FloatZone: Accelerating Memory Error Detection using the Floating Point Unit

Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos, *Vrije Universiteit Amsterdam*

https://www.usenix.org/conference/usenixsecurity23/presentation/gorter

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

FloatZone: Accelerating Memory Error Detection using the Floating Point Unit

Floris Gorter*
f.c.gorter@vu.nl

Enrico Barberis*
e.barberis@vu.nl

Raphael Isemann
r.isemann@vu.nl

Erik van der Kouwe
vdkouwe@cs.vu.nl

Cristiano Giuffrida
giuffrida@cs.vu.nl

Herbert Bos
herbertb@cs.vu.nl

Vrije Universiteit Amsterdam
* Equal contribution joint first authors

Abstract

Memory sanitizers are powerful tools to detect spatial and temporal memory errors, such as buffer overflows and use-after-frees. Fuzzers and software testers often rely on these tools to discover the presence of bugs. Sanitizers, however, incur significant runtime overhead. For example, AddressSanitizer (ASan), the most widely used sanitizer, incurs a slowdown of 2x. The main source of this overhead consists of the sanitizer *checks*, which involve at least a memory lookup, a comparison, and a conditional branch instruction. Applying these checks to confirm the validity of the memory accesses in a program can greatly slow down the execution.

We introduce FloatZone, a compiler-based sanitizer to detect spatial and temporal memory errors in C/C++ programs using lightweight checks that leverage the Floating Point Unit (FPU). We show that the combined effects of "lookup, compare, and branch" can be achieved with a single floating point addition that triggers an underflow exception in the case of a memory violation. This novel method to detect illegal accesses greatly improves performance by avoiding the drawbacks of traditional comparisons: it prevents branch mispredictions, enables higher instruction-level parallelism due to offloading to the FPU, and also reduces the cache miss rate due to the lack of shadow memory.

Our evaluation shows that FloatZone significantly outperforms existing systems, with just 37% runtime overhead on SPEC CPU2006 and CPU2017. Moreover, we measure an average 2.87x increase in fuzzing throughput compared to the state of the art. Finally, we confirm that FloatZone offers detection capabilities comparable with ASan on the Juliet test suite and a collection of OSS-Fuzz bugs.

1 Introduction

Sanitization for memory safety has become a standard technique for bug discovery in software testing in general and fuzzing in particular. Driven by the many security incidents involving memory errors in system software, many bug detectors have been proposed [7, 15, 17--19, 27, 36, 43, 50], with the most widespread (compiler-based) sanitizer being AddressSanitizer (ASan) [39]. By placing "redzones" around memory allocations and checking for every load and store if they hit the redzone and are therefore invalid, ASan eliminates contiguous buffer overflows entirely and other spatial memory errors probabilistically. In addition, it uses memory quarantining to detect temporal memory errors. Sanitizers are crucial for identifying potentially exploitable bugs in unsafe languages like C and C++. Unfortunately, the checks inserted by sanitizers incur a significant runtime overhead, negatively impacting, say, the number of executions in a fuzzing campaign.

To reduce the overhead, researchers have proposed to eliminate checks as much possible [29, 46, 47, 49, 51], or to optimize the associated management of metadata [16, 18, 24, 26]. However, the bulk of the overhead is still caused by the checks themselves. For instance, a recent analysis attributes approximately 80% of ASan's overhead to the checks [51]. While hardware extensions can help bring down the cost [30, 41, 50], they are not always available and often the checks and/or associated metadata management are still expensive. Moreover, many of the solutions that promise cheap checks limit their security guarantees to either (specific flavors of) temporal or spatial memory safety [9, 15, 27], but not both. Since the techniques are often incompatible, merging them to provide both spatial and temporal memory safety is very expensive [33, 50].

In this paper, we optimize the checks for buffer over- *and* underflows, such as performed by ASan [39] and similar redzone-based solutions [6, 18, 51]. Like LBC [18], we use in-band redzones containing poison values for performance and, like ReZZan [6], we raise an alarm immediately upon a load or store to such poisoned areas, as we weed out false positives in a separate verification stage. Moreover, we ensure that our solution integrates well with modern quarantining techniques to provide temporal memory error detection.

The question we ask is: What makes the checks so costly and what can we do about it? Without hardware support, checking for a security property consists of a combination of compare and branch instructions. For instance, on every load and store ASan looks up the corresponding metadata,

compares it to see if it is part of a redzone and branches to exit code that raises an alarm if this is the case. The branch instruction pollutes the branch predictor and contends with the application for CPU execution units. In addition, the accesses to shadow memory add pressure on the caches and TLBs.

The key insight in this paper is that sanitizer checks never fail in the normal case and should add little overhead except in the event of a violation of memory safety. In an ideal world, the sanitizer should use a special, fast instruction that is branchless, does not contend with the application for CPU execution units, and checks the validity of memory implicitly—raising an exception upon a violation. While modern CPUs lack such a targetted instruction, we will show that they do have instructions that *approximate* exactly this behavior.

In particular, we find that a *floating point addition* can be made to generate an exception if it processes redzone data. We achieve this by configuring a single floating point addition to result in an *underflow exception* only if one of the operands is equal to our redzone *poison* value. By instrumenting vulnerable loads/stores with the addition, we ensure that redzone accesses raise an alarm. Moreover, the addition is fast and branchless and executes on an execution unit that is underutilized in most programs. As a result, the solution ensures high instruction-level parallelism and much better performance than prior techniques. While we focus primarily on Intel x86, we also show such benefits generalize to other architectures.

Using the floating point checks, we implement FloatZone, an ASan-like sanitizer for spatial and temporal memory errors. We opt for a design that solely relies on in-band redzones to promote high memory locality. Omitting *shadow memory* results in high-performance fuzz testing [6]. The tradeoff for such performance gains is accepting some false positives, where accesses to regular program memory cannot be distinguished from redzones. Nonetheless, in software testing, dealing with (infrequent) false positives is not problematic, as bugs can trivially be confirmed during triaging by using an oracle (e.g., ASan) to provide ground truth.

In our evaluation, we investigate the performance of using floating point additions to perform branchless comparisons. Using the SPEC CPU2006 benchmarking suite, we show that the FPU on commodity hardware grows faster with every generation and floating point checks are now twice as fast as the equivalent comparison-and-branch instructions. As a result, FloatZone reports a geomean runtime overhead on SPEC CPU2006 and CPU2017 of 36.4% and 37.0% respectively, which is significantly faster than existing systems. The tradeoff for our fast checks is accepting slightly reduced detection guarantees, namely FloatZone cannot detect underflows up to three bytes. Nonetheless, we show that FloatZone provides security guarantees comparable to ASan on the Juliet test suite, the Linux Flaw project, and a collection of OSS-Fuzz bugs. We also evaluate the increase in performance in fuzzing when using FloatZone as a sanitizer. FloatZone provides an average increase in throughput of 2.87x compared to ASan-- [51].

Contributions We make the following contributions:

- We introduce a novel method to express a comparison operation using floating point arithmetic.
- We show that these *float checks* are significantly faster than traditional comparisons.
- We present FloatZone: a design to use float checks to detect both spatial and temporal memory errors.
- We implement a prototype of FloatZone and we evaluate it against the state of the art sanitizers.

Availability https://github.com/vusec/floatzone

2 Background

2.1 Floating Point Exceptions

The IEEE-754 [1] standard defines five possible exceptions that can arise from Floating Point (FP) operations: Invalid Operation, Division by Zero, Overflow, Underflow, and Inexact. All main architectures (Intel [20], AMD [4], and ARM [5]) offer hardware support for synchronous FP exceptions, together with the possibility to provide user-defined exception handlers. On these architectures, an underflow exception happens when the result of a FP operation is denormal, i.e., the result is so small that it cannot be represented in the normal FP form. Denormals are FP numbers with the smallest possible exponent (2^{-126} for single precision), and the mantissa with an implicit leading zero (the implied leading digit is one for normal numbers). For example, when subtracting $1.0 \cdot 2^{-126}$ from $1.5 \cdot 2^{-126}$, the resulting value $0.5 \cdot 2^{-126}$ can only be represented in denormal form since, without the possibility to use an exponent smaller than 2^{-126}, a leading zero is needed in the mantissa. If so configured, this results in an exception to warn about loss of precision caused by the denormal representation. FP exceptions are disabled by default, but the user can enable them by setting the corresponding FP Control Register (e.g., `MXCSR` on x86-64). For performance reasons [4], x86-64 architectures do not fully respect the IEEE-754 standard. They generate underflow exceptions only with flush-to-zero (FTZ) enabled, zeroing out every underflow result.

2.2 Memory Sanitizers

Memory sanitizers are tools to detect memory violations in programs developed in unsafe languages such as C and C++. The introduced memory safety guarantees are usually divided in two main categories: (i) *Temporal safety*: all memory accesses to an object must happen during its lifetime. For example, use-after-free and double-free bugs are violations of temporal safety. (ii) *Spatial safety*: all memory accesses must occur within bounds of the referenced object. For example, heap and stack buffer overflows are violations of spatial safety.

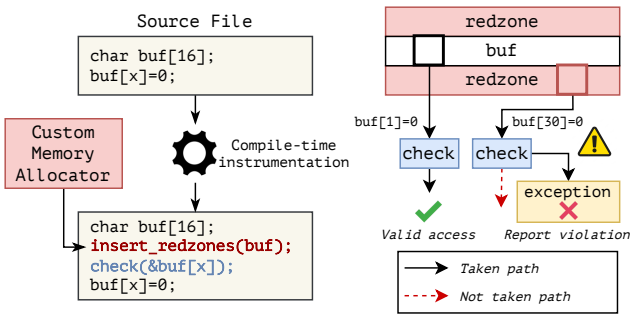

Figure 1: Overview of FloatZone's components and workflow.

A common method to achieve memory safety is through the use of *redzones*, where the sanitizer inserts padding between memory objects. The sanitizer ensures memory accesses within the redzone fault, thereby detecting spatial memory errors that do not skip over the redzone. Deallocated memory can be similarly guarded to detect temporal memory errors until reallocation. We will use the term redzone in a broad sense, including deallocated memory. In order to implement this functionality, sanitizers accompany every memory access with a runtime check for validity. Note that memory sanitizers are detection tools, and are generally not suitable to be used as defense mechanisms against exploitation.

