RustSan: Retrofitting AddressSanitizer for Efficient Sanitization of Rust

Kyuwon Cho, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee, *Sungkyunkwan University*

https://www.usenix.org/conference/usenixsecurity24/presentation/cho-kyuwon

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

RUSTSAN: Retrofitting AddressSanitizer for Efficient Sanitization of Rust

Kyuwon Cho
kyuwon.cho@skku.edu
Sungkyunkwan University

Jongyoon Kim
jongyoon.kim@skku.edu
Sungkyunkwan University

Kha Dinh Duy
khadinh@skku.edu
Sungkyunkwan University

Hajeong Lim
hajeong.lim@skku.edu
Sungkyunkwan University

Hojoon Lee *
hojoon.lee@skku.edu
Sungkyunkwan University

Abstract

Rust is gaining traction as a safe systems programming language with its strong type and memory safety guarantees. However, Rust's guarantees are not infallible. The use of unsafe Rust, a subvariant of Rust, allows the programmer to temporarily escape the strict Rust language semantics to trade security for flexibility. Memory errors within unsafe blocks in Rust have far-reaching ramifications for the program's safety. As a result, the conventional dynamic memory error detection (e.g., fuzzing) has been adapted as a common practice for Rust and proved its effectiveness through a trophy case of discovered CVEs.

RUSTSAN is a retrofitted design of AddressSanitizer (ASan) for efficient dynamic memory error detection of Rust programs. Our observation is that a significant portion of instrumented memory access sites in a Rust program compiled with ASan is redundant, as the Rust security guarantees can still be valid at the site. RUSTSAN identifies and instruments the sites that definitely or may undermine Rust security guarantees while lifting instrumentation on safe sites. To this end, RUSTSAN employs a cross-IR program analysis for accurate tracking of unsafe sites and also extends ASan's shadow memory scheme for checking non-uniform memory access validation necessary for Rust. We conduct a comprehensive evaluation of RUSTSAN in terms of detection capability and performance using 57 Rust crates. RUSTSAN successfully detected all 31 tested cases of CVE-issued memory errors. Also, RUSTSAN shows an average of 62.3% performance increase against ASan in general benchmarks that involved 20 Rust crates. In the fuzzing experiment with 6 crates, RUSTSAN marked an average of 23.52%, and up to 57.08% of performance improvement.

1 Introduction

Rust has been gaining traction as a practical *safe* system programming language. It guarantees memory safety through strict compile-time rules and lightweight runtime checking. Many new developments have adopted Rust as the main programming language [2, 4–8, 17, 21, 41]. Also, the inclusion of Rust infrastructure in the Linux kernel [2] was a landmark in the ongoing widespread adoption of Rust.

However, Rust's safety guarantees are not achieved without a price. Rust draws the programmer's cooperation by imposing its strict language semantics. By doing so, the language design and the programmer together yield code whose memory safety can be validated by the compiler and minimal runtime checks. Rust's safety model can be too restrictive for certain use cases requiring fine-grained touch. For this reason, Rust provides a variant of itself called *unsafe Rust* that lives within a code block declared using the **`unsafe`** keyword. The unsafe Rust enjoys unconfined access to language semantics prohibited in Rust, such as raw pointer access and bypassing strict ownership enforcement [9]. The use of **`unsafe`** can be inevitable in certain programs (e.g., interfacing with low-level components) or a programmer's choice to trade safety for flexibility.

Previous works have studied the common practices regarding using **`unsafe`** in Rust and their ramifications of using **`unsafe`** in Rust programs [22, 56]. The findings in these works indicate that the use of unsafe Rust is nearly the sole source of memory errors in Rust programs [56]. In response, researchers have proposed static analysis for discovering unsafe Rust memory errors [14, 26, 36, 37] and runtime isolation of safe Rust from unsafe in Rust programs [15, 31, 33, 38, 45].

Static analysis methods have limited detection capability on highly complex bugs or those that reveal themselves during runtime. Runtime isolation frameworks have laid the foundation for identifying insecure program subsets of Rust programs to protect safe Rust parts. Runtime isolation contains the impact of, rather than detect, the memory errors that stem from unsafe Rust. In addition, the proposed isolation solutions accompany hardware feature dependency (e.g., *Memory Protection Key (MPK)*) [15, 31] that hinders portability. Efforts are being made towards revamping the existing techniques to secure Rust amid the rise of safe languages. An efficient and

*Corresponding author

portable dynamic memory error detection tool (i.e., sanitizers) specifically designed for Rust is lacking to the best of our knowledge.

The Rust community, in fact, has already embraced fuzzing for Rust and reported an abundant number of memory safety bugs in Rust programs with the method [1, 3, 13]. The Rust compiler supports compiling Rust programs with *AddressSanitizer (ASan)* and other sanitizers with compiler flags [3]. Unfortunately, the existing dynamic testing practices and infrastructure for unsafe languages (e.g., C/C++) are inherited without consideration for the nature of Rust memory errors.

ASan [24, 47] established its position as a de facto standard sanitizer in dynamic memory error detection thanks to its detection capability and portability. However, its runtime performance and memory overhead are known to be substantial. Previous works endeavored to improve ASan's runtime and memory overhead [27, 59, 60]. Optimization of sanitizer metadata has been shown to significantly reduce the runtime overhead of sanitizers including ASan [27]. Recent works have shown that the elimination of redundant sanitizer checks imposed on the memory access sites is a promising direction toward optimizing ASan [59, 60].

ASan and other sanitizers are designed for unsafe languages such as C/C++ and assume the prevalence of potential memory errors anywhere in the program. However, a large portion of a Rust program retains the safety guarantees even in the presence of its unsafe neighbor [15, 31, 38]. This means an opportunity exists to significantly reduce ASan's runtime overhead on Rust programs without sacrificing its detection capabilities.

We present RustSan, a retrofitted design of ASan that fully exploits the unique nature of Rust programs in which the vast majority of memory accesses retain the language's safety guarantees. RustSan accurately identifies the safety of the memory object and the memory access sites, enabling the elimination of costly runtime sanitizer checks on *safe* access sites.

RustSan's *cross-IR* analysis is a key design component of RustSan that introduces a fine-grained Rust *High-level IR (HIR)* and *Mid-level IR (MIR)*-level analysis. Our finding is that Rust semantics such as the **`unsafe`** is not properly propagated to the backend LLVM IR level, therefore necessitating the advancement of Rust-specific analysis techniques. Moreover, as our analysis will show, a close inspection of the Rust's IRs allows us to improve accuracy and reduce complications in the later stage of analysis.

With the analysis providing accurate safety information about the objects and sites, RustSan's shadow memory scheme extends that of ASan to apply *selective instrumentation*. RustSan frees the safe sites of shadow memory checks while supporting non-uniform memory access validation for access sites with different safety classifications.

We conduct an extensive evaluation of RustSan through a total of 57 Rust crates. With the reproduction of 31 cases of CVE-issued memory error detection, we validate that RustSan retains ASan's detection capabilities even when a large number of redundant sanitizer checks are lifted. Also, we measure RustSan's performance through a benchmark of 18 general Rust programs to report an average of 62.3% advantage over ASan. In fuzzing scenarios, RustSan provides an average of 23.52% and up to 57.08% of performance improvement.

In all, we summarize our contributions as follows:

- We propose a retrofitted ASan design that significantly reduces runtime sanitizer overhead through selective instrumentation for Rust programs.
- We incorporate a cross-IR static analysis to accurately identify Rust `unsafe` blocks and their data-flow propagation.
- We retrofit ASan's shadow memory scheme to support selective instrumentation and non-uniform memory validation model for safe Rust and unsafe Rust.
- We conduct a comprehensive evaluation of RustSan with 57 Rust crates to investigate its detection capability, scalability, and performance improvements.
- We make RustSan publicly available[1] in the hopes of community adoption for efficient dynamic testing of Rust programs.

2 Background

In this section, we concisely explain the concepts that can aid in understanding this paper.

2.1 AddressSanitizer

ASan [24, 47] is a versatile memory error detector that is widely used thanks to its compatibility and availability in commodity compilers. Notably, ASan is a de facto standard sanitizer component in many dynamic testing scenarios, including fuzzing.

ASan provides memory safety by maintaining a shadow memory that represents the validity of the process virtual address space. A shadow memory byte can be encoded to represent so-called *redzones*, to mark the corresponding memory address as *invalid* for access. ASan encodes several redzone values in its shadow memory to distinguish between types of errors. On detecting invalid access to a redzone with the non-zero shadow value (e.g., `0xfd`), ASan uses the redzone value to index a predefined table that contains the root causes and uses the result to generate reports. All memory access instructions in programs compiled with ASan are to be validated using the shadow memory before execution.

ASan detects spatial memory safety violations on memory objects by inserting redzones before and after the objects.

[1] RustSan git repository that includes source code and detailed documentation will be available in the final version

Thus, out-of-bounds access to objects will be detected through sanitizer checks that consult the shadow memory. ASan also detects temporal memory safety violations on heap (use-after-free) and optionally on stack objects. By marking the object address range with redzones, the object is *invalidated* in the shadow memory. For instance, ASan maintains a quarantined set of recently freed heap objects by marking the objects with redzones to catch use-after-free.

2.2 Rust safety model

Rust provides memory safety through its *ownership* model. In this model, the Rust compiler statically enforces that every memory object belongs to exactly one *owner* variable. When a variable goes out of scope, the compiler automatically inserts the deallocation of its owned memory objects, relieving the programmer of memory management responsibilities. However, such an ownership model might be too restrictive for general programs. To support more flexible programming models, Rust also supports the *borrowing* of memory objects to allow access to resources without ownership. To safely enable borrowing, Rust associates *lifetime* to the resources and employs a *borrow checker* to guarantee that the borrowed references do not outlive the lifetime of the underlying resource.

Unsafe Rust and Rust safety guarantees. Rust provides the unsafe sub-language, denoted by the **`unsafe`** keyword, to temporarily bypass its strict ownership model. The use of unsafe Rust is quite prevalent, as reported by existing works [22], and also can be seen in our evaluation targets. The *unsafeness* of **`unsafe`** is not contained within the **`unsafe`** blocks. In fact, the objects that undergo interactions inside **`unsafe`** blocks are propagated outside through the data flow path, endangering the safety assumptions with the safe Rust [15, 31, 38]. The corrupted objects inside **`unsafe`** may be consumed in the safe Rust to coerce the safe Rust to commit memory errors. As such, even the memory access sites that are in safe Rust (i.e., not inside **`unsafe`** blocks) can be rendered unsafe.

