# RangeSanitizer: Detecting Memory Errors with Efficient Range Checks

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

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

## Abstract

Sanitizers for spatial and temporal memory errors have become a cornerstone of security testing. Popular redzone-based sanitizers such as AddressSanitizer (ASan) offer high compatibility and effectiveness through the use of redzones, but incur significant runtime overhead. A major cause of this overhead is the traditional use of per-object redzone metadata, which constrains the sanitizer to check individual *addresses* rather than entire *ranges* of memory at once—as is done by classic bounds checkers based on per-pointer metadata.

In this paper, we introduce RangeSanitizer (RSan), a redzone-based sanitizer that introduces a novel metadata and check paradigm. RSan combines the compatibility of redzones with a rich per-object metadata format that allows for *range* (rather than address) checks and powerful optimizations. RSan stores *bounds* information inside the underflow redzone associated with each memory object. By combining *pointer tagging* with power-of-two size classes, RSan can swiftly locate metadata and validate an access to an arbitrary memory range with a *single* check. RSan incurs a geomean runtime overhead of 44% on SPEC CPU2017, faster than all state-of-the-art redzone-based sanitizers and twice as fast as ASan. Additionally, fuzzing with AFL++ and RSan as sanitizer improves state-of-the-art throughput by up to 70%.

## 1 Introduction

Memory errors such as buffer overflows and use-after-frees are still the leading cause of security issues in production [23, 45], with serious real-world consequences as also evidenced by the recent CrowdStrike incident [14]. Over time, different solutions have been proposed to detect memory errors. *Bounds checkers* first addressed spatial memory errors by associating *base* and *bound* information with every pointer. By tracking such *per-pointer metadata* throughout the execution, out-of-bounds pointer dereferences can raise an alarm. Early adopters use rudimentary schemes based on *fat pointers* [5, 34, 49, 59]. To improve compatibility, later installments store bounds metadata in a disjoint structure [17, 35, 53, 64] and allow nondereferenced out-of-bounds pointers [2, 16, 18, 38, 47, 55, 58, 65, 68]. Nonetheless, per-pointer metadata tracking still poses compatibility challenges (e.g., supporting pointer to integer casting [9]) other than incurring significant performance overhead [52].

Roughly a decade ago, AddressSanitizer (ASan) [56] and LBC [29] shifted away from the per-pointer metadata tracking model, and instead popularized a per-object metadata design. These solutions use *redzones* [30, 51, 54] to denote invalid regions of memory (e.g., inter-object padding) and track per-object metadata to distinguish valid from redzone addresses. Although spatial detection guarantees are reduced due to the limited redzone size, this strategy improved compatibility and performance while also extending detection capabilities to temporal errors. Since then, redzone-based sanitizers have been widely adopted by modern fuzz testing campaigns and also received much attention in literature, with adaptations to improve metadata management [33] as well as remove [39, 62, 63, 67, 69] or accelerate [25, 42] checks.

While redzone-based systems offer many benefits, a valuable asset was lost in the transition from the per-pointer metadata world: the ability to efficiently validate an entire *range* of memory in one go, as done by classic bounds checkers. For instance, consider a `memset(buf, 0, x)` operation. With per-pointer bounds metadata, two comparisons suffice to check that addresses `buf` and `buf+x` are within bounds. In contrast, a solution like ASan needs to iteratively scan the entire `[buf, buf+x]` range for the presence of redzones. Despite recent optimizations to reduce the number of scan iterations [42, 69], the performance of existing redzone-based sanitizers is fundamentally limited by the need to check individual *addresses* rather than entire memory *ranges* at once.

In this paper, we introduce RangeSanitizer (RSan), a redzone-based sanitizer for heap, stack, and global memory with a novel metadata and check format that enables fast range-based checks as seen in traditional bounds checkers. RSan combines the compatibility and effectiveness of redzones with rich per-object metadata and optimizations inspired by per-pointer metadata solutions—a best of both worlds. As a result, RSan is able to check an arbitrary (range and non-range) access for buffer underflow, overflow, and use-after-free, all with *one* metadata lookup and *one* comparison.

Conceptually, RSan functions by quickly retrieving the *base address* of any memory object, and locating the upper *bound* information (i.e., where the object ends) stored inside the *redzone* that directly precedes the base. RSan ascertains the validity of memory accesses by evaluating whether an access is within the upper bound associated with the corresponding object. As a result, RSan avoids disjoint metadata structures (like *shadow memory* [50]), without introducing any unwanted false positive bug detections [25].

More specifically, RSan finds the base address of objects by combining a power-of-two allocator with *size classes* and *pointer tagging*. Upon allocation, RSan pads objects with redzones, and stores the corresponding power-of-two size class in the upper bits of the resulting pointer. Subsequent memory accesses can quickly retrieve the base address by reading the pointer tag and masking off the size class from the pointer. The upper bound metadata then resides in the underflow redzone, exactly eight bytes before the base.

We show that this design enables powerful compiler-based optimizations that are traditionally unavailable for address-based sanitizers. In contrast to recent work, we avoid non-conservative optimizations that violate the C standard [42] or allow potential corruption of metadata [69]. Furthermore, we show that modern address masking features (e.g., Arm Top-Byte Ignore [4] and Intel Linear Address Masking [32]) improve the memory overhead of RSan. However, RSan also supports systems where address masking features are not available (e.g., legacy x86) through *implicit pointer tagging*.

We evaluate the performance of RSan and show that, with 44% geomean runtime overhead on the SPEC CPU2017 benchmarking suite, RSan outperforms all the state-of-the-art redzone-based sanitizers [25, 42, 56, 69]. Our evaluation also shows that RSan is a generic solution that performs well regardless of the underlying hardware, in contrast to a more hardware-sensitive sanitizer like FloatZone [25] which relies on acceleration from a powerful FPU. Additionally, we show that *fuzzing* with AFL++ and RSan as sanitizer increases state-of-the-art throughput by up to 70%. Finally, we show that RSan provides the same memory error detection guarantees as ASan and even covers some more bug scenarios.

**Contributions** We make the following contributions:

- We introduce a novel metadata format that stores bound information inside redzones and pointer tags.
- We show that the resulting design allows for efficient *range checks* to detect spatial and temporal errors.
- We present and open source RSan for both x86 and Arm, and evaluate it against state-of-the-art sanitizers.

**Source** `https://github.com/vusec/rangesanitizer`

## 2 Background

We consider sanitizers to detect *spatial* and *temporal* memory errors. Spatial errors such as buffer overflows stem from accesses that occur outside the bounds of an object. Temporal errors such as use-after-free stem from accesses that occur outside the lifetime of an object. We consider two main metadata formats to detect such errors: pointer-based and redzone-based. Alternative strategies do exist, but such solutions are less favorable for general-purpose security testing, being often limited to a single vulnerability class [15, 24, 26, 37] or relying on specific hardware extensions [27, 40, 70].

**Pointer-based** By tracking *base* and *bound* (or size) metadata for every pointer, bounds checkers enforce that pointers only refer to or access the intended object. The metadata is either stored inside the pointer or is associated with the pointer in a disjoint data structure. The checks evaluate whether a memory access is within the valid range, i.e., above the lower bound and below the upper bound. One significant drawback of pointer-based metadata is the incompatibility with integer arithmetic on pointers, for example due to alignment [9].

**Redzone-based** With redzones, metadata is associated with memory *objects* instead of pointers. Fundamentally, redzones serve as inter-object *padding* and sanitizers detect bugs by evaluating whether an access operates on a redzone or regular (valid) memory. Freed memory can similarly be marked as a redzone to detect temporal errors. Redzone-based solutions often use shadow memory [50], a disjoint metadata structure that tracks which areas of memory are valid.

## 3 Overview

RSan's primary goal is to sanitize spatial and temporal memory errors with high performance and compatibility. To this end, RSan introduces a metadata format that enables checking the validity of a *range* of memory with a single comparison. Fundamentally, RSan detects bugs in programs written in unsafe languages such as C or C++ through compiler-based instrumentation and a modified memory allocator. Figure 1 displays a high-level overview of RSan's components.

At its core, RSan operates on memory objects (e.g., heap allocations), pointers to objects, and accesses to objects (i.e., load and store operations). Crucially, RSan finds memory errors at runtime by equipping the source code with sanitizer *checks*. These checks evaluate all load and store operations for validity and abort the program in the case of a memory error. In order for the checks to function, RSan requires memory objects and pointers to follow a specific organization.

More specifically, RSan uses a custom memory allocator to protect memory objects by surrounding them with inaccessible memory areas (commonly called *redzones*). When

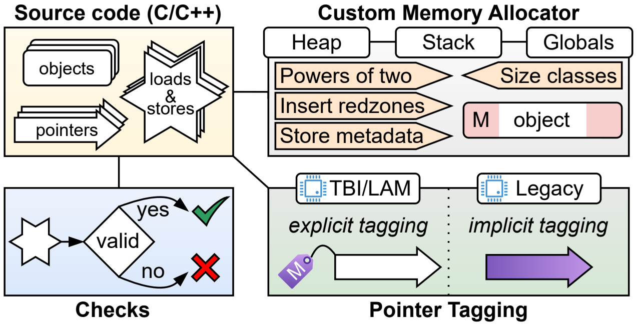
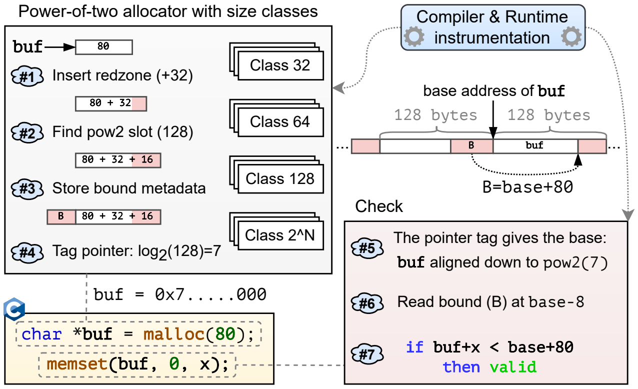


Figure 1: High-level overview of RSan's components.



```
char *buf = malloc(80);
  memset(buf, 0, x);
```

```
if buf+x < base+80
   then valid
```

Figure 2: Overview of RSan's workflow.

errorenous memory operations (e.g., a buffer overflow) access a redzone, RSan aborts the program. RSan introduces a novel metadata format where it stores object bound information *inside* the redzones. To ensure RSan can quickly retrieve the metadata to perform its checks, it uses two additional properties. First, the memory allocator uses power-of-two *size classes* to guarantee objects start at predictable alignment. Second, RSan relies on storing some metadata inside pointers by means of *pointer tagging*. On modern architectures, RSan tags pointers *explicitly* by using the unused (ignored) bits thanks to Arm's Top-Byte Ignore and Intel's Linear Address Masking. On legacy architectures, where such address masking features are unavailable, RSan instead performs *implicit* tagging, where the tags are encoded inside the pointer.

**Workflow** Figure 2 further clarifies the overall workflow of RSan by showing how RSan detects bugs in seven steps. The figure contains an example C program that performs a memory allocation of 80 bytes, followed by a `memset` operation that sets the first `x` bytes to zero. It is the task of a bug sanitizer to confirm that the access range `[buf, buf+x]` is valid with respect to the memory object (i.e., not out-of-bounds and not deallocated). RSan evaluates the validity of this access using the following paradigm: it performs a metadata lookup on the *lower* address (`buf`) to retrieve the bounds information of the object, and then compares whether the *higher* address (`buf+x`) exceeds the retrieved upper bound.