Traditionally, sanitizers rely on *shadow memory*, which is a separate memory region that stores redzone metadata of the application to provide the ground truth on which memory areas are accessible. The shadow memory has to be managed upon allocation and deallocation of memory. Alternatively, sanitizers may use in-band poisoning of redzones. Initializing redzones with an (uncommon) poison value, they check loads and stores to see if they access such poisoned areas. Of course, false positives may arise if the program legitimately uses the poison value. The sanitizer may weed out false positives immediately with a slow check (e.g., using shadow memory [18]), or leave them for later, to be checked in a separate verification step [6]. Commonly, in both cases temporal safety is provided by marking heap objects as redzones upon deallocation, combined with a *quarantine* that delays re-using the redzoned memory region for new objects.

3 Overview

FloatZone detects spatial and temporal memory violations by protecting memory accesses using exception-based runtime checks. Figure 1 provides an overview of FloatZone's components and workflow. At the core of our sanitizer, there are two main components: a custom memory allocator to manage redzones and quarantine freed memory, and compile-time instrumentation to accompany every memory access with a check to detect redzone accesses. Since the check is performed on every load and store, speed is of the essence. As shown in Figure 1, FloatZone introduces a novel method to detect redzone accesses through highly efficient *exception-based checks*.

Traditionally, sanitizers perform checks using at least compare and branch instructions, and potentially a shadow memory lookup. Unfortunately, this method of checking introduces a significant runtime overhead [51]. Ideally, we avoid this overhead by using a cheap `check` instruction that implicitly checks for redzone accesses and interrupts the program execution if that is the case, all without branching and or contending for the same heavily used execution units as the normal code.

While no dedicated `check` instruction for this purpose exists in modern architectures, we identify an instruction that approximates this behavior by generating an *exception* in the case of a memory violation. More specifically, we map floating point underflow exceptions to redzone access checks by carefully selecting the operands of a floating point addition.

4 Checks using Floating Point Exceptions

The structure for a memory safety check of sanitizers generally follows the same two-step pattern: (i) evaluate whether the memory location is a redzone (compare), and (ii) diverge the control flow in case of detecting a violation (branch). This pattern has been standardized in existing sanitizers, for example in ASan [39], which for each load and store operation performs a *comparison* on the corresponding shadow memory, followed by a conditional *branch* to an error-reporting function. Unfortunately, this method of checking incurs a runtime overhead of approximately 2x [39]. Recent work has highlighted two dominant sources of this overhead. First, performing the check itself constitutes 80% of the overhead [51]. Second, consulting shadow memory results in a significant increase in the number of page faults [24, 51].

Aside from the known instrumentation costs, sanitizer checks also introduce microarchitectural penalties: (i) loads from shadow memory can evict application cache lines and TLB entries, (ii) frequent comparisons pollute the branch predictor buffers and can result in branch mispredictions, and (iii) performance bottlenecks caused by the competition for execution units (e.g., load, branch, and address generation units). For instance, in our experiments we observe that sanitizer checks significantly increase the branch misprediction rate, suggesting that crucial predictor entries are being evicted. Note that sanitizer branches are rarely taken, so their behavior can be predicted without relying on the branch predictor.

An ideal implementation of a sanitizer check leverages a cheap instruction to distinguish between valid and invalid memory, uses an underutilized execution unit, and does not pollute caches and branch predictor buffers. Although it is impossible to completely avoid these inherent drawbacks on commodity hardware, we identify that *floating point arithmetic* can closely approximate such an ideal check. The core idea behind this is that we can map memory safety violations to floating point exceptions. Exceptions can be seen as con-

Listing 1 Pseudocode representation of the implicit check as a result of performing `float(y) + float(0x0b8b8b8a)`.

```
if( y == 0x8b8b8b8b || y == 0x8b8b8b89 ) {
    goto exception_handler;
}
```

ditional branches that redirect the control flow in the case of an error. Out of all the available exception types, we find that floating point exceptions, due to their flexibility, are the most suitable for expressing invalid memory accesses. By carefully selecting the operands, we can configure a single floating point operation to result in an exception *only* if one of the operands is equal to a constant value. Hence, we introduce a sanitizer check of the form: `fp_operation(mem[addr], const_value)`. This provides the means to express a comparison that evaluates whether the value at `addr` is equal to a constant value and, if so, raises an exception.

By encapsulating a comparison as a floating point operation, we gain three major benefits. First, exception-based checks use a fast implicit branch through the CPU to verify whether an exception occurred. Second, since floating point arithmetic never uses branch prediction resources, it prevents pollution of microarchitectural buffers and expensive branch mispredictions. Although the floating point *exception* is slower than taking a branch, this slow path is rarely executed. Third, the checks result in higher instruction-level parallelism in the common case of programs that underutilize the floating point unit (FPU).

Operation Details To find the most suitable floating point arithmetic configuration, we identify that the performance of the operation is a critical aspect. Hence, we avoid instructions with low throughput such as divisions, and instead limit ourselves to additions and subtractions, which are equivalent due to the sign bit. Next, we require certain constraints on the pair(s) of values that generate an exception when summed or subtracted. First, we need to avoid *collisions* with other numbers as much as possible to avoid false positives. In essence, we look for a fixed value x such that we can find only one (or few) y value(s) where $x+y$ generates an exception. Second, we require that y follows a byte-wise repetitive pattern (e.g., `0x4a4a4a4a`). The aforementioned is a property we require for memory alignment reasons, which we explain later.

With all these constraints in mind, by searching the floating point number space in a brute-force manner, we discover a suitable configuration. Specifically, with $x = 5.375081 \cdot 10^{-32}$ (`x=0x0b8b8b8a`), $x+y$ causes an underflow exception only with $y = -5.3750813 \cdot 10^{-32}$ or $y = -5.37508 \cdot 10^{-32}$ (`y=0x8b8b8b8b` or `y=0x8b8b8b89`). Note that `y=0x8b8b8b8b` is a byte-wise repetitive pattern. The specific combination of numbers that we discover allows us to express the comparison in Listing 1 by performing `float(y) + float(0x0b8b8b8a)`. Note that the exception is only raised if y is equal to one of the two known constant values.

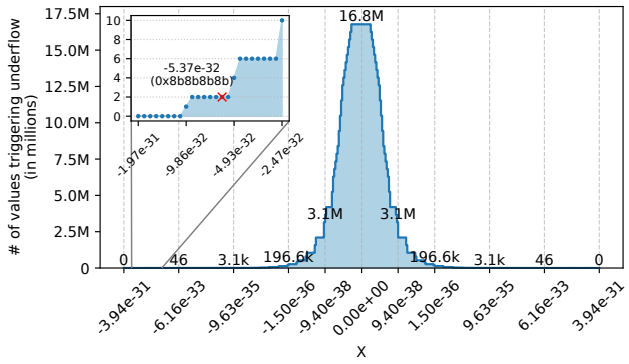

Figure 2: Underflow exception distribution by doing $x+y$ where x is fixed (x-axis) and $y \in$ 32-bit space of single precision FP numbers. In the zoomed subgraph we highlight the repetitive number `0x8b8b8b8b` that can underflow with only two possible values.

To understand why underflows are the most convenient exception for our goal, we visualize the distribution of underflows in Figure 2. We fix x and try all possible 32-bit y values. We count the number of underflow exceptions generated by computing $x+y$. We can see that x values close to 0 are more susceptible to underflow exceptions, while increasing the absolute value of x reduces the frequency of underflows. This is because an underflow exception happens only if $|x+y| < 1.0 \cdot 2^{-126}$. In other words, the difference between x and y must be so small that it can only be represented using the denormal representation. By having large x values, the floating point precision becomes too little to generate small results. In the subgraph of Figure 2 we can also see that there is a sweet spot of values where the number of possible underflows is very small, and thanks to the large range of these values we manage to find a number with a repetitive pattern (`0x8b8b8b8b`), satisfying all our requirements.

5 Sanitizing Memory Errors

Now that we have a fast means to evaluate if a four-byte memory location holds `0x8b8b8b8b` or `0x8b8b8b89`, we describe how to build a sanitizer for spatial and temporal memory.

Redzones are an effective technique to detect memory errors. Most notably, ASan implements redzones through a 1-to-8 byte compressed shadow memory, where the shadow bytes signify whether memory is valid or not. However, recent work has highlighted that shadow memory significantly degrades performance [6, 24, 51] through an increase in the number of page faults, as well as misses in the TLB and caches. To avoid such overhead, the faster alternative is to mark the redzones *in-band* with a special poison value and raise an alarm for any memory operation that accesses a poisoned address [18]. While false positives occur if a program legitimately uses the

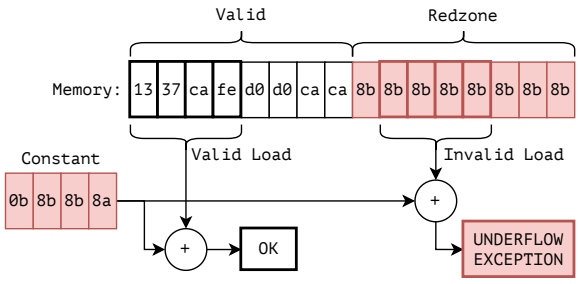

Figure 3: Verifying memory accesses using floating point additions.

Listing 2 Float check instrumentation. A single `vaddss` instruction is sufficient to fully instrument load and store operations. We must check stores before they are performed to avoid overwriting redzones, while we check loads afterwards to improve caching.

```
1 ;Assume xmm14=0x0b8b8b8a and xmm15 clobber reg
2 mov    rax, [load_addr]
3 vaddss xmm15, xmm14, [load_addr]   ;Load check
4
5 vaddss xmm15, xmm14, [store_addr]  ;Store check
6 mov    [store_addr], rax
```

poison value, this situation is rare and a slow-path check using shadow memory can weed these cases out. More efficient still, modern variants drop the precise and inline slow-path analysis altogether, because it needlessly slows down the execution and we can always filter out false positives at a later stage. As we will see later in this section, we opt for a similar model, with additional constraints to reduce false positives.