Runtime isolation of unsafe Rust. Existing works proposed runtime isolation that protects the safe Rust from the memory unsafe **`unsafe`** [10, 15, 31, 33, 38, 45]. The task involved containing not only the operations of **`unsafe`** blocks but possible occurrences of vulnerabilities due to **`unsafe`**-bound corrupted objects in the safe Rust [15, 38].

3 RustSan overview

Figure 1 shows an overview of RustSan. In this section, we first discuss the concepts and terminologies that must be explained before we elaborate on RustSan design (§3.1). Then, we provide the overview of the RustSan's compile time (§3.2) and runtime operations in (§3.3).

3.1 Definitions and memory validation model

Here, we introduce our terminology structure used throughout the paper to describe RustSan's design.

Objects and access sites. RustSan maintains the validity state of memory *objects* in shadow memory. Memory access instructions (i.e., load and store) within the program are instrumented to consult the shadow memory for validity before execution. We refer to these instructions as *access sites*, or simply sites.

Safety of objects. We distinguish *safe* and *unsafe* objects, similar to the existing works [15, 38]. When an object is *modified* inside an **`unsafe`** block, it becomes a *source* for unsafeness propagation. Unsafe objects are all objects affected by the unsafe source through the data flow. The term safe object is self-explanatory; all objects that are *not* unsafe are considered safe. Another type of object safety that RustSan defines is called *overlapping* objects. We will revisit this concept after explaining other intertwined concepts.

Safety of sites. The safety of sites is dictated by the objects that *can* be accessed at the site. For instance, a safe site, by definition, is a site that can only access safe objects regardless of control flow. An unsafe site is a site that is within an **`unsafe`** block. We use the term *false-safe site* to refer to the sites within safe Rust that may access unsafe objects. More concretely, a site is false-safe if it is not inside an **`unsafe`** block but can access unsafe objects during runtime.

Overlapping objects. Overlapping objects should be explained through the points-to relation [11]. When a false-safe site may access both safe and unsafe objects during runtime, all objects that can be pointed by the site form a *may-point-to* set. We define overlapping objects as safe objects that belong to such may-point-to set along with unsafe objects. This type of object safety has significance in RustSan's memory validation model for its shadow memory scheme. We visit this concept again as we elaborate on our program analysis later.

Memory validation model of sites. While eliminating sanitizer checks on the safe sites, RustSan enforces separated access validation for the remaining unsafe and false-safe memory access sites in addition to the ASan's redzone-based out-of-bounds memory error checking. RustSan detects all access to *safe objects* from *unsafe* memory access sites. Note that safe objects, by definition, are not reachable from unsafe memory access sites, as identified during the offline analysis. For false-safe memory access sites, RustSan only allows memory accesses to 1) the *overlapping objects*, a subset of safe objects, and 2) unsafe objects.

`unsafe` keyword and unsafe. Now that (un)safety of the sites and objects is explained, we again accentuate the difference between **`unsafe`** and the word "unsafe" in our terminology. When referring to the **`unsafe`** Rust keyword, we always refer to them as **`unsafe`** (bold and typewriter). The word "unsafe" is used in a general sense to refer to potentially corrupted objects that can be accessed on every site.

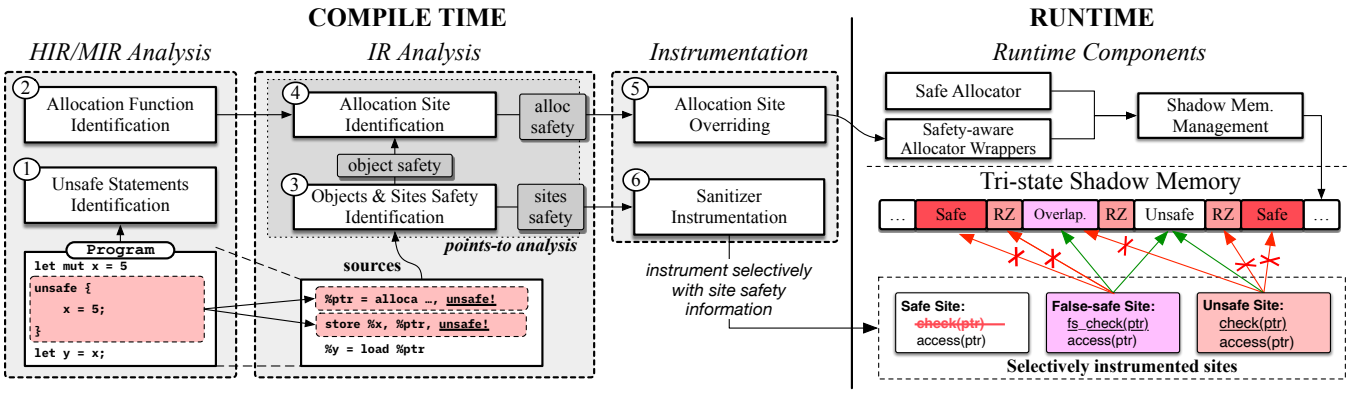

Figure 1: RUSTSAN overview.

3.2 Cross-IR analysis

The first task that RUSTSAN must do is to classify the safety of objects and sites according to the definitions described previously. To this end, RUSTSAN implements a fine-grained analysis to extract information only available in the Rust IR forms (HIR and MIR). One such information being extracted is the memory-modifying statements **`unsafe`** blocks, which is used in the subsequent analyses in LLVM IR as the *source* of unsafeness when determining the safety of objects and sites (① in Figure 1). Extracting and delivering more fine-grained information to the LLVM IR analysis renders the analyses at this stage more efficient. The next piece of information is the Rust-specific allocation functions that our LLVM IR analysis must be aware of to track heap allocation sites reliably (②).

More concrete analysis and instrumentation are performed at the LLVM IR phase of compilation. In LLVM, RUSTSAN utilizes points-to analysis [11] that relies on the Value-Flow Graph [50] to iteratively determine the safety of objects and sites, starting from the distilled *sources* analyzed in the previous MIR/HIR analysis (③). It also tracks the allocation sites in Rust with an allocation site identification scheme to reliably override the allocation of heap objects in its instrumentation (④).

3.3 RUSTSAN shadow memory scheme

The information and facilities from the compile-time time analyses can then be materialized into shadow memory management and instrumentation in RUSTSAN's shadow memory scheme. RUSTSAN retrofit its non-uniform access validation model into ASan's shadow memory scheme to enforce different *memory views* for the classified safe, unsafe, and false-safe sites.

RUSTSAN first introduces an allocator overriding scheme that *colors* the safe and unsafe/overlapping objects accordingly in the shadow memory. This is achieved by replacing the heap memory allocation sites of the unsafe and overlapping object allocation sites (⑤) with RUSTSAN's allocator wrappers. RUSTSAN now extends ASan's instrumentation scheme using the site safety information obtained from its cross-IR analyses. It lifts ASan instrumentation on memory access sites classified as safe, and instruments unsafe and false-safe sites to consult the shadow memory for the safety of the object being accessed to achieve its safety-aware memory validation model (⑥).

4 Cross-IR: Rust HIR/MIR analysis

RUSTSAN employs a fine-grained Rust HIR/MIR-level analyzer to render the analysis of Rust-specific information accurate and efficient. Existing works [15, 38] also used MIR. However, they are limited to marking all statements within **`unsafe`** such that they are propagated to the LLVM IR level. Our findings indicate that the **`unsafe`** semantics of Rust do not propagate well to LLVM IR. **`unsafe`** is a Rust language semantic meant to be consumed by the HIR/MIR compilation stage, and the resulting LLVM IR does not convey the information. Also, the LLVM IR instructions that correspond to the **`unsafe`** HIR instructions become indistinguishable during the multiple stages of transformation. Moreover, certain analyses can be made much more efficient by involving HIR/MIR-level pre-analysis. Our HIR/MIR-level analysis must identify the following four key *information* in the HIR/MIR stage:

- **I1** Memory access statements in **`unsafe`** blocks
- **I2** Memory modification statements in **`unsafe`** blocks
- **I3** Heap memory allocating Rust functions
- **I4** Rust-specific methods that may allocate heap memory

This information can be lost during translation or complicate the later stage analysis if not collected in this stage. **I1** is used during memory access site during safety classification in the LLVM IR and ultimately used to apply selective instrumentation in RUSTSAN's shadow memory scheme (§3.1). **I2** are the birthplace of *unsafe* objects; any object that are modified within **`unsafe`** at least once is considered unsafe by definition. These statements, therefore, create the sources for

the propagation of object unsafety. Hence, precisely differentiating memory-modifying statements and their superset (**I1**) can reduce the overtainting of unsafe objects. **I3** and **I4** are necessary for the accurate identification of heap allocation sites in the later LLVM analysis.

In the process of extracting this information, RUSTSAN's HIR/MIR-level analyzer introduces three novel techniques, namely 1) *Statement-level memory accesses tracking*, 2) *Recursive safety scope analysis*, and 3) *Allocation function identification*. Now we explain the more detailed operations of RUSTSAN's analysis.

4.1 Background: Rust HIR/MIR

The Rust front-end compiler, rustc, internally uses two IR forms, namely HIR and MIR. **`unsafe`** blocks can only be identified at the HIR-level and become indiscernible as the code is translated to MIR format. For this reason, RUSTSAN uses HIR to extract information regarding the **`unsafe`** blocks. However, it uses the analyzer-friendly MIR for more detailed statement-level analysis. Also, there is a one-to-one mapping between MIR and HIR, such that an analysis can query the compiler for HIR-specific information from a MIR statement.

4.2 Statement-level memory access tracking

RUSTSAN implements a statement-level analysis to precisely track memory access statements (**I1**) and further distill writes to objects (**I2**) among them. Overapproximation on **I2** set would increase the number of unnecessary unsafe objects in the LLVM IR analysis, eventually increasing the number of unsafe and false-safe sites in a cascaded way. Therefore, minimizing even a small portion of **I2** with MIR analysis is crucial to RUSTSAN's performance gain, whose source is the elimination of checks on safe sites.

Differentiating write statements. We differentiate write statements from read statements by analyzing the MIR syntax of **I1** statements. We closely examined Rust MIR syntax to list all possible MIR statement forms of memory writes. Thus, RUSTSAN can create the **I2** set, a subset of **I1** that includes the statements that can modify the memory object. Note that while we track down only write statements in `unsafe` as they are the sources of unsafe objects, sanitizer checks on the resulting unsafe objects from this analysis detect both reads and writes.