In the first step (see Figure 2), RSan pads the object with a redzone, which, in our example, increases the size by 32 bytes. In step two, RSan extends the redzone of the object such that the total size of the object becomes a power of two. This ensures the object can reside in a power-of-two *size class*, a region of memory with objects of the same size. In step three, RSan stores the upper bound of this allocation in the underflow redzone (i.e., the overflow redzone of the preceding object). Then, right before returning the resulting pointer, in step four, RSan tags the upper bits of the pointer with the binary logarithmic power-of-two size class (7, in this case).

Before the program executes the `memset` operation, RSan checks whether the access is valid. To perform this check, RSan needs to retrieve the bound metadata for the target object. It achieves this by first locating the *base* address of the object in step five by aligning down the lower address (`buf`) using the pointer tag. Afterwards, in step six, RSan retrieves the bound metadata by reading at the end of the underflow redzone: address `base-8`. Finally, RSan confirms the validity of the access in step seven by comparing whether the higher address of the access (`buf+x`) is within the retrieved bound.

RSan's design comes with multiple benefits that promote high performance. For instance, RSan does not require disjoint metadata (e.g., *shadow memory* [50]) to track where the redzones reside, and instead solely relies on in-band metadata. Note that RSan does not introduce false positive bug detections as seen in other in-band metadata formats [6, 25]. Another crucial aspect is that RSan checks for buffer under- and overflow, as well as temporal errors, in a range, all with a *single* comparison. In contrast, existing redzone-based sanitizers need to iteratively scan memory for redzones to validate common range operations, and existing *bounds checkers* require at least two checks: one for underflows, one for overflows, and even a third if temporal errors are to be considered [46].

## 4 Design

Central to RSan's design is the ability to quickly check the validity of any arbitrary range of memory from a lower to a higher address. We make the key observation that even regular load and store accesses are (small) range operations. For instance, a 4-byte load at address `p` can in fact be represented as a range check on `[p,p+3]`. A key requirement of such checks is that RSan can find the upper bound (i.e., the metadata) of the object corresponding to any arbitrary pointer, regardless of where (i.e., what offset) the pointer points to in the object. RSan solves this challenge by storing the power-of-two size class in the unused bits of pointers. By knowing the size class, any pointer can be *aligned down* (i.e., masked) to its base address, after which the metadata can be retrieved at `base-8` (inside the padding serving as redzone).

Note that RSan implicitly checks for buffer underflows as well as for temporal errors with its (upper) bound check. Buffer underflows are caught as a result of negative out-of-bound pointers reading the upper bound metadata of a *previous* (i.e., lower address) object. Conceptually, a buffer underflow of object `N` can be viewed as a buffer overflow from the perspective of object `N-i` (with `i>0`). RSan detects temporal memory errors by updating the bound metadata to zero upon deallocation, causing any future bound validity check on such an object to fail until (delayed) reallocation. In the following, we describe the design of RSan's components in more detail.

## 4.1 Allocator

RSan relies on a memory allocator using power-of-two *size classes*. Size classes are regions of memory where objects of the same (slot) size are allocated. Since size classes provide a uniform layout of objects next to each other, if the start of the size class region is aligned to the slot size, then so are all the following objects. As a result, for any arbitrary pointer to a memory object, if we know its size class we can find the base of that object by aligning the pointer down to the slot size.

Since storing and fetching additional disjoint metadata (e.g., in a map) to track base addresses incurs large overhead penalties, RSan instead finds the base of an object purely through pointer arithmetic. To that end, upon allocation RSan stores some metadata in the unused upper pointer bits to enable retrieving the base. This technique is also referred to as *pointer tagging*. Since the number of bits of the *tag* are constrained (e.g., 8 bits with Arm Top-Byte Ignore), storing the base as a complete address embedded within the pointer is infeasible without breaking the pointer format (as seen with *fat pointers*). Instead, RSan uses object alignment properties to store an encoded representation of the base in the available bits.

To this end, RSan restricts the size classes to powers of two, which allows us to encode size classes up to `2^64` in just six bits by storing the binary logarithm (`log2`) of the power-of-two as pointer tag. For the heap, popular allocators such as TCMalloc, Scudo, and jemalloc already provide size classes, and can trivially be restricted to power-of-two classes. For the stack and global memory, we draw from the designs proposed by recent work [27]. The stack is split into multiple stacks, each one dedicated to objects of a particular power-of-two size class. Global variables are moved into global arrays, each array containing entries for a particular size class.

## 4.2 Pointer tagging

As mentioned before, a key ingredient of RSan is the ability to quickly locate the base address (i.e., the start) of memory objects. RSan achieves this through *pointer tagging*.

**Explicit tagging** On modern Arm and Intel CPU architectures, the TBI and LAM features allow for software to use the most significant (unused) bits of pointers. With such an *address masking* feature available, the hardware ignores these bits when accessing memory and tagging pointers becomes trivial. Upon allocation, RSan computes the pointer tag by taking the `log2` of the power-of-two size class corresponding to the object. For this purpose, RSan relies on the `ctz` instruction (available on both Arm and x86) that counts the number of trailing zeroes of a value. The pointer tag is then set using a regular `OR` binary arithmetic operation, and ultimately the (explicitly) tagged pointer is returned to the user program. The address mask of TBI starts at bit 56 (8 bits tag), while LAM excludes bit 63 and starts at bit 57 (6 bits tag) or bit 48 (15 bits tag) depending on the configuration of the system.

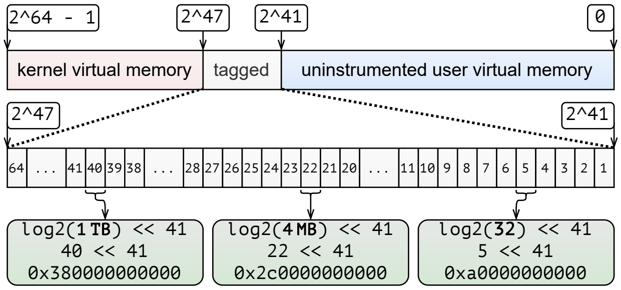


Figure 3: Address space partitioning for implicit pointer tagging on legacy architectures. Starting addresses of three size classes are depicted: 32 bytes, 4 MB, and 1 TB.

**Implicit tagging** While AMD has announced an address masking feature (Upper Address Ignore [3]), it is not yet available in commodity hardware. Furthermore, Intel only recently introduced LAM in its latest processors. Hence, for RSan to support common (legacy) architectures, we rely on *implicit pointer tagging*, similar, in spirit, to the tagging scheme adopted by Low-Fat pointers [18]. Specifically, we rearrange the 48-bit user-space address layout so that all objects of a particular size class are allocated in a dedicated address range. As a result, the objects are *implicitly* tagged with their size class, based on where they reside in the address space.

More specifically, we interpret bits [46:41] of every user pointer as *implicit* pointer tags. Conceptually, these six bits encode the power-of-two size class just as with explicit pointer tagging. However, since the hardware does not ignore these bits, RSan needs to ensure objects of the corresponding size class are allocated in the correct address range (such that the tag matches the class). Figure 3 displays how RSan rearranges the address space to support implicit pointer tags. The address space range between `2^41` and `2^47` is partitioned into equal chunks of `2^41` bytes each. As a result, each class hosts 2 TB of virtual memory, which is hence also the maximum object size. No such size limitation exists for *explicit* tagging. Each chunk represents the power-of-two size class that corresponds to the implicit tag bits being set. For instance, the fifth chunk

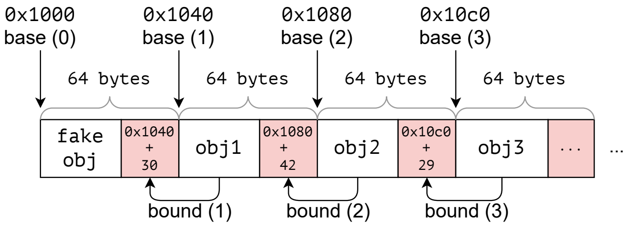


Figure 4: Redzone and metadata management in RSan. Three example objects are created with sizes 30, 42, and 29 bytes. The figure denotes the aligned base addresses and the locations where bound metadata can be retrieved for each object.

represents the size class of 32 bytes, because when interpreting bit 41 as the least significant bit, the six-bit binary value is equal to 32. Note that the remaining upper bits [63:47] are all zero for user pointers. Hence, the implicit pointer tag is extracted similarly to the address masking variant: a constant shift operation (`ptr >> 41` for implicit tagging).

The uninstrumented virtual memory region is used for any memory objects or mappings that do not (or cannot) require sanitization. This includes external libraries and file mappings, but also provably safe stack allocations. RSan guarantees that the program does not accidentally map in the implicit tagging range. Otherwise, accesses to this memory would wrongfully be interpreted as tagged, and the subsequent spurious metadata lookup causes undefined behavior. To this end, at program startup RSan moves the regular stack (typically mapped in the tagged area) into the uninstrumented area. Additionally, RSan ensures the entire tagged address space portion is reserved by the allocator, which prevents future uninstrumented mappings from residing in the tagged area.

## 4.3 Metadata

Similar to prior solutions [20,25,27,29], RSan repurposes redzones to not only serve as spatial guards, but also to contain in-band metadata. Similarly, RSan only allows the instrumentation to load and store metadata in the redzones, not the program itself. In contrast to prior solutions, RSan stores the metadata of the current object in the redzone padding of the *previous* object slot, which, as we will explain, is crucial for the properties we desire for RSan's sanitizer checks.

Figure 4 depicts RSan's redzone and metadata layout. Upon allocation, RSan pads the object with a redzone, whose size is larger for larger objects. The redzone is then (possibly) extended for the padded object to fit in a power-of-two size class slot. RSan stores the bound information (i.e., where the object ends) right before the start of the object. This location concerns the underflow redzone, which is the right-hand side padding inserted by the previous object. The bound metadata value is equal to the base address plus the originally requested (nonpadded) object size. The figure shows where the bound information for each object can be found, e.g., "bound (1)" denotes the bound metadata location for object 1 accesses.

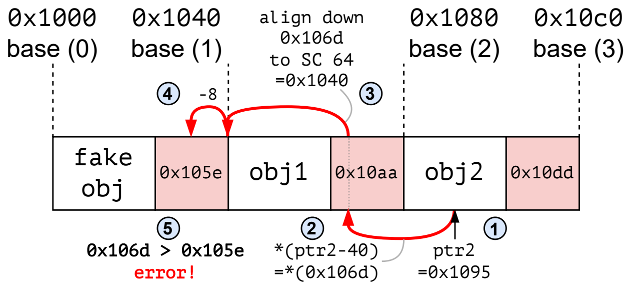


Figure 5: How RSan detects buffer underflows. In this example `ptr2` initially points to the middle of `obj2`, after which it is offset to `ptr2-40` (underflow) and dereferenced. RSan retrieves the upper bound metadata by locating the base address, and reports an error due to an out-of-bounds violation. This example concerns a 1-byte access (i.e., no range offset)

There are two reasons why RSan stores its metadata *before* objects, instead of *after* objects. First, computing the location of the metadata is a cheaper operation when inserted before the object, because aligning down requires less arithmetic than aligning up. Second, we experimentally measured that storing the metadata before the start of the object provides better memory locality (and therefore less runtime overhead), especially when considering large objects. We suspect that it is simply more common for programs to first access the lower part of memory allocations (e.g., a linear upwards pattern).