Besides false positives, padding and alignment are additional challenges for modern in-band redzone solutions [6,42]. Alignment is necessary because even if a pointer does not point to the start of the poison value, dereferencing it should still raise an alarm if the access (partially) overlaps with the redzone. Existing solutions explicitly align the target of the check to a multiple of the size of the poison value, incurring even more overhead because of the necessary addition and modulo operations [6,42]. With alignment, padding occurs whenever an object does not end at a boundary that is a multiple of the size of the poison value. Detecting out-of-bounds accesses to the padding requires even more instrumentation. For instance, ReZZan [6] requires multiple additions and subtractions and two modulo operations to perform the check.

Adopting a similar design, FloatZone not only removes the compare-and-branch operations, but also sidesteps these challenges by carefully choosing a poison value that is insensitive to alignment and padding. In particular, we place our four-byte float constant around each memory object, repeating it as necessary, and then instrument all memory accesses with a float addition. If a memory access operates on our poison value, the corresponding addition results in an underflow exception. Figure 3 visualizes the design. By performing the sanitizer checks directly on the memory address of the target operation, we avoid the memory locality penalties caused by shadow memory accesses. To avoid the padding and alignment issues, we employ a repetitive poison pattern (`0x8b8b8b8b`), allowing us to read four bytes from the starting point of any memory access without alignment. This concept is visualized in Figure 3, where it is clear that any four-byte access within the redzone results in the same poison pattern being read.

Redzone Configuration When discovering our repetitive poison value, we noticed that there is one additional colliding value: `0x8b8b8b89`. Note the last byte is `0x89` rather than `0x8b`. While this seems at first to be a drawback, we use the `0x89` byte to our benefit. Specifically, since the `0x89` byte is stored as the first byte in little endian representation, we can use it as start marker for our redzone, which helps to reduce false positives. To clarify, suppose the valid memory in Figure 3 ends in `0x8b8b` instead of `0xcaca`, and we perform a two-byte access to read this value. Our instrumentation inserts a float addition on the four-byte value starting at the memory access, which would be `0x8b8b8b8b` (first half valid, second half redzone). Consequently, this valid access would result in an unwanted exception. However, when we use `0x89` as the first redzone byte, the pattern is broken: `0x8b8b898b` does not generate an exception. The `0x89` byte ensures that we can separate objects ending in `0x8b` from the start of the redzone.

Instrumentation FloatZone implements the checks by inserting just a single instruction, as shown in Listing 2. In particular, a `vaddss` instruction after a load and before a store operation is enough the check the validity of the memory access. The `vaddss` assembly instruction performs a scalar single-precision (i.e., 32-bit) floating point addition, and stores the result in a specified register. This addition implicitly performs all sanitizer check steps: load the target, compare against the poison value, and trigger an exception if the access is invalid. Additionally, floating point additions are a cheap operation: on x86-64, the CPU can schedule up to two `vaddss` instructions per clock cycle [2].

Spatial and Temporal Memory Safety We improve spatial memory safety by extending (i.e., padding) stack, heap, and global memory objects with 16-byte underflow and overflow redzones. Figure 4 visualizes the creation and destruction of memory objects and their accompanying redzones. We fill the redzone areas with our poison value, starting with the `0x89` byte marker. The user of these memory objects is unaware of the presence of the redzones, hence the program is unaffected unless memory violations occur. Additionally, the underlying heap allocator may pad the object size to a multiple of 16 (for alignment). In this case, our overflow redzone is enlarged such that it fills the complete usable size of the object.

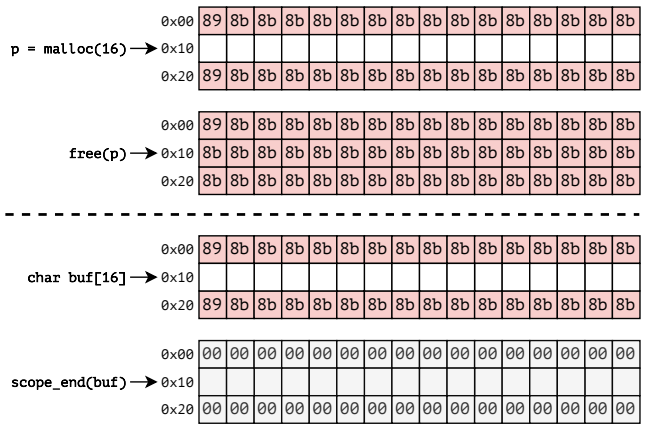

Figure 4: Redzone management on the heap (top) and stack (bottom).

The instrumentation upon destruction of memory objects is different for the stack and the heap. When a stack object reaches the end of its lifetime (i.e., goes out of scope), we clear out the redzone to avoid running into phantom redzone bytes on the stack at a later time (when new variables are initialized). On heap object deallocation, we poison the entire object (as shown in Figure 4) to detect temporal memory errors. To prevent the heap memory area from being re-initialized by a later allocation, we introduce a *heap quarantine* that delays the actual deallocation of the object. The quarantine follows a least recently used replacement policy, and starts deallocating objects when it is full (default 256 MB). We ensure that the object, including its redzones, is zeroed out upon leaving the quarantine to avoid phantom redzone bytes on the heap. Heap quarantines are a common technique to detect temporal memory errors, as seen in related work [7, 15, 17, 36, 39].

Security Guarantees Due to the four-byte granularity of our float values, we have to instrument memory accesses with a four-byte addition, regardless of the original access size. As a consequence of this design, there are some limitations when a complete pattern of four bytes cannot be read. Specifically, with a 16-byte redzone, accesses that are off-by-{13,14,15} bytes operate on an incomplete pattern. However, if desired, a larger redzone size can be used to detect further overflows. Additionally, as visualized in Figure 5, underflows of {1,2,3} bytes cannot be detected, since the four-byte pattern will be partially inside the (valid) memory object, and hence not equal to the poison value. As we will see, our experiments show that underflows of three or less bytes are not common, and hence the aforementioned drawback has minimal impact on our detection capabilities.

Furthermore, *partial* overflows are an additional case that requires special attention. This concerns memory accesses that are partially in-bounds and partially out-of-bounds. Such overflows can only be detected if we offset the memory operation by its access size, i.e., a 4-byte load at address x will be

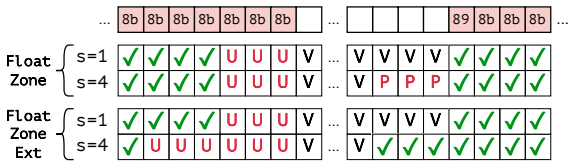

Figure 5: Visualization of FloatZone(Ext)'s detection capabilities. Each cell indicates the start of a memory access (i.e., load/store) of size `s`. The symbol `V` denotes a valid (in-bound) memory access, ✓ a true positive out-of-bound, `U` a false negative underflow, and `P` a false negative partial overflow.

checked at $x + memsize(x) - 1$, which in this case equates to $x+3$. The additional offset effectively evaluates whether the *last* byte of the memory access is within bounds. However, this optional extension, which we refer to as FloatZoneExt, comes with the drawback of further reducing underflow detection capabilities. As highlighted in Figure 5, if we perform a four-byte load (`s=4`) on an address that is off by -4 bytes (i.e., an underflow), the resulting shifted check reads the last underflow redzone byte together with the first three bytes of the object, meaning no exception is generated.

In our design, so far we have only considered 32-bit floating point values. However, floating point exceptions can also be generated with 64-bit and 16-bit (requiring AVX512 on Intel) numbers. For our use case, we found that 32-bit numbers offer the best trade-off between bug detection granularity and the probability of spurious exceptions. To clarify, 64-bit numbers reduce security guarantees (e.g., missing underflows up to 7 bytes), while 16-bit numbers statistically make unwanted exceptions more likely.

5.1 False Positives

As mentioned, the main drawback of relying exclusively on poisoned in-band redzones is the possibility of false positives, as a program that happens to use our poison value in memory results in an unwanted exception. FloatZone recovers from these exceptions if the four-byte poison value is not part of a complete redzone (i.e., one `0x89` byte followed by at least 15 `0x8b` bytes). As a result, false positives occur only under the following conditions: (i) the program contains a complete redzone (not from our instrumentation), or (ii) the program contains a memory object that starts with `0x8b8b8b8b`. Note that in the latter case, we are unable to distinguish the memory object from the redzone before it, since we do not have a marker to signify the end of a redzone, and hence the redzone effectively merges with the start of the object.

Recent work [6] shows that using a memory sanitizer without a backend for ground truth (i.e., no shadow memory) can be accurate and performant. We argue that, in software testing, potentially dealing with false positives is not problematic, as the test cases can be confirmed using an alternative backend or system (e.g., ASan). While alternatives to detect false

positives exist, such as tree-structured shadow memory [24], our evaluation shows that false positives are rare enough that confirming them offline (i.e., asynchronously) is sufficient.

5.2 Exception Handling

Since floating point exceptions are disabled by default, we enable per-process underflow `SIGFPE` exceptions at program startup, together with flush-to-zero (needed on x86). We register an exception handler such that the instrumented program enters our handler on every floating point underflow. Such an event can arise under two circumstances: our inserted `vaddss` instruction underflows, or from any generic floating point underflow operation present in the target program.

In the case of a `SIGFPE` caused by our instrumentation, we have to confirm whether this was a memory violation. To do so, we first have to recover the faulting address that corresponds to the triggered exception. Unfortunately, the `SIGFPE` exception does not come with the faulting address in the exception context (in contrast to `SIGSEGV`). However, since our instrumentation is designed to always be a single `vaddss` with a memory operand, we can disassemble the faulting instruction to recover the location of the memory operand. Once we obtain the faulting address, we filter out false positives by scanning for our redzone pattern (e.g., confirm the `0x89` redzone start marker is present).