Excluding strictly-local writes. The statement-level analysis also allows us to reduce further write statements in the **I2** set. RUSTSAN utilizes an MIR data-flow analysis to find statements that modify *strictly-local* variables. These variables do not have data-flow edges outward from their current unsafe scope. It is safe to remove them since they can corrupt no safe variable. For statements that modify the strictly-local variables, RUSTSAN marks the stack allocation for the local variable as allocating an unsafe object and removes the statement from **I2** to reduce the load on the RUSTSAN's subsequent LLVM IR analysis.

4.3 Recursive safety scope analysis

In addition to our efforts to limit overapproximation, we also mitigated a case of false negatives that can arise during MIR analysis. The scope is the basic unit of the code block in Rust. For instance, **`unsafe`**`{...}` itself uses a scope to enclose its statements. We found that nested scopes inside **`unsafe`** frequently appear when the MIR-level optimizer internally inserts scopes (i.e., curly brackets) around *inlined* function calls inside **`unsafe`** blocks.

Recall that the safety (i.e., is the block inside **`unsafe`**?) of a block is not visible in MIR. Therefore, during MIR analysis, determining the scope safety requires consulting the HIR with scope ID. We found that without a conscious effort to make the scope analysis recursive (i.e., traverse up to the outermost scope) and HIR-aware, the nested scopes inside **`unsafe`** can be misidentified as safe. A consequence of this shortcoming can be missing unsafe memory access statements inside the entire inlined function. An existing work [38] did not correctly identify these nested scopes inside **`unsafe`**, creating false negative cases on **I1**. We could not confirm such a problem for TRust [15] as their implementation is yet to be made public at the time of writing.

4.4 Allocation function identification

Finally, RUSTSAN introduces two Rust-specific heuristics that rely on type information only available in MIR to identify possible *heap* allocation functions/methods. First, if the type implements allocator-based traits (e.g., `Global` or `bumpalo`), we mark the implemented methods (e.g., `Global::alloc()`) as heap allocating functions (**I3**). The SVF-based points-to analysis in LLVM later uses the information to model heap object allocations in Rust. Second, on the types that contain allocator-based traits as bounds (e.g., `Vec<T, A:Allocator>`), RUSTSAN regards all of their implemented methods as *potential* heap-allocating method **I4**. Our analysis in LLVM further refines these potential methods into the concrete set of allocation methods by determining if they actually trigger heap allocations on their call graphs (§5.2). Identifying the allocation functions and using them as taint sources reduces the complexity of the program analysis. This heuristic eliminates the need for tainting from the Rust allocation function all the way down to the native allocation function (i.e., C library's `malloc`) in the LLVM analysis, which could unnecessarily increase complexity.

5 Cross-IR: LLVM IR analysis

Figure 2 shows RUSTSAN's analyses performed at LLVM IR level. RUSTSAN's LLVM IR analysis engages as the MIR has

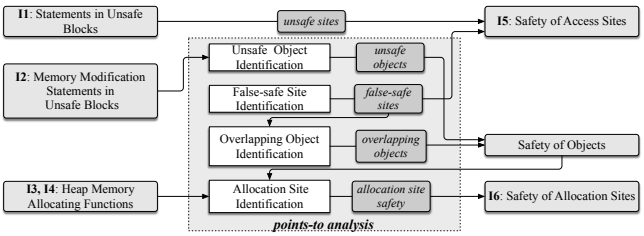

Figure 2: RUSTSAN's LLVM IR analysis.

now been translated to LLVM IR along with the key information (**I1**-**I4**) extracted in the previous stage. Now, RUSTSAN must use them to identify the following information:

I5 Safety of memory access sites (safe/unsafe/false-safe)
I6 Satety of allocation sites (safe/unsafe/overlapping)

The above information is necessary for RUSTSAN's runtime components to manage shadow memory and instrumentation of access sites.

I5 allows RUSTSAN's sanitizer checks to be selectively applied according to the site's safety. The instrumentation to safe sites would be omitted, while the unsafe and false-safe sites will have differing memory access views, as we will explain in the next section (§6). **I6** is necessary to identify all occurrences of Rust allocation functions (e.g., already obtained **I3**, **I4** set) that may be used to allocate unsafe/overlapping heap objects. These *allocation sites* will be replaced with wrapper allocation functions that update the shadow memory entries of the object with the safety information.

5.1 Object and site classification pipeline

RUSTSAN uses SVF's implementation of field-sensitive, context and flow-insensitive Andersen's points-to analysis [11], to classify the safety of memory objects and memory access sites (**I5**). The context-insensitive and flow-sensitive analysis offers reasonable precision while also allowing to scale with large programs, an imperative requirement for a sanitizer.

We follow the memory model of SVF [50] to model memory objects as their allocation sites, which could either stack allocations (`alloca` instructions) or heap allocations (calls of heap allocation functions such as `exchange_malloc`, identified in **I3**). RUSTSAN uses points-to and value flow analyses similarly to the previous works [15, 38] to find objects that a memory access site may access, i.e., its may-points-to set. The points-to information is used to classify the safety of objects and sites according to its definitions in §3.1, which we will explain in detail as follows.

Unsafe site set. We first classify the memory accessing statements in **`unsafe`** as *unsafe sites*. This is achieved by inserting the matching translated LLVM instructions of the statements in **I1** into the unsafe site set.

Unsafe object set. RUSTSAN then identifies the *unsafe objects*. Recall that all objects that are modified by an **`unsafe`** block are considered unsafe objects. The refined write statement set **I2** from HIR/MIR analysis has also been translated into a set of LLVM IRs. RUSTSAN performs points-to analysis with this set to find memory objects to which these instructions can point – a set of objects that can be modified by **`unsafe`**. These objects are then classified as the *unsafe* object set.

False-safe site set. At this stage, all sites not in the unsafe site set are currently considered safe sites. RUSTSAN must identify and separate the false-safe site set from the current safe site set. By our definition, any sites that do not belong in an **`unsafe`** block but access at least one of the objects in the unsafe object set are false-safe. The points-to analysis is performed on the safe set with this definition, resulting in the separated false-safe set.

Overlapping object set. Finally, we identify the overlapping object set. The overlapping object set is a subset of the safe object set (i.e., complementary of the unsafe object set) to be separated. An object in the overlapping set has a *pointed-by* relationship with at least one false-safe site. We iteratively perform the points-to analysis on the identified false-safe site set and obtain the overlapping object set. The overlapping objects and unsafe objects are the tenants of the same may-points-to set of false-safe sites, and hence the name *overlapping*.

5.2 Allocation site safety identification

Now, we have the information necessary to determine the safety of allocation sites (**I6**). RUSTSAN must identify the allocation sites of the objects such that they can be overridden to mark the object's safety information in the shadow memory through instrumentation (explained in §6). Rust exhibits indirect heap allocation behavior through Rust object's *traits*. Hence, along with handling the direct heap allocation sites, we introduce a Rust-specific method for handling the indirect heap allocation sites.

Direct heap allocation sites. We first determine the safety of heap memory allocation directly obtained from a heap through allocation functions in **I3**. For such allocation sites, their safety is already classified by our analysis in 5.1. Hence, we simply mark the allocation site with the safety of the allocated object so that the later instrumentation can override them accordingly.

Indirect heap allocation sites. RUSTSAN implements a scheme to classify the safety of the *indirect heap allocation* triggered by trait method invocation. We demonstrate such indirect heap allocation in Listing 1a. Here, a Rust object is defined at Line 1 and is modified by **`unsafe`** at Line 4. Its compiled LLVM IR counterpart is shown in Listing 1b, where the invocation of `Vec::set_len` (Line 4) is classified as an unsafe site, and the stack object allocated at Line 1 is classified as an unsafe object. On line 3, `Vec::reserve()` is invoked, which is a trait invocation that will trigger a heap allocation

```
1 let mut v = Vec::new();
2   v.reserve(10);
3   unsafe{
4     v.set_len(100);
5   }
```

(a) Rust source code

```
1 %v = alloca ...
2 call void Vec::new(%v)
3 invoke Vec::reserve(10, %v)
4 invoke Vec::set_len(100, %v)
  !unsafe
```

(b) Simplified LLVM IR

: candidate allocation method (**I4**)
: unsafe object : *indirect* unsafe allocation site (**I6**)

Listing 1: RUSTSAN allocation site safety identification for indirect heap allocation.

and store the result into the stack object `%v`. However, such heap object allocation will not be detected if we only find the points-to set for the argument `%v` at Line 4, since it will only contain the stack object `%v` in its points-to set.

To track the safety of such allocation sites, RUSTSAN starts by analyzing the LLVM invocation of methods identified in **I4**. On those invocations, RUSTSAN checks for the following conditions. First, the object method that is being invoked must be connected to the heap allocation function (**I3**) on its call graph. Second, one of the arguments used in the method invocation must point to an unsafe/overlapping object. If the two conditions are met for a method invocation, RUSTSAN marks the method invocation as a heap allocation site and attaches the safety of the pointed-to object to the site. Using this heuristic, RUSTSAN identifies Line 5 in Listing 1b as an indirect heap allocation site of unsafe objects.

5.3 Adapting analysis techniques to Rust

Our LLVM IR analysis depends on the SVF [50]'s state-of-the-art Value-Flow analysis and Andersen's points-to analysis [11]. However, we made several modifications to SVF to improve its compatibility with Rust.

Rust-specific allocation functions. First, we modify SVF such that it can be aware of Rust's heap allocation functions that we found through HIR/MIR analysis (**I3**). This way, SVF can recognize the object pointers returned from such functions as newly allocated heap objects.

Emulation of pointer operation traits. If an object type's pointer operators are not overloaded, the default symbols that are invoked for the overloadable operators are called from the Rust standard library's operator module (`std::ops`). We found that we could minimize the complexity of points-to analysis by simply replacing these with the default pointer referencing and dereferencing behavior.

Supporting commonly used instruction. We add support for LLVM instructions such as `ExtractValue` and `InsertValue` to SVF. These instructions are heavily used in Rust compilation but are not recognized by SVF. A very recent paper [15] also mentions a similar change in SVF. Since their implementation is not yet available at the time of writing, our SVF modification is our own.

6 RUSTSAN shadow memory scheme

RUSTSAN's retrofitted ASan shadow memory scheme takes advantage of the information obtained from compile-time (**I5**, **I6**) to perform safety-aware object allocation and selective instrumentation.