Next, there are two design options for how to place the in-band metadata before objects, i.e., how to insert the redzones. As mentioned before, RSan stores its per-object metadata in the redzone padding of the *previous* object. Since the allocations reside in a size class, redzones in-between objects can be shared, and hence from the perspective of a size class slot, the padding can either be inserted at the start of the slot (i.e., in the current slot, before the object), or at the end of the slot (i.e., in the previous slot, after the object). The crucial benefit of storing redzones at the end of the slot is that the start of the object remains aligned with the alignment of the slot. Hence, as soon as a pointer underflows, aligning down the pointer results in reading the metadata of the *previous* object, which (correctly) triggers an out-of-bounds violation.

In contrast, consider the unwanted consequence of inserting redzones at the *start* of size class slots. This effectively moves up the (true) start address of the object within the slot, meaning the first byte of data no longer starts at the alignment of the class. In that case, buffer underflows are no longer easily detectable with a single upper-bound comparison, since a pointer that goes out-of-bounds below the start of the object can still point inside the size class slot, and hence is considered valid for the associated upper bound.

Figure 5 shows how RSan detects out-of-bounds violations in both directions with a single comparison. To this end, size classes need to be prefixed with a *fake object* to host the meta-

**Listing 1** Computing the base address of an object using the size class pointer tag and two variable shifts.

```
sizeclass_tag = ptr >> 41;  // 56 on Arm
base = (ptr >> sizeclass_tag) << sizeclass_tag;
```

**Listing 2** Improved computation of the base address of an object on x86 architectures using the `bzhi` instruction.

```
sizeclass_tag = ptr >> 41;
base = ptr ^ bzhi(ptr, sizeclass_tag);
```

**Listing 3** Complete sanitizer check on `[ptr, ptr+offset]`: metadata lookup, bound comparison, and slow-path check. `offset` denotes the number of bytes the operation spans (e.g., 8 for an 8-byte load, or 100 for a `memset(ptr, 0, 100)`).

```
1 tag = ptr >> 41; // 56 on Arm
2 base = ptr ^ bzhi(ptr, tag); // varshift on Arm
3 bound = *(base - 8); // load metadata
4 if( ptr + offset > bound ){
5     // untagged memory slow-path check
6     if( tag != 0 ){
7         // error!
8     }
9 }
```

data of the first real object (see Figure 4 and 5). Additionally, since RSan also needs to detect underflows on the first (real) object, a second piece of prefix padding is necessary to store a metadata bound of size zero for the fake object itself (not visualized). Since size class slots can grow large, we replace the prefix for the fake object with a guard page whenever favorable in terms of memory consumption.

Finally, RSan also detects temporal errors such as use-after-free and double-free. This is done by setting the bound metadata to zero upon object deallocation. As such, when a deallocated memory object is accessed, the corresponding check always reports an error, because every address fails the bounds check when the stored upper bound address is zero. RSan delays the reuse of deallocated memory using a *heap quarantine*, as also done in existing redzone-based sanitizers [25, 28, 56]. Note that the necessary metadata update upon deallocation is minimal in RSan, with only a single store to the base metadata. In contrast, existing redzone-based sanitizers invalidate metadata with a loop that depends on the object size.

## 4.4 Checks

With all metadata and memory organization set up, RSan can perform its sanitizer checks with a fixed paradigm: for any access from address `L` to `H`, look up the bound metadata on the lower address `L` and compare the bound with the higher address `H`. Note that for a one-byte access `L` and `H` are identical.

Since programs tend to contain many load and store operations, the speed of the check is essential for the overall performance. In order for RSan to find the base address of any arbitrary pointer, we require a fast method to align down pointers using the size class stored as pointer tag. Thanks to the alignment properties RSan enforces with its memory allocator, any pointer can be cut down to its base address using the pointer tag and two variable shifts. Listing 1 shows how two variable shifts effectively mask off the lower bits of a pointer up to the point of the size class alignment. For example, using the pointer `0x106d` from Figure 5 with size class 64 bytes (tag 6): `(0x106d >> 6) << 6 = 0x1040`. Note that the pointer tag (6) is omitted from the pointer in this example.

We conducted some initial performance experiments using this variable shifting technique, which indicated that the Intel CPUs we tested (generations 10 up to 14) do not seem to be optimized for shifting with a variable (i.e., non-constant) operand. Interestingly, we found that the AMD and Arm platforms do not experience a similar significant slowdown from variable shifting. To avoid variable shifting becoming a performance bottleneck on Intel CPUs, we explored alternative methods to implement the base address computation. On x86 architectures, we found a suitable alternative in the `bzhi` instruction (Zero High Bits), which is available in the x86 bit manipulation instruction set since the Haswell and Excavator generations in Intel and AMD CPUs, respectively.

The `bzhi` instruction zeroes out the upper bits of a value starting with a specified (variable) bit position. Note that this operation effectively does the *opposite* of what RSan requires: RSan needs to zero out the *lower* bits with a variable bit position. Therefore, we combine the `bzhi` instruction with a `XOR` operation to achieve our desired computation. This results in the improved base address computation displayed in Listing 2. Note that the binary logarithmic encoding of the size class is beneficial here, since the pointer tag directly represents the starting index for `bzhi` to zero. We find that `bzhi` with a `XOR` performs significantly better on Intel CPUs (nearly halving the total sanitizer runtime overhead), while on AMD the difference is more modest (since variable shifting is relatively fast) but an improvement nonetheless. On Arm—which does not feature similar instructions—we resort to variable shifting, which fortunately already provides strong performance.

Aside from performance considerations, RSan needs to account for the possibility of performing checks on uninstrumented memory. Since uninstrumented memory is untagged, the base computation leaves the pointer intact (i.e., aligning to pointer tag zero). The metadata lookup then reads whatever data is stored 8 bytes before the pointer (which RSan ensures to be always mapped), potentially causing the bounds check to spuriously fail (i.e., a false positive bug detection). To rule out false positives, RSan introduces a *slow-path check* that is only executed when a bounds violation occurs. The slow-path check simply evaluates whether the pointer tag is zero, which indicates uninstrumented memory and can thereby be ignored. Listing 3 displays the complete algorithm of RSan's sanitizer check. After computing the base address of the object (lines 1

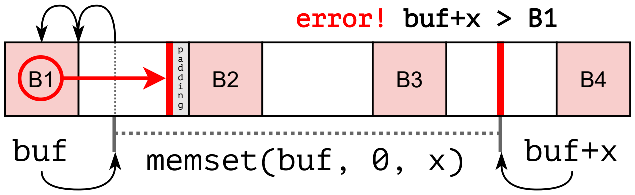


Figure 6: An example range check. The metadata lookup on address `buf` retrieves object bound `B1`. The upper address of the range (`buf+x`) is out-of-bounds with respect to `B1`.

and 2), the bound metadata is read at address `base-8` (line 3). The *higher* address of the access (i.e., the end of the *range*) is then compared with the retrieved bound (line 4).

Summarizing, Figure 6 illustrates how RSan checks a range of memory. This example concerns an out-of-bounds `memset` operation spanning `x` bytes on a target allocation `buf`. RSan retrieves the bound metadata (`B1`, which points to the end of `buf`) by aligning down the lower address of the range (`buf`). RSan then compares the upper address of the range (`buf+x`) to the retrieved bound and reports an out-of-bounds violation. Note that *within a range* there is effectively an unlimited redzone, since any address that exceeds the retrieved bound is invalid no matter how many redzones are crossed. We also highlight that RSan's range checks are constant-time operations, unlike traditional loop-based redzone checks [69].

## 5 Optimizations

With the ability to check ranges of memory, RSan unlocks a large space of optimizations that are historically inapplicable to redzone-based sanitizers. Before detailing such optimizations (Section 5.2 and 5.3) we first describe the generic ones (Section 5.1) our RSan prototype adopts from prior work.

### 5.1 Existing optimizations

Aside from optimizations that RSan unlocks with its metadata format, RSan also includes generic optimizations that were introduced by ASan-- [69]—note that not all optimizations are applicable, as some rely on ASan-specific shadow memory.

First, we remove unsatisfiable checks, which means we omit checks for memory accesses that are statically proven as safe because both the size of the corresponding object and the offset of the access are known (e.g., for a static stack allocation). Next, we remove recurring checks, where we deduplicate checks on the same memory location if there exists a check that is guaranteed to precede another. Last, we optimize neighboring checks. This optimizations considers three or more memory accesses to the same object, where the checks for the accesses spatially in the middle are skipped since the neighboring checks on both sides guarantee validity.

**Listing 4** Hoisting a loop invariant check.

```
check(&ptr[x])
for(uint i = 0; i < n; i++)
  ptr[x] = 0; // ptr[x] is loop invariant
```

**Listing 5** Hoisting a loop variant *range* check.

```
range_check(&ptr+s, &ptr+e)
for(uint i = s; i <= e; i++)
  ptr[i] = 0; // ptr[i] is loop variant
```

### 5.2 Loop optimizations

By optimizing sanitizer checks in loops we aim to reduce runtime overhead by moving computation outside of the loop as much as possible. Different optimizations are applicable depending on the characteristics of the memory operation: accesses can be executed (un)conditionally, and the target pointer of an access may be loop (in)variant.

We point out that moving checks (or computation through caching) out of a loop can conflict with use-after-free detection. Since heap memory can be freed inside a loop, a check *before* the loop may suffer from false negatives (i.e., checking too early), while a check *after* the loop may report a false positive (i.e., checking too late). ASan-- [69] addresses this by avoiding optimizations if loops contain a call of which an argument is the target pointer (which indicates a potential deallocation). Similarly, GiantSan [42] includes a post-loop check for use-after-free to accompany their metadata caching, because if an object gets freed inside a loop the cached metadata does not reflect this. However, it appears GiantSan does not consider the scenario where a program frees its memory in a valid manner (i.e., no use-after-free occurs in the loop), in which case the post-loop check reports a false positive.

We extend upon ASan--'s strategy by being more conservative: we include alias analysis to ensure that a loop does not execute calls that contain an argument which (possibly) aliases the target pointer, and otherwise we do not optimize. Note that there can still exist (rare) cases where a called function loads (an alias to) the pointer from memory and frees it, and a pre-loop check does not detect a potential use-after-free.

