ERASAN: Efficient Rust Address Sanitizer

Jiun Min*
Department of Computer Science
UNIST

Dongyeon Yu*
Department of Computer Science
UNIST

Seongyun Jeong
Department of Computer Science
UNIST

Dokyung Song
Department of Computer Science
Yonsei University

Yuseok Jeon†
Department of Computer Science
UNIST

***Abstract*—Rust is a rapidly growing system programming language that ensures a speed comparable to traditional C/C++ system programming languages, along with the additional benefit of guaranteed memory safety. However, Rust's strict security rules make implementing and executing some features challenging. To address this, Rust has introduced unsafe Rust, which is less constrained by these strict rules. Nevertheless, these unsafe Rust, where strict Rust security rules are not fully applied, can cause temporal and spatial memory bugs that account for 22% of the Rust bugs reported between 2016 and 2023.**

In this paper, we propose an efficient address sanitizer design customized for Rust, called ERASAN, to detect memory bugs in Rust programs more efficiently than prior work. Based on our thorough analysis of safe and unsafe Rust programming language standards as well as memory bugs found in real-world Rust programs over the past years, we design and implement ERASAN to only instrument memory accesses in both safe and unsafe code areas where Rust cannot guarantee safety. We evaluate ERASAN with several real-world applications. ERASAN removes an average of 90.03% of ASan's memory access checks. Due to this, ERASAN significantly reduces ASan's performance overhead by an average of 239.05% without harming its bug-finding ability.

1. Introduction

Traditional C/C++ system programming languages do not guarantee memory safety, leading to memory bugs. However, the newly emerging Rust programming language [9] guarantees memory safety without sacrificing performance, and for this reason, Rust is now widely used. For example, due to Rust's performance and security, major operating systems such as Linux [19], Android [3, 8], and Windows [10] are being partly implemented in Rust.

To guarantee memory safety, Rust employs strict safety rules, including ownership [17], lifetime [14], and borrowing [4]. However, these strict rules make it impossible to implement some features (e.g., double-linked lists) or to ensure memory safety during certain operations (e.g., interacting with foreign languages). Therefore, an unsafe Rust [26] is proposed to allow the implementation and execution of these functions without restrictions imposed by Rust's security rules. However, some operations in these unsafe Rust, which are not fully restricted by Rust security policies, lead to various memory bugs. According to our analysis, 22% of all 581 Rust bugs reported in the RustSec Advisory Database [20] over the seven years (from 2016 to 2023) are buffer overflow, use-after-free, and double free.

To address these memory bugs in Rust, several approaches have been proposed, and these approaches can generally be classified into static analysis-based [29, 31, 38, 43, 44], dynamic analysis-based [11, 49] bug detection approaches, and isolation approaches [30, 40, 41, 55]. Isolation techniques generally separate safe memory areas from unsafe areas to deny illegal access (e.g., due to memory bugs) to safe memory areas from unsafe areas. However, these techniques cannot fundamentally eliminate memory bugs, since these approaches cannot detect and remove memory bugs. Static analysis-based approaches suffer from high false positive issues (e.g., Rudra [29] shows a false positive up to 84%) mainly due to the over-approximated analysis of static analysis (e.g., indirect call target analysis). Additionally, existing static analysis tools have limited bug detection coverage, as these tools cover only a specific portion of memory bugs.

Unlike static analysis-based techniques, dynamic analysis-based approaches (e.g., sanitizers [50]) have fewer false positives and are the most widely used approaches for finding memory bugs. These approaches are generally combined with fuzzing approaches [46], which are actively being researched and used to increase detection capability further. The MIRI [11] interpreter for Rust's mid-level intermediate representation (MIR) is designed to detect memory bugs. However, it has a significant high runtime overhead limitation, as it interprets all code at execution time. Additionally, MIRI cannot handle specific behaviors, such as releasing memory from user-defined destructors.

Address Sanitizer (ASan) [49] is the most widely used

*. Equal Contribution
†. Corresponding author

approach [50] and has detected a large number of bugs in C/C++. For example, ASan has detected more than 10,000 memory bugs [34, 36] across various applications, including over 3,000 memory bugs in Chrome [34] and more than 3,000 in Google's server software [34], as well as in the Linux kernel (over 1,000 memory bugs [35, 37]), through a customized Kernel Address Sanitizer (KASan) [37]. ASan can also be applied to Rust, as Rust internally uses LLVM as its backend optimizing compiler. Therefore, ASan is widely used to detect memory bugs in Rust due to its strengths (e.g., low false positive and broad detection coverage) and the Rust compiler's support [12, 18] for ASan, making it easy to apply without manual efforts. However, ASan inherently checks all memory accesses, leading to around two times [34] runtime overhead. Although several existing works [53, 56, 57] have been proposed to reduce this high runtime check overhead in ASan, they can still not eliminate unnecessary checks for memory accesses whose safety is guaranteed by Rust's strict memory safety rules.

We present ERASAN, an efficient address sanitizer customized for the Rust environment, enhancing efficiency by removing unnecessary checks in regions where memory safety is guaranteed per the strict rules of Rust without trading bug detection precision for performance. To achieve this, we first conducted a wide-ranging and in-depth analysis for various standards [4, 5, 14, 17, 20, 26], and all 581 bugs reported in the `RustSec advisory database` [20] over seven years from 2016 to 2023. Through this analysis, we draw memory bug patterns in Rust. We also find that most unsafe areas are still protected by the robust security policies of Rust, which guarantee memory safety. Consequently, we identified that memory bugs occur only in areas related to raw pointers (including even safe Rust). Raw pointers are the only type of pointers not checked by the Rust compiler (`rustc`). More specifically, this raw pointer can point to invalid memory, free'd memory, or even NULL, since Rust safety checks (e.g., ownership, borrow checks, or bounds check both at compile time and runtime) do not apply to raw pointers. As a result, raw pointers and their related code area (including Rust safe) can break Rust safety barriers and lead to memory bugs.

To detect the potential memory bugs in these areas, ERASAN identifies raw pointer-related areas, including safe Rust, to thoroughly check memory accesses in these identified areas. In addition, ERASAN further eliminates memory access checks in identified areas if these accesses have already been verified by Rust's