6.1 Safety-aware object allocation

RUSTSAN implements safety-aware objection allocation that encodes the object's safety information (**I6**) in the shadow memory. To this end, RUSTSAN extends ASan's heap memory allocator that manages shadow memory to reflect the safety of objects. RUSTSAN's instrumentation then replaces the object allocation sites with one of the three (safe, unsafe, and overlapping) allocator wrappers. The wrappers perform shadow memory entry updates for the new heap memory address returned by the successful heap memory allocation.

Allocation site override. RUSTSAN hooks the default heap allocation function used by the program to use the safe allocator, using the existing method of by ASan. For allocation sites of unsafe and overlapping objects, RUSTSAN compiler replaces the allocation sites using the safety information from **I6**. It starts by visiting the call graphs of the function invoked at the allocation sites. For each of the called functions on the call graph, RUSTSAN clone the functions into a function that is only used when allocating an object with a particular safety, e.g., `foo` is cloned into `foo`, `foo_unsafe` and `foo_overlapping`. RUSTSAN then replaces the function call on the call graph with the new function based on the allocation site's safety. Finally, it replaces occurrences of heap allocation calls on the call graph with RUSTSAN's allocator wrapper that performs shadow memory management.

Shadow byte scheme. RUSTSAN colors the objects according to the safety of the object. The unsafe objects are non-colored, while we conceptually associate safe objects with magenta and overlapping objects with pink (i.e., pinkzone) to aid reader understanding as shown in Figure 3. RUSTSAN chooses 2 bits, bits 4 and 5, in the upper 5 bits of the shadow byte that represent various types of redzone, to represent safe and overlapping objects, respectively. That is, the shadow byte representation of the safe and overlapping objects are `0b00001aaa` and `0b00010aaa`. Note that the `aaa` field is used

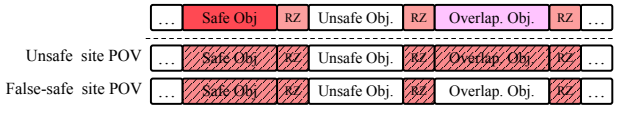

Figure 3: RUSTSAN's shadow memory coloring scheme. Safe objects are colored with *magenta*, overlapping objects with *pink*, and unsafe objects are non-colored.

by ASan to express addressable bytes within the 8-byte memory mapped by the shadow memory.

In addition to the safe and overlapping object coloring, we use two previously unused 5-bit combinations to represent *quarantined* unsafe and overlapping objects. For example, when an overlapping object is deallocated, the shadow byte of the object memory is updated accordingly to the quarantined pink object. Then, we extend ASan's `report()` function that is called upon memory error detection such that it can report the cases of use-after-free on unsafe or overlapping objects.

6.2 Selective instrumentation

With the safety of objects classified (**I5**), RUSTSAN now must instrument the memory access sites according to their safety. RUSTSAN brings a significant runtime overhead reduction by lifting the costly shadow memory-based checks on the safe sites. However, the false-safe and unsafe sites are subject to RUSTSAN's cross-safety memory access validation as well as ASan's existing memory error detection. We now discuss the two aspects in more detail.

Cross-safety memory access validation. RUSTSAN introduces a unique cross-safety memory access validation model. It applies different memory access validation logic on false-safe and unsafe sites, as shown in Listing 2. For unsafe sites, RUSTSAN reports an invalid access error whenever the shadow value is non-zero (Listing 2a). This means their access to the magenta-colored safe and pink-colored overlapping objects will be detected during sanitizer checks. On the other hand, false-safe sites are allowed to access pink-colored objects, and thus can access both unsafe and overlapping objects (Listing 2b). Hence, access to safe objects is detected in both unsafe and false-safe sites.

Note how RUSTSAN makes a conscious design choice on the false-safe sites. ASan's address-based sanitization is inherently incapable of performing context-sensitive checks since there is no feasible way to discern which data-flow path the objects were delivered from. Hence, our design choice was to inherit ASan's compatibility and performance while relaxing the detection coverage on the false-safe sites.

Interoperability with existing ASan capabilities. RUSTSAN's shadow memory scheme fully retains the original ASan detection capabilities on the unsafe and false-safe sites. Recall that ASan's detection mechanism involves two methods of utilizing the redzones: overflow detection through redzones in-between objects and invalidation of objects themselves by marking object address range with redzones as they are freed. Since the former, inter-object redzones, are placed for object allocation regardless of safety, overflows that occur at unsafe and false-safe sites can be detected. For instance, if a *legal* object access (e.g., overlapping or unsafe access at a false-safe site) goes out-of-bound to touch the object-end redzone, it will be detected and reported.

Likewise, unsafe and overlapping objects are also subject to ASan's quarantine-based use-after-free detection scheme implemented through object invalidation. RUSTSAN's heap allocator retains the ASan allocator's object invalidation and quarantining scheme. That is, the heap allocator invalidates the object by marking the object range with redzone and placing it in the quarantine list. One key addition in RUSTSAN is the ability to represent quarantined-unsafe and quarantined-overlapping leveraging the unused bits in the shadow byte, as we already explained. This aids in generating reports of site and object safety upon error detection.

```
#define PINKZONE_MASK ~0b00010000
```

```
shadow = *memToShadow(ptr);
// Disallow PinkZone access

if (shadow != 0)
  report();
// Memory access site
*ptr = 0xdeadbeef;
```

(a) Unsafe site

```
shadow = *memToShadow(ptr);
// Allow PinkZone access
shadow &= PZ_MASK;
if (shadow != 0)
  report();
// Memory access site
*ptr = 0xdeadbeef;
```

(b) False-safe site

Listing 2: RUSTSAN's instrumentation for unsafe sites (left) and false-safe sites (right).

7 Evaluation

In this section, we evaluate the RUSTSAN implementation that is based on LLVM 13.0.0 [39] and rustc 1.66.0 nightly [46]. All experiments were conducted on a workstation with an AMD Ryzen Threadripper 3990X (64 cores@2.9GHz), 256GB RAM, and Ubuntu 20.04 LTS.

We evaluate RUSTSAN with large test sets in terms of its detection capabilities, scalability, and performance improvement over ASan. We built Rust program test sets for our experiments, namely the CVE *reproduction*, *scalability* evaluation, *general performance* benchmark, and *fuzzing* benchmark sets. The CVE reproduction set is a specific version of programs that contain one or more CVE-issued memory errors and deterministic input sequences that trigger them. We empirically evaluate RUSTSAN's detection capability by reproducing the CVEs in §7.2. The scalability set contains programs with relatively larger code bases. With the set, we evaluate the scalability of RUSTSAN's program analysis passes with large Rust MIRs and LLVM IRs in §7.4. The general performance and fuzzing benchmark set are used to measure the runtime overhead of RUSTSAN imposed on target Rust programs (§7.6 and §7.7).

7.1 Unsafe Rust usage statistics

Table 1 shows the statistics on the occurrence and safety statistics access sites of all Rust crates we used for our evaluation. The table illustrates the total number of memory access instructions, including the memory intrinsics. The average

	Target	Whole Program			Unsafe Instructions (`unsafe` + false-safe)						False-safe Instructions						Unsafe Obj / All Obj alloc.
		Load / Store	Intrinsic	Total	Load / Store		Intrinsic		Total		Load / Store		Intrinsic		Total		
Scalability Set (S)	bat	1050235	77225	1127735	41604	(3.96%)	2914	(3.77%)	44568	(3.95%)	24166	(2.30%)	2914	(3.77%)	27130	(2.41%)	12485 / 717529 (1.74%)
	fd	625885	41775	668020	27209	(4.35%)	848	(2.03%)	28187	(4.22%)	18188	(2.91%)	848	(2.03%)	19166	(2.87%)	5646 / 409130 (1.38%)
	ripgrep	587696	35591	623487	37830	(6.44%)	727	(2.04%)	38632	(6.20%)	22448	(3.82%)	727	(2.04%)	23250	(3.73%)	7022 / 116451 (6.03%)
	tokio	397079	41507	439176	21902	(5.52%)	1093	(2.63%)	23165	(5.27%)	15432	(3.89%)	1093	(2.63%)	16695	(3.80%)	5405 / 76885 (7.03%)
	firecracker	354015	24312	378512	19898	(5.62%)	1635	(6.73%)	21588	(5.70%)	12891	(3.64%)	1635	(6.73%)	14581	(3.85%)	5493 / 341180 (1.61%)
	hyper	219199	21522	241166	11758	(5.36%)	703	(3.27%)	12536	(5.20%)	8692	(3.97%)	703	(3.27%)	9470	(3.93%)	2780 / 24301 (11.44%)
	Rocket	2560541	214122	2775973	144394	(5.64%)	8699	(4.06%)	153613	(5.53%)	101860	(3.98%)	8699	(4.06%)	111079	(4.00%)	30762 / 1680984 (1.83%)
B ∩ S	wasmtime†	3259408	213306	3473659	273751	(8.40%)	15271	(7.16%)	289392	(8.33%)	213541	(6.55%)	15271	(7.16%)	229182	(6.60%)	48740 / 2014050 (2.42%)
	RustPython†	3061746	229165	3291721	258251	(8.43%)	12447	(5.43%)	271098	(8.24%)	220621	(7.21%)	12447	(5.43%)	233468	(7.09%)	54304 / 1872552 (2.90%)
Benchmark Set (B)	uuid	1286	126	1412	9	(0.70%)	2	(1.59%)	11	(0.78%)	7	(0.54%)	2	(1.59%)	9	(0.64%)	6 / 203 (2.95%)
	chrono	373383	37322	410910	6765	(1.81%)	863	(2.31%)	7663	(1.86%)	4959	(1.33%)	863	(2.31%)	5857	(1.43%)	1627 / 325400 (0.50%)
	adler	351077	35617	386899	6521	(1.86%)	840	(2.36%)	7396	(1.91%)	4733	(1.35%)	840	(2.36%)	5608	(1.45%)	1580 / 309804 (0.51%)
	unicode-xid	347317	35394	382916	6519	(1.88%)	841	(2.38%)	7395	(1.93%)	4731	(1.36%)	841	(2.38%)	5607	(1.46%)	1556 / 305098 (0.51%)
	base64	372304	37193	409707	12442	(3.34%)	1182	(3.18%)	13659	(3.33%)	6541	(1.76%)	1182	(3.18%)	7758	(1.89%)	5715 / 317500 (1.80%)
	btree	52118	5363	57486	5542	(10.63%)	187	(3.49%)	5729	(9.97%)	1451	(2.78%)	187	(3.49%)	1638	(2.85%)	3924 / 286423 (1.37%)
	image	624398	73205	698093	23139	(3.71%)	1934	(2.64%)	25253	(3.62%)	19159	(3.07%)	1934	(2.64%)	21273	(3.05%)	3410 / 454667 (0.75%)
	slice	42494	2504	45003	5605	(13.19%)	192	(7.67%)	5797	(12.88%)	1514	(3.56%)	192	(7.67%)	1706	(3.79%)	3932 / 206947 (1.90%)
	regex	228994	24707	253796	17475	(7.63%)	521	(2.11%)	18001	(7.09%)	10103	(4.41%)	521	(2.11%)	10629	(4.19%)	3576 / 55875 (6.40%)
	vec	33181	2600	35786	5565	(16.77%)	193	(7.42%)	5758	(16.09%)	1470	(4.43%)	193	(7.42%)	1663	(4.65%)	3930 / 124762 (3.15%)
	str	32953	1819	34777	5583	(16.94%)	187	(10.28%)	5770	(16.59%)	1492	(4.53%)	187	(10.28%)	1679	(4.83%)	3925 / 282374 (1.39%)
	byteorder	38437	2235	40677	7178	(18.67%)	196	(8.77%)	7374	(18.13%)	3073	(7.99%)	196	(8.77%)	3269	(8.04%)	4094 / 115000 (3.56%)
	string	19057	1256	20318	5542	(29.08%)	187	(14.89%)	5729	(28.20%)	1451	(7.61%)	187	(14.89%)	1638	(8.06%)	3924 / 67539 (5.81%)
	binary_heap	19119	1230	20354	5564	(29.10%)	187	(15.20%)	5751	(28.25%)	1473	(7.70%)	187	(15.20%)	1660	(8.16%)	3925 / 50580 (7.76%)
	linked_list	17850	1187	19042	5542	(31.05%)	187	(15.75%)	5729	(30.09%)	1451	(8.13%)	187	(15.75%)	1638	(8.60%)	3924 / 51496 (7.62%)
	bytes	20092	1289	21426	6043	(30.08%)	248	(19.24%)	6291	(29.36%)	1724	(8.58%)	248	(19.24%)	1972	(9.20%)	4065 / 26603 (15.28%)
	vec_deque	16019	1109	17133	5542	(34.60%)	187	(16.86%)	5729	(33.44%)	1451	(9.06%)	187	(16.86%)	1638	(9.56%)	3924 / 66508 (5.90%)
	json	9901	610	10511	2253	(22.76%)	62	(10.16%)	2315	(22.02%)	1492	(15.07%)	62	(10.16%)	1554	(14.78%)	283 / 3997 (7.08%)
† Included in both scalability and benchmark set.																	
Fuzzing Set	image-tiff	87460	4539	92089	1519	(1.74%)	24	(0.53%)	1543	(1.68%)	783	(0.90%)	24	(0.53%)	807	(0.88%)	679 / 52231 (1.30%)
	minidump	437131	45187	482483	10268	(2.35%)	150	(0.33%)	10438	(2.16%)	4460	(1.02%)	150	(0.33%)	4630	(0.96%)	1535 / 255833 (0.60%)
	pdf	480686	55125	535971	15076	(3.14%)	1846	(3.35%)	16962	(3.16%)	8460	(1.76%)	1846	(3.35%)	10346	(1.93%)	5051 / 315688 (1.60%)
	png	37591	2236	39827	2212	(5.88%)	96	(4.29%)	2308	(5.80%)	1077	(2.87%)	96	(4.29%)	1173	(2.95%)	863 / 22891 (3.77%)
	cpp_demangle	28564	4637	33201	2078	(7.27%)	20	(0.43%)	2098	(6.32%)	2076	(7.27%)	20	(0.43%)	2096	(6.31%)	88 / 21463 (0.41%)
	brotli-rs	17709	854	18563	1185	(6.69%)	0	(0%)	1185	(6.38%)	1185	(6.69%)	0	(0%)	1185	(6.38%)	141 / 9400 (1.50%)