**Unconditionally executed accesses** We distinguish between two different cases for memory accesses that execute unconditionally inside a loop. If we can prove the pointer is loop invariant, then we hoist the check on that particular pointer out of the loop. Otherwise, if the pointer is loop variant, we employ LLVM's Scalar Evolution (SCEV) loop analysis to compute the start and end values of the pointer whenever possible. We then hoist out a check that validates the complete *range* the pointer spans throughout the loop. To determine whether a pointer is loop invariant, we use the algorithm introduced by ASan-- [69], which is more extensive than LLVM's loop invariance API. Unlike ASan--, we do not

**Listing 6** Caching the bound metadata in a local variable for loop variant conditional accesses. `check_ret_meta()` is a regular check that returns the found metadata for reuse.

```
1 uint64_t bound = 0;
2 for(uint i = 0; i < n; i++){
3   if( cond ){ // conditional
4     if( &ptr[i]+access_size > bound ){
5       bound = check_ret_meta(&ptr[i]);
6     }
7     ptr[i] = 0; // ptr[i] is loop variant
8   }
9 }
```

move checks for store operations *after* the loop, since such a delayed check allows for potential corruption of metadata.

Listing 4 shows a memory access with a loop invariant pointer, for which RSan hoists out the check out of the loop. In contrast, Listing 5 shows a loop variant pointer, for which RSan performs a range check by querying the SCEV API to obtain the start and end addresses. Note that the start and end are not always constant, and hence in order for RSan to find the lower and higher addresses, it queries SCEV for the minimal and maximal expressions. In practice, if these are not statically deducible, this results in a runtime comparison for whichever pointer is greater. The performance benefits of validating the access for the entire loop in a hoisted range check significantly outweighs the cost of comparing the start and end address once to determine which address is greater.

**Conditionally executed accesses** We also consider memory accesses inside loops that execute *conditionally*. For such cases, it is difficult to hoist out checks, because the access may never occur or only occur at certain iterations. Instead of hoisting, RSan *caches* the results of metadata lookups.

For loop variant pointers, we cache the metadata lookup if we can guarantee the pointer only *increases* throughout the loop. Note that if the pointer *decreases*, metadata caching is not possible, since detecting underflows depends on metadata lookups on a previous object (as discussed in Section 4.3). Listing 6 shows how RSan caches the metadata in a local variable the first time that the memory access executes. Essentially, RSan emits a quick comparison of the current pointer with the cached metadata, which is always true at the first access because the metadata is initialized to zero. RSan then performs its complete sanitizer check, and stores the retrieved metadata inside the local variable. Subsequent memory accesses reuse this cached metadata value, which saves having to perform the pointer arithmetic of the complete check. In the uncommon case where the pointer goes out-of-bounds, the quick comparison (line 4) will evaluate to true, and the subsequent complete check (line 5) causes a sanitizer error.

We also investigated an optimization for conditional loop *invariant* pointers where we cache the result of the check, and another for conditional accesses where the condition itself is loop invariant, meaning we can hoist the check if we evaluate all the conditions before the loop. For both cases, we did not observe performance benefits, which we suspect is due to the existing loop invariant code motion and loop unswitching compiler optimizations already handling these cases.

**Listing 7** Merging the check for a constant offset range.

```
range_check(&ptr[-10], &ptr[80]);
ptr[-10] = 0; ptr[15] = 0; ptr[80] = 0;
```

## 5.3 Range check merging

Aside from loop-based optimizations, RSan also allows for more generic range-based optimizations. We discuss multiple conditions that enable two or more checks to be merged into a single check that spans a larger range. Unlike optimizations seen in related work [42], RSan does not make assumptions about `GetElementPtr` (GEP) [44] base addresses being valid, because the C standard allows the GEP base to be out-of-bounds, while the GEP offsets bring the pointer back into bounds before dereferencing. In order to avoid false positive bug detections, our optimizations refrain from relying on the spatial validity of the GEP base address pointer.

**Constant offset merging** If we find multiple memory accesses that dereference the same memory object with different *constant* offsets, we merge these checks into a larger range check that spans the smallest (minimum) and largest (maximum) offsets. This optimization synergizes well with for example loop unrolling, where memory accesses in a loop get duplicated and operate on known constant offsets.

We find memory accesses that operate on the same object by searching for GEP instructions, and employing LLVM's *alias analysis* on the base of the GEP. Since we do not wish to merge sanitizer checks for accesses that may never execute, we use LLVM's *dominance* analysis to confirm a memory access has to succeed another. We group constant offset memory accesses by their GEP base address, and select the lowest and highest GEP offset. Listing 7 shows how we insert a range check that covers the complete offset range [min, max]. Note that the check can always be moved up to the first operation, since all GEP offsets are constant, and all GEP bases must be aliases, hence no data dependencies exist. To ensure the early check does not conflict with use-after-free detection, we scan for calls that may free the pointer (as with loop optimizations).

**Negative-positive pairs** While constant offsets allow for convenient merging, variable offsets are more challenging. If we find two memory accesses that operate on a GEP with variable offsets (and an aliasing base and dominance properties), we can only merge the checks if we prove one of the offsets is smaller than the other. We identify a scenario in

**Listing 8** Merging the check for a negative and positive pair.

```
int neg = -x; uint pos = x; // proven neg/pos
ptr[neg];                   // load
range_check(&ptr[neg], &ptr[pos]);
ptr[pos] = 0;               // store
```

**Listing 9** Merging the check for two pointers to the same object by computing the lower and higher address.

```
1 int x = ..., y = ...; // both variable
2 ptr[x];               // load
3 if( &ptr+x > &ptr+y )
4   range_check(&ptr[y], &ptr[x]);
5 else
6   range_check(&ptr[x], &ptr[y]);
7 ptr[y] = 0;           // store
```

which it is statically known which variable offset is larger than the other: if one offset is provably negative, and the other is provably positive. The SCEV API provides the ability to deduce whether a GEP offset is provably negative or positive (or undecided) based on type information, for example. Listing 8 shows how RSan merges the checks of a negative and positive offset pair of accesses. Note that this optimization only operates on pairs, since the signedness property does not extend to a third offset. Currently, we only explicitly apply this optimization on pairs for which the first operation is a load, such that the check can be delayed to the second operation, which guarantees temporal validity, the availability of both pointer operands, and avoids metadata corruption.

**Lower-higher pairs** For the remaining memory accesses, we aim to merge checks based on a runtime comparison for the greatest address. More specifically, for any pair of checks that operate on the same GEP base and that are guaranteed to execute together (dominance and post-dominance), we compute the unsigned minimum and maximum between the two addresses. As shown in Listing 9, we can turn the two checks into a single range check by knowing which address is greater than the other. With this transformation, we exchange the cost of one complete check (including pointer arithmetic and metadata lookup) with the min-max comparison on line 3. Like the previous optimization, we only apply this optimization on pairs that start with a load, such that the check can be delayed to the second access. We found an extension towards merging *chains* of checks not to be fruitful, likely due to the pressure imposed by propagating the pointers down to the last operation to perform multiple address comparisons.

## 6 Implementation

We implement a prototype of RSan using the LLVM (16.0.6) LTO compiler framework combined with a modified TCMalloc (2.15) memory allocator. For size classes on the stack we use the modified SafeStack design introduced by previous work [27]. Each size class on the stack is a separate region that gets allocated using the modified heap allocator.

**Implicit tagging** In order to implement implicit pointer tagging for legacy architectures, we modify the address space layout. First, we restrict TCMalloc's allocator to only allocate size class regions in the address space area corresponding to that size class (see Figure 3). This is done by dedicating a separate *freelist* of available memory to every size class. Next, we use a linker script to move global variables into the memory ranges matching their size class. We enable the *large code model* to allow the distance between the program and the global variables to exceed 4 GB. Finally, we create a custom dynamic linker to interpose on the true entry point of the program. By doing so, we can move the original (uninstrumented) stack below the implicit tagging space and plug the implicit tagging space with dummy mappings. This ensures that all subsequent mappings (e.g., libc, other libraries, calls to `mmap`, etc.) do not end up in the implicit tagging space.

## 7 Evaluation

In this section, we evaluate RSan in terms of its ability to detect bugs, as well as its overhead. For our experiments, we use an Intel i9-13900K machine (for implicit tagging) with Ubuntu 22.04 and 64GB RAM, an Arm-based Macbook M2 Pro (for TBI) with Debian 12 and 16GB RAM, and an Intel Ultra 9 285K machine (for LAM) with Ubuntu 24.04 and 128GB RAM. By default, we report results of our implicit tagging design, since the vast majority of existing systems do not support address masking features yet. We evaluate RSan with explicit pointer tagging separately in Section 7.4. All reported overhead numbers are the median of five iterations.

### 7.1 Security

First, we further detail and evaluate the security guarantees of RSan in terms of what bugs it detects. For temporal errors, RSan's detection guarantees are identical to ASan and fundamentally limited by the size of the heap quarantine to delay reuse of the memory. For spatial errors, we configure the padding RSan inserts for redzones to be at least as large as ASan, but often the effective size of the redzone becomes larger due to RSan's power-of-two allocator. Since RSan is a redzone-based solution, skipping over redzones remains possible (i.e., a non-linear out-of-bounds access that lands in valid non-redzone memory). However, our range optimizations do improve upon this drawback compared to ASan.

For example, consider the invalid memory access on line 5 in Listing 10. This concerns an out-of-bounds access that ASan does not detect, since the access skips over the redzone. However, RSan detects this bug thanks to its check merging

**Listing 10** Example bugs RSan detects but ASan does not.

```
1 char *ptr1 = malloc(16);
2 char *ptr2 = malloc(16); // to land into
3 range_check(&ptr1[10], &ptr1[32]);
4 ptr1[10] = 0; // in bound (valid)
5 ptr1[32] = 0; // skip redzone (invalid)
6
7 check(&ptr1+12);         // access offset = 8
8 *(uint64_t*)(ptr1+12);  // partial overflow
```

| Description (CWE) | Total | ASan | RSan |
|---|---|---|---|
| Stack buffer overflow (121) | 2,885 | 2,791 | 2,885 |
| Heap buffer overflow (122) | 3,365 | 3,318 | 3,365 |
| Buffer underwrite (124) | 1,001 | 907 | 1,001 |
| Buffer overread (126) | 657 | 563 | 657 |
| Buffer underread (127) | 1,001 | 907 | 1,001 |
| Double free (415) | 799 | 799 | 799 |
| Use-after-free (416) | 374 | 374 | 374 |

Table 1: Juliet Test Suite bug detection results.

capabilities, in this case specifically by the constant offset merging optimization. This property applies to all of RSan's range-based optimization (including loops).

Another scenario where RSan provides better detection guarantees than ASan concerns *partial* buffer overflows. Specifically, ASan cannot detect partial buffer overflows unless the start address of the access is 8-byte aligned, as a consequence of its compressed metadata. In contrast, RSan can detect any partial overflow thanks to its (range) checking paradigm. For example, the halfway out-of-bounds partial overflow on line 8 in Listing 10 goes undetected by ASan (because the 8-byte access is 4-byte aligned), while RSan detects partial overflows regardless of their alignment.

Next, to showcase RSan's capabilities to detect bugs, we use the NIST Juliet Test Suite [36] and real-world CVEs. Additionally, RSan successfully detects the known bugs in the SPEC CPU benchmarking suite, as well as a previously unknown bug: a buffer overflow on `argv` in 403.gcc. ASan does not detect this bug because it does not instrument `argv`—a non-fundamental limitation.