In the case of entering the exception handler due to a spurious `SIGFPE` trigger, we restore the execution back to the point of interruption, since otherwise the program would default into process termination. For these unwanted exceptions, we also have to ensure the correct result of the instruction is applied. Note that simply ignoring the exception is not sufficient, as the execution would continue by faulting again. Therefore, we re-execute the faulting instruction ourselves by temporarily disabling flush-to-zero mode to compute the originally intended result, and then restore the execution starting from the next instruction. Thereby we also avoid issues where flush-to-zero unintentionally changes the output of operations. For example, flushing a small denominator to zero leads to an unintended division by zero. Similarly, for false `vaddss` exceptions, we simply move the instruction pointer to the next instruction and continue executing (here, obtaining the result of the instruction is redundant).

5.3 Optimizations

We reduce the total number of required memory safety checks by applying the optimizations described in the ASan-- paper [51]. These optimizations filter out memory accesses that are proven to be safe. We apply all optimizations except those that assume shadow memory and are therefore not applicable.

First, we successfully integrate the *removing recurring checks* optimization. With this optimization, we identify checks that concern the same memory location and access size, and hence we can deduplicate them into a single check. Next, we partially apply the *relocating invariant checks in loops* optimization. This optimization recognizes checks inside loops that operate on a constant pointer, and hence a single deduplicated check *after* the loop is sufficient. However, we cannot move checks on *store* operations outside of loops due to the store potentially overwriting our redzone, hence we restrict this optimization to *loads*. Fortunately, we measure that 80% of the cases where this optimization applies in SPEC CPU2006 concern load operations.

6 Implementation

We implement a prototype of *FloatZone* as a compile-time instrumentation pass in LLVM 14. Additionally, we override the default heap memory allocation functions to serve as an overlay allocator. We overload the common `mem*` and `str*` family functions of the C standard library to insert memory checks. For our exception handler, we use the Intel XED library [22] to disassemble instructions at runtime.

There are some special cases we need to consider to ensure our instrumentation functions properly. Most importantly, redzones on the stack must be cleaned up after the lifetime of the associated object. Otherwise, the checks may run into phantom (i.e., old remnant) redzone bytes, resulting in false positives. Most of these cases are already described in prior work [11], such as `setjmp/longjmp` and (C++) exceptions.

For FloatZone specifically, we have found a problematic case where the `XSAVE` and `XRSTOR` instructions are used for saving and restoring the processor state on the stack. It is possible that a complete redzone is saved on the stack due to still being present in an XMM register. The only observed occurrence of this behavior is inside the dynamic lookup of symbols. We addressed this issue by compiling the target binary with bind-now symbol resolution.

7 Evaluation

In this section, we evaluate both the performance and the security of FloatZone. We first demonstrate the performance impact of the float check itself, by comparing against traditional comparison instructions. Then, we evaluate the performance and accuracy of FloatZone. We measured runtime and memory consumption overhead using the SPEC CPU2006 and CPU2017 benchmarking suites. We show the effectiveness of FloatZone's security guarantees using the Juliet Test Suite, the Linux Flaw project, and OSS-Fuzz test cases. We also evaluated the performance gain in one of the main use cases for FloatZone, as part of a fuzzing campaign. We ran the experiments on a machine with Ubuntu 22.04, 64 GB RAM, and an Intel i9-13900K CPU of which we use the 8 performance cores. All reported overhead numbers are the median of 5 iterations, unless specified otherwise. For SPEC CPU2017,

we enable OpenMP where available (using 16 threads).

The FloatZone design can be configured in two different modes: including or excluding partial overflow detection. In our experiments, we refer to the main design as FloatZone, and the extended version including partial overflows as FloatZoneExt. As described in Section 5, the FloatZoneExt design involves offsetting the checks on memory operations by its access size. This allows for precise partial overflow detection, at the cost of reduced underflow detection guarantees and slightly slower checks.

7.1 Float Arithmetic Checks

We investigate the performance difference between a traditional branch (`cmp+je`) and a float addition (`vaddss`) in terms of multiple metrics: the imposed runtime overhead, the strain on the branch predictor, the increase in binary code size, and the effect on the instruction-level parallelism.

First, we measure the runtime overhead of the two different checking methods. We create two compile-time instrumentation passes that either insert a `vaddss` or a `cmp+je` on every memory access, and apply this pass to SPEC CPU2006. Since we are interested in the isolated overhead of the check itself, neither pass contain a backend (i.e., no shadow memory, and no exception handler), and additionally neither pass contains any of the (orthogonal) optimizations mentioned in Section 5.3. The `cmp+je` pass compares against (repetitions of) `0x8b` and contains a branch to an effective NOP to introduce minimal overhead in the case of a false positive. For the `cmp+je` instructions, we test three different comparison configurations: 1-byte, 4-byte, and N-byte, where N is equal to the access size of the corresponding instrumented memory access. We measure that the N-byte `cmp+je` performs the best, and this configuration in fact resembles the design of LBC [18] ported to 64-bit. This result suggests the N-byte configuration enjoys the most favorable trade-off between false-positives and data re-use. To clarify, the 1-byte `cmp+je` configuration experiences more false positives due to the constant small granularity, while the 4-byte `cmp+je` version requires reloading data for 1- and 2-byte accesses.

When comparing the N-byte `cmp+je` pass to the `vaddss` pass, we empirically confirm that the total number of inserted checks is equivalent. Figure 6 shows the performance progression of both passes across various CPU generations. Each reported overhead is with respect to the baseline measured on the respective machine. For the `cmp+je` instrumentation, we observe that across all microarchitectures the relative overhead remains nearly identical at a geomean of 50%. Note that the absolute speed of the baseline increased significantly: geomean 42% less overall runtime between Skylake and Raptor Cove. The overhead of the `vaddss` instrumentation is significantly lower than the `cmp+je` counterpart, and newer architectures widen the gap even further. In the most recent generation Intel CPU (i9-13900K), the geomean overhead of

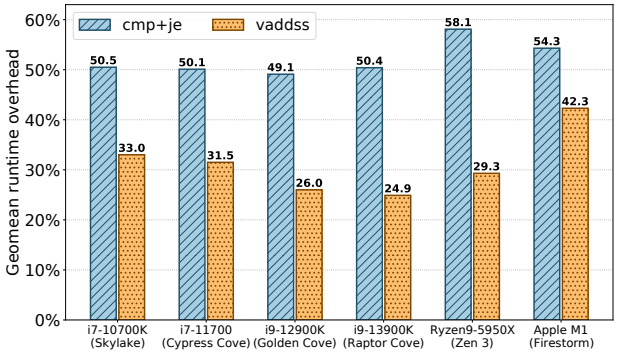

Figure 6: SPEC CPU2006 geomean runtime overhead of `vaddss` and `cmp+je` across CPU generations.

inserting a `vaddss` is half the relative cost of a `cmp+je`, and also 8 percentage points lower than the `vaddss` pass on the less recent i7-10700K. These results suggest that the FPU has become significantly faster in recent Intel generations. This hypothesis is supported by the fact that on the Golden Cove microarchitecture, two additional FADD (Fast Add) units were added to the pipeline [21]. We additionally verified that the `vaddss` benefits are not Intel-specific by performing the same evaluation on an AMD Ryzen 9 5950X with the Zen 3 microarchitecture and an Apple M1 (Firestorm). We observe an improvement for both architectures. On AMD we measure 58.1% overhead for `cmp+je` and 29.3% for `vaddss`. While on the M1, using Arm's equivalent instructions for the comparison and floating point addition (`fadd`), we manage to reduce the overhead from 54.3% to 42.3%.

Microarchitectural benefits Next, we investigate the impact of the inserted code on the branch misprediction rate. Note that the memory violation branches are generally never taken, hence the ideal outcome is that they have minimal influence on the overall misprediction rate. Using `perf` [31] as performance monitor, we collect the number of branch mispredictions on SPEC CPU2006. We observe that the `cmp+je` pass increases the number of branch mispredictions with a geomean of 22.2%. In contrast, the `vaddss` pass results in a geomean increase of 6.6%. The negative impact of the `vaddss` pass is mostly attributed to the libc instrumentation, which for example requires additional comparisons to check the size passed to `memcpy` and similar functions. When excluding the libc interposition, the rate reduces from 6.6% to 1.5% for the `vaddss` pass, while the impact of the `cmp+je` pass remains relatively high at 17.8%. The residual 1.5% increase in mispredictions originates from benchmarks that are sensitive to code size changes. In fact, inserting an equivalent number of `NOP` instructions to match the `vaddss` pass causes the same misprediction effect. From this experiment we conclude that inserting comparison checks that rarely fail has a notable negative impact on the branch misprediction rate, while float

additions, as expected, have a minimal effect.

Additionally, inserting the `vaddss` instructions increases the size of the `.text` region of SPEC CPU2006 binaries by a geomean of 31.8%, while the `cmp+je` inflates the code size by 52.0%. This is a consequence of comparison instructions requiring a split in basic blocks and the accompanied jumps, while float additions keep the control flow as-is.

Furthermore, `vaddss` offers high instruction-level parallelism for multiple reasons: (i) The CPU can freely schedule our `vaddss` instructions thanks to the lack of data dependencies (the addition results are never used). (ii) The `vaddss` instruction puts less pressure on the CPU reservation stations by requiring only two micro-operations (AGU and FP) in contrast to the three required by `cmp+je` (AGU, ALU, JMP) [2]. (iii) On x86-64, the CPU is capable of scheduling up to two `vaddss` instructions per clock cycle. (iv) The CPU can handle a high throughput of `vaddss` thanks to the wide availability of FPU execution units. In fact, on a modern Intel microarchitecture (e.g., Golden Cove [23]), floating point additions can be scheduled on a total of six units (2x FP ALU, 2x FMA, and 2x FastADD). In contrast, there are only two execution units for branch instructions. Lastly, we point out that the `vaddss` instruction is not slowed down by micro-code assists, which we confirm by never observing the `FP_ASSIST` event.

To conclude, our experiments highlight the benefits of using a float addition as an alternative for a comparison: lower runtime overhead, less microarchitectural interference, a smaller code footprint, and higher instruction-level parallelism.

7.2 Security Evaluation

We compare the detection capabilities of FloatZone and FloatZoneExt against ASan through various metrics. We use the Juliet Test Suite, as seen in previous work [6, 24, 51], to test a wide range of synthetic bugs. We detect CVEs from the Linux Flaw project [49, 51] to demonstrate our effectiveness with realistic bugs. We investigate the impact of our inherent limitations on fuzzing by analyzing OSS-Fuzz [14] test cases. Additionally, FloatZone successfully detects bugs that are known in SPEC, such as a global buffer overflow in 464.h264ref [39], and a buffer underflow in 602.gcc_s.

Juliet Test Suite We measure FloatZone's security guarantees using the NIST Juliet Test Suite (v1.3) [25]. This test suite contains hundreds of test cases to detect memory safety errors. We select the bug categories that are relevant to spatial and temporal memory errors, which are the same categories as reported in the ASan-- paper. We deduplicate the Juliet test cases to obtain a unique set of bugs, as many programs share bugs that are functionally identical at runtime.

Table 1 shows that FloatZone provides nearly identical security guarantees to ASan when evaluated on the Juliet test suite. For CWE 121 and 122, it misses a partial buffer overflow that overrides the loop iterator, but FloatZoneExt does successfully detect these cases. For CWE 122, FloatZone can detect one more heap buffer overflow than ASan, due to ASan lacking instrumentation for `wmemset`. CWE 124 shows the drawback of FloatZoneExt offsetting the checks on underflows, where two off by -4 bytes buffer underflows are missed, as explained under the security guarantees in Section 5. However, these binaries take input from `stdin`, and the effectiveness depends on which input is provided. With input '-1', FloatZone and ASan pass, and with '-2' FloatZoneExt also detects the bug. For CWE 126, FloatZone manages to detect three more cases than ASan, but the impact of these bugs depends on undefined data on the stack, due to a non-null terminated `printf` that may or may not cause an overread. For CWE 127, FloatZone misses an access that is off by -20 bytes (5 integers underflow), since our redzone is 16 bytes. There is no preceding guarded stack object of which the overflow redzone could be hit. Furthermore, FloatZoneExt (potentially) misses two bugs due to the same input dependency mentioned before. Finally, all temporal bugs are detected by both systems. In total, we can see that FloatZone can detect one more bug, and 98.7% of the bugs that ASan detects.

Description (CWE)	Total	ASan	FloatZone	FloatZoneExt
Stack overflow (121)	72	72	71	72
Heap overflow (122)	78	74	74	75
Buffer underwrite (124)	24	24	24	22
Buffer overread (126)	19	16	19	19
Buffer underread (127)	24	24	23	21
Double free (415)	17	17	17	17
Use-after-free (416)	18	18	18	18

Table 1: Juliet Test Suite results. Overflow implies buffer overflow.

Linux Flaw project We evaluate FloatZone's bug detection capabilities on certain CVEs of the Linux Flaw project [32], as was also done in the ASan-- and SANRAZOR [49] papers. We execute all the (relevant) test cases from both papers, although some were not reproducible on our machine. We omit some types of bugs that we do not target, such as stack exhaustion and null pointer dereferences.

Table 2 shows the detection results per test case. There are two cases where FloatZone fails to detect the bug. First, CVE-2009-2285, which concerns an off-by-one byte underflow, which is a known limitation of our design. Second, CVE-2017-7263, which is off by 1016 bytes on a 4-byte object. Clearly, our 16-byte overflow redzone is too small to detect this bug. This shows a limitation of in-band redzones when there are no subsequent objects with redzones. ASan manages to detect these wild pointer accesses since they fall into a 'not mapped is poison' area in shadow memory. On the whole, we detect 16 out of 18 realistic bugs.

OSS-Fuzz To further investigate the impact of the underflow and partial overflow limitations, we test FloatZone

CVE	Type	ASan	FloatZone
CVE-2006-2362	stack-buffer-overflow	✓	✓
CVE-2009-1759	stack-buffer-overflow	✓	✓
CVE-2009-2285	heap-buffer-overflow	✓	✗
CVE-2013-4243	heap-buffer-overflow	✓	✓
CVE-2015-8668	heap-buffer-overflow	✓	✓
CVE-2017-12858	heap-use-after-free	✓	✓
CVE-2015-9101	heap-buffer-overflow	✓	✓
CVE-2016-10095	stack-buffer-overflow	✓	✓
CVE-2016-10270	heap-buffer-overflow	✓	✓
CVE-2016-10269	heap-buffer-overflow	✓	✓
CVE-2017-5976	heap-buffer-overflow	✓	✓
CVE-2017-5977	heap-buffer-overflow	✓	✓
CVE-2017-7263	heap-buffer-overflow	✓	✗
CVE-2017-12858	heap-use-after-free	✓	✓
CVE-2017-12937	heap-buffer-overflow	✓	✓
CVE-2017-14407	stack-buffer-overflow	✓	✓
CVE-2017-14408	stack-buffer-overflow	✓	✓
CVE-2017-14409	global-buffer-overflow	✓	✓

Table 2: Linux Flaw Project CVE detection by FloatZone and ASan.

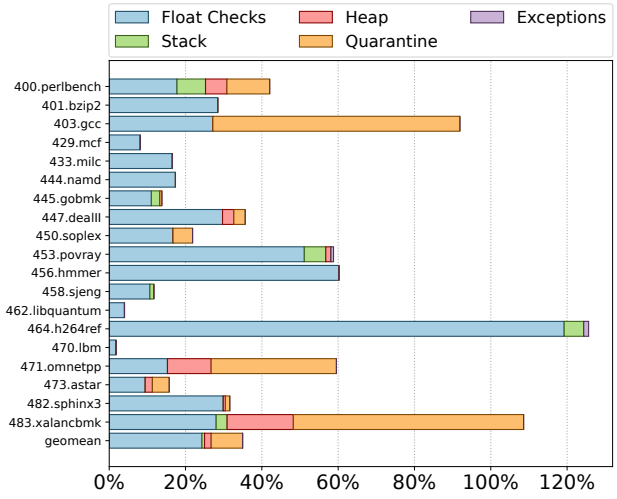

Figure 7: SPEC CPU2006 runtime overhead buildup of FloatZone.

against a large collection of heap buffer overflows obtained from the OSS-Fuzz project [14], with the goal of finding how many of them are undetectable. We extract a total of 2942 test cases with heap buffer overflows, and manage to reproduce 480. Unfortunately, many bugs were not reproducible due to building or versioning issues, since we have to estimate the target build commit from the bug report date.

We execute the test cases with ASan to establish ground truth on the reproducibility of the original OSS-Fuzz report. We obtain the overflow parameters from the ASan bug reports, after which we match them against the detection capabilities of the FloatZone design. Additionally, we run AFL's *crash exploration mode* to fuzz the bug for five minutes, where we attempt to find alternative triggers of the same bug (e.g., a different faulting offset).

Out of the 480 total bugs, we find that 424 are overflows, 42 are underflows, and 14 crash without reporting the faulting offset. To evaluate the number of false negatives introduced by FloatZone, we consider the two relevant cases: detecting partial overflows and detecting underflows. Of the 42 underflows, 22 have a negative offset greater than 3, meaning FloatZone can detect them. The remaining 20 cases are mostly off-by-one. When also considering the bug exploration, we found that two out of the 20 bugs have an alternative trigger that is detectable. Hence, 18 underflows remain undetected.

Regarding partially out-of-bounds accesses, excluding bugs that report a non-partial variant in the same run (i.e., recovery mode continuation) or in the AFL exploration, we find a total of 6 overflows and 2 underflows. For the 6 partial overflows, FloatZoneExt can accurately provide detection guarantees, whereas the partial underflows would require an additional check altogether. We point out that the partial overflow statistics are limited due to the fact that ASan itself can only detect partial overflows if the starting address of the fault is 8-byte aligned due to the shadow memory compression.

In conclusion, the security guarantees experiments show that FloatZone can accurately detect bugs, excluding known limitations. We additionally point out that these limitations are acceptable, with undetectable underflows being rare.

7.3 Overhead Buildup

Figure 7 displays the runtime overhead of FloatZone on each individual SPEC CPU2006 binary. On average, FloatZone results in a geomean runtime overhead of 36.4%. To better understand the source of the overhead, we measure the impact of multiple different components separately. More specifically, we distinguish between the cost of: (1) load/store float checks, (2) stack redzone initialization and destruction, (3) heap redzone initialization and destruction, (4) heap object quarantining, and (5) enabling exceptions (flush-to-zero, handler, etc.). Note that component 2 and 3 constitute spatial bug detection, while component 4 introduces the temporal aspect. The overhead of inserting redzones around global variables is omitted visually, as the runtime penalty is negligible.

From the buildup graph we conclude that most of the overhead originates from the floating point checks. This overhead includes the optimizations proposed by ASan--, although we find that the ones we implement have minimal benefit for our design (less than one percentage point improvement). The 464.h264ref binary stands out with a relatively large overhead caused by our checks. We observe that the overhead for this particular binary is dominated by the checks for libc `mem*`-family functions. Currently, we apply the checks to these standard library calls prior to the function. This can be optimized by performing the checks inside the function (e.g., merged into the `memcpy` loop to avoid looping twice). Furthermore, we observe that the stack instrumentation is relatively cheap on all binaries, as well as the heap redzones in general, although some allocation intensive binaries (e.g., 471.omnetpp

Program	Underflow	Float Hit	False Positive
401.bzip2	0	41703	1
433.milc	0	46	0
444.namd	0	4	0
450.soplex	0	14	0
453.povray	444672	2	0
456.hmmer	2	0	0
464.h264ref	0	313620	858
470.lbm	0	75	0
473.astar	4	12	0
482.sphinx3	3	96	0
602.gcc_s	0	148	0
619.lbm_s	0	166	0
625.x264_s	0	115358837	2233669
631.deepsjeng_s	0	11525	0
638.imagick_s	98234	10371	49
641.leela_s	0	8	2
644.nab_s	0	14	0
657.xz_s	0	76	2

Table 3: Exception handler hits in SPEC CPU2006 and CPU2017. Binaries with zero hits are omitted.

and 483.xalancbmk) experience noticeable overhead from the additional operations in the heap allocator.

After the float checks, the heap quarantine is the second largest factor in the overhead buildup. A quarantine for memory objects degrades the spatial locality of the program. Additionally, poisoning the objects upon deallocation has negative impact on the cache. It is possible to further optimize the quarantine, such as reducing cache-miss latency by prefetching the next-to-be-evicted quarantine entry (as seen in ASan). Moreover, we could use *non-temporal stores* upon deallocation to avoid caching freshly poisoned data. Currently, FloatZone's overhead on the 623.xalancbmk_s (SPEC CPU 2017) binary is higher than ASan's, as a result of the overhead being dominated by our unoptimized quarantine. However, we decided to stick with a simple design, since optimization techniques for the quarantine are orthogonal to our contribution.

The last overhead component concerns our handler for floating point exceptions. As shown in the buildup figure, there is only a noticeable overhead for 453.povray and 464.h264ref, which accurately correlates to the number of exceptions shown in Table 3. The table displays how many times and why the exception handler was entered for each SPEC CPU2006 and CPU2017 binary. Note that binaries with zero hits are omitted. The *Underflow* column provides the number of floating point arithmetic underflows unrelated to FloatZone's instrumentation. Next, *Float Hit* implies a load or store was performed on `0x8b8b8b8b` or `0x8b8b8b89`, but it did not concern a redzone. Finally, *False Positives* are float hits that could not be distinguished from a redzone. From the number of exceptions and the corresponding overhead component we conclude that the exception handler is not a bottleneck.

In total, we experience unavoidable false positives in two

	Runtime		Memory	
	SPEC 06	*SPEC 17*	*SPEC 06*	*SPEC 17*
ASan	77.8%	62.0%	237%	139%
ASan--	65.9%	53.5%	236%	139%
FloatZone	36.4%	37.0%	182%	82%
FloatZoneExt	47.4%	42.9%	182%	82%

Table 4: SPEC CPU geomean overhead comparison summary.

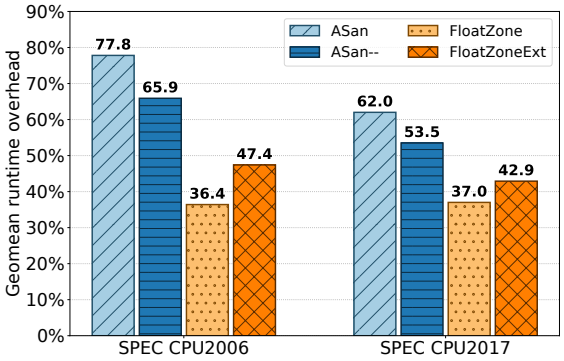

Figure 8: SPEC CPU2006 and CPU2017 geomean runtime overhead.

SPEC CPU2006 and four SPEC CPU2017 binaries. We observe that video compression algorithms (464.h264ref and 625.x264_s) frequently operate on our float poison value. We suspect it is common for our pattern to be contained in image representations, where 15 pixels contain the same value (`0x8b`), preceded by a slightly adjusted pixel (`0x89`). Note that the difference between `0x8b` and `0x89` is only 2. Excluding these binaries, the false positives are mostly caused by containing `0x8b8b8b8b8b` in the start of memory objects. The float constant originates from sources such as randomly generating an integer (in 641.leela_s). We point out that it is not difficult to identify these cases as false positives, since they can be asynchronously confirmed using for example ASan.

Our results show that offloading memory safety checks to the FPU does not disproportionally affect floating point benchmarks. In fact, the subset of floating point benchmarks in SPEC CPU2006 (7 binaries) has a partial geomean runtime overhead of 24.7%, while the integer subset (11 binaries) is 43.7%. The integer benchmarks generally concern more allocation-heavy workloads, and therefore experience more overhead from the heap instrumentation.

7.4 Comparison against State-of-the-Art

In this section, we evaluate the performance of FloatZone as a memory sanitizer in terms of runtime and memory overhead. We measure performance using the SPEC CPU2006 and SPECspeed 2017 benchmarking suites. To put the per-

formance of FloatZone into better perspective, we compare the overhead to related state-of-the-art systems. We evaluate against ASan [39] and ASan-- [51], while we do not include systems such as ReZZan [6] and FuZZan [24] because their designs are optimized for very short-lived applications, so they would be misrepresented by the SPEC CPU benchmarks. In fact, ReZZan crashes on SPEC CPU2006 due to running out of memory. For ASan and ASan-- we disable the functionality that we do not implement, such as use-after-scope detection. We port ASan-- to LLVM 14 to equalize the baseline.

Figure 8 visualizes the geomean runtime overhead of FloatZone and FloatZoneExt (the configuration with memory access size offsetting) in comparison to ASan and ASan--. A complete summary of the SPEC CPU overhead results is shown in Table 4. In our reproduction of ASan--'s evaluation we find that it is 15% and 13.7% (11.9 and 8.5 percentage points) faster than ASan on SPEC CPU2006 and CPU2017, respectively. In the original paper, ASan-- reports an overhead reduction of 41.7% (44.5 percentage points) against ASan.

There are some differences in our setup that may explain this deviation. We do not run ASan(--) in recovery mode, and instead apply patches to SPEC to fix any present bugs [40], as is done in the ASan paper. This avoids wrongfully benchmarking the error reporting component of ASan. Additionally, for the overhead aggregation we use the geomean, as opposed to the mean in ASan--. However, taking the mean does not make the overhead results more favorable. Finally, our numbers include the 471.omnetpp SPEC CPU binary, which was reported to not build in the ASan-- paper.

In comparison, FloatZone's overhead is less than half of ASan's on SPEC CPU2006, and 25 percentage points lower on SPEC CPU2017. Although ASan-- successfully reduces the overhead of ASan, FloatZone is significantly faster, with a runtime overhead of 36.4% and 36.9% on SPEC CPU2006 and CPU2017, respectively. ASan-- reported compatibility issues with SPEC CPU2017 in their paper, resulting in limited results. We observe that most of the SPECspeed binaries run successfully, although 602.gcc_s and 265.x264_s report a false positive use-after-free. Hence, we run these two binaries in recovery mode for ASan--, as fixing this bug is not trivial.

Regarding potential FPU overutilization resulting in bottlenecks, for SPEC CPU2017 we observe a similar trend as for SPEC CPU2006 (discussed in the previous section). The three floating point binaries experience a geomean runtime overhead of 11%. Note that all floating point binaries have OpenMP directives, which is beneficial for distributing the load across multiple FPUs. For 644.nab_s we observe a severe slowdown when limiting the execution to a single core, which we suspect is due to FPU overutilization.

The FloatZoneExt configuration manages to outperform ASan-- by a significant margin, where the (optional) partial overflow extension increases the overhead of FloatZone from 36.4% to 47.4% on SPEC CPU2006. For SPEC CPU2017, the overhead increases from 37.0% to 42.9%. The increased overhead is fully attributed to the additional pointer arithmetic required to offset the checks by the memory access size. For a use case where accurate partial overflow detection is crucial, this mode can prove beneficial.

Regarding the memory overhead, FloatZone uses notably less memory, which is a logical consequence from the fact that we do not employ shadow memory. More specifically, FloatZone consumes 55 and 57 percentage points less memory than ASan on SPEC CPU2006 and CPU2017, respectively. However, we also note that ASan employs variably sized redzones, with a *minimum* of 12 and 16 bytes for the stack and heap, where the size of the redzone depends on the allocation size. Hence, using larger redzones naturally increases memory consumption. As expected, memory overheads are identical for FloatZone and FloatZoneExt, as for ASan and ASan--, since the changes only affect runtime performance.

In conclusion, the benchmarking experiments on SPEC CPU highlight the benefit of implementing a sanitizer using our approximation of an ideal memory safety check. While we showed in Section 7.1 that a `vaddss` can outperform a `cmp` instruction, this section furthermore underlines that the subsequent sanitizer built from such an alternative checking method is significantly more performant.

Microarchitectural bottlenecks To investigate the microarchitectural differences between ASan and FloatZone, we identify three performance metrics that highlight the negative impact of inserting comparison-based checks combined with shadow memory. Using `perf` [31] as performance monitor on SPEC CPU2006, we observe that ASan's instrumentation increases the number of TLB misses by a geomean of 145%, while FloatZone only causes a 56% increase. Furthermore, ASan causes the number of L3 misses to increase by 144%, while FloatZone causes a 107% increase. The reduction in TLB and L3 misses shows the benefit of FloatZone using in-band redzones to improve memory locality.

Additionally, we find that ASan increases the number of branch mispredictions by 11.3%, and FloatZone does so by 4.9%. This highlights the benefit of branchless checks on the branch predictor buffer. The remaining increase in branch mispredictions by FloatZone is mostly attributed the libc interposition, and to the binaries that experience many false positive exceptions (see Table 3), and hence frequently execute XED's decoding code. Note that the branch misprediction increase is lower in ASan than the `cmp+je` pass in Section 7.1, since the `cmp+je` pass experiences more prediction interference from false positive paths being taken.

7.5 Fuzzing

A common use case of memory sanitizers is fuzz testing to discover bugs in applications. We evaluate the increase in throughput and (edge) coverage in fuzzing when using

Benchmark	Total Execution Increase		Coverage Increase	
	ASan--	ReZZan	ASan--	ReZZan
file	97.0%	114.8%	-7.1%	-5.5%
libpng	178.1%	6.0%	0.6% †	-2.1% †
tcpdump	434.9%	4.5%	16.6%	0.5% †
cxxfilt	179.7%	25.9%	2.1%	0.2% †
nm	207.0%	116.5%	2.2%	1.2%
size	150.1%	206.1%	4.1%	4.2%
objdump	158.7%	122.7%	4.3%	2.1%
geomean	187.6%	71.8%	3.1%	0.1%

Table 5: Results of the fuzzing evaluation. The numbers concern the increase in total executions and branch coverage when using FloatZone compared to ReZZan and ASan--. Statistically *insignificant* results (Mann–Whitney U test p-value > 0.05) are marked with †.

FloatZone as sanitizer compared to ASan-- [51] and ReZZan [6]. We use the same benchmarks and configuration as ASan--, and perform 15 fuzzing iterations of 24 hours each. All reported results are the median of 15 runs. We select AFL++ [12] 4.05a in fork-mode as our fuzzer, and measure all results on the same hardware described in Section 7. We point out that ReZZan does not contain the optimizations mentioned in Section 5.3, however in our evaluation we observed they provide minimal benefit to FloatZone, hence we argue their omission from ReZZan is insignificant. Additionally, we fixed a confirmed issue regarding the missing instrumentation of `memset` in the ReZZan prototype [38], which would otherwise result in a misrepresentative overhead comparison.

The results in Table 5 show that FloatZone provides a considerable increase in the total number of executions in 24 hours: a geomean increase of 187.6% and 71.8% compared to ASan-- and ReZZan, respectively. As a result of the increase in throughput, fuzzing with FloatZone improved coverage up to 16.6% (in `tcpdump`) and 4.2% (in `size`) compared to ASan-- and ReZZan. In the coverage plots in Figure 9 (in the Appendix) we can observe that fuzzing with FloatZone usually causes the coverage plateau to be reached quicker than with the competing sanitizers, although all systems tend to reach similar total coverage after 24 hours. As pointed out by related work [13], an increase in throughput is mostly visible in the coverage graph up until the plateau, while exploring additional coverage beyond the plateau is not easily achieved solely by a larger throughput. To clarify, we observe notably higher relative coverage after 4 hours compared to the 24 hours counterpart. Specifically for `nm`, `objdump`, and `size`, after 4 hours FloatZone reaches 12%, 13%, and 6% more coverage than ASan--, and 5%, 6%, and 12% for ReZZan.

During the fuzzing campaign we observed the presence of some false positive bug detections. To remove these spurious results we simply provided the same crashing inputs to the target benchmark, using ASan to confirm or deny the presence of a (true positive) memory violation. Overall, we never observed more than 1900 false positives (originating from a total of 790 million executions) in a 24 hours run. Additionally, we verified that for all buggy benchmarks, FloatZone is capable of detecting the known true positive crashes.

For two benchmarks (`libpng` and `tcpdump`), FloatZone performs similarly compared to ReZZan. This is attributed to the fact that forking is a major bottleneck in these benchmarks, hence the benefits of FloatZone are less apparent. For coverage, the only (statistically significant) outlier is `file`, where we reach slightly worse final coverage. We verified that by configuring our sanitizer to never abort, thus avoiding crashes from false positives, we observe an almost identical coverage compared to ASan-- and ReZZan. This suggests that for `file` a false positive crash prevents AFL++ from exploring new paths. This issue can be solved by modifying AFL++ to use ASan as oracle to discard such blocking false positives.

In conclusion, our evaluation shows that FloatZone can increase fuzzing throughput considerably, reaching the same coverage while using less time and therefore less power.

8 Discussion

Limitations While we have addressed the accuracy limitations of FloatZone throughout this paper (see Sections 5, 5.1, 7.2, 7.5) here we point out some fundamental drawbacks of redzone-based sanitizers: limited detection for non-linear and inter-struct overflows, as well as engineering difficulties such as having to instrument custom memory allocators (e.g., `ngx_calloc` in Nginx [37]) and recompiling libraries (e.g., to instrument the `C++` standard library).

Hardware Floating point additions are an approximation of the ideal checking instruction and have the characteristics of being fast, branchless, and exception-based. The actual ideal instruction would execute on a dedicated execution unit to avoid interfering with the target program as much as possible. Since FloatZone assumes the FPU is generally underutilized, our design may not translate well to devices with limited FPU resources. Moreover, if the hardware can support efficient shadow memory management, the ideal instruction can avoid relying on 4-byte poison values, thereby avoiding alignment issues hampering detection capabilities.

In conclusion, we have shown FloatZone's floating point addition to be an efficient approximation of the ideal instruction on commodity hardware. For instance, the isolated SPEC CPU2006 geomean overhead of performing checks is 25% (cf. Figure 6), and the remaining 11.2% overhead is a *lower bound* for applying redzones and heap quarantining. Given the difficulty of finding better approximations on commodity hardware, future work may investigate hardware extensions to more closely match the performance of the ideal instruction.

9 Related Work

Sanitizers aim to *detect* bugs, such that they can be fixed. For this reason, we do not discuss related work that focuses on *mitigating* bugs.

Sanitizers There exist many different memory sanitizers, especially when including sanitizers that specifically target a single bug category. There are systems that only detect spatial memory errors [3,10,11,18,27,28,34], as well as systems only detecting temporal bugs [8,9,15,35,44]. Recent work [33,50] points out that combining two distinct spatial and temporal safety systems into a joined sanitizer results in high overhead.

In contrast, there exist sanitizers that by design provide both spatial and temporal detection guarantees, such as ASan [39], MEDS [17], DrMemory [7], and Memcheck [36]. Excluding sanitizers that require special hardware features [30,41,50], the existing memory sanitizers still incur a significant runtime overhead, despite modern optimizations [29,46,47,49,51].

FloatZone shares a part of its design with prior work. In-band redzones have been introduced by LBC [18], and have been re-visited by ReZZan [6]. More specifically, ReZZan optimized the LBC design by omitting shadow memory at the cost of requiring more expensive (branch-based) checks due to padding and alignment issues. FloatZone avoids these complications by leveraging a byte-wise repetitive poison pattern. Additionally, we demonstrate that we can avoid multiple microarchitectural penalties by offloading the checks to the FPU, resulting in a significantly faster comparison. On top of this, we highlight the benefits of using exception-based checks, through which we shift the overhead away from the common path towards the case of a memory violation.

To the best of our knowledge, we are the first to leverage byte-wise repetitive redzones with a start marker, and floating point arithmetic to perform exception-based checks. Specialized faster sanitizer checks have been proposed before, for example using the MMU [9, 15, 27], however FloatZone's checks express a more generic comparison that can offer both spatial and temporal bug detection. There is related work that relies on custom hardware to raise an exception upon a memory violation [42, 45]. In contrast, in FloatZone the exceptions themselves determine the validity of accesses. Finally, heap quarantines are common in temporal safety designs [7, 15, 17, 36, 39], and our implementation is similar.

Fuzzing Recent work introduced memory sanitizers specifically designed for fuzzing: FuZZan [24], and ReZZan [6]. These systems are designed to perform well for short-lived applications, at the expense of scaling poorly to long-running executions. FuZZan points out that the linear shadow memory of ASan can be optimized for fuzzing, while ReZZan shows that in-band redzones can outperform the tree-structured shadow memory of FuZZan. With FloatZone, we show that by introducing a fast method of performing checks, we can perform well both on short-lived (i.e., fuzzing) and long-lived (i.e., SPEC benchmarks) applications. Other literature to improve fuzzing throughput uses snapshots to reduce time spent executing target programs [13, 48]. Like FloatZone, these aim to relieve a bottleneck in fuzzing, but they are orthogonal as we specifically target the sanitizer. Both approaches can be combined to achieve further speedups.

10 Conclusion

Sanitizers for memory safety have become a standard in software testing, and despite recent optimizations, state-of-the-art bug detection tools still incur significant runtime overhead. We further improve performance by introducing a faster check for validity on commodity hardware. We show that we can use floating point arithmetic to express the common compare-and-branch sanitizer paradigm. We use this primitive to build a memory sanitizer called FloatZone, that relies on carefully crafted floating point underflow exceptions to identify memory violations. We show these checks using floating point additions are indeed notably faster than comparison instructions, thanks to multiple microarchitectural benefits. Moreover, we show that our resulting sanitizer significantly outperforms the state-of-the-art in both runtime and memory overhead.

Acknowledgments

We thank the anonymous reviewers for their feedback. We also thank Johannes Blaser for his valuable support with LLVM. This work was supported by Intel Corporation through the "Allocamelus" project, the Dutch Ministry of Economic Affairs and Climate through the AVR program ("Memo" project), the Dutch Science Organization (NWO) through projects "TROPICS", "Theseus", and "Intersect".

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std. 754-2019*, 2019.

[2] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673--686, New York, NY, USA, 2019. ACM.

[3] 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*, volume 10, 2009.

[4] AMD. AMD64 Architecture Programmer's Manual. 40332 Rev 4.00, 2020.

[5] Arm. Architecture Reference Manual - Armv8, for A-profile architecture. DDI 0487G.b (ID072021), 2021.

[6] Jinsheng Ba, Gregory J. Duck, and Abhik Roychoudhury. Efficient Greybox Fuzzing to Detect Memory Errors. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

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

[8] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *2012 International Symposium on Software Testing and Analysis*, pages 133--143, 2012.

[9] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *USENIX Security Symposium*, pages 815--832, 2017.

[10] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103--114, 2008.

[11] Gregory J. Duck, Roland H.C. Yap, and Lorenzo Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *NDSS Symposium*, pages 1--15, 2017.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[13] Elia Geretto, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Snappy: Efficient Fuzzing with Adaptive and Mutable Snapshots. In *Annual Computer Security Applications Conference (ACSAC)*, pages 375--387, 2022.

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

[15] 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 (CCS)*, pages 1307--1322, 2022.

[16] Istvan Haller, Erik Van Der Kouwe, Cristiano Giuffrida, and Herbert Bos. METAlloc: Efficient and Comprehensive Metadata Management for Software Security Hardening. In *9th European Workshop on System Security (EuroSec)*, pages 1901--1915, 2016.

[17] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing memory error detection for large-scale applications and fuzz testing. In *Network and Distributed Systems Security (NDSS) Symposium*, 2018.

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

[19] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter 1992 Technical Conference*, pages 125--136, 1992.

[20] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual combined volumes. 325462-070US, 2019.

[21] Intel. Intel Architecture Day 2021 Presentation. Online, 2021. https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf.

[22] Intel. X86 Encoder Decoder (XED). Online, 2022. https://intelxed.github.io/.

[23] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. 248966-046, 2023.

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

[25] Frederick Boland Jr. and Paul Black. Juliet 1.1 C/C++ and Java Test Suite. In *IEEE Computer*, pages 88--90, 2012.

[26] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Fast and generic metadata management with mid-fat pointers. In *10th European Workshop on Systems Security (EuroSec)*, pages 1--6, 2017.

[27] 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.

[28] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr, and Andre DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *2013 ACM SIGSAC conference on*

Computer & communications security (CCS), pages 721--732, 2013.

[29] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, , and Michael Franz. PartiSan: fast and flexible sanitization via run-time partitioning. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 403--422, 2018.

[30] 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 *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1901--1915, 2022.

[31] Linux. Profiling with performance counters . Online. https://perf.wiki.kernel.org.

[32] Dongliang Mu. LinuxFlaw. Online, 2015. https://github.com/mudongliang/LinuxFlaw.

[33] Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL)*, 2015.

[34] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. 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 (PLDI)*, pages 245--258, 2009.

[35] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In *2010 International Symposium on Memory Management (ISMM)*, pages 31--40, 2010.

[36] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, pages 89--100, 2007.

[37] Nginx. Memory Management API. Online. https://www.nginx.com/resources/wiki/extending/api/alloc/#c-ngx-calloc.

[38] ReZZan. Missing memset instrumentation. Online. https://github.com/bajinsheng/ReZZan/issues/2.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 309--318, 2012.

[40] Kostya Serebryany. SPEC2006 ASan patch. Online, 2013. https://github.com/google/sanitizers/blob/3d102187c4b096a8b32b62558d8aca6ab633d406/address-sanitizer/spec/spec2006-asan.patch.

[41] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory Tagging and how it improves C/C++ memory safety. 2018.

[42] Kanad Sinha and Simha Sethumadhavan. Practical Memory Safety with REST. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 600--611, 2018.

[43] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46--55, 2015.

[44] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *Twelfth European Conference on Computer Systems (EuroSys)*, pages 405--419, 2017.

[45] Sterling Vinson, Rachel Stonehirsch, Joel Coffman, and Jim Stevens. Preventing Zero-Day Exploits of Memory Vulnerabilities with Guard Lines. In *SSPREW9*, pages 600--611, 2018.

[46] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High System-Code Security with Low Overhead. In *IEEE Symposium on Security and Privacy (S&P)*, pages 866--879, 2022.

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

[48] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2313--2328, 2017.

[49] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 479--494, 2021.

[50] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *ASPLOS*, pages 631--644, 2019.

[51] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating Address Sanitizer. In *USENIX Security Symposium*, 2022.

A FloatZone instrumentation example

We show a small example of what the assembly code generated by the FloatZone compiler pass looks like. Listing 3 shows a simple function affected by an out-of-bound access, and Listing 4 shows the function in assembly after adding instrumentation.

Listing 3 C source code of `buggy`.

```
char buggy(unsigned int idx) {
    char buf[16];
    fill_buffer(buf);
    return buf[idx];
}
```

Listing 4 Instrumented assembly code (Intel syntax) of `buggy`.

```
buggy:
  push   rbx
  sub    rsp,0x30
  mov    ebx,edi
  lea    rdi,[rsp+0x10]

  ;Apply redzones to buf[16]
  movaps xmm0,XMMWORD PTR [rip+0xe8d] ;898b8b8b8b..
  movaps XMMWORD PTR [rsp],xmm0       ;Underflow rz
  movaps XMMWORD PTR [rsp+0x20],xmm0  ;Overflow  rz

  ;fill_buffer(buf)
  call   fill_buffer

  ;return buf[idx];
  mov    ecx,ebx
  mov    al,BYTE PTR [rsp+rcx*1+0x10]

  ;FloatZone check
  movss  xmm0,DWORD PTR [rip+0xe45]   ;0b8b8b8a
  vaddss xmm15,xmm0,DWORD PTR [rsp+rcx*1+0x10]

  ;Zero out buf[16] redzones
  xorps  xmm0,xmm0
  movaps XMMWORD PTR [rsp],xmm0       ;Underflow rz
  movaps XMMWORD PTR [rsp+0x20],xmm0  ;Overflow  rz

  add    rsp,0x30
  pop    rbx
  ret
```

B Fuzzing evaluation

Table 6 shows the benchmarks setup used in the fuzzing evaluation, while Figure 9 offers an overall comparison of the obtained coverage among FloatZone, ReZZan and ASan--.

Benchmark	Version	AFL++ Command Line
file	1.62	`-m magic.mgc @@`
libpng	1.6.38	`@@`
tcpdump	5.0.0	`-n -e -r @@`
cxxfilt	2.31	`-n`
nm	2.31	`@@`
size	2.31	`@@`
objdump	2.31	`-a @@`

Table 6: Fuzzing evaluation setup taken from ASan-- [51].

C Exception-based checks search space

Modern CPUs can generate various exceptions, such as *page fault*, *alignment check* and *division by zero*. However, in our analysis, we found that very few of them can be used to express an equality condition. Thanks to the inherent mathematical properties, Floating Point exceptions offer a good degree of freedom in selecting the conditions that will trigger this event. Specifically for addition and subtraction operations, we demonstrated (cf. Figure 2) how the underflow exception allows to express equality conditions with a large pool of constants. The remaining exceptions offer less ideal properties:

- **Invalid**: An *SNaN* is sufficient to trigger this exception, and since there are almost 2^{23} *SNaN* encodings, it is not suitable to express an equality condition.
- **Division by zero**: Only possible with the expensive division operation.
- **Inexact**: Not suitable since this exception is almost always triggered. Operation results are rarely perfectly represented in floating point format.
- **Overflow**: There is a sweet-spot of numbers that experience overflow only with a small count of numbers, therefore making it a suitable candidate. For example, the number `0x73000000` will only overflow when added to `0x7f7fffff` (i.e., `FLT_MAX`). However, overflow is not ideal to express comparisons due to the low choice of faulting numbers, since it is always a set containing `FLT_MAX` and its closest numbers, i.e. `0x7f7ffffe`, `0x7f7ffffd`, etc.).

On the other hand, a limitation of underflows regarding the FloatZone design is that only numbers close to the selected constant will trigger exceptions. This makes it impossible to find an equivalent of `0x8b8b8b8b`**`89`** for the underflow redzone (i.e. `0x`**`89`**`8b8b8b`) since the most significant byte predominantly encodes the floating point exponent.

Benchmark	ASan	ASan--	FZ	FZExt
400.perlbench	3.93	3.50	1.46	1.85
401.bzip2	1.53	1.47	1.29	1.54
403.gcc	2.72	2.68	1.92	2.02
429.mcf	1.30	1.19	1.08	1.23
433.milc	1.13	1.09	1.11	1.17
444.namd	1.55	1.44	1.17	1.21
445.gobmk	1.56	1.46	1.19	1.23
447.dealII	2.14	1.99	1.34	1.42
450.soplex	1.50	1.47	1.22	1.30
453.povray	2.25	1.98	1.65	1.83
456.hmmer	2.30	2.29	1.58	2.32
458.sjeng	1.73	1.45	1.14	1.16
462.libquantum	1.07	1.05	1.02	1.03
464.h264ref	2.35	1.85	2.28	2.35
470.lbm	1.21	1.14	1.02	1.02
471.omnetpp	1.88	1.75	1.61	1.62
473.astar	1.42	1.30	1.18	1.19
482.sphinx3	1.62	1.60	1.32	1.34
483.xalancbmk	2.56	2.56	2.09	2.23
600.perlbench_s	2.17	1.75	1.60	2.06
602.gcc_s	2.02	2.07	1.44	1.48
605.mcf_s	1.20	1.16	1.11	1.17
619.lbm_s	0.97	0.97	0.99	0.99
620.omnetpp_s	2.33	2.24	1.78	1.80
623.xalancbmk_s	1.79	1.76	2.02	2.09
625.x264_s	1.95	1.81	2.10	1.98
631.deepsjeng_s	1.54	1.38	1.19	1.27
638.imagick_s	1.61	1.50	1.25	1.26
641.leela_s	1.68	1.64	1.27	1.28
644.nab_s	1.47	1.39	1.10	1.10
657.xz_s	1.29	1.26	1.12	1.22

Table 7: SPEC CPU runtime overheads. FZ = FloatZone

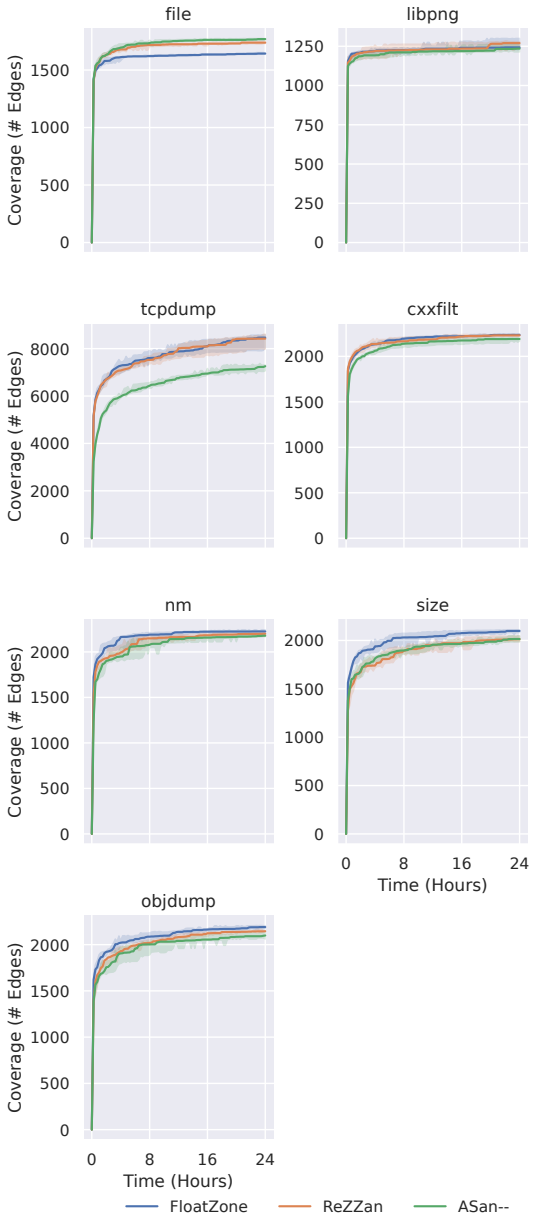

Figure 9: Coverage plots of our fuzzing evaluation. The plots show the median of 15 runs with the corresponding 95% confidence intervals.