Table 1: Instrumentation statistics for scalability, general application set, and fuzzing set Rust programs. *intrinsic* include memory transfer intrinsics such as `memmove` or `memcpy`.

proportion of unsafe sites (**`unsafe`** blocks + false-safe) is measured to be 10.24% across the crates. Notably, crates that require raw pointer manipulation (e.g., `linked_list`) or process low-level data (e.g., `string`, `bytes`) show a higher ratio of unsafe Rust.

The ratio of unsafe sites supports RUSTSAN's motivation and approach; The memory access sites that require sanitizer checks are approximately 34.60% (`vec_deque`) of the total sites at the most, while they can be as low as 0.7% (`uuid`). However, these numbers do not directly indicate runtime overhead that RUSTSAN can eliminate since the runtime performance would depend on the number of sanitizer checks *encountered* during program execution. We discuss the runtime overhead reduction through crate benchmarks §7.6 and fuzzing experiment §7.7 later in this section.

7.2 Detection capability: robustness of selective instrumentation

Regarding detection capability, we empirically validate RUSTSAN's for possible *false-negatives*. In other words, we validate that the eliminated sanitizer checks in RUSTSAN do not cause false negatives. Towards this goal, we test a total of 31 memory-related CVEs that can be detected with ASan, then we reproduce the results with RUSTSAN.

CVE test set collection. We amass a test set of CVE-issued memory vulnerabilities in common Rust programs that can be reproduced and detectable by ASan. The vulnerabilities are collected from the RustSec Advisory Database [55], where the CVEs reported on Rust programs from `crates.io` are accumulated. We collected *all* 408 CVEs listed in the advisory database and assessed their reproducibility. Among these 408 entries, we used the category column on the advisory database to have identified 227 entries as memory-related vulnerabilities. 91 out of 227 entries included a PoC code that allowed us to reproduce the vulnerability, of which *52* were successfully detected by ASan. RUSTSAN was also able to detect all of these 52 cases.

Detection results. Table 2 highlights 31 cases out of 52 cases that demonstrate the robustness of RUSTSAN. These cases are detected through inlined shadow memory checks that are subject to RUSTSAN's selective instrumentation. While RUSTSAN drastically reduces shadow memory checks in Rust programs as shown in Table 1, all CVE cases were successfully detected. The other 21 reproduced cases not shown in Table 2, include cases such as double free. These cases are detected through intercepting standard library calls (e.g., `free`) which remain unaffected by RUSTSAN's check elimination and obviously detected by RUSTSAN as expected.

Memory errors in false-safe sites. The ***FS/U**** column in Table 2 reports that 21 out of 31 reproduced memory errors were detected at false-safe sites. We identified these cases by placing additional instrumentation on memory instructions within the **`unsafe`** that identified 10 within-**`unsafe`** cases. The abundance of memory errors detected at false-safe memory instructions reassures the accuracy of RUSTSAN's false-safe site identification and the correctness of the retrofitted shadow memory scheme. We manually validated selected CVEs in the 21 false-sate site detection cases. Among them, we choose one case of heap overflow (CVE-2018-21000) and use-after-

Crate Name	CVE	Vuln. Class	Detected	*FS/U*[*]
base64	CVE-2017-1000430	Heap Ovf.	✓	U
bumpalo	CVE-2020-35861	Heap Ovf.	✓	U
generic-array	CVE-2020-36465	UAF	✓	FS
smallvec	CVE-2018-20991	UAF	✓	FS
smallvec	CVE-2021-25900	Heap Ovf.	✓	U
smallvec	CVE-2019-15551	UAF	✓	FS
futures-task	CVE-2020-35906	UAF	✓	U
http	CVE-2019-25009	UAF	✓	FS
http	CVE-2020-25574	UAF	✓	FS
prost	CVE-2020-35858	Stack Ovf.	✓	FS
lru	CVE-2021-45720	UAF	✓	U
sized-chunks	CVE-2020-25792	Stack Ovf.	✓	FS
sized-chunks	CVE-2020-25791	Stack Ovf.	✓	FS
sized-chunks	CVE-2020-25795	UAF	✓	FS
rusqlite	CVE-2021-45713	UAF	✓	FS
heapless	CVE-2020-36464	UAF	✓	U
sys-info	CVE-2020-36434	UAF	✓	U
string-interner	CVE-2019-16882	UAF	✓	FS
safe-transmute	CVE-2018-21000	Heap Ovf.	✓	FS
chttp	CVE-2019-16140	UAF	✓	FS
cbox	CVE-2020-35860	UAF	✓	U
id-map	CVE-2021-30455	UAF	✓	FS
id-map	CVE-2021-30457	UAF	✓	FS
simple-slab	CVE-2020-35892	Heap Ovf.	✓	U
simple-slab	CVE-2020-35893	Heap Ovf.	✓	U
scratchpad	CVE-2021-28031	UAF	✓	FS
toodee	CVE-2021-28028	UAF	✓	FS
rdiff	CVE-2021-45694	Heap Ovf.	✓	FS
qwutils	CVE-2021-26954	UAF	✓	FS
insert_many	CVE-2021-29933	UAF	✓	FS
ordnung	CVE-2020-35891	UAF	✓	FS

[*] detected at false-safe (FS) / unsafe (U) site.

Table 2: Reproduction experiment result of RUSTSAN with CVE reproduction test set.

free (CVE-2021-45713) in Appendix A with code examples.

7.3 Detection capability: cross-safety object access

Here we evaluate the detection capability unique to RUSTSAN, detection of safe object access detection explained in §6.2. We prepare two synthetic examples of safe object corruption from unsafe and false-safe shown in Listing 3.

We resort to crafting synthetic examples instead of real-world bugs due to the difficulty of identifying and reproducing them. Many real-world memory errors are initially discovered with ASan through its *inter-object* redzones. We found that a memory error at an unsafe or false-safe site that does not touch any of *inter-object* redzones but corrupts the safe objects, thus not triggering the ASan alarm is too specific to identify and reproduce. Both examples, inspired by real-world CVEs (CVE-2017-1000430 and CVE-2018-21000) but slightly altered, allow arbitrary memory corruption on safe objects. In Listing 3a both and Listing 3b, the adversary can control the argument to the function to corrupt the pointer to point to safe objects at an `unsafe` and false-safe sites respectively.

These cases do not necessarily touch the redzones, but can corrupt only the *contents* of safe objects (i.e., intra-object corruption). These examples illustrate RUSTSAN's Rust cross-safety memory access detection; the unmodified ASan would not have detected the illustrated memory errors, while RUSTSAN extends the ASan with the capability to detect a new class of memory error specific to Rust.

7.4 Compile time with RUSTSAN

We now measure the compile time of RUSTSAN in larger Rust crates to evaluate its scalability.

Scalability test set collection. We selected the nine crates based on the crate's popularity (e.g., Github stars over 10k) and codebase size. Table 3 shows RUSTSAN's compile time for large programs in the scalability set.

Result discussion. Overall, RUSTSAN increased the compile time of the Rust crates by 19.50×. `wasmtime`, a WebAssembly interpreter, produced the largest whole-program LLVM IR of 2253MB. The compile time for the crate with RUSTSAN took 29 minutes and 56 seconds, 30.45× longer than the instrumented normal build. The second largest was `RustPython`, with 2127MB of LLVM IR size and 1688s (28 minutes and 8 seconds) of compile time, which is 24.82× slower than the normal build.

We concluded that the computational complexity of RUSTSAN's analysis remains reasonable even for large programs. The refined information extraction in the HIR/MIR analysis is a contributing factor, as it reduced the amount of taint source (**I2** from §4) by 12.86% compared to a previous work [38] in our experiment (Appendix B).

Considering the typical dynamic testing scenario in which RUSTSAN would be used, the cost of building Rust pro-

```rust
pub fn encode_config_buf(len: usize, buf: &mut Vec<u8>) {
  let mut output_ptr = buf.as_mut_ptr();
  unsafe {
    // new_ptr now points to safe object.
    let new_ptr = output_ptr.offset(len);
    ptr::write(new_ptr, ...);
  }
```

(a) Case for safe object corruption at **`unsafe`** site

```rust
fn to_bytes_vec(ptr: *mut T, offset: usize) {
  let v = unsafe{
    Vec::from_raw_parts(ptr, USIZE_MAX, 0);
  }
  // v[offset] now points to safe object.
  v[offset] = ...;
}
```

(b) Case for safe object corruption at false-safe site

Listing 3: Cases of cross-safety memory corruption detected by RUSTSAN

Target	IR Size (MB)	Compile Time		
		Normal	ASan	RustSan
bat	712	28.48s	29.94s (1.05×)	479.00s (16.81x)
fd	410	23.91s	24.80s (1.02×)	289.00s (12.08x)
ripgrep	387	26.23s	27.41s (1.04×)	232.00s (8.84x)
tokio	289	26.60s	27.20s (1.02×)	178.00s (6.69x)
firecracker	244	19.89s	20.87s (1.04×)	137.00s (6.68x)
hyper	157	23.72s	24.10s (1.02×)	104.00s (4.38x)
Rocket	1911	40.11s	41.05s (1.02×)	1259.00s (31.38x)
wasmtime	2253	58.97s	62.00s (1.05×)	1796.00s (30.45x)
RustPython	2127	68.00s	72.00s (1.06×)	1688.00s (24.82x)
Avg.	943	35.10s	36.59s (1.04×)	684.67s (19.50x)

Table 3: Compile time of scalability set crates with ASan and RustSan.

grams with RustSan once before testing would be trivial. TRust [15] mentions its strategy for dealing with heavy programs that fall back to less precise context-sensitive analysis. The work's evaluation mentions `tokio` and `hyper` (both are also included in our set in Table 3) as examples of large programs. A direct comparison is infeasible due to the implementation's unavailability as of writing. However, we showed that RustSan could build much larger programs with manageable compile times.

7.5 Microbenchmarks

We conducted microbenchmarks on RustSan's safety-aware shadow memory scheme and modified heap allocator. A closer look into the overhead of the two components provides a context into the general application and fuzzing benchmark presented later in this section.

Shadow memory checks. RustSan's shadow memory scheme introduces an additional shadow byte masking in the false-safe sites as shown in Listing 2b. For this reason, while RustSan improves sanitization performance through sanitizer check elimination, its false-safe site checks may add a small overhead. In order to measure the isolated overhead from such *mask-then-branch* checks, we compiled *nbench* [52] into two versions: one with unmodified ASan and the other with theoretic *worst-case* RustSan. The worst-case RustSan instruments all memory access sites with the mask-then-branch checks of false-safe sites and does not eliminate any checks. Figure 4 shows the comparison between the theoretic worst-case RustSan and the unmodified ASan. On average, the ASan version showed 2.06x runtime overhead, and the theoretical worst-case RustSan version showed 2.08x. The experiment results indicate that the performance overhead from the false-safe site checks is minimal.

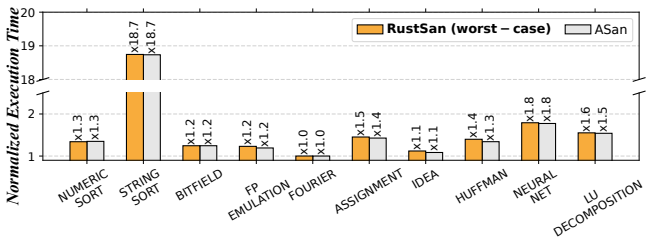

Figure 4: Result of nbench-byte of ASan and worst-case of RustSan instrumentation normalized to uninstrumented version of the benchmark.

Heap allocator. Our benchmark reported that RustSan's heap allocator incurs 5.52% additional overhead in addition to that of ASan. Due to its negligible overhead and straightforward experiment method, we discuss the topic in Appendix C.

7.6 Runtime overhead in general applications

We measured the runtime overhead of RustSan in comparison to ASan in general Rust programs with each crate's built-in benchmark set (i.e., `cargo bench`).

General application benchmark set collection. We collected the most downloaded Rust programs from Crates.io [20], and also referenced benchmarked programs from previous works [15, 38]. We manually verified if the included benchmarks worked correctly and yielded a single-number result that allowed us to compare the performance of RustSan and ASan conveniently.

Benchmark results. Figure 5 compares the runtime overhead of ASan and RustSan with average execution times taken for benchmark completion normalized to those of the uninstrumented versions. The average normalized execution time was 2.40× (=140.3% overhead), and 1.53× (=52.9% overhead) for ASan and RustSan respectively, RustSan reduce the overhead to 62.3% in average. In order to validate that the performance increase is due to the eliminated sanitizer checks, we devised a metric that we call sanitizer check hit decrease rate (**Check Hit Decr. (%)**) in Figure 5. The metric explains *the percentage of ASan runtime sanitizer hit count* RustSan *eliminates*. We confirm that the performance increase is directly proportional to the metric.

The crate with the highest overhead reduction was `adler`. The ASan-instrumented version suffers 5.4× slower runtime performance, while RustSan removed 100% of the sanitizer checks and thus achieved near-native speed. On the other hand, several crates showed little to no ASan overhead reduction. `vec_deque` and `vec` are such examples. We suspect these programs seldom encounter unsafe or false-safe sites and, therefore, do not benefit from RustSan's check removal. Among the evaluated crates, `wasmtime` and `RustPython` are some of the most complex and large Rust ecosystem programs, also included in our scalability set. The experiment, therefore, shows the feasibility of RustSan on larger Rust programs, not to mention the significant performance improvement in both (64% and 63% improvement over ASan, respectively).

Besides the performance, the crates did not experience

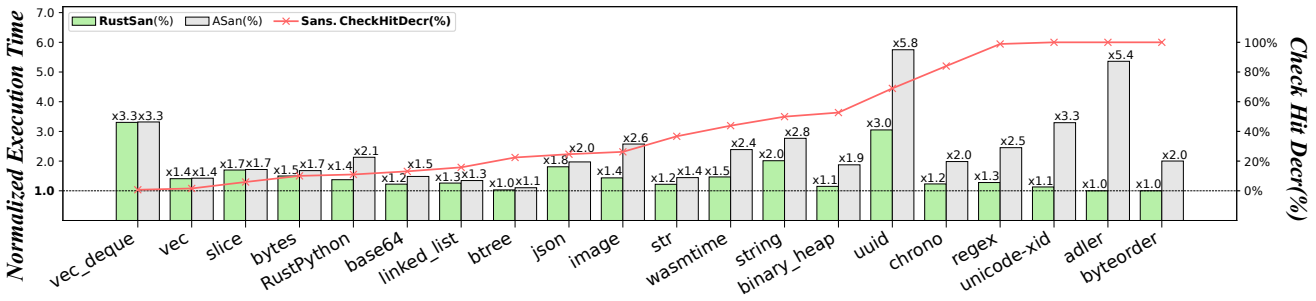

Figure 5: General application performance benchmark with RUSTSAN and ASan. **Normalized Execution Time** (Y1 axis) represents the execution time overhead normalized to execution time of uninstrumented versions of crates. Sanitizer Check Hit Count Decrease (**Check Hit Decr (%)**, Y2 axis) is calculated as the following: (#Checks$_{ASan}$ - #Checks$_{\text{RUSTSAN}}$) / #Checks$_{ASan}$.

program crashes due to RUSTSAN-induced false positives during the benchmark.

7.7 Fuzzing

We further investigate the performance gain that RUSTSAN offers over ASan in the context of fuzzing.

Fuzzing test set collection. We selected our target programs from the Rust trophy-case [13]. We chose the top six crates that have the highest count of reported bugs and also provide built-in *fuzzing harnesses*.

Performance measurements. As with the general application benchmark set, we compiled the crates with RUSTSAN and ASan for comparison. Then, we fuzzed two versions of each crate in the fuzzing set with AFL++ [23] version 4.05c for *24 hours*. We used the built-in harness and seed set included in each crate. We illustrate RUSTSAN's performance gain through the difference *run executions per second*, a conventionally accepted statistic in fuzzing performance evaluation [23, 60]. Thanks to RUSTSAN's runtime overhead reduction, the fuzzer can perform more runs with RUSTSAN than ASan given a fixed duration (24 hours).

Table 4 shows the comparison of RUSTSAN with ASan. We first compared the performance in terms of *execution per second*, in which RUSTSAN showed an average of 23.52% of performance increase (**Incr(%)**) over ASan. To better illustrate the source of the performance gain, we measured *average checks per execution*. Since RUSTSAN has a total execution number than ASan, we use the average number of checks rather than the total encounter checks. Among the targets, `brotli-rs` exhibited the most significant margin of improvement (57%). During runtime, RUSTSAN removed approximately 390K sanitizer checks for each run, which explains the performance improvement. In the case of `cpp_demangle`, the overhead reduction is minimal (0.87%), although the reduction rate is very high (98.37%). This is because the limited number of memory access occurrences (2535.67 checks/sec) is dominated by other sources of overhead in fuzzing, such as seed mutation and relaunching of the process.

Target	Exec/s		Incr. (%)	Avg. checks/exec		Reduc. (%)
	AS	RS		AS	RS	
image-tiff	879.41	952.49	8.31%	7599.75	28.92	99.62%
minidump	432.57	571.33	32.08%	20618.15	76.61	99.63%
pdf	296.68	414.75	39.80%	288990.49	2771.96	99.04%
cpp_demangle	846.63	853.98	0.87%	2535.67	41.43	98.37%
png	795.89	819.38	2.95%	1606.78	935.19	40.68%
brotli-rs	142.46	223.77	57.08%	390218.61	704.22	99.82%

Table 4: Fuzzing performance benchmark of ASan (**AS**) and RUSTSAN (**RS**). RUSTSAN significantly increases (**Incr.**) the number of executed runs per second.

Result discussion. RUSTSAN shows substantial performance gains in several Rust crate fuzzing (e.g., `brotli-rs`). However, the performance of fuzzing a target depends on other influential factors. For instance, load and store instructions may have a limited proportion in the target program. Also, the duration of the pure target program execution time in each run can be relatively shorter in certain programs, allowing the operations of the fuzzer itself (e.g., respawn target process, seed mutation) to dominate the overall execution time. RUSTSAN can only contribute to minimizing the shadow memory checks (i.e., **Reduc. %** in Table 4). The experiment focused on presenting the *isolated* performance gain from RUSTSAN in real-world fuzzing scenarios. We expect that RUSTSAN can be adapted with other ASan optimizations to alleviate the aforementioned factors. For instance, FuZZan [27] optimizes the shadow memory initialization, which directly affects the fuzzer's process respawning time.

Detected errors. Neither ASan nor RUSTSAN found cases of crashes induced by memory error detection. Judging from the included fuzzing harnesses and documentation, we suspect that it is because these crates have been fuzzed to a certain extent already (e.g., more than 24 hours). Hence, easily reachable bugs are likely to have been fixed. Even so, this result at least shows that RUSTSAN did not exhibit any cases of false positives.

8 Security and robustness discussion

Sanitizers are always best-effort solutions since they must balance detection coverage, portability, and performance to be practical. In this sense, we argue that RustSan has shown its value as a practical solution through extensive empirical testing. Here we revisit our evaluation with a robustness perspective and also qualitatively discuss possible sources of incorrectness in RustSan.

8.1 Empirical validation

Our evaluation confirms RustSan's performance advantages and also provides empirical validation to a certain extent given the large number of tested crates. As discussed in our CVE reproduction experiment (§7.2), we confirmed that RustSan successfully detected all CVE cases.

RustSan did not produce any false positives, i.e., program crashes, during the benchmarking. The crates' built-in benchmarks are not meant to trigger bugs in the program; therefore, they also serve as an isolated robustness test on RustSan. In retrospect, RustSan checked approximately 1500 million memory access sites across 18 Rust crates. If a single safe site were marked as unsafe or a false-safe site, then it would have caused a program-crashing false-positive case, as these sites would then not be able to access safe objects.

During the fuzzing experiment, RustSan checked a total of 89 billion memory access sites throughout 24 hours of fuzzing for 6 crates. False positives were also absent in this experiment. As with the CVE experiment, the fuzzing experiment could serve as a false negative validation. However, as we mentioned, the baseline ASan also did not discover any previously unknown memory error during this period. Understanding relatively untested Rust crates and manually generating fuzzing harnesses for fuzzing tests would be a rather daunting task that we could not include in this work.

8.2 Qualitative analysis

While RustSan makes a conscious effort to be conservative in its program analysis techniques, complex program analysis techniques can often be challenging to achieve perfect completeness and soundness. We use Table 5 to discuss the potential issues of the RustSan validation model systematically.

Incomplete HIR/MIR analysis. MC1 and **MC2** can happen due to an incomplete HIR/MIR analysis. Recall that the unsafe sites are memory access sites within the **`unsafe`** blocks by definition and are to be identified by the MIR-level analysis. The consequence of these misclassifications is that the unsafe site would be able to silently corrupt safe objects (in the case of **MC1**) or overlapping objects (in the case of **MC2**). Therefore, these are false negative cases that may undermine RustSan's memory validation model.

Unsound HIR/MIR analysis. Conversely, the HIR/MIR analysis may misidentify safe or false-safe sites that are *outside* the **`unsafe`** blocks as unsafe sites (**MC3**, **MC5**). **MC5** is a false positive case where the misclassified unsafe site would cause a false alarm as it attempts to (legitimately) access safe objects. Another implication of the **MC5** cases is that they diminish the performance advantage of RustSan, as unnecessary sanitizer checks are added in this case. **MC3** is also a false positive because false-safe sites, under RustSan's validation model, should be able to access safe objects. A false alarm will be raised as the site must access the safe objects received through a safe data flow during runtime.

Robustness of RustSan HIR/MIR anlaysis. Our MIR analysis is a targetted analysis of the **`unsafe`** blocks that only amount to a very small portion of Rust programs. For this reason, the MIR analysis can be made complete and sound. Furthermore, RustSan's introduction of recursive scope safety in MIR analysis addressed a source of false negatives present in a previous work [38]. As such, we argue that HIR/MIR is robust, and the occurrences of related misclassification cases would be extremely rare. Also, our evaluation empirically proved the absence of false positive cases.

Incomplete points-to analysis. Incomplete data flow and points-to analysis may result in the cases of **MC4**. The analysis may mistakenly omit the points-to relation between a site and an unsafe object. Consequently, a false-safe site that must access only overlapping and unsafe objects is falsely vindicated of the sanitizer check. RustSan would be oblivious of such a site corrupting safe objects; therefore, this is a false negative case.

Unsound points-to analysis. An unsound points-to analysis may cause the case of **MC6**, in which a safe site is misclassified as a false-safe site. As with **MC5**, this case would cause a false positive and performance degradation due to unnecessary sanitizer checks.

Robustness of points-to anlaysis. RustSan employed a state-of-the-art program analysis tool [50] that has been widely used in the domain of software security [15, 18, 28–30, 38, 44]. Our evaluation indicated that the cases of **MC6**-type false positives would be rare. However, the risk of **MC4** remains an external dependency, while we still deem the tool reliable given its extensive usage in the domain.

	Misclassification	False {Pos, Neg}	Cause
MC1	Unsafe→Safe	False Neg	Incomplete HIR/MIR
MC2	Unsafe→False-safe	False Neg	Incomplete HIR/MIR
MC3	False-safe→Unsafe	False Pos	Unsound HIR/MIR
MC4	False-safe→Safe	False Neg	Incomplete points-to
MC5	Safe→Unsafe	False Pos	Unsound HIR/MIR
MC6	Safe→False-safe	False Pos	Unsound points-to

Table 5: Classification of potential site misclassifications.

9 Related Work

9.1 Sanitizers for detecting memory bugs

Sanitizers have long been used for detecting bugs in programs written in memory-unsafe languages to detect memory erros [12, 34, 40, 42, 47–49, 53, 58]. Both spatial memory error sanitization [12, 19, 40, 42, 47, 54], and temporal memory error sanitizers [34, 47, 53, 58] have been well-studied offers multiple alternatives in program dynamic testing.

Among the sanitizer designs, ASan [24, 47] is the most widely-used sanitizer integrated into the mainstream compiler thanks to its lightweight shadow memory-based checking mechanism and high compatibility [48]. RUSTSAN aims to retrofit ASan for Rust by eliminating redundant checks while inheriting its portability and compatibility.

9.2 Optimizing sanitizers for performance

Sanitizers often induce a very high runtime overhead, so previous works have endeavored to optimize them.

Eliminating sanitizer checks. Identifying and removing redundant sanitizer checks has been one of the approaches for lowering the runtime overhead [16, 25, 32, 43, 51, 60]. Several works have located and removed unnecessary checks through accurate static analysis methods without compromising detection coverage. For instance, SANRAZOR [59] combines dynamic code coverage data and static data dependencies of checks to find redundant sanitizer checks. ASan-- [60] perform lightweight static analysis to detect and remove recurring checks and optimize sanitizer checks. These works have sought to optimize ASan for its intended target programs, such as the ones written in C/C++. On the other hand, RUSTSAN points out ASan's inefficiency for only partially-unsafe Rust programs and proposes a solution.

Making tradeoffs. Some works propose a trade-off between detection capability and performance for a chosen performance budget. ASAP [47] profiles the target programs to determine frequently accessed code sections with the most performance gain when sanitizer checks are removed. SANRAZOR [59] offers a configuration with a sanity level for varied performance trade-offs. Unlike these works, RUSTSAN retains ASan's detection capabilities while providing additional rust-specific safe object protection.

Optimizing sanitizer runtime. A few works focused on optimizing the sanitizer runtime to make it more suitable for certain workloads, such as fuzzing. Fuzzan [27] designed an efficient metadata for redzone management to accelerate fuzzing with ASan. PartiSan [35] partitions the application into sanitized and unsanitized *slices* so that the unsanitized slice can run without the overheads. Bunshin [57] divides sanitizer checks among various programs that are run concurrently. We expect that RUSTSAN can be combined with these techniques to further optimize its performance.

9.3 Securing Rust Programs

Static analysis for securing Rust. Static analyzers have been introduced to detect bugs in unsafe Rust. Rudra [14] introduced a scalable static analysis in Rust's MIR and HIR to find bugs with specific patterns. MirChecker [37] combines static numerical analysis and symbolic execution in MIR to detect potential runtime panic and memory errors. Rupair [26] automatically detects the overflow inside the `unsafe` code and rectifies it using a lightweight dataflow analysis in MIR. RUSTSAN also adapts HIR and MIR analysis to achieve a more fine-grained information extraction on the Rust `unsafe` blocks. Static and dynamic testing of programs offers varying advantages and disadvantages depending on the use case. Unlike these works, RUSTSAN is a runtime sanitizer that detects memory errors during program execution (e.g., through fuzzing).

Runtime isolation for Rust programs. Many proposed runtime isolation mechanisms to *contains* the ramifications of memory errors triggered by `unsafe` blocks and memory-unsafe external libraries. XRust [38] and TRust [15] introduce a split memory allocator scheme and place objects touched by `unsafe` into the unsafe heap. They also include custom instrumentation frameworks that contain safe memory instructions affected by unsafe objects (*false-safe* as explained in §3.1). MPK-based isolation is also explored to isolate the memory access by the C libraries [15, 31, 45]. PKRUSafe [31] automatically identifies objects shared between Rust and C through dynamic profiling and isolates memory accesses to those objects using MPK.

The static analysis methods for identifying the safety of Rust program instructions and objects [15, 38] inspired RUSTSAN, while RUSTSAN improves them with its statement-level analysis in MIR. Moreover, RUSTSAN is a sanitizer and therefore must detect memory errors *as they occur*. RUSTSAN consciously inherits the compatibility and portability of ASan. One concrete design decision in this regard is the avoidance of architecture-specific features such as MPK. In addition, it avoids computationally expensive techniques such as the context-sensitive points-to analysis [15] to ensure scalability.

10 Conclusion

In this paper, we presented RUSTSAN, a retrofitted AddressSanitizer (ASan) design for Rust programs. RUSTSAN significantly improves the performance of AddressSanitizer on Rust programs through selective memory access site instrumentation. The key insight was that a large number of sites still retain Rust security guarantees and therefore identified and freed of sanitizer checks. Our evaluation empirically proved RUSTSAN's detection capability through CVE reproduction. Also, RUSTSAN showed 62.3% of performance gain against ASan in the application benchmark, and 23.52% in the fuzzing experiment.

Acknowledgments

We deeply appreciate our shepherd and the anonymous reviewers for their constructive comments and feedback. This work was supported by the Mobile eXperience division, Samsung Electronics Co., Ltd. This work was also supported by grants funded by the Korean government: the National Research Foundation of Korea (NRF) grant (NRF-2022R1C1C1010494), Institute of Information & Communications Technology Planning & Evaluation (IITP) grants (No. 2022-0-00688, No. 2022-0-01199, IITP-2024-RS-2023-00259497), and Korea Internet & Security Agency (KISA) grant (1781000009).

References

[1] https://github.com/rust-fuzz, 2022.

[2] Rust for linux. https://rust-for-linux.com/, 2022.

[3] The rust unstable book: sanitizer. https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html, 2022.

[4] Fast and safe http for the rust language.o. https://hyper.rs/, 2023.

[5] A fast and secure runtime for webassembly. https://wasmtime.dev, 2023.

[6] Rocket is a web framework for rust. https://rocket.rs/, 2023.

[7] Rust-based platform for the web. https://swc.rs/, 2023.

[8] Secure and fast microvms for serverless computing. https://firecracker-microvm.github.io/, 2023.

[9] Unsafe rust - the rust programming language. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html, 2023.

[10] Hussain M. J. Almohri and David Evans. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 248–255, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[12] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Not.*, 29(6):290–301, jun 1994.

[13] The Rust Fuzzing Authority. Trophy case. https://github.com/rust-fuzz/trophy-case, 2023.

[14] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 84–99, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. Trust: A compilation framework for in-process isolation to protect safe rust against untrusted code. In *32st USENIX Security Symposium (USENIX Security 23)*, pages 4345–4363, Boston, MA, August 2023. USENIX Association.

[16] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[17] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, November 2020.

[18] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 715–733, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Crispan Cowan. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*. USENIX Association, 1998.

[20] Crate.io. The rust community's crate registry. https://crates.io/, 2023.

[21] The Servo Project Developers. Servo, the parallel browser engine. https://servo.org/, 2023.

[22] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 246–257, New York, NY, USA, 2020. Association for Computing Machinery.

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

[24] Google. Addresssanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer, 2023.

[25] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. Software Eng.*, 3(3):243–250, 1977.

[26] Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. Rupair: Towards automatic buffer overflow detection and rectification for rust. In *Annual Computer Security Applications Conference*, ACSAC '21, page 812–823, New York, NY, USA, 2021. Association for Computing Machinery.

[27] 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.

[28] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019.

[29] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings 2020 Network and Distributed System Security Symposium*, San Diego, CA, 2020. Internet Society.

[30] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave (Jing) Tian. PASAN: Detecting peripheral access concurrency bugs within Bare-Metal embedded applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 249–266. USENIX Association, August 2021.

[31] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 132–148, New York, NY, USA, 2022. Association for Computing Machinery.

[32] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *PLDI*, pages 270–278. ACM, 1995.

[33] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, page 51–57, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[35] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. Partisan: fast and flexible sanitization via runtime partitioning. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*, pages 403–422. Springer, 2018.

[36] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. Detecting cross-language memory management issues in rust. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, pages 680–700, Cham, 2022. Springer Nature Switzerland.

[37] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. Mirchecker: Detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2183–2196, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with xrust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 234–245, 2020.

[39] LLVM. The llvm compiler infrastructure. https://llvm.org/, 2012.

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. 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*, PLDI '09, page 245–258, New York, NY, USA, 2009. Association for Computing Machinery.

[41] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020.

[42] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, may 2005.

[43] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI*, pages 333–344. ACM, 1998.

[44] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1937, 2021.

[45] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe rust safe with galeed. In *Annual Computer Security Applications Conference*, ACSAC '21, page 824–836, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Rust. The rust programming language. https://github.com/rust-lang/rust/tree/1.66.0, 2022.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

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

[49] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: Fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, 2015.

[50] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[51] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Trans. Reliab.*, 65(4):1682–1699, 2016.

[52] Uwe F. Mayer. Linux/Unix nbench. https://www.math.utah.edu/~mayer/linux/bmark.html, 2017. Last accessed March 08 , 2023,.

[53] 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*, EuroSys '17, page 405–419, New York, NY, USA, 2017. Association for Computing Machinery.

[54] Zhilong Wang, Xuhua Ding, Chengbin Pang, Jian Guo, Jun Zhu, and Bing Mao. To detect stack buffer overflow with polymorphic canaries. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 243–254, 2018.

[55] Rust Secure Code WG. Rustsec advisory database. https://rustsec.org/advisories/, 2023.

[56] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. Memory-safety challenge considered solved? an in-depth study with all rust cves. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–25, 2021.

[57] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: Compositing security mechanisms through diversification. In *USENIX Annual Technical Conference*, pages 271–283. USENIX Association, 2017.

[58] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[59] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 479–494, 2021.

[60] 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, Boston, MA, August 2022. USENIX Association.

A CVE case study

```
1 fn to_bytes_vec(mut from: Vec<T>) -> Vec<u8> {
2   ...
3   let capacity = bytes.capacity() / size_of::<T>();
4   let len = bytes.len() / size_of::<T>();
5   unsafe{
6     Vec::from_raw_parts(ptr as *mut T, capacity, len)
7   }
8 }
```

(a) Heap overflow vulnerability in CVE-2018-21000.

```
1  let db = Connection::open_in_memory().unwrap();
2  {
3    let obj = Box::new(...);
4    let closure = |...| {
5      *obj = ...; // obj is captured here
6    };
7    db.update_hook(Some(closure));
8  }
9  fn update_hook<'c, F>(&'c self, hook: Option<F>)
10   where
11   F: FnMut(Action, &str, &str, i64) + Send + 'c,
12 {
13  ...
14  let boxed_hook: *mut F = Box::into_raw(Box::new(hook));
15  unsafe {
16    ffi::sqlite3_update_hook(..., boxed_hook as *mut _)
17  }
18 }
```

(b) Use-after-free vulnerability in CVE-2021-45713.

Listing 4: CVE Case Studies

We provide the case studies that analyze the root cause of memory error patterns in Rust and RUSTSAN detection on them below.

Case Study 1: CVE-2018-21000. This CVE is a case of heap overflow on the Rust implementation of transmute. Line 5 of Listing 4a is the culprit of the problem. `Vec::from_raw_parts` is a Rust standard library function that constructs a new vector object from a raw pointer, taking vector length and capacity arguments. The mistake here is the reversed second and third argument order. In Rust terminology, the capacity of a vector is the maximum space allocated accounting for the future insertion of elements, while length refers to the current number of elements in the vector. Hence, the constructed vector is highly like to cause out-of-object-bound memory access in later use. In our reproduction, RUSTSAN reported object-end redzone access on an unsafe object at a false-safe site.

Case Study 2: CVE-2021-45713. The CVE in `rusqlite` was a case of use-after-free caused by a violation of the Rust object lifetime guarantee in unsafe Rust. In Line 16 of Listing 4b, `update_hook` API casts the closure to a function pointer and registers it to the external foreign (C/C++) library inside the **`unsafe`** block. The unsafe Rust and the foreign libraries do not conform to Rust's lifetime guarantees. As a result, the callback invoking when the `closure` and `obj` dropped results in a use-after-free. With this CVE reproduction, RUSTSAN reported quarantined safe object access at a false-safe site.

B Taint source reduction with HIR/MIR analysis

Scale. Set	IR Size	Identified taint source		
		XRust [38]	RUSTSAN	Decrease (%).
bat	712	3772	3066	18.72%
fd	410	8075	7007	13.23%
ripgrep	387	7907	6470	18.17%
tokio	289	16373	15382	6.05%
firecracker	244	9932	9021	9.17%
hyper	157	19210	17438	9.22%
Rocket	1911	47547	42534	10.54%
wasmtime	2253	44076	37630	14.62%
RustPython	2127	71697	60210	16.02%

Table 6: The identified taint source number with RUSTSAN for large programs.

We also evaluated RUSTSAN's and XRust [38]'s method of HIR/MIR analysis for taint source identification (i.e., **I1** and **I2** explained in §4). Note that XRust uses **I1**, all statements in **`unsafe`** blocks, directly as a taint source while RUSTSAN uses **I2**, which is a refined subset of **I1** that only contains write statements. Also, RUSTSAN can identify the statements inside inlined functions, while XRust misses such statements. Hence using the **I2** set reduces the number of taint sources, while the inlined functions increase RUSTSAN's set compared to that of XRust. Even so, RUSTSAN showed an average of 12.86% decrease in the taint source set. This means that RUSTSAN delivers more refined analysis results in the HIR/MIR stage to the LLVM stage, thereby attenuating the complexity in the LLVM IR analyses.

C Heap allocator microbenchmark

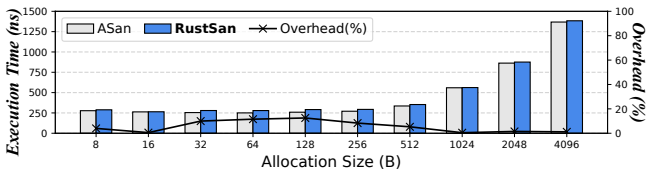

Figure 6: Microbenchmark of RUSTSAN runtime heap allocation function (`malloc()`

RUSTSAN modifies the ASan's heap allocator to color the allocated memory with RUSTSAN's shadow memory scheme. To measure the overhead of the runtime shadow memory management, we compare the average execution time of `malloc()` for varied allocation sizes using RUSTSAN and ASan. The

allocation (`malloc()`) for each memory size was repeated 10 million times.

Figure 6 shows the experiment results; Against the allocator of ASan, RUSTSAN's allocator shows an average overhead of 5.52%. Based on this microbenchmark, we consider the influence of heap allocator changes negligible in general application and fuzzing benchmarks.