**Juliet Test Suite** We select the relevant categories for spatial and temporal memory errors from the NIST Juliet Test Suite (v1.3) and exclude test cases that do not (deterministically) contain a bug. Table 1 shows the results of this experiment. RSan reports a 100% detection rate of all bugs across all categories. In comparison, ASan misses some cases. For the heap buffer overflow category (CWE122), ASan lacks instrumentation for wide string operations (e.g., `wcscpy`), which is not a fundamental limitation. In the other categories, ASan misses buffer overflow errors where the out-of-bounds access skips over the redzone. In contrast, RSan successfully detects these bugs because the accesses land in uninitialized memory. Since RSan's allocator maps its size class regions with `MAP_ANONYMOUS`, the corresponding memory gets zero-initialized. As a result, the metadata lookup in this memory (after skipping the redzone) reads the bound value *zero*, which generates a sanitizer error due to the failed bounds check (any non-zero address is larger than zero). We point out that ASan does detect these bugs if we execute the test cases with a smaller input offset (within the redzone).

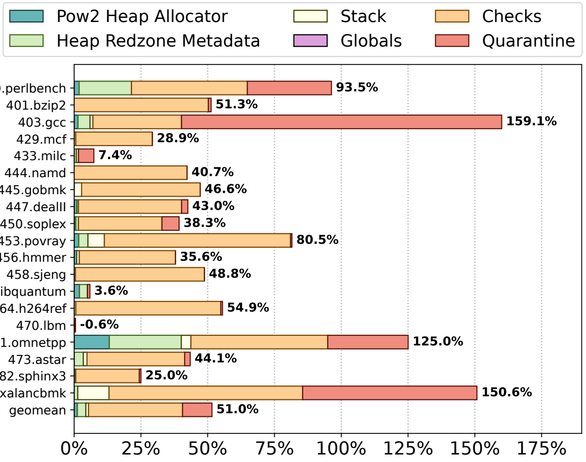


Figure 7: SPEC CPU2006 runtime overhead buildup of RSan.

**CVEs** To confirm RSan also detects real-world bugs, we use the same CVEs evaluated in recent work [25, 67, 69]. We execute all the relevant programs, however one test case is not reproducible on our machine due to legacy code clashing with a modern compiler and execution environment. Table 4 (in the Appendix) shows the results of this experiment. Both RSan and ASan successfully detect all the 16 evaluated CVEs.

## 7.2 Performance overhead buildup

On the SPEC CPU2006 benchmarking suite, RSan incurs a geomean runtime overhead of 51% compared to an unmodified LLVM and TCMalloc baseline. To better understand what this overhead consists of, we measured the slowdown of RSan's key components separately. Figure 7 displays the overhead buildup of RSan for each SPEC CPU2006 program.

RSan's memory allocator contributes 5 percentage points (5*pp*) of the total geomean runtime overhead, consisting of roughly 1pp from the power-of-two heap allocator (including implicit pointer tagging), 3pp from the redzone padding and setting the metadata, 1pp from splitting the stack into size classes, and no measurable slowdown from moving global variables into size classes. The remaining two components have a larger impact: 35pp from the sanitizer checks, and another 11pp from the heap quarantine. The heap quarantine in-

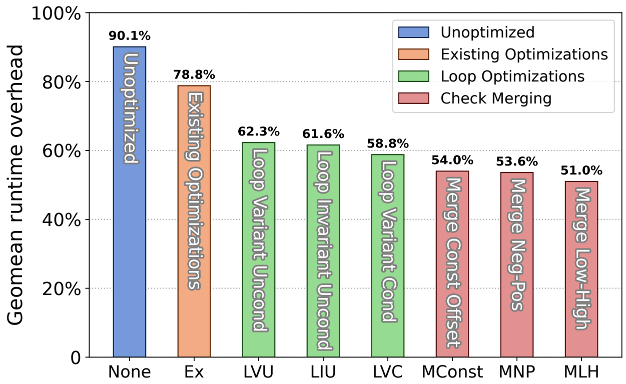


Figure 8: Runtime overhead progression of RSan on SPEC CPU2006 when enabling optimizations one-by-one.

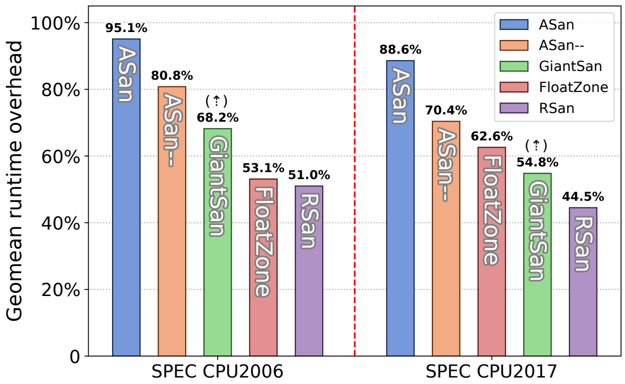


Figure 9: Runtime overhead comparison of RSan and the state of the art sanitizers on SPEC CPU2006 and CPU2017.

curs a large slowdown on the allocation intensive benchmarks (e.g., 126pp for 403.gcc). We confirmed that the performance penalty from the quarantine is caused by memory fragmentation, and not by any bottleneck in our quarantine data structure implementation (a simple thread-safe ring buffer).

**Optimizations impact** Next, we evaluate the performance benefits of the optimizations introduced in Section 5 on SPEC CPU2006. Figure 8 shows the cumulative geomean runtime overhead of RSan where each bar represents an optimization being enabled. As a starting point, unoptimized RSan incurs an overhead of 90.1%. This drops to 78.8% by including the existing ASan-- optimizations. Next, enabling all three loop optimizations reduces overhead to 58.8%, with the biggest contributor being the hoisted range checks for unconditionally executed loop-variant accesses (16.5pp). Finally, the three check merging optimizations reduce the total overhead by another 7.8pp, resulting in the final 51% geomean slowdown.

## 7.3 Comparison against the state of the art

Next, we put RSan's performance in perspective by comparing it to state of the art sanitizers. We compare RSan with ASan [56], ASan-- [69], FloatZone [25], and GiantSan [42]. To equalize the performance results, we measure the overhead of all of the sanitizers compared to a TCMalloc baseline. We port the mostly allocator-agnostic code of FloatZone to TCMalloc such that we do not attribute a slower underlying allocator as sanitizer overhead. Additionally, for an accurate comparison we also modify FloatZone to use the same minimal scaling redzone sizes as RSan and ASan, and we use FloatZone's extended mode that detects partial overflows (since RSan does too). ASan-- and GiantSan use ASan's custom memory allocator, which completely replaces the underlying heap allocator, hence using TCMalloc makes no difference. We disable all additional functionality of ASan that RSan does not implement (e.g., stack use-after-scope). We ensure that all compatible optimizations are enabled (e.g., FloatZone includes the applicable optimizations from ASan--). As an aside, we discover a concerning trend in recent related work pertaining evaluation practices, which we detail in Appendix A.2.

Figure 9 displays the geomean runtime overhead of each sanitizer on the SPEC CPU2006 and CPU2017 (SPECspeed) benchmarking suites. The results show that RSan provides the best performance with 51.0% and 44.5% overhead (respectively), being slightly faster than its closest competitor on CPU2006 (FloatZone), and 10 percentage points faster on CPU2017 (GiantSan). We point out that the measured overhead for GiantSan is incomplete, due to limitations in the available artifact[1]. Note that the overhead for ASan and ASan-- is relatively high compared to previous measurements [25, 69] because we compare to a faster baseline (TCMalloc). More detailed SPEC CPU results are available in Appendix A.3.

For SPEC CPU2017, we parallelize the four (out of eleven) SPECspeed benchmarks where OpenMP is available across all 32 performance (P) and efficiency (E) cores of the i9-13900K CPU. We observe that FloatZone's performance is highly sensitive to which cores it executes on. First, benchmarking singlethreaded programs on a P-core and OpenMP-enabled programs on all 32 P- and E-cores results in the 62.6% geomean runtime overhead displayed in Figure 9. Next, if we restrict the execution to P-cores only, FloatZone's overhead shrinks to 42%, comparable to RSan's (mixed-cores) overhead. However, if we restrict FloatZone to E-cores only, the overhead explodes. For instance, we measured a 27x runtime overhead for 600.perlbench running on an E-core. Clearly, FloatZone's performance is deeply intertwined with the performance of the FPU. In contrast, we observe that restricting RSan to only P-cores or only E-cores does not drastically

[1] The GiantSan artifact is closed source and contains a bug that results in missing memory errors in loops. After reaching out, the authors confirmed the issue and indicated that the released artifact does not match the one evaluated in the paper. The authors did not respond to our question whether the artifact serves at least as a lower bound for the overhead due to the missing checks.

| | HWASan | ASan | ASan-- | FZ | RSan |
|---|---|---|---|---|---|
| | | | *(Arm TBI)* | | |
| **Rt** | ~131.6% | 86.7% | 75.4% | N/A | 54.0% |
| **Mem** | ~6% | 193% | 215% | N/A | 228% |
| | | | *(Intel LAM)* | | |
| **Rt** | ~268.3% | 159.1% | 143.1% | 60.0% | 54.0% |
| **Mem** | ~4% | 187% | 188% | 159% | 207% |

Table 2: SPEC CPU2006 runtime and memory overhead comparison on CPUs supporting Arm TBI and Intel LAM. RSan uses explicit tagging. FloatZone (FZ) does not support Arm.

affect performance. For instance, on 600.perlbench RSan incurs a 1.8x overhead on a P-core, and 2.0x overhead on an E-core. For context, ASan incurs a 2.64x and 3.26x overhead on 600.perlbench on a P- and E-core, respectively. In summary, from this experiment we conclude that RSan provides the best performance while also being a generic solution.

Regarding memory consumption, the memory overhead of each sanitizer is dominated by the impact of the heap quarantine. RSan incurs a memory overhead of 239% in total and 78% when the quarantine is disabled. This overhead is slightly elevated compared to GiantSan (203%), ASan(--) (190%), and FloatZone (172%). The increased memory footprint is a consequence of the address space restrictions for implicit tagging, and the padding RSan inserts for power-of-two allocations. We point out that the power-of-two padding increases the effective size of the redzones (beyond ASan's redzone sizes), which can be beneficial for detecting non-linear buffer overflows [28]. Note that increasing the redzone size (e.g., to match the size class requirements or even beyond the current configuration) does not incur extra metadata overhead.

## 7.4 Address masking

In this section, we evaluate the performance of RSan in combination with address masking (Arm TBI and Intel LAM), which showcases the benefits of explicit pointer tagging. Table 2 displays the geomean runtime and memory overhead of RSan compared to HWASan [57], ASan, ASan--, and FloatZone on the SPEC CPU2006 suite. We omit GiantSan due to the limitations of its artifact (see footnote[1]). The geomean values for HWASan concern a partial result because (this version of) HWASan causes the 453.povray benchmark to crash.

First, we observe that ASan-style instrumentation (including HWASan) experiences a large slowdown on our last-generation Intel CPU. For reference, ASan introduces 95.1% overhead on the slightly older i9-13900K (Figure 9). The overhead increases to 159.1% on the latest Ultra 9 285K. We disclosed this behavior to Intel, which they reproduced and are currently investigating. In contrast, RSan's performance is unaffected and stable across the different architectures.

