a proven and effective testing method for the detection of security vulnerabilities and other bugs in modern software systems. Greybox fuzzing can also be used in combination with a *sanitizer*, such as AddressSanitizer (ASAN), to further enhance the detection of certain classes of bugs such as buffer overflow and use-after-free errors. However, sanitizers also introduce additional performance overheads, and this can degrade the performance of greybox mode fuzzing—measured in the order of 2.36× for fuzzing with ASAN—partially negating the benefit of using a sanitizer in the first place. Recent research attributes the extra overhead to program startup/teardown costs that can dominate fork-mode fuzzing.

In this paper, we present a new memory error sanitizer design that is specifically optimized for fork-mode fuzzing. The basic idea is to mark object boundaries using *randomized tokens* rather than disjoint metadata (as used by traditional sanitizer designs). All read/write operations are then instrumented to check for the token, and if present, a memory error will be detected. Since our design does not use a disjoint metadata, it is also very lightweight, meaning that program startup and teardown costs are minimized for the benefit of fork-mode fuzzing. We implement our design in the form of the ReZZan tool, and show an improved fuzzing performance overhead of 1.14-1.27×, depending on the configuration.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Dynamic analysis**.

KEYWORDS

Fuzz testing, greybox fuzzing, memory errors, sanitizers.

ACM Reference Format:
Jinsheng Ba, Gregory J. Duck, and Abhik Roychoudhury. 2022. Efficient Greybox Fuzzing to Detect Memory Errors. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3561161

1 INTRODUCTION

Fuzz testing is a proven method for detecting bugs and security vulnerabilities in real-world software. Fuzz testing can be seen as a biased random search over the domain of program inputs, with the goal of uncovering inputs that cause the program to crash or hang. The biased random search may be guided by an objective function, as in the case of *greybox* fuzzing, which aims to maximize code coverage. Alternatively, the biased random search can be guided by other forms of feedback, such as symbolic formulae, as is the case with *whitebox* fuzzing based on symbolic execution. However, because of the scalability challenges in conducting symbolic execution on real-world software, coverage-guided greybox fuzzing is more widely adopted in the practice of security vulnerability detection. Usually, greybox fuzzing uses instrumentation that is inserted at compile-time, and then random mutations of provided seed inputs are generated and tested over the lifetime of the fuzzing campaign. If a mutated input is found to traverse new instrumented locations/edges, it is retained and prioritized for further mutation. In this way, via repeated mutation, the fuzzing campaign *covers* (a portion of) the program input space, and seeks to find vulnerabilities. Coverage-guided greybox fuzzing is a well-known technique for finding vulnerabilities today and is embodied by popular tools such as AFL [41] and LibFuzzer [14].

The core aim of greybox fuzzing is to detect vulnerabilities in the target program. One important class of bug is *memory errors*, which include *spatial memory errors* such as object-bounds errors (including buffer overflows/underflows), and *temporal memory errors* that access an object after it has been `free()`'ed. Memory errors are a common occurrence in software implemented in *unsafe programming languages*, such as `C`/`C++`, which, for performance reasons, use manual memory management and no bounds checking by default. Furthermore, experience with the industry has shown that memory errors are a common source of security vulnerabilities [21]. This is because memory errors may grant attackers the ability to change the contents of memory, and this can form the basis of information disclosure and control-flow hijacking attacks.

Importantly, memory errors do not necessarily cause the program to immediately crash, and such "silent" memory errors can be difficult to detect during a fuzz campaign. To address this problem, the target program may be instrumented using a *sanitizer*, which uses a program transformation to insert additional code before each memory operation. The additional code checks for object-bounds and use-after-free errors, and if detected, will immediately abort (crash) the program, essentially making memory errors visible. Such instrumentation is quite natural to incorporate into greybox fuzzing. Memory error sanitizers, such as AddressSanitizer [29], have been implemented as part of the LLVM Compiler Infrastructure Project [20] and can be enabled by passing a suitable switch (`-fsanitize=address`) to the compiler.

However, the combination of fork-mode fuzzing and memory error sanitizers is known to suffer from significant performance overheads. For example, under our experiments, the combination of AFL+AddressSanitizer runs at a ∼58% reduction of throughput

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3561161

(execs/sec) compared to AFL alone. Recent work [16] attributes this reduction in performance to the negative interaction between the sanitizer implementation and the fuzzing process. Specifically, traditional memory error sanitizer designs use a *disjoint metadata* to track memory state (e.g., is the memory free or out-of-bounds?). However, maintaining disjoint metadata can negatively impact program startup/teardown costs due to additional overheads, such as an increased number page faults (metadata access) and additional page fault handling costs [16]. This interacts poorly with the fuzzer which must fork a new process for each generated test input.

In this paper, we present a new memory error sanitizer design that is specially optimized for fork-mode greybox fuzzing. Since the disjoint metadata is the main source of additional page faults and associated overheads, we propose a memory error sanitizer design that eliminates metadata based on the idea of *Randomized Embedded Tokens* (RETs) pioneered by tools such as LBC [15] and REST [32]. The key idea is to track memory state by using a special token that is initialized to some predetermined random nonce value. With a suitable run-time environment, out-of-bounds (redzone) and `free()`'ed memory can be "poisoned" by writing the nonce value to these locations. Next, a program transformation inserts instrumentation that checks all memory operations to see if the nonce value is present, indicating a memory error. By representing memory state using the memory itself, we eliminate the additional page faults that otherwise would have been generated by accessing the disjoint metadata, meaning that a RET-based design has the potential to improve the overall sanitizer+fuzzing performance.

That said, existing RET-based sanitizers suffer from various limitations, such as coarse-grained memory error detection [32] or the retention of a disjoint metadata in order to resolve *false detections* [15]. Here, a "false detection" may occur if the nonce value happens to be generated by chance during normal program execution. If this occurs, then legitimate memory may be deemed poisoned, resulting in a "false detection". However, for the application of fuzz testing, we argue that false detections can be tolerated provided the occurrence is sufficiently rare and can be eliminated by automatically re-executing the program with a new random nonce value. In our experiments, no false detection was observed during 19200 hours (~2.2 years) of combined CPU time. Another problem with the basic RET-based design is the memory error detection granularity. This is an artifact of multi-byte token sizes, where some bounds overflows may never reach a token, resulting in a "missed detection". For this we propose *refined boundary check*, which encodes boundary information directly into the token, allowing for byte-precise overflow detection. We show that our design can detect the same class of memory error as traditional sanitizers, such as AddressSanitizer.

We have implemented our design in the form of the (*REt+fuZZing +sANitizer*) ReZZan tool. We show that ReZZan is significantly faster under fork-mode fuzzing, running at a 1.27× overhead over "native" AFL fuzzing without any sanitizer. In comparison, AddressSanitizer incurs an overhead of 2.36×. We also present a simplified configuration without refined boundary checking, which runs at a 1.14× overhead. This configuration exchanges an increased throughput for a modest reduction of error detection capability.

Contributions. In summary, our main contributions are:

- We propose a memory error sanitizer design based on the concept of *Randomized Embedded Tokens* (RETs) [15, 32]. We show that a RET-based sanitizer design can minimize program startup/teardown costs, and can therefore optimize the combination of sanitizers and fork-mode fuzzing.
- We tune our design so that false detections are very rare in practice, meaning that a disjoint metadata (and associated overheads) is not necessary and can be eliminated. We also introduce the notion of *refined boundary checking* for byte-precise memory error detection under a RET-based design.
- We have implemented our design in the form of the ReZZan tool. We have integrated ReZZan with popular greybox fuzzers.
- Our evaluation results against contemporary works (AddressSanitizer (ASAN) [29] and FuZZan [16]) show a clear benefit from our approach, with a modest 1.27× overhead with respect to native throughput, compared to 2.36× for AddressSanitizer.

Open Source Release. The ReZZan tool is available open source:

https://github.com/bajinsheng/ReZZan

An archived artifact [3] and pre-print [4] are also published online.

2 BACKGROUND

Fuzz Testing. Fuzz testing, or "fuzzing", is a method for automated software testing using a (biased) random search over the input space. Popular fuzz testing tools, such as LibFuzzer [14] and the *American Fuzzy Lop* (AFL) [41], are configured with a target program P and an initial seed corpus T. During the fuzzing process, new inputs for P are automatically generated using random mutation over the elements of T. Each newly generated input t is then tested against program P to detect *crashes* (e.g., `SIGSEGV`), indicating a bug or security vulnerability. The fuzzing process can be purely random (e.g., *blackbox* fuzzing) or use information derived from the program P to guide test selection (e.g., *whitebox* or *greybox* fuzzing). In this paper we focus on greybox fuzzers, such as AFL, which collect *branch coverage information* for each newly generated input t. Inputs which increase branch coverage, i.e., cause the execution of new code branches, are deemed "interesting" and will be added into the corpus T. This effectively biases the random search towards inputs that explore more paths, thereby allowing for more bugs to be discovered.

Fuzz testing tools, such as AFL, need to run an instance of the target program P for each newly generated test input t. This is implemented using a *fork server* which is illustrated in Algorithm 1.

```
while recvMsg() do
    pid = fork();
    if pid == 0 then
        main(); // Execute the test case
    else
        waitpid(pid, &status);
        sendMsg(status);
    end
end
```

Algorithm 1: Fork server loop.

The fork server is injected into the target program during program initialization (i.e., before `main()` is called). The fork server essentially implements a simple *Remote Procedure Call* (RPC) loop, where the external fuzzer sends a message for each new input t ready for testing. This induces the target program to fork, creating a child process copy of the original (parent) process. The child process (i.e., where pid `==` 0) calls `main()` to execute the test case t. The parent process waits for the child to finish executing, and communicates the *exit status* (normal execution or *crash*) back to the external fuzz testing tool. For a typical fuzz testing application, the target program P will be forked hundreds or thousands of times per second—once for each generated input t.

Alternatives to the fork server design exist, such as *in-process fuzzing* used by LibFuzzer [14] or AFL's *persistent mode*. However, this design requires the developer to manually create a *driver* which guides the fuzzing process, as well as reset the program state between tests. The fork server design avoids the need for a manual driver and can fully automate the fuzzing process. Most existing literature [5, 7, 12, 16, 18, 27, 43] assumes fork-mode fuzzing.

Memory Error Sanitizers. Like fuzz testing tools, the aim of *sanitizers* [33] is to detect bugs in software. Sanitizers typically use a program transformation and/or a runtime environment modification in order to make bugs more visible. Many popular sanitizers are implemented as compiler extensions (e.g., an *LLVM Compiler Infrastructure* [20] pass) that insert *instrumentation* to enforce safety properties before critical operations. In the case of memory error sanitizers, the instrumentation aims to detect memory errors (e.g., buffer overflows, (re)use-after-free), and will be inserted before all memory read and write operations. Since memory errors will not always cause a crash, a memory error sanitizer is necessary for reliable detection.

Since memory errors are a major source of security vulnerabilities in modern software [21], the detection of memory errors is of paramount importance. As such, many different memory error sanitizer designs have been proposed, including [1, 8–11, 16, 23, 24, 29, 33, 34, 40], each with different performance and capabilities. Each design has its pros and cons in terms of detection capability, performance, and implementation maturity. A summary of some popular memory error sanitizers is shown in Table 1. Here, each memory error sanitizer can detect at least one of five classes of memory error (object *overwrites*, *overreads*, *underwrites*, *underreads*, and *use-after-free* errors) over (*heap*, *stack*, and *global*) objects.

Each sanitizer can be differentiated based on memory error detection capability, with some sanitizers being specialized and others being more general. For example, *stack canaries* are specialized to *stack buffer overwrites*, LowFat [8, 10] and *Lightweight Bounds Checking* (LBC) [15] are specialized to overflows/underflows only, and FreeSentry [39] is specialized to use-after-free errors only. Furthermore, the detection of memory errors may be partial or imprecise even if supported. For example, GWP-ASAN [19] only applies protection to randomly selected heap objects, and LowFat/REST [32] tolerate "small" overflows that do not intersect with other objects.

Sanitizers such as AddressSanitizer [29] aim to be *general*, and are able to detect all classes of memory error with *byte-level precision*, meaning that even "small" overflows will be detected. To do so, AddressSanitizer implements a form of *memory poisoning*

Table 1: Comparison of popular memory error sanitizers.

	Error Detection					*Memory*					*Impl.*		
Sanitizer	*Overwrite*	*Overread*	*Underwrite*	*Underread*	*Use-after-free*	*Heap*	*Stack*	*Global*	*Locality*	*Overhead*	*Maintained?**	`x86_64`*?*	*+Fuzzer?†*
Stack Canaries	◕	○	○	○	○	-	✓	-	●	●	✓	✓	✓
GWP-ASAN [19]	◔	◔	○	○	◔	✓	-	-	●	◕	✓	✓	-
efence [28]	◕	◕	○	○	●	✓	-	-	●	○	✓	✓	-
FreeSentry [39]	○	○	○	○	●	✓	✓	-	○	n/a	-	-	-
LowFat [8, 10]	◕	◕	●	●	○	✓	✓	✓	◑	●	-	✓	-
REST [32]	◕	◕	●	●	●	✓	✓	✓	●	◕	-	-	-
LBC [15]	●	●	●	●	○	✓	✓	✓	◑	◑	-	-	-
MemCheck [24]	●	●	●	●	●	✓	-	-	○	○	✓	✓	-
RedFat [11]	●	●	●	●	●	✓	-	-	●	●	✓	✓	-
AddressSanitizer [29]	●	●	●	●	●	✓	✓	✓	○	○	✓	✓	✓
FuZZan [16]	●	●	●	●	●	✓	✓	✓	○	◑	✓	✓	✓
ReZZan	●	●	●	●	●	✓	✓	✓	●	●	✓	✓	✓

Key: *Error Detection*: ○ = no support, ◔ = partial,random, ◕ = byte imprecise, ● = byte precise; *Mem. Locality*: ○ = disjoint, ◑ = mixed, ● = none/unified; *Mem. Overhead*: ○ = high, >2×, ◑ = mixed, ◕ = mixed,minimal, ● = minimal, n/a = data not avialable

∗ Maintained? = public repository with recent (<1 year) commits.
† +Fuzzer? = known public fuzzer integration.
Impl. = Implementation feature similarity.

as illustrated in Figure 1. The basic idea is to mark memory as "poisoned" if it can only be accessed using a memory error. This includes:

(1) Poisoning a small `REDZONE` region that is inserted between each valid allocated object. The redzone region is used to detect object bounds overflow/underflow errors.

(2) Poisoning `free()`'ed memory (`FREE`) to detect use-after-free errors.

AddressSanitizer uses a runtime support library to (1) insert and poison redzones between allocated objects, and (2) poison `free()`'ed memory. In order to detect memory errors, AddressSanitizer instruments all memory access operations to check for poisoned memory:

```
if (poisoned(p))            // Instrumentation
  error();
*p = v; /* or */ v = *p;   // Access
```

Here, `poisoned`(p) holds iff the corresponding memory at address p is poisoned. If so, `error()` will report the memory error and abort the program.

Memory poisoning is a popular technique implemented by many different sanitizers, such as [15, 16, 24, 29, 32]. The main distinction is *how* memory poisoning is implemented. For example, AddressSanitizer implements memory poisoning by dividing the program's virtual address space into two parts: *application memory* and *shadow memory*. The shadow memory tracks the (un)poisoned state of each byte of application memory. To do so, each 8-byte word in application memory is mapped to a corresponding shadow byte as follows: ($addr_{shadow}$ = $offset_{shadow}$ + ($addr$ / 8)). The shadow byte tracks which of the corresponding application bytes have been poisoned, allowing for byte-precise memory error detection. Shadow memory

is a form of *disjoint metadata*—i.e., an additional metadata that is (1) maintained by the sanitizer, and (2) disjoint from the application memory/data. Disjoint metadata adds memory *overheads* (i.e., the extra space for the shadow memory) and affects memory *locality* (i.e., the application and shadow memory are disjoint).

Alternative implementations of memory poisoning are possible. One example is *Randomized Embedded Tokens* (RETs) as implemented by REST [32] and LBC [15]. Here, poisoned memory is represented by a special *token* that is initialized to some predetermined random nonce value. Memory is deemed "poisoned" if it directly stores the nonce value:

```
poisoned(p) = (*(p - p % sizeof(Token))==NONCE)
```

This approach represents the memory (un)poisoned state using the same memory itself. Since there is no disjoint metadata, both the memory overhead and locality are improved.

That said, a memory poisoning implementation based on RETs may suffer from limitations, such as *false detections* and a coarse-grained memory error detection *granularity*. Here, a false detection will occur if the `NONCE` value happens to collide with a legitimate value created by the program during normal execution, meaning that legitimate access may be flagged as a memory error. To counter this, REST uses a very large token size (a whole 512 bit cache line) combined with a strong pseudo-random source, meaning that collisions are essentially impossible over practical timescales. That said, large (multi-byte) tokens introduce a new problem: a reduced memory error detection granularity. Specifically, due to size constraints, REST only stores tokens in addresses that are a multiple of the token size. This also means that it is necessary to "pad" allocated objects to the nearest token size multiple. For example, given the 512 bit (64 byte) token size of REST, a call to `malloc(27)` will:

(1) pad the allocation size by 64−27=37 bytes,
(2) allocate the object aligned a 64 byte boundary, and
(3) store a token in bytes 64..127 to implement the redzone.

This means that any overflow into the padding bytes (27..63) will not access the token and will therefore not be detected as a memory error. This is a *missed detection* (i.e., *false negative*) meaning that REST is not byte-precise.

The idea of randomized tokens is also used by (and first pioneered by) *Lightweight Bounds Checking* (LBC) [15].[1] Unlike REST, LBC uses a single byte (8 bit) token size, which allows for byte precise memory error detection, but also means that collisions are inevitable. To avoid false detections, LBC implements a hybrid approach that retains a disjoint metadata for distinguishing collisions from legitimate memory errors.

Problem Statement: Fuzzing+Sanitizers. It is natural to combine memory error sanitizers with fuzz testing. The fuzzer will explore the input space, and the sanitizer will detect "silent" memory errors that would otherwise go undetected. In principle, the fuzz testing and sanitizers should work together synergistically, allowing for the detection of more memory error bugs than either tool alone. In practice, however, the combination of fuzz testing and sanitizers suffers from poor performance [16].

[1]LBC also uses a different terminology, with *guard zone value* used in place of *randomized embedded token*.

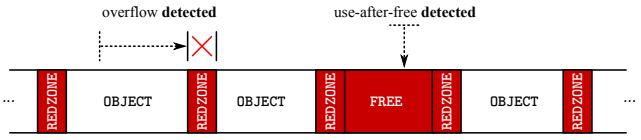

Figure 1: An illustration of memory poisoning. Here, (a) each allocated `OBJECT` is padded with a *poisoned redzone* to detect bounds over/under flows, and (b) free'ed memory is also poisoned to detect use-after-free errors. The memory (un)poisoning operations are implemented using program transformation (for global/stack objects) and a modified memory allocator (for heap objects).

The root cause of the problem lies with the interaction between the *copy-on-write* (COW) semantics of the `fork()` system call, and the initialization/use of any disjoint metadata by the sanitizer. The basic idea of COW is to delay the copying of memory by initially sharing all physical pages between the parent and child process. To do so, all writable memory will be marked as *copy-on-write* by the `fork()` system call, meaning that a *page fault* will be generated when the corresponding page is first written to by the child process. This allows the kernel to intercept writes and copy pages *lazily*, which is useful for optimizing the common `fork+execv` use-case. However, in the case of fuzzing+sanitizers, this design can lead to a proliferation of page faults since memory must be modified in two distinct locations: once for allocated objects and once again for the disjoint metadata. Since page fault handling is a relatively expensive operation, especially for a disjoint (non-local) address space, this can become the main source of overhead. In addition to page faults, AddressSanitizer also introduces other sources of overheads relating to `fork()`, such as the copying of kernel data structures including the *Virtual Memory Areas* (VMAs), page tables, and associated teardown costs. We refer the reader to [16] for a more detailed analysis.

One idea would be to choose an alternative memory error sanitizer with lower memory overheads and higher locality. However, as shown in Table 1, no existing sanitizer design satisfies the dual requirements of optimal memory usage and high memory error detection coverage. For example, some sanitizers, such as stack canaries, GWP-ASAN, *efence*, LowFat, LBC and REST, achieve excellent/good locality and overheads. However, this comes at the cost of a reduced memory error detection coverage and/or the lack of support for standard `x86_64` hardware. Another idea would be to optimize the disjoint metadata representation. For example, FuZZan retains AddressSanitizer's error detection coverage, but can also switch metadata representations to a more compact representation using feedback based on fuzzing performance [16]. Nevertheless, FuZZan's approach still uses a disjoint metadata, meaning that the overheads are mitigated rather than eliminated altogether.

2.1 Our Design

Our aim is to optimize fuzzing+sanitizer performance without reducing memory error detection coverage. To do so, we propose a new sanitizer design based on a variant of *Randomized Embedded Tokens* (RETs) that does not use *shadow map* or other *disjoint metadata* representation. The key idea is that, by tracking the (un)poisoned

state using the memory itself, we avoid any additional page fault (and other overhead) that would be generated by the disjoint metadata initialization and access. Furthermore, since the instrumented check and corresponding memory operation both access the same memory, the overall number of page faults generated by the instrumented program remains roughly equivalent (i.e., within the order of magnitude) as the uninstrumented program, except for a modest increase due the insertion of redzones and quarantines. We argue that a RET-based sanitizer design is optimized for memory *locality*, and this improves the performance of fork-based fuzzing. We summarize the main elements of our design as follows:

Token Size. As shown in Table 1, some existing sanitizers already use a RET-based design. However, the existing tools suffer from various limitations and are not optimized for fuzzing. For example, REST [32] uses a very large token size (512 bits) resulting in imprecise memory error detection. In contrast, LBC [15] uses a very small token size (8 bits), but must retain a disjoint metadata to avoid false detections. We argue that, for the application of fuzzing, some small level of false detections can be tolerated provided that the real errors (i.e., "true" positives) are not overwhelmed. We therefore propose a "medium" token size of 64 bits which is sufficient to avoid false detections in practice without the need for a separate disjoint metadata. For example, assuming a 64 bit token size and one billion randomized writes per second, we would expect the first false detection to occur after an average of ~584.9 years.

Finally, we note that, for the application of fuzzing, false detections can be also mitigated by re-executing the test case with a different randomized `NONCE` value. This process can be automated, similar to how AFL handles false positives due to *hangs*/*timeouts*.

Memory Error Detection Granularity. Another design challenge is the memory error detection granularity. Under the basic RET-design, tokens are stored on token-size aligned boundaries, which means an 8-byte alignment for 64 bit tokens. Although this is an improvement over REST, it nevertheless means that object bounds errors can only be detected within a granularity of 8 bytes. Small overflow of 1..7 bytes into object padding will not be detected (unlike tools such as AddressSanitizer which are byte-precise). To address the issue of granularity, we propose a refinement to the basic RET-based design which additionally encodes object boundary information directly into the token representation itself. This information can then be retrieved at runtime, and compared against the bounds of the memory access, allowing for fine-grained (byte-precise) detection. This enables a similar memory error detection capability compared to the current state-of-the-art memory error sanitizers while still avoiding disjoint metadata.

That said, by encoding boundary information into the token, we must reduce the effective nonce size by 3 bits. This lowers the expected time to the first false detection from centuries to decades (e.g., to ~73.1 years for the example above). We believe this is still a tolerable false detection rate.

Hardware. The final design challenge is a practical implementation. REST [32] is implemented as a non-standard hardware extension, and LBC [15] is specialized to 32bit `x86` systems only. In contrast, our RET-based design is the first to target standard hardware (`x86_64`) and standard fuzzers.

```
1  /* Randomized Embedded Token check */
2  void *ub = ptr + sizeof(*ptr) - 1;
3  void *tptr = ub - (ub % sizeof(Token));
4  Token token = *(Token *)tptr;
5  if (token.random == NONCE)
6      error();
7
8  /* Boundary check */
9  tptr += sizeof(Token);
10 token = *(Token *)tptr;
11 if (token.random == NONCE &&
12     ub % sizeof(Token) > token.boundary)
13     error();
14
15 /* The original memory access */
16 *ptr = val;    or    val = *ptr;
```

Figure 2: Pseudo-code for the *Randomized Embedded Token* (RET) check and the optional *Refined Boundary* check. The RET-check is highlighted in lines 2-6, and the boundary-check is highlighted in lines 9-13.

3 BASIC MEMORY ERROR CHECKING

For ease of presentation, we define *Randomized Embedded Tokens* (RETs) using the following structure type:

```
struct Token { uint64_t random; };
```

For a value t of type `Token` to be a valid RET, the `random` bits must match a predetermined 64 bit randomized constant denoted by the name `NONCE`, i.e., `(t.random == NONCE)`. The `NONCE` constant is initialized once during program initialization using a suitable pseudorandom source. We define RETs as structures to allow for extensions under the refined design (see Section 4).

Instrumentation Schema. As with other memory error sanitizer designs, our underlying approach is to transform the program (e.g., using an *LLVM compiler infrastructure* pass [20]) to insert *instrumentation* before each memory access. The instrumentation is additional code that checks whether the given *safety property* has been violated or not, and if it has, the program will abort. In the case of our sanitizer design, the safety property is that the corresponding accessed memory is not *poisoned*. Later, we combine the instrumentation with a suitable runtime environment that poisons *free* and *redzone* memory in order to enforce memory safety.

The baseline instrumentation schema is highlighted in Figure 2 lines 2–6, and is inserted before each memory access operation represented by line 16. Lines 9–13 contain additional instrumentation that we shall ignore for now. We can define the *range* of a memory access in terms of the *lower bound* (*lb*) and *upper bound* (*ub*):

$$lb..ub = \texttt{ptr} \,..\, \texttt{ptr} + \texttt{sizeof}(\texttt{*ptr}) - 1$$

The lower and upper bounds are the addresses of the first and last byte accessed by the memory operation.

Figure 2 line 2 calculates the upper bound. Line 3 calculates the corresponding *token pointer* (`tptr`) by *aligning* the *ub* to the nearest token-sized multiple. This effectively discards the original alignment of *ub*, meaning that any arbitrary overlap between the memory operation and the token can be detected. Finally, lines 4–6 read the corresponding token from memory and compare the value with the predetermined `NONCE` constant. If the values match, the

corresponding memory is deemed *poisoned*, and the execution of the program will be *aborted* (line 6).

The instrumentation in Figure 2 lines 2–6 consists of pointer arithmetic (lines 2–3), memory dereference (line 4), and an error check (lines 5–6). Importantly, the memory dereference (line 4) will only access memory that is to be accessed anyway by the memory operation (line 16). In other words, the line 4 memory dereference will not generate an extra page fault that would not have been generated by line 16 anyway. In addition to line 4, the error check (lines 5–6) also accesses memory to retrieve the `NONCE` value that is stored in a global variable. Since the `NONCE` is stored in a single location, this will generate at most one additional page fault under normal conditions, so can be treated as a once-off cost.

Runtime Support. To enforce *memory safety*, the runtime environment is also modified in order to *poison* both *redzone* and *free* memory, thereby allowing the instrumentation to detect the error. Each class of object (heap/stack/global) is handled differently.

Heap Allocated Objects. Standard heap allocation functions, i.e., `malloc`, `realloc`, `new`, etc., are replaced by new versions that place *redzone*s around each allocated object. The process is similar to how redzones are implemented in other memory error sanitizers, such as AddressSanitizer, except:

- Poisoning is implemented by writing a `NONCE`-initialized token directly into redzone memory.
- The redzone size is 1 or 2 tokens (depending on alignment).
- The redzone is placed at the end of the object. Underflows are detected using the redzone of the previous object allocated in memory.

For heap objects, we have implemented a simple custom memory allocator that allocates objects *contiguously*, i.e., there are no gaps between objects except for redzones.

For heap *deallocation*, the free'ed object is poisoned by filling the corresponding memory with a `NONCE`-initialized token. Any subsequent access to the object will therefore be detected as an error, i.e., *use-after-free* error. To help mitigate *reuse-after-free* errors (i.e., if a dangling pointer is accessed after the underlying memory was reallocated) our solution also implements a *quarantine* for free'ed objects. A quarantine is essentially a queue for free'ed objects which aims to delay reallocation, making it more likely that reuse-after-free errors will be detected. After an object is removed from the quarantine in order to be reallocated, the corresponding memory will be *zeroed* to "*unpoison*" the memory before use.

Stack Allocated Objects. Stack allocated objects are handled using a program transformation implemented as an LLVM [20] pass (similar to the instrumentation pass). To add redzones to stack objects, the allocation size is first modified to include space for both the original object as well as the redzone memory. The augmented object is then allocated from the stack as per usual, and the redzone memory is poisoned by writing a `NONCE`-initialized token, as with the case with heap-allocated objects. The remaining memory is zeroed to remove any residual token values that may be leftover from previous stack allocations.

Global Objects. Global objects are similarly implemented using an LLVM [20] pass. The idea is the same: the object size is extended to include some additional space for redzone memory, which is poisoned using a `NONCE`-initialized token.

4 REFINED BOUNDARY CHECKING

The basic RET-check of Figure 2 can protect object bounds overflow errors up to a granularity of the *token size*, i.e., `sizeof(Token)`=8 bytes. Since embedded tokens must be aligned to the token size, allocated objects must therefore be *padded* to the nearest 8-byte boundary (see Section 2.1). To address the issue of granularity, we propose a refinement of the original *Randomized Embedded Token* (RET) design. The basic idea is to encode *object boundary information* into embedded tokens in addition to the randomized `NONCE`. This boundary information can be retrieved at runtime and checked against the bounds of the memory access. The refined token design therefore consists of two components:

- *random*: The `NONCE` value, same as before.
- *boundary*: An encoding of the *object boundary* in the form:

$$size \bmod \texttt{sizeof(Token)}$$

where *size* is the object size.

Conceptually, the refined token design is represented by a structure with two bitfields:

```
struct Token {
    uint64_t random:61;  // NONCE
    uint64_t boundary:3; // Boundary encoding
};
```

The *boundary* field must be at least $\log_2 \texttt{sizeof(Token)} = 3$ bits in order to represent all possible boundary values. The *random* field has also been reduced to 61 bits (from 64).

In order to detect overflows into padding, memory access must be instrumented with an additional *boundary check*. The basic idea is illustrated in Figure 3. As before, we assume the allocated `OBJECT` is immediately followed by a redzone poisoned by a `NONCE`-initialized token. In this example, we assume the object *size* is not a perfect multiple of the token size, e.g., $(size \bmod \texttt{sizeof(Token)}) = 5$, meaning that an additional $\texttt{sizeof(Token)}-5 = 3$ bytes of padding is used. Overflows into the padding will not be detected by the basic RET-check alone, since the padding is too small to store a token. To detect such overflows, we instrument the memory access with an additional *boundary check* that:

(1) Examines the *next* word in memory from the *current* word that is to be accessed;
(2) If the *next* word is **not** a token (i.e., the *random* bits do not match the `NONCE`), then the memory access is allowed;
(3) Otherwise if the next word is a token, then we retrieve the *boundary* field and compare it against the memory access range *lb*..*ub*. A memory error detected if the following (Boundary-Check) condition does **not** hold:

$$(ub \bmod \texttt{sizeof(Token)}) \leq boundary$$

Note that, by examining the *next* word in memory, we essentially bypass the problem of the padding having insufficient space. In the example from Figure 3, any memory access that overlaps with the *padding* will not satisfy (Boundary-Check), and will therefore be detected. All other access within the object will be deemed valid,

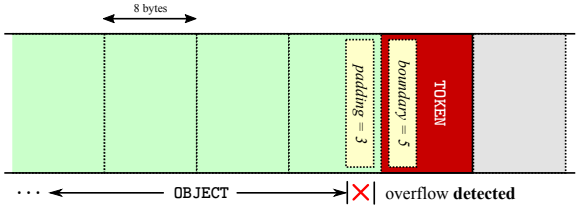

Figure 3: Example of accurate overflow detection with *boundary checking*. Here, in addition to the random value, the token also encodes the object *boundary* modulo the token size, i.e., *boundary*=`sizeof(Token)`−*padding* Any access into *padding* can therefore be detected by comparing the offset (modulo the token size) with the encoded boundary.

either because the next word in memory is not a token, or the token *boundary* is consistent with the memory access range.

Instrumentation Schema. The boundary-check instrumentation is highlighted in Figure 2 lines 9–13. Here, we assume that the memory access (line 16) has already passed the RET-check (lines 2–6), meaning that the *current* word pointed to by the memory access *upper bound* (*ub*) does *not* contain a token. However, it is still possible that the current word contains the object/padding boundary, and that the *ub* exceeds this boundary (an overflow error). The purpose of the *boundary check* is to detect this case.

In order to complete the boundary check, the *next* word in memory must be examined. Here, lines 9–10 load the *next* word after the upper bound (*ub*) into the `token` variable. Line 11 compares `token.random` against the `NONCE`, and if there is a match, the following information can be deduced:

- The *next* word is part of the *redzone* of the current object;
- The *current* word contains the object/padding boundary which is encoded in the `token.boundary` field.

Line 12 therefore checks whether the upper bound (*ub*) exceeds the `token.boundary` value, and if it does, an error is reported and execution is aborted (line 13).

The next word may reside in a different page, which may be inaccessible. This could be handled by disabling the refined check over page boundaries, for a slight loss of precision. Alternatively, all mappings could be extended by a `NONCE`-initialized page. This can be implemented lazily, by using a signal handler to detect boundary-check induced faults, and then extending the corresponding mapping "on-demand".

5 EXPERIMENTAL SETUP

We experimentally validate our sanitizer design in terms of error detection capability, performance, coverage, bug finding capability, flexibility, as well as false detections. In this section, we give an overview of the experimental setup.

We have implemented our design in the form of the *REt+fuZZing +sANitzer* (ReZZan) for the `x86_64`. We will evaluate two main configurations of ReZZan:

- ReZZan: Fine-granularity memory error detection with both RET (Section 3) and byte-accurate boundary checking (Section 4); and
- $\text{ReZZ}_{\text{lite}}$: Reduced-granularity memory error detection with RET-checking only. This version is faster but may not detect some overflows into object padding.

Our ReZZan implementation comprises two parts: an *LLVM Compiler Infrastructure* (LLVM) [20] pass and runtime library. The ReZZan LLVM pass:

- Transforms all memory operations (e.g., `load`/`store`) to insert RET and boundary-checking instrumentation. In the case of $\text{ReZZ}_{\text{lite}}$, the boundary checking is omitted.
- Transform all stack allocations (e.g., `alloca`) and global objects to new versions that are protected by redzones.

The ReZZan runtime library implements replacement heap allocation functions (e.g., `malloc`, `free`, etc.) which insert redzones as well as poisons free'ed memory.

Research Questions. Our main hypotheses are that a RET-based sanitizer design can (1) exhibit a lower performance overhead under fuzz testing environments, and (2) achieve a similar memory error detection capability as more traditional sanitizer designs, such as AddressSanitizer (ASan). We investigate these hypotheses, from the performance and effectiveness perspectives, with the following research questions:

RQ.1 (Detection Capability) Does ReZZan detect the same class of memory errors as ASan?
RQ.2 (Execution Speed) How much faster is ReZZan compared to ASan under fuzz testing environments?
RQ.3 (Branch Coverage) How does the branch coverage for ReZZan compare to ASan?
RQ.4 (Bug Finding Effectiveness) How much faster can ReZZan expose bugs compared to ASan?
RQ.5 (Flexibility) Can ReZZan be used to fuzz huge programs? Is ReZZan compatible with other fuzzers?
RQ.6 (False Detections) What is the false detection rate of ReZZan in real execution environments?

Infrastructure. We run our experiments on an Intel Xeon CPU E5-2660v3 processor with 28 physical and 56 logical cores clocked at 2.4GHz. Our test machine uses Ubuntu 16.04 (64 bit) LTS with 64GB of RAM, and a maximum utilization of 26 cores.

For the baseline, we compare against ASan (LLVM-12) and FuZZan (with dynamic metadata structure switching mode). Both ASan and FuZZan are: (1) maintained, (2) support the `x86_64`, and (3) have existing fuzzer integration (see Table 1). As far as we are aware, ReZZan is the first RET-based design to support the `x86_64` as well as being integrated with a fuzzer. For the fuzzing engine, we use AFL (v2.57b), which is (1) the base of most modern fuzzers, and (2) supported by all sanitizers used in the evaluation. For the memory error detection capability experiments (RQ1), we use the Juliet [25] benchmark suite. Juliet is a collection of test cases containing common vulnerabilities based on a *Common Weakness Enumeration* (CWE), including heap-buffer-overflow, use-after-free, etc. For the execution speed and branch coverage experiments (RQ2, RQ3), we use `cxxfilt`, `nm`, `objdump`, `size` (all from binutils-2.31), `file` (from coreutils version 5.35), `jerryscript` (version 2.4.0), `mupdf` (version 1.19.0), `libpng` (version 1.6.38), `openssl` (version 1.0.1f), `sqlite3` (version 3.36.0), and `tcpdump` (version 4.10.0). Our test

Table 2: Detection capability based on the bad test cases (bug triggered). This checks for false negatives.

CWE (ID)[Bad]	Total	ASan	ReZZan	$\text{ReZZ}_{\text{lite}}$
Stack Buffer Overflow (121)	2860	2856	2860	2380
Heap Buffer Overflow (122)	3246	3189	3246	2724
Buffer Underwrite (124)	928	928	890	890
Buffer Overread (126)	630	610	630	630
Buffer underread (127)	928	928	880	880
Use After Free (416)	392	392	392	392
Pass rate:		99.10%	99.04%	87.89%

Table 3: Detection capability based on the good test cases (bug is not triggered). This checks for false positives.

CWE (ID)[Good]	Total	ASan	ReZZan	$\text{ReZZ}_{\text{lite}}$
Stack Buffer Overflow (121)	2860	2860	2860	2860
Heap Buffer Overflow (122)	3246	3246	3246	3246
Buffer Underwrite (124)	928	928	928	928
Buffer Overread (126)	630	630	630	630
Buffer underread (127)	928	928	928	928
Use After Free (416)	392	392	392	392
Pass rate:		100.00%	100.00%	100.00%

subjects are widely used by recent fuzzing works [2, 5, 7, 18, 27] as well as FuZZan [16]. For the bug finding effectiveness experiments (RQ4), we use Google's fuzzer-test-suite[2], which provides a collection of subjects and bugs for testing fuzzing engines. We use the same bugs as studied by [16]. For the initial seed corpus in (RQ2, RQ3) we use the same number of valid inputs. For the bug finding effectiveness experiments (RQ4), we use the single input provided by the fuzzer-test-suite, or else an empty file if no input is provided. Each trial is run for 24 hours and repeated 20 times (the reported result takes the average).

6 EVALUATION RESULTS

In this section, we present the experimental results for the six research questions from Section 5.

RQ.1 Detection Capability. To evaluate the detection capability of ReZZan and $\text{ReZZ}_{\text{lite}}$ against memory errors, we select the following tests from the Juliet test suite: stack buffer overflow (CWE: 121), heap buffer overflow (CWE: 122), buffer underwrite (CWE: 124), buffer overread (CWE: 126), buffer underread (CWE: 127), and use-after-free (CWE: 416). We exclude test cases where the bug is triggered by data from a socket, standard input, or are only triggered under 32 bit operating system environments. Some other test cases use a random value that determines whether the bug should be triggered. For these cases, we fix the value to ensure that the bug is always triggered. Each test case provides a *bad* and a *good* function. The *bad* function will trigger the bug whereas the *good* function will not, allowing for the detection of false negatives and positives respectively. Since FuZZan has the same detection capability as ASan [16], we focus our evaluation on ASan only.

Table 2 shows the evaluation results of ASan, $\text{ReZZ}_{\text{lite}}$, and ReZZan on all *bad* test cases. The results show that ReZZan and ASan have a similar error detection capability, at 99.04% and 99.10% respectively. There is a slight deviation due to implementation differences, such as library support and redzone size. In contrast,

[2]https://github.com/google/fuzzer-test-suite

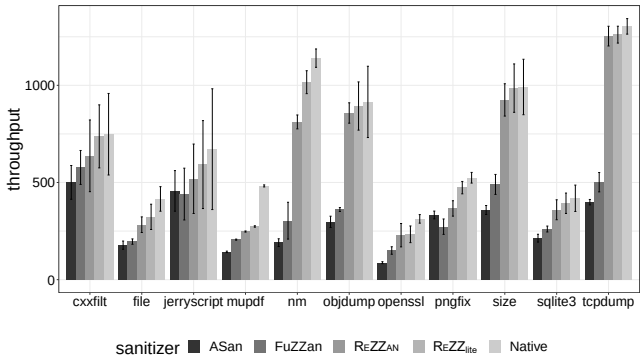

Figure 4: The average throughput (execs/sec) of AFL with four sanitizers and one without (Native).

the memory error detection capability of $\text{ReZZ}_{\text{lite}}$ is somewhat reduced, at 87.89%. This reduction is expected since $\text{ReZZ}_{\text{lite}}$ does not support byte-accurate overflow detection into the padding, meaning that some test cases with (e.g., with off-by-one overflows) will not be detected. As will be seen, $\text{ReZZ}_{\text{lite}}$ trades error detection capability for greater fuzzing throughput. ReZZan and $\text{ReZZ}_{\text{lite}}$ also perform slightly worse than ASan for underflow detection (CWE 124 and 127). However, this is because ASan uses a double-wide (32byte) redzone for stack objects by default. When ReZZan is similarly configured, both ReZZan and ASan can detect 100% of all underflow errors.

Table 3 shows the results of ASan, $\text{ReZZ}_{\text{lite}}$, and ReZZan on all the *good* test cases. For these results, we see that all of ASan, $\text{ReZZ}_{\text{lite}}$, ReZZan pass all test cases.

For memory error bugs (CWE 121, 122, 124, 126, 127, 416) in the Juliet test suite, ReZZan passes 99.04% and $\text{ReZZ}_{\text{lite}}$ passes 87.89% of the *bad* test cases respectively.

RQ.2 Execution Speed. The design of ReZZan has been specifically optimized for performance under fuzz testing environments. To measure the performance, we conduct a fuzzing campaign with AFL+ReZZan against the 11 real-world subjects listed in Section 5. We also compare the performance against ASan [29], FuZZan [16], as well as the "Native" AFL performance with no sanitizer.

The results are shown in Table 4 and illustrated in Figure 4. Here, we arrange the results from lowest to highest throughput. Overall, we see that ASan has the lowest executions per second, with a 57.67% reduction of throughput (2.36× overhead) compared to Native. Under our experiments, FuZZan further improves the ASan throughput, with a 50.11% reduction of throughput (2.00× overhead) compared to Native. In contrast, both ReZZan and $\text{ReZZ}_{\text{lite}}$ show a significantly improved throughput reduction. Under the coarse-grained $\text{ReZZ}_{\text{lite}}$ configuration, which can still detect most memory errors including overflows into adjacent objects, the observed throughput reduction is 12.45% over Native (1.14× overhead). Even with the byte-accurate ReZZan configuration, which uses a more complicated instrumentation, the throughput reduction is a relatively modest 21.34% over Native (1.27× overhead). To compare against ASan, Table 4 also includes a statistical comparison. Here,

Table 4: The average throughput (execs/sec) of AFL with four sanitizers and one without (Native). Each value is averaged over 20 trials and 24 hours. The percentages in the brackets represent the performance loss for each sanitizer compared to the Native campaign. $\hat{A}_{12}$ represents the Vargha Delaney A measure, U represents Wilcoxon signed rank, and *Improv.* (Improvement) is the throughput gain of ReZZan versus ASan.

Subject	ASan	FuZZan	ReZZan	ReZZ$_{\text{lite}}$	Native	ReZZan vs. ASan		
						$\hat{A}_{12}$	U	Improv.
cxxfilt	500.56 (-33.12%)	577.38 (-22.85%)	637.25 (-14.85%)	737.38 (-1.47%)	748.39 (0.00%)	0.76	<0.01	27.31%
file	177.46 (-57.28%)	195.87 (-52.85%)	282.80 (-31.92%)	323.17 (-22.20%)	415.41 (0.00%)	0.95	<0.01	59.36%
jerryscript	456.74 (-31.98%)	440.45 (-34.41%)	519.13 (-22.69%)	592.16 (-11.82%)	671.50 (0.00%)	0.61	0.24	13.66%
mupdf	142.96 (-70.36%)	205.93 (-57.31%)	247.60 (-48.65%)	273.66 (-43.27%)	482.39 (0.00%)	1.00	< 0.01	73.26%
nm	190.46 (-83.29%)	303.46 (-73.38%)	811.71 (-28.79%)	1016.20 (-10.86%)	1139.96 (0.00%)	1.00	<0.01	326.18%
objdump	297.51 (-67.48%)	360.95 (-60.54%)	857.65 (-6.24%)	893.54 (-2.32%)	914.76 (0.00%)	1.00	<0.01	188.28%
openssl	87.27 (-72.09%)	150.00 (-52.04%)	228.83 (-26.83%)	233.39 (-25.37%)	312.73 (0.00%)	1.00	<0.01	162.20%
libpng	332.89 (-36.53%)	272.65 (-48.01%)	366.46 (-30.13%)	473.54 (-9.71%)	524.46 (0.00%)	0.86	<0.01	10.08%
size	358.46 (-63.85%)	490.08 (-50.57%)	924.95 (-6.71%)	985.67 (-0.59%)	991.49 (0.00%)	1.00	<0.01	158.03%
sqlite3	213.43 (-49.03%)	260.59 (-37.77%)	359.67 (-14.11%)	393.12 (-6.12%)	418.77 (0.00%)	1.00	<0.01	68.52%
tcpdump	398.67 (-69.41%)	501.49 (-61.52%)	1253.28 (-3.85%)	1260.82 (-3.27%)	1303.40 (0.00%)	1.00	<0.01	214.37%
Avg Loss:	-57.67%	-50.11%	-21.34%	-12.45%			**Avg:**	118.30%

$\hat{A}_{12}$ is the Vargha Delaney value measuring *effect size* [35] and U is the Wilcoxon rank sum test. With U<0.05, we see that ReZZan outperforms ASan with statistical significance.

Table 5: Average page faults over 1000 runs. The *Factor* compares ReZZan to ASan.

Subject	ASan	FuZZan	ReZZan	ReZZ$_{\text{lite}}$	Native	Factor
cxxfilt	2893.54	2962.79	195.57	195.49	98.55	14.80
file	4131.51	3621.84	437.95	564.88	253.24	9.43
jerryscript	3152.13	3019.65	176.15	176.75	109.67	17.89
mupdf	4475.74	4423.24	647.00	653.71	305.01	6.92
nm	3328.70	3293.03	273.66	277.02	128.81	12.16
objdump	3416.45	3474.61	294.21	294.19	133.71	11.61
openssl	4692.02	1349.85	362.16	373.14	227.90	12.96
libpng	4089.07	3691.06	1204.38	1207.24	914.74	3.40
size	3348.35	3251.60	491.02	494.61	129.01	6.82
sqlite3	3515.80	3464.81	282.90	283.89	71.00	12.43
tcpdump	4007.29	4023.99	420.97	425.87	232.09	9.52
					Avg:	10.72×

Page faults. ReZZan is specifically designed to optimize the startup/teardown costs caused by the sanitizer [16]. One major source of overhead are page faults arising from the interaction between the *copy-on-write* (COW) semantics of `fork()` and disjoint metadata. To quantify the impact, we randomly choose 1000 inputs of each subject generated from the experiments in Table 4 and measure the average number of page faults for each sanitizer. The results are shown in Table 4. Overall we see that the number of page faults is greatly reduced, with a 10.72× reduction over ASan on average, and is comparable to the Native execution. These results are reflected in Table 4, and validate the RET-based design of ReZZan in the context of fuzz testing.

When combined with fuzz testing, the overheads of ReZZan (1.27×) and ReZZ$_{\text{lite}}$ (1.14×) are lower than that of traditional sanitizers ASan (2.36×) and FuZZan (2.00×). The performance of ReZZan and ReZZ$_{\text{lite}}$ is comparable to fuzz testing without any memory error sanitization, as are the number of page faults.

Table 6: The average branch coverage achieved by each fuzzing campaign with four sanitizers and one without.

Subject	ASan	FuZZan	ReZZan	ReZZ$_{\text{lite}}$	Native	ReZZan vs. ASan		
						$\hat{A}_{12}$	U	Impr.
cxxfilt	1284.95	1285.90	1287.90	1290.35	1292.95	0.53	0.78	0.23%
file	1393.50	1401.42	1453.80	1516.60	1548.00	0.52	0.85	4.33%
jerry	8318.90	8295.15	8434.85	8440.80	8485.15	0.65	0.11	1.39%
mutool	2642.25	2629.70	2656.70	2661.40	2676.35	0.77	<0.01	0.55%
nm	1938.75	1938.40	2206.00	2228.70	2260.10	1.00	<0.01	13.78%
objdump	1292.25	1296.53	1348.00	1300.85	1352.70	1.00	<0.01	4.31%
openssl	2944.70	2415.90	3344.50	3363.40	3373.85	0.64	0.15	13.58%
libpng	1939.80	1911.60	1940.15	1936.55	1941.10	0.68	0.06	0.02%
size	1273.70	1293.65	1310.50	1320.15	1328.75	0.95	<0.01	2.89%
sqlite3	11261.05	11088.80	11369.30	11401.25	11585.28	0.58	0.41	0.96%
tcpdump	5960.45	5665.60	7086.95	7168.15	7171.70	1.00	<0.01	18.90%
							Avg:	5.54%

RQ.3 Branch Coverage. Greybox fuzzing aims to increase the branch *coverage* since this can lead to the discovery of new bugs. For the same setup, a higher fuzzer throughput ought to translate into higher code coverage as more tests can be explored for the same time budget. Table 6 shows the average branch coverage achieved by the fuzzing campaigns with all 4 sanitizers and without (Native). We used the `gcov`[3] to measure the branch coverage on the inputs generated in the RQ2. For all benchmarks, the branch coverage for ReZZan and ReZZ$_{\text{lite}}$ are close to that of Native execution, with the coarse-grained ReZZ$_{\text{lite}}$ achieving a slightly higher coverage due to greater throughput. The results for ASan and FuZZan generally show lower coverage, with FuZZan performing similarly to ASan.

On average, ReZZan and ReZZ$_{\text{lite}}$ achieve branch coverage similar to Native. The fuzzing campaign with ReZZan explores 5.54% more code branches than ASan within 24 hours.

RQ.4 Bug Finding Effectiveness. Since ReZZan/ReZZ$_{\text{lite}}$ have lower performance overheads and higher coverage, they ought to be more effective in finding bugs. To test our hypothesis, we use the same benchmark from [16], which consists of 6 errors

[3] https://man7.org/linux/man-pages/man1/gcov.1.html

chosen from the Google fuzzer-test-suite [13]. Here, each test is a C program/library containing a bug, including: `c-ares` (CVE-2016-5180), `libxml2` (CVE-2015-8317), `openssl` (A: heartbleed, and B: CVE-2017-3735), and `pcre2`. The `json` bug triggers an assertion failure, so is not a memory error. Nevertheless, it is interesting to include non-memory error bugs which may also benefit from greater throughput and branch coverage.

Table 7: The average time (in seconds) needed to expose the corresponding bug in the Google fuzzer test suite, averaged over 20 trials. The libxml2 and openssl (B) benchmarks sometimes exceed the 24 hour timeout, leading to a partial result (*) averaged over 8 and 4 successful trials respectively. Here, (–) means the bug was not exposed in any run.

Subject	ASan	FuZZan	ReZZan	ReZZ$_{\text{lite}}$	ReZZan vs. ASan		
					$\hat{A}_{12}$	U	Factor
c-ares	80.00	47.65	22.65	171.95	0.93	<0.01	3.53
json	485.70	410.70	320.05	148.85	0.67	0.07	1.52
libxml2*	>29328.75	>21462.88	>6301.00	>6318.63	1.00	<0.01	4.65
openssl (A)	1736.40	223.50	210.15	219.25	0.95	<0.01	8.26
openssl (B)*	>26589.50	>21431.00	>12750.00	–	1.00	<0.01	2.09
pcre2	7994.80	6438.60	3900.30	3090.95	0.94	<0.01	2.05
						Avg:	3.68×

The results are shown in Table 7. Overall we see that ReZZan can expose the corresponding bugs faster than ASan and FuZZan. Interestingly, some bugs (`json`, `pcre2`) are exposed faster using the coarse-grained ReZZ$_{\text{lite}}$ configuration rather than the fine-grained ReZZan configuration. These benchmarks are either not memory errors (`json`), or are overflows beyond the object padding (`pcre2`), and therefore benefit more from higher throughput. Other bugs can only be detected by ReZZan and not ReZZ$_{\text{lite}}$. For example, consider the `openssl` (B) bug (CVE-2017-3735), which occurs in the following function:

```
unsigned X509v3_addr_get_afi(IPAddressFamily *f) {
    return (f != NULL && ...
            ? (f->addressFamily->data[0] << 8) |
               f->addressFamily->data[1] : 0);
}
```

At runtime, it is allowable for `f->addressFamily->data` to point to a single byte array, meaning that accessing index `1` results in a single-byte overflow. Since this (small) overflow only affects allocation padding, it is undetectable by the coarse-grained ReZZ$_{\text{lite}}$ and is never exposed. This example demonstrates how fine-grained ReZZan can detect more bugs in practice, and we therefore recommend ReZZan as the default configuration.

ReZZan exposes bugs around 3.68× faster than ASan. ReZZan can also detect more bugs than ReZZ$_{\text{lite}}$ in practice.

RQ.5 Flexibility. For this research question, we run additional experiments to demonstrate the overall flexibility of ReZZan with respect to fuzzer support, fuzzing modes, and scalability. These results are intended to augment the main results of Table 4.

Fuzzer Support. We chose AFL for our main benchmarks since it is natively supported by both ASan and FuZZan. However, ReZZan is not intended to be limited to one specific fuzzer or fuzzer version. To demonstrate this, we have also integrated ReZZan into AFL++[12], which is a more modern fuzzer derived from standard AFL. We conduct a set of limited experiments with AFL++ against ASan, ReZZan, ReZZ$_{\text{lite}}$ and Native (FuZZan does not provide AFL++ support) and 6 subjects (`file`, `nm`, `objdump` `libpng`, `openssl`, `sqlite3`) from our main benchmarks (Table 4). We select subjects that (1) are included in FuzzBench framework [4], and (2) have a fuzzing harness that executes the similar program logic in both persistent (in-memory) and fork fuzzing modes.

Table 8: Average throughput (execs/sec) of AFL++.

Subject	ASan	ReZZan	ReZZ$_{\text{lite}}$	Native	vs. ASan	
					ReZZan	ReZZ$_{\text{lite}}$
file	284.71	431.69	487.774	725.87	51.62%	71.32%
nm	374.33	907.31	922.43	1072.73	142.38%	146.42%
objdump	375.50	897.62	998.35	1061.97	97.79%	108.29%
libpng	929.07	1915.70	2038.84	2223.96	106.20%	119.45%
openssl	191.95	212.48	238.054	395.75	10.70%	24.02%
sqlite3	128.48	398.28	412.974	500.90	209.99%	221.43%
				Avg:	103.12%	115.15%

The Table 8 shows the average throughput of AFL++ for both sanitizer and native execution. We note that AFL++ implements several optimizations and improvements over standard AFL [12], and generally achieves an overall higher throughput. Nevertheless, ReZZan achieves an overall improvement of 103.12% versus ASan which is consistent with our main results. The results also show that the performance of ReZZan is not tied to one specific fuzzer, and that ReZZan can be integrated into other fuzzers.

Persistent (In-Memory) Mode Fuzzing. In the interest of completeness, we also tested ReZZan against (a.k.a., *in-memory*) mode fuzzing. Persistent mode aims to eliminate (or significantly reduce) the reliance on `fork()` to reset the program state between tests. To do so, persistent mode relies on a developer-provided *test harness* to manually reset the program state. Unlike fork-mode fuzzing, persistent-mode is not automatic, so is not the default in standard fuzzing tools such as AFL and AFL++.

For this experiment, we use the test harnesses provided by FuzzBench. Overall we measured a modest reduction in performance of -19.31% for ReZZan and -11.47% for ReZZ$_{\text{lite}}$ compared to ASan using the same subjects as Table 8. Since ReZZan is specifically optimized for fork-mode fuzzing and uses a more complex instrumentation (see Figure 2), a reduction in performance for persistent mode fuzzing is the expected result. Nevertheless, the results show that (1) ReZZan can be applied to both fork and persistent modes, and (2) the asymptotic performance of ReZZan over longer runtimes will eventually approach that of ASan. These results also show that FuZZan-style metadata switching may only have a marginal benefit under our sanitizer design.

Table 9: Average throughput (execs/sec) of AFL with sanitizers and native against Firefox (fork-mode).

Target	ASan	ReZZan	ReZZ$_{\text{lite}}$	Native	vs. ASan	
					ReZZan	ReZZ$_{\text{lite}}$
ContentParentIPC	0.73	1.55	1.57	3.02	112.60%	114.86%
StunParser	0.73	1.56	1.58	3.04	113.31%	116.43%
				Avg:	112.95%	115.64%

[4] https://github.com/google/fuzzbench

Scalability. To test the *scalability* of ReZZan we test the Firefox browser version 91.3.0 (extended support). To do so, we integrate ReZZan into the Firefox-customized version of AFL. Since the Firefox project only supports fuzzing individual components (rather than whole program fuzzing) so we select two targets (namely, `ContentParentIPC` and `StunParser`) from Firefox-specific WebRTC/IPC subsystems respectively. For these experiments we use fork-mode fuzzing.

The results are shown in Table 9. Given the size of Firefox, the overall fuzzing throughput is much slower compared to the other benchmarks. Nevertheless, both ReZZan and $\text{ReZZ}_{\text{lite}}$ still outperform ASan with 112.60% and 114.86% improvement respectively. These results are consistent with the other benchmarks, and demonstrate that ReZZan is scalable.

RQ.6 False Detections. The ReZZan sanitizer design allows for a small chance of false detections. Our experiments comprise more than 19200 hours (~2.2 years) of combined CPU time, during which no false detection was observed. This result is in line with expectations, where decades of CPU time would be required before we expect to observe the first false detection.

No false detections were observed for ReZZan and $\text{ReZZ}_{\text{lite}}$ during 19200 hours (~2.2 years) of CPU time.

7 RELATED WORK

We briefly summarize the related work in this section.

Memory Poisoning. As described in Section 2, one common approach (and also used by ReZZan) is to *poison* memory that should not be accessed. This approach has been implemented by many tools [6, 15, 16, 24, 29, 31, 32]. Most existing tools implement memory poisoning using disjoint metadata, as opposed to the RET-based design of ReZZan. However, as shown by our experiments, disjoint metadata can lead to high performance overheads under fuzzing environments.

Guard Pages. Another approach is to insert inaccessible guard pages between memory objects [22, 28]. Accessing a guard page will trigger a memory fault (`SIGSEGV`) and terminate the program. Both *efence* [28] and GWP-ASAN [19] implement this approach. However, guard pages also mean that each protected object must reside in its own page of virtual memory—possibly leading to very high memory overheads.

Canaries. The ReZZan design has some similarities with *stack canaries* [36]. Stack canaries aim to detect stack-buffer-overwrites by delimiting stack buffers with a randomized canary value. The idea has also been generalized to the heap [38]. However, this approach is not instrumentation-based, and is limited to stack/heap-overwrites only. In contrast, ReZZan can detect more classes of memory error, including overreads and use-after-free.

Low Fat Pointers and Pointer Tagging. Another idea is to encode metadata within the pointer representation itself, such as with *low fat pointers* [8, 10]. Unlike ReZZan, this is limited to object bounds errors only, and is not byte-accurate. HWAsan [30] also tags each pointer with a random value that is associated with a given object. However, this approach still uses a disjoint meta data, and assumes a compatible instruction set architecture.

Probabilistic Methods. DieHarder [26] randomizes the heap object layout in order to mitigate against memory errors. This approach is probabilistic, detects heaps errors only, and is primarily intended for hardening rather than bug detection via fuzz testing. In contrast, ReZZan is designed to protect heap/stack/global objects with minimal fuzzing overheads.

Improving Sanitizer Performance. ASAP [37] removes checks in code that is more often executed. PartiSan [17] creates different versions of the target program, in which some are more sanitized while others are not. These approaches essentially trade error coverage for improved performance. SANRAZOR [42] and ASAN-- [43] aim to only remove redundant checks, thereby preserving coverage. Such optimizations are orthogonal to ReZZan and may be integrated as future work.

Improving Fuzzing Performance. FuZZan [16] similarly aims to optimize the combination of fuzz testing with sanitizers. Unlike ReZZan, FuZZan still uses disjoint metadata, but may use more compact representation (based on *RB-trees*) to minimize startup/teardown costs. This approach can be more general, since different kinds of metadata can be supported, whereas ReZZan is specialized to the same class of memory errors that are covered by ASan. Under our experiments, the disjoint metadata-free design of ReZZan further improves the throughput of fork-mode fuzzing.

8 DISCUSSION

It is natural to combine fuzz testing with state-of-the-art memory error sanitizers, such as AddressSanitizer (ASan). However, for fork-mode fuzzing, it has been shown that such a combination leads to poor performance [16]. The underlying cause relates to the heavyweight metadata structures maintained by the sanitizer, leading to high program startup and teardown costs.

In this paper we introduced ReZZan—a lightweight sanitizer design based on *Randomized Embedded Tokens* (RETs). The basic idea is to poison memory directly using a randomized nonce, rather than relying on disjoint metadata. We also show that the basic RET design can be further refined with an encoding of object boundary information—allowing for byte-accurate memory error detection without a significant difference in performance. By eliminating the use of disjoint metadata, we show that ReZZan achieves a modest performance overhead of 1.27× (or 1.14× for coarse-grained checking) under fuzz testing environments, compared to a 2.36× overhead for the state-of-the-art (ASan). Our result helps to remove one of the main impediments to the fuzzer+sanitizer combination, and enhances the overall effectiveness of memory error detection during fuzz campaigns.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful suggestions. This research was partially supported by the National Research Foundation Singapore (National Satellite of Excellence in Trustworthy Software Systems).

REFERENCES

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors. In *USENIX Security Symposium*. USENIX.

[2] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *NDSS*. Internet Society.

[3] J. Ba. 2022. *The ReZZan Artifact*. https://doi.org/10.5281/zenodo.6965396

[4] J. Ba, G. Duck, and A. Roychoudhury. 2022. Fast Fuzzing for Memory Errors. https://arxiv.org/abs/2204.02773

[5] M. Böhme, V. Pham, and A. Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Computer and Communications Security*. ACM.

[6] D. Bruening and Q. Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Code Generation and Optimization*. IEEE.

[7] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Security and Privacy*. IEEE.

[8] G. Duck and R. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Compiler Construction*. ACM.

[9] G. Duck and R. Yap. 2018. EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++. In *Programming Language Design and Implementation*. ACM.

[10] G. Duck, R. Yap, and L. Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Network and Distributed System Security Symposium*. Internet Society.

[11] G. Duck, Y. Zhang, and R. Yap. 2022. Hardening Binaries against More Memory Errors. In *European Conference on Computer Systems*. ACM.

[12] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Workshop on Offensive Technologies*. USENIX.

[13] Google. 2022. *Google Fuzzer Test Suite*. https://github.com/google/fuzzer-test-suite

[14] Google. 2022. *LibFuzzer – A Library for Coverage-guided Fuzz Testing*. https://llvm.org/docs/LibFuzzer.html

[15] N. Hasabnis, A. Misra, and R. Sekar. 2012. Light-weight Bounds Checking. In *Code Generation and Optimization*. ACM.

[16] Y. Jeon, W. Han, N. Burow, and M. Payer. 2020. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *Annual Technical Conference*. USENIX.

[17] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz. 2018. PartiSan: Fast and Flexible Sanitization via Run-time Partitioning. In *Research in Attacks, Intrusions, and Defenses*. Springer.

[18] Y. Li, B. Chen, M. Chandramohan, S. Lin, Y. Liu, and A. Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Foundations of Software Engineering*. ACM.

[19] LLVM. 2022. *GWP-ASan*. https://llvm.org/docs/GwpAsan.html

[20] LLVM. 2022. *The LLVM Compiler Infrastructure*. https://llvm.org

[21] Microsoft. 2019. *Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape*. https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL

[22] Microsoft. 2022. *Using Page Heap Verification to Find a Bug*. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/example-12---using-page-heap-verification-to-find-a-bug

[23] S. Nagarakatte, Z. Santosh, M. Jianzhou, M. Martin, and S. Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Programming Language Design and Implementation*. ACM.

[24] N. Nethercote and J. Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Programming Language Design and Implementation*. ACM.

[25] NIST. 2022. *Juliet Test Suite for C/C++ v1.3*. https://samate.nist.gov/SARD/testsuite.php

[26] G. Novark and E. Berger. 2010. DieHarder: Securing the Heap. In *Computer and Communications Security*. ACM.

[27] H. Peng, Y. Shoshitaishvili, and M. Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *Security and Privacy*. IEEE.

[28] B. Perens. 2022. *Electric Fence Malloc Debugger*. https://linux.die.net/man/3/efence

[29] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Annual Technical Conference*. USENIX.

[30] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov. 2018. Memory Tagging and how it Improves C/C++ Memory Safety. In *Hot Topics in Security*. USENIX.

[31] J. Seward and N. Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Annual Technical Conference*. USENIX.

[32] K. Sinha and S. Sethumadhavan. 2018. Practical Memory Safety with REST. In *Computer Architecture*. IEEE.

[33] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. 2019. SoK: Sanitizing for Security. In *Security and Privacy*. IEEE.

[34] E. Stepanov and K. Serebryany. 2015. MemorySanitizer: Fast Detector of Uninitialized Memory use in C++. In *Code Generation and Optimization*. IEEE.

[35] A. Vargha and H. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[36] P. Wagle and C. Cowan. 2004. StackGuard: Simple Stack Smash Protection for GCC. In *GCC Developers Summit*.

[37] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. 2015. High System-code Security with Low Overhead. In *Security and Privacy*. IEEE.

[38] G. Watson. 2022. *Dmalloc*. https://dmalloc.com/

[39] Y. Younan. 2015. FreeSentry: Protecting Against Use-after-free Vulnerabilities due to Dangling Pointers. In *Network and Distributed System Security*. Internet Society.

[40] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and Wouter W. Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Computer and Communications Security*. ACM.

[41] M. Zalewski. 2022. *American Fuzzy Lop*. https://lcamtuf.coredump.cx/afl

[42] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su. 2021. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs. In *Operating Systems Design and Implementation*. USENIX.

[43] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu. 2022. Debloating Address Sanitizer. In *Security Symposium*. USENIX.