Additionally, we measured that HWASan, a hardware-assisted ASan variant which uses address masking to tag pointers, incurs a large runtime overhead of 131.6% and 268.3%, even though we configured HWASan in the most favorable way (e.g., inlining the checks). We find that RSan's runtime outperforms the other sanitizers by a significant margin, with 54.0% runtime overhead on both Arm and Intel being 21 percentage points faster than ASan-- with 75.4% overhead on Arm. Moreover, RSan is nearly three times as fast as ASan on our last-generation Intel CPU. Compared to the 60.0% overhead of FloatZone, RSan is 6 percentage points faster, while RSan does not introduce any false positives. Additionally, RSan's runtime overhead is slightly higher compared to the implicit tagging design (51.0%), which can be attributed to explicitly assigning tags to allocation pointers (`OR` operation).

Next, we show that explicit pointer tagging through address masking features improves the memory overhead of RSan. Although the heap quarantine somewhat conceals these benefits, the memory overhead of RSan without its quarantine is 42% on both architectures, which is significantly lower than the 78% measured with the implicit tagging design (Section 7.3). Note that the memory overhead in general can vary across architectures, for example due to the M2's 16 KB page size. We also observe that the heap quarantine imposes a lower *runtime* penalty on the Arm platform: 5.1pp on Arm compared to 11pp on x86 (both with implicit and explicit tagging).

Overall, RSan incurs a memory overhead of 228% on Arm and 207% on Intel, not far from ASan--'s 215% and 188% memory overhead, which shows that address masking features successfully reduce the memory overhead gap between RSan and ASan. FloatZone's memory overhead is slightly lower, at 159%. Furthermore, we measure that the (partial geomean) memory overhead of HWASan is minimal with 6% and 4%. A low memory footprint is one of HWASan's key features [11], which is a logical consequence of its memory tagging approach, as opposed to using redzones. We do point out that the missing benchmark (453.povray) is a heavy contributor to the overhead of the other sanitizers (roughly 19x memory overhead for both RSan and ASan).

## 7.5 Fuzzing

For the final part of RSan's performance evaluation, we compare the fuzzing throughput and coverage of AFL++ (v4.21c) using RSan and its closest functional competitors (FloatZone, and ASan--) as sanitizers. Since sanitizers naturally rely on the availability of source code, we configure AFL++ to use persistent mode rather than (the more binary-targeted) fork mode. This is also to evaluate the mode used in *all professional fuzzing*, as stated in the AFL++ documentation [1]. We fuzz the same programs as evaluated in previous work [25,69] for 10 iterations of 24 hours each (on P-cores). Since the three sanitizers apply different instrumentation to the target programs, we cannot directly compare the resulting edge cov-

| Benchmark | Throughput increase | | Coverage Increase | |
|---|---|---|---|---|
| | ASan-- | FloatZone | ASan-- | FloatZone |
| file | 70.1% | 2.1% † | 0.1% † | 12.2% |
| libpng | 24.9% | 42.2% | 0.1% † | 2.2% |
| tcpdump | 13.9% | 13.1% | - | - |
| cxxfilt | 27.1% † | -14.9% † | 0.6% † | 4.3% |
| nm | 32.1% | 67.6% | 1.0% † | 2.8% |
| size | 6.6% † | 32.5% | 0.2% † | 1.0% |
| objdump | -30.8% † | -14.3% † | -0.3% † | 0.6% † |
| geomean | 33.7% | 37.5% | † | 4.4% |

Table 3: RSan's increase in total executions and edge coverage for persistent mode fuzzing. Statistically *insignificant* results (Mann–Whitney U test $p$-value $> 0.05$) are marked with †. Geomean results include statistically significant values only.

erage [60]. Instead, we re-execute the resulting AFL++ *queue* (i.e., the inputs that increase coverage) of each fuzzing iteration through an *uninstrumented* binary to equalize the coverage metric. Additionally, we confirmed that RSan detects all the bugs found by ASan-- and vice versa, and that no new bugs were found by any of the sanitizers.

Table 3 displays the median increase in throughput (total executions) and (edge) coverage of RSan compared to the other sanitizers. We observe that RSan increases throughput by up to 70.1% and 67.7% compared to ASan-- and FloatZone, respectively. Moreover, RSan increases throughput by a geomean of 33.7% and 37.5% (only counting the statistically significant results). From these results, we conclude that RSan notably accelerates fuzzing throughput. We observe a large variance in throughput for the statistically insignificant results. For example, the highest measured total executions for `cxxfilt` is 5x larger than the lowest (with the same sanitizer).

In terms of coverage, after 24 hours, our results show that RSan does not explore more edges with statistical significance compared to ASan--, which can be explained by the evaluated set of (small) programs saturating their coverage too quickly [21, 25]. Nonetheless, after one hour of fuzzing, we observe a statistically significant median coverage increase of 2.4% on the `file` benchmark compared to ASan--. For the `tcpdump` benchmark, we were unable to obtain valid coverage results by replaying the queue (for all of the sanitizers). Compared to FloatZone, we observe a geomean increase of 4.4% in final coverage. Note that FloatZone suffers from false positive bug detections that block AFL++ from exploring new paths, as for example highlighted by the `file` program [25]. Overall, we found that, while RSan tends to discover edges more quickly early on, the coverage of the evaluated programs is saturated after 24 hours.

To confirm our intuition that the evaluated programs are relatively small and hence saturate quickly (especially with the high throughput of persistent mode fuzzing), we also considered a larger program from the Magma benchmarking suite [31]. In particular, we selected `libxml2` for its large code size and high "stability" metric according to AFL++ (and thus low noise). For this program, RSan increases throughput by 33% compared to ASan-- (statistically significant). Figure 10 shows the resulting edge coverage progression of fuzzing `libxml2` with RSan and ASan-- as sanitizers. After 24 hours, we measured a statistically significant coverage increase of 0.7% for RSan. Additionally, we measured a statistically significant peak coverage increase of 5.8% after approximately one hour of fuzzing. Compared to FloatZone, for `libxml2` there is no statistically significant difference in throughput nor coverage. These results show that the increased throughput of RSan results in more coverage if the program does not saturate too quickly. Moreover, in line with with prior work, our results show that exploring new (undiscovered) code requires exponentially more throughput [7].

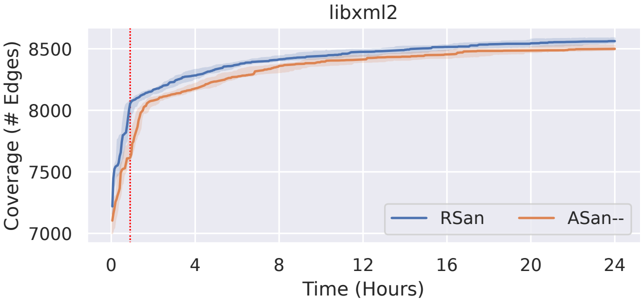


Figure 10: Median coverage progression (with 95% confidence intervals) over 24 hours of fuzzing `libxml2` (10 runs). The red dotted line marks the peak coverage increase (5.8%).

In summary, our performance and security evaluation shows that RSan is effective at detecting bugs and provides better performance than state-of-the-art sanitizers, for fuzzing as well as on the SPEC CPU benchmarks. We also highlight that RSan is a generic solution that supports both Arm and x86 architectures, performing well even on low-end CPUs.

## 8 Limitations

Aside from the well-known drawbacks of (redzone-based) sanitizers, like not detecting intra-object overflows, and conflicts with custom memory allocators, RSan also introduces some technical limitations. First, RSan's relies on LLVM's SafeStack for its size classes on the stack, which currently does not support instrumenting dynamic libraries [12]. Second, RSan's metadata lookup on `base-8` causes segmentation faults on uninstrumented memory if `base-8` is unmapped. This can only occur if `base` resides in the first seven bytes of a page with a preceding unmapped page. We address this limitation by inserting a `MAP_NORESERVE` accessible page before every mapping (where needed). Third, RSan's pointer tagging can clash with programs that implement their own pointer tagging. None of these limitations are fundamental, but they do require engineering effort to address. Finally, RSan does not yet take full advantage of its large redzones and the pointer bits provided by Intel LAM [32], both of which may help further reduce the overhead (see Appendix A.1 for details).

## 9 Related work

There exist many different solutions for detecting and mitigating memory errors. There are systems that specifically target spatial memory errors [2, 18, 19, 24, 27, 29, 37, 47] or temporal errors [10, 15, 26, 48, 61]. Despite many dedicated optimizations [33, 39, 42, 62, 63, 67, 69], solutions that detect both bug categories [8, 28, 51, 56] remain expensive in terms of runtime overhead or require special hardware features [40, 70]. Additionally, joining two methods for combined spatial and temporal memory error detection results in high overhead [46, 68]. Existing solutions also sometimes improve performance at the cost of detecting *false-positive* errors [18, 25, 52].

Multiple components of RSan are similar to or shared with concepts seen in related work. First, *redzones* have been employed by various sanitizers [25,29,56], and similarly to RSan, RedFat [20] also stores *metadata* directly inside the redzone padding. However, RedFat requires additional memory loads to locate the metadata and for its checks either requires multiple comparisons (e.g., one for the lower and one for the upper bound), or a single comparison with extensive arithmetic to represent the lower bound with an integer underflow. In contrast, RSan retrieves the metadata location solely with quick pointer arithmetic thanks to the alignment properties of its allocator and checks for validity with a single comparison. Second, RSan stores per-object metadata in-between objects, which is a common idiom seen in memory allocators, for example the inline *chunk headers* in GNU's allocator. GNU's inline metadata can in turn become a small redzone if memory tagging hardware features (e.g., Arm MTE) are available [22].

Furthermore, RSan's implicit pointer tagging design is inspired by Low-Fat's [18] partitioning of the virtual address space. Similarly, Low-Fat splits the address space into fixed regions to service allocations of a specific size range. In Low-Fat, the *region index* (i.e., the pointer tag) is used as key in a metadata table, requiring an integer division and a memory load from the lookup table to find the actual slot size corresponding to a pointer. RSan improves upon this strategy by directly encoding the size class in the pointer tag, eliminating the need for expensive memory accesses or divisions.

Additionally, RSan uses the design introduced by StickyTags [27] to split the stack and global variables into size classes. Both RSan and StickyTags use size classes to guarantee a predictable layout of memory objects. StickyTags uses Arm's Memory Tagging Extension (MTE) to protect against spatial memory errors and the size classes help with reducing the cost of tagging memory. In contrast, RSan uses size classes to track and manage its own metadata. The pointer tags in StickyTags represent one of the 16 possible MTE colors, while RSan uses pointer tags to encode size classes. Similarly to BaggyBounds [2], RSan uses power-of-two alignment properties to calculate base addresses. However, unlike RSan, BaggyBounds' bounds check does not detect temporal errors and out-of-bounds accesses inside padding bytes.

Finally, RSan was in part inspired by GiantSan [42]. GiantSan aims to improve redzone-based sanitization by increasing the *protection density* of a single metadata unit in ASan [56]. More specifically, GiantSan reduces the number of required metadata lookups and checks by folding multiple consecutive chunks of valid (shadow) memory into larger *segments*. As a result, GiantSan accelerates sanitizer checks for ranges of memory by scanning for validity iterating one segment at a time. In contrast, RSan introduces a stronger range check and validates complete ranges of memory with a single metadata lookup and comparison, without any loops.

## 10 Conclusion

After decades of research on detecting memory errors, redzone-based sanitizers like AddressSanitizer have thrived and became indispensable for security testing. When transitioning from per-pointer metadata (in *bounds checkers*) to per-object metadata, sanitizers lost the ability of checking *ranges* of memory for validity in one go. In this paper, we show that with a novel metadata and check format, we can reintroduce the ability to perform *range checks* to redzone-based sanitizers. The resulting sanitizer design, called RSan, detects spatial and temporal memory errors with high performance. With its range checking paradigm, RSan enables a vast space of optimizations and our evaluation shows that RSan detects bugs with a geomean runtime overhead of 44% on SPEC CPU2017, faster than all state-of-the-art sanitizers.

## Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This work was supported by Intel Corporation through the "Allocamelus" project, by NWO through project "Theseus" and the Dutch Prize for ICT research, and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

## Ethics considerations

All the evaluated CVEs in this work (see Table 4) were already publicly known and hence no new vulnerabilities have been discovered. We strived to evaluate the performance of RSan and all competing systems in the most fair manner. For our experiments, we ensured an equalized baseline, optimal configurations for all solutions, and reported median results of multiple runs. For our fuzzing evaluation, we relied on the best practices from the community [60].

## Open science

We comply with the open science policy by releasing the RSan prototype (modified LLVM compiler, modified TCMalloc allocator, linking scripts, and installation scripts) as open source and partaking in Artifact Evaluation. The artifacts for RSan can be found at https://zenodo.org/records/14701524

## References

[1] AFL++. llvm_mode persistent mode. Online. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md.

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, page 96, 2009.

[3] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Section 5.10: Upper Address Ignore.

[4] Arm. Arm Cortex-A Series: Programmer's Guide for ARMv8-A. Chapter 12.5.1: Virtual Address tagging.

[5] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, 1994.

[6] Jinsheng Ba, Gregory J Duck, and Abhik Roychoudhury. Efficient greybox fuzzing to detect memory errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[7] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *ESEC/FSE*, pages 713–724, 2020.

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

[9] Marc Brünink, Martin Süßkraut, and Christof Fetzer. Boundless memory allocations for memory safety and high availability. In *International Conference on Dependable Systems & Networks (DSN)*, 2011.

[10] 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.

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

[12] Clang/LLVM. SafeStack. Online. https://clang.llvm.org/docs/SafeStack.html.

[13] Joe Costa. Calculating Geometric Means. Online. https://www.waterboards.ca.gov/water_issues/programs/swamp/docs/cwt/guidance/3413.pdf.

[14] CrowdStrike. CrowdStrike PIR Executive Summary. Online. https://www.crowdstrike.com/wp-content/uploads/2024/07/CrowdStrike-PIR-Executive-Summary.pdf.

[15] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX security symposium (USENIX security 17)*, pages 815–832, 2017.

[16] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, pages 162–171, 2006.

[17] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. *ACM SIGPLAN Notices*, 2006.

[18] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[19] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, volume 17, pages 1–15, 2017.

[20] Gregory J Duck, Yuntong Zhang, and Roland HC Yap. Hardening binaries against more memory errors. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 117–131, 2022.

[21] Elia Geretto, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Snappy: Efficient fuzzing with adaptive and mutable snapshots. In *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022.

[22] GNU. Malloc memory tagging. https://elixir.bootlin.com/glibc/glibc-2.40/source/malloc/malloc.c#L383.

[23] Google. Memory safety. Online. https://www.chromium.org/Home/chromium-security/memory-safety.

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

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

[26] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. DangZero: Efficient use-after-free detection via direct page table access. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1307–1322, 2022.

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

[28] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing memory error detection for large-scale applications and fuzz testing. In *NDSS*, 2018.

[29] Niranjan Hasabnis, Ashish Misra, and R Sekar. Lightweight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.

[30] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in c and c++ programs. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, 1992.

[31] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. In *ACM SIGMETRICS*, pages 81–82, 2021.

[32] Intel. Intel Architecture Instruction Set Extensions and Future Features. Chapter 6: Linear Address Masking.

[33] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient sanitizer metadata design for fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 249–263, 2020.

[34] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[35] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG*, pages 13–26, 1997.

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

[37] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[38] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221, 2017.

[39] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. Partisan: fast and flexible sanitization via run-time partitioning. In *RAID*, 2018.

[40] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *ACM CCS*, 2022.

[41] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. Camp: Compiler and allocator-based heap memory protection. In *33rd USENIX Security Symposium (USENIX Security)*, 2024.

[42] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. Giantsan: Efficient memory sanitization with segment folding. In *ASPLOS*, 2024.

[43] LLVM. llvm-project. Online. https://github.com/llvm/llvm-project/blob/llvmorg-18.1.8/llvm/lib/Analysis/ValueTracking.cpp#L7074.

[44] LLVM. The Often Misunderstood GEP Instruction. https://llvm.org/docs/GetElementPtr.html.

[45] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. Online, 2019. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL.

[46] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2015.

[47] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[48] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, 2010.

[49] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002.

[50] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74, 2007.

[51] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[52] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–30, 2018.

[53] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software: Practice and Experience*, 27(1):87–110, 1997.

[54] Bruce Perens. Electric Fence, 1987. https://elinux.org/Electric_Fence.

[55] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 2004, pages 159–169, 2004.

[56] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012.

[57] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.

[58] Matthew S Simpson and Rajeev K Barua. Memsafe: ensuring the spatial and temporal memory safety of c at runtime. *Software: Practice and Experience*, 2013.

[59] J Steffen. Adding run-time checking to the portable c compiler. *Software: Practice and Experience*, 1992.

[60] Moritz Schloegel *et al.* Sok: Prudent evaluation practices for fuzzing. In *IEEE (S&P)*, 2024.

[61] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419, 2017.

[62] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy (S&P)*, pages 866–879. IEEE, 2015.

[63] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: compositing security mechanisms through diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 271–283, 2017.

[64] Wei Xu, Daniel C DuVarney, and R Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 2004.

[65] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R Sekar, Frank Piessens, and Wouter Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[66] Zheng Yu, Ganxiang Yang, and Xinyu Xing. ShadowBound: Efficient heap memory protection through advanced metadata management and customized compiler optimization. In *USENIX Security*, 2024.

[67] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs. In *OSDI*, 2021.

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

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

[70] Mohamed Tarek Ibn Ziad, Miguel A Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-fat: Architectural support for low overhead memory safety checks. In *ISCA*, 2021.

# A Appendix

## A.1 Future work

Currently, RSan encodes size classes as binary logarithmic powers of two inside the pointer tags due to the limited 8-bits of space from Arm TBI. In future work, we aim to explore the benefits of the larger address mask of Intel LAM. For instance, with LAM providing a 15 bits tag space, we could support size classes up to 2ˆ15 (32kB) in the pointer tag without any encoding. This could remove the powers-of-two requirement from RSan's allocator and improve memory overhead.

Another strategy to reduce memory overhead when address masking is available is to host smaller sized objects in the redzone space of larger size classes. This can take effect on a sub-page granularity, e.g., when an object of 2,048 bytes leaves $4,096 - 2,048 - 8 = 2,040$ redzone bytes available to host smaller objects. However, we can also save memory across pages if we reuse the virtual address space of redzones in large size classes for new regions, which in turn improves address space locality and reduces the number of required page table pages.

Furthermore, we believe the optimization space for RSan is not exhausted. While this paper covers the core optimizations, more exotic optimizations are possible, such as: interprocedural check merging, interplay between optimizations (e.g., caching the metadata of merged checks), and reusing metadata across non-loop accesses. Additionally, certain analyses of LLVM can also be extended, such as: a less constrained alias analysis that finds pointers to the same object, and the `FIXME` in LLVM's API that finds conditional loop accesses [43].

### A.2 Evaluation practices

We observed a concerning trend in recent related work pertaining evaluation practices. Specifically, we found that some recent papers miscalculate geometric means (geomeans), omit benchmarks, and use *training* workloads.

> "Like zero, it is impossible to calculate Geometric Mean with negative numbers. [...] Incidentally, if you do not have a negative percent value in a data set, you should still convert the percent values to the decimal equivalent multiplier. It is important to recognize that when dealing with percents, the geometric mean of percent values does not equal the geometric mean of the decimal multiplier equivalents." [13]

CAMP [41] reports a geomean of percentages for its runtime overhead, which becomes significantly higher when (correctly [13]) recalculated with decimal multipliers. However, CAMP does calculate its *memory* overhead correctly, as one zero value makes a geomean of percentages impossible. We recalculated their reported 21.3% SPECspeed 2017 runtime overhead as 48.6% using the numbers from the paper. For SPEC CPU2006, the reported 54.9% becomes 76.1%, excluding the 447.dealII and 471.omnetpp benchmarks. The authors state that these benchmarks are omitted due to incompatibility with *competing* systems. We believe all results should be reported regardless of what the (current) competition supports.

ShadowBound [66] reports a geomean runtime overhead of percentages on SPEC CPU2006. When recalculating the geomean using multipliers, we found an overhead of 20.1% compared to the reported 10.6%. Since the exact overhead (or runtime) values are not reported, we had to estimate them from the bar plot. The geomean excludes 401.bzip2 and 471.omnetpp because LLVM 15 reportedly fails to compile these benchmarks due the programs being too dated. We found no such issues with LLVM 16 in our evaluation.

| CVE | Type | ASan | RSan |
|---|---|---|---|
| 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-2017-12937 | heap-buffer-overflow | ✓ | ✓ |
| CVE-2017-14407 | stack-buffer-overflow | ✓ | ✓ |
| CVE-2017-14408 | stack-buffer-overflow | ✓ | ✓ |
| CVE-2017-14409 | global-buffer-overflow | ✓ | ✓ |

Table 4: CVE detection results of ASan and RSan.

GiantSan [42] reports a geomean of percentages and also takes the geomean of SPECspeed 2017 and SPECrate 2017 combined as a whole. Additionally, GiantSan reports normalized percentage representations (presumably to avoid negative and zero values), where for example a runtime increase from 358 to 718 seconds becomes a change of 200%, while we would call this a 2.0x decimal multiplier and a 100% increase. The recalculated geomean (using the numbers from the paper) is favorable to GiantSan: GiantSan's 146% runtime overhead (actually 46%) becomes 39.4%. However, we found a large discrepancy with these results in our evaluation.

ASan-- [69] omits 471.omnetpp from their SPEC CPU2006 evaluation because their solution fails to compile this program. ASan-- also only includes 5 out of 11 SPECspeed 2017 benchmarks due to compiler compatibility issues. Since unmodified ASan [56] supports SPEC CPU without issues, we believe that follow-up compiler optimizations should not introduce breakage—unless any limitations (e.g., nonconservative optimization behavior) are adequately documented.

PACMem [40] reports a subset of SPECspeed 2017 (8 out of 11 benchmarks, specifically only the C programs, omitting C++) and unexpectedly also includes runtime overhead on Nginx in the geomean. Excluding Nginx shifts the partial geomean from 68.7% (as reported) to 77.5%. PACMem also uses the SPEC CPU *training* workload (i.e., input) for four benchmarks because their competition cannot run with the *reference* workload. We believe the reference workload overhead should still be reported for the PACMem design, regardless of what the competition supports.

The dissimilarity (miscalculated geomeans, missing benchmarks, training workloads) between SPEC CPU results across different (but closely related) papers causes the aggregate overheads to no longer be directly comparable, an alarming trend that, if not reversed, may hinder progress in the area.

## A.3 SPEC CPU results

Table 5 displays the impact of the compiler-based optimizations on SPEC CPU2006 for each individual benchmark. Table 6, 7 and 8 contain the SPEC CPU2006 and SPECspeed2017 runtime overhead results of RSan and related solutions for each individual benchmark.

| (x86, implicit) | **None** | **Ex** | **LVU** | **LIU** | **LVC** | **MC** | **MNP** | **MLH** |
|---|---|---|---|---|---|---|---|---|
| 400.perlbench | 2.49 | 2.01 | 2.01 | 2.00 | 2.01 | 1.94 | 1.94 | 1.94 |
| 401.bzip2 | 1.64 | 1.61 | 1.59 | 1.58 | 1.58 | 1.57 | 1.57 | 1.51 |
| 403.gcc | 2.76 | 2.69 | 2.66 | 2.65 | 2.66 | 2.66 | 2.66 | 2.59 |
| 429.mcf | 1.41 | 1.38 | 1.36 | 1.36 | 1.35 | 1.30 | 1.30 | 1.29 |
| 433.milc | 1.17 | 1.15 | 1.11 | 1.11 | 1.11 | 1.11 | 1.10 | 1.07 |
| 444.namd | 1.70 | 1.63 | 1.50 | 1.48 | 1.48 | 1.41 | 1.41 | 1.41 |
| 445.gobmk | 1.60 | 1.52 | 1.50 | 1.50 | 1.51 | 1.48 | 1.48 | 1.47 |
| 447.dealII | 1.74 | 1.73 | 1.59 | 1.52 | 1.52 | 1.45 | 1.44 | 1.43 |
| 450.soplex | 1.53 | 1.52 | 1.42 | 1.42 | 1.42 | 1.41 | 1.41 | 1.38 |
| 453.povray | 2.51 | 2.12 | 2.12 | 2.12 | 2.10 | 1.83 | 1.83 | 1.81 |
| 456.hmmer | 3.13 | 3.10 | 1.52 | 1.50 | 1.36 | 1.37 | 1.36 | 1.36 |
| 458.sjeng | 1.79 | 1.59 | 1.54 | 1.54 | 1.54 | 1.54 | 1.54 | 1.49 |
| 462.libquantum | 1.29 | 1.29 | 1.13 | 1.13 | 1.04 | 1.06 | 1.06 | 1.04 |
| 464.h264ref | 2.46 | 1.86 | 1.77 | 1.77 | 1.75 | 1.71 | 1.69 | 1.55 |
| 470.lbm | 1.30 | 1.30 | 1.24 | 1.24 | 1.24 | 1.00 | 1.00 | 0.99 |
| 471.omnetpp | 2.54 | 2.30 | 2.29 | 2.27 | 2.30 | 2.25 | 2.25 | 2.25 |
| 473.astar | 1.55 | 1.46 | 1.46 | 1.46 | 1.46 | 1.46 | 1.46 | 1.44 |
| 482.sphinx3 | 2.07 | 2.07 | 1.27 | 1.27 | 1.27 | 1.25 | 1.25 | 1.25 |
| 483.xalancbmk | 3.16 | 3.09 | 2.89 | 2.89 | 2.55 | 2.53 | 2.53 | 2.51 |
| geomean | 1.901 | 1.788 | 1.623 | 1.616 | 1.588 | 1.540 | 1.536 | 1.510 |

Table 5: Optimizations runtime impact on SPEC CPU2006.

| (Arm, TBI) | **RSan** | **ASan** | **ASan--** | **HWASan** |
|---|---|---|---|---|
| 400.perlbench | 1.95 | 3.34 | 2.73 | 3.67 |
| 401.bzip2 | 1.60 | 1.53 | 1.44 | 1.64 |
| 403.gcc | 1.52 | 1.92 | 1.90 | 5.38 |
| 429.mcf | 1.39 | 1.40 | 1.31 | 1.67 |
| 433.milc | 1.24 | 2.07 | 1.71 | 4.52 |
| 444.namd | 1.41 | 1.46 | 1.21 | 1.62 |
| 445.gobmk | 1.49 | 1.57 | 1.46 | 1.83 |
| 447.dealII | 1.50 | 2.23 | 2.14 | 2.37 |
| 450.soplex | 1.52 | 1.55 | 1.49 | 1.67 |
| 453.povray | 1.99 | 2.44 | 2.53 | - |
| 456.hmmer | 1.35 | 2.39 | 2.09 | 3.12 |
| 458.sjeng | 1.47 | 1.59 | 1.36 | 1.99 |
| 462.libquantum | 1.07 | 1.37 | 1.19 | 2.64 |
| 464.h264ref | 1.82 | 2.04 | 1.88 | 3.14 |
| 470.lbm | 1.05 | 1.33 | 1.42 | 1.44 |
| 471.omnetpp | 2.13 | 2.40 | 2.62 | 1.94 |
| 473.astar | 1.48 | 1.44 | 1.35 | 1.59 |
| 482.sphinx3 | 1.43 | 1.60 | 1.73 | 2.30 |
| 483.xalancbmk | 2.54 | 3.32 | 3.29 | 2.44 |
| geomean | 1.540 | 1.867 | 1.754 | 2.316 |

Table 6: Runtime overhead of RSan and other state of the art solutions on SPEC CPU2006 (explicit tagging, Arm TBI).

| (x86, implicit) | **RSan** | **ASan** | **ASan--** | **GiantSan** | **FZ** |
|---|---|---|---|---|---|
| 400.perlbench | 1.94 | 4.21 | 3.90 | 3.80 | 1.87 |
| 401.bzip2 | 1.51 | 1.49 | 1.46 | 1.29 | 1.52 |
| 403.gcc | 2.59 | 3.01 | 2.96 | 2.68 | 2.16 |
| 429.mcf | 1.29 | 1.29 | 1.20 | 1.26 | 1.25 |
| 433.milc | 1.07 | 1.46 | 1.22 | 1.59 | 1.24 |
| 444.namd | 1.41 | 1.50 | 1.45 | 1.42 | 1.25 |
| 445.gobmk | 1.47 | 1.60 | 1.51 | 1.56 | 1.24 |
| 447.dealII | 1.43 | 2.41 | 2.28 | 2.72 | 1.33 |
| 450.soplex | 1.38 | 1.54 | 1.49 | 1.46 | 1.36 |
| 453.povray | 1.81 | 2.49 | 2.21 | 2.81 | 2.17 |
| 456.hmmer | 1.36 | 2.75 | 2.35 | 1.39 | 2.39 |
| 458.sjeng | 1.49 | 1.75 | 1.51 | 1.46 | 1.29 |
| 462.libquantum | 1.04 | 1.29 | 1.18 | 0.99 | 1.08 |
| 464.h264ref | 1.55 | 2.31 | 1.99 | 1.65 | 2.37 |
| 470.lbm | 0.99 | 1.21 | 1.15 | 1.10 | 1.03 |
| 471.omnetpp | 2.25 | 2.90 | 2.64 | 2.49 | 1.94 |
| 473.astar | 1.44 | 1.43 | 1.38 | 1.11 | 1.14 |
| 482.sphinx3 | 1.25 | 1.68 | 1.62 | 1.21 | 1.34 |
| 483.xalancbmk | 2.51 | 3.42 | 3.35 | 2.49 | 2.35 |
| geomean | 1.510 | 1.951 | 1.808 | 1.682 | 1.531 |
| 600.perlbench_s | 1.79 | 2.64 | 2.03 | 1.96 | 1.72 |
| 602.gcc_s | 1.73 | 2.20 | 2.25 | 1.91 | 1.50 |
| 605.mcf_s | 1.20 | 1.43 | 1.22 | 1.25 | 1.17 |
| 620.omnetpp_s | 2.53 | 3.57 | 3.53 | 3.22 | 2.00 |
| 623.xalancbmk_s | 1.86 | 2.31 | 2.13 | 1.91 | 1.83 |
| 625.x264_s | 1.48 | 2.00 | 1.91 | 1.45 | 2.07 |
| 631.deepsjeng_s | 1.34 | 1.62 | 1.47 | 1.70 | 1.22 |
| 641.leela_s | 1.74 | 1.86 | 1.76 | 1.38 | 1.42 |
| 619.lbm_s | 1.01 | 1.00 | 1.01 | 1.00 | 1.07 |
| 638.imagick_s | 1.10 | 2.24 | 1.52 | 1.07 | 2.53 |
| 644.nab_s | 1.07 | 1.77 | 1.58 | 1.44 | 1.95 |
| 657.xz_s | 1.15 | 1.24 | 1.19 | 1.25 | 1.64 |
| geomean | 1.445 | 1.886 | 1.704 | 1.548 | 1.626 |

Table 7: Runtime overhead of RSan and other state of the art solutions on SPEC CPU2006 & 2017 (implicit tagging x86).

| (x86, LAM) | **RSan** | **ASan** | **ASan--** | **HWASan** | **FZ** |
|---|---|---|---|---|---|
| 400.perlbench | 2.14 | 5.52 | 5.54 | 10.62 | 2.38 |
| 401.bzip2 | 1.47 | 1.96 | 1.93 | 3.15 | 1.59 |
| 403.gcc | 1.88 | 4.42 | 4.35 | 5.33 | 2.22 |
| 429.mcf | 1.45 | 3.29 | 2.77 | 2.15 | 1.34 |
| 433.milc | 1.16 | 2.64 | 2.20 | 2.38 | 1.25 |
| 444.namd | 1.32 | 1.36 | 1.35 | 3.20 | 1.31 |
| 445.gobmk | 1.45 | 1.62 | 1.56 | 3.48 | 1.26 |
| 447.dealII | 1.45 | 3.65 | 3.42 | 4.58 | 1.40 |
| 450.soplex | 1.38 | 2.34 | 2.11 | 2.53 | 1.32 |
| 453.povray | 1.86 | 2.40 | 2.17 | - | 2.42 |
| 456.hmmer | 1.50 | 2.45 | 2.16 | 8.48 | 2.36 |
| 458.sjeng | 1.39 | 1.61 | 1.43 | 3.95 | 1.31 |
| 462.libquantum | 1.04 | 4.58 | 4.44 | 1.80 | 1.04 |
| 464.h264ref | 1.52 | 2.13 | 1.81 | 5.27 | 2.23 |
| 470.lbm | 1.11 | 2.91 | 2.86 | 2.79 | 1.04 |
| 471.omnetpp | 2.53 | 2.93 | 2.84 | 2.93 | 2.13 |
| 473.astar | 1.41 | 1.59 | 1.60 | 2.33 | 1.14 |
| 482.sphinx3 | 1.25 | 1.92 | 1.94 | 4.11 | 1.47 |
| 483.xalancbmk | 3.21 | 3.83 | 3.66 | 5.34 | 2.75 |
| geomean | 1.540 | 2.591 | 2.431 | 3.683 | 1.600 |

Table 8: Runtime overhead of RSan and other state of the art solutions on SPEC CPU2006 (explicit tagging, Intel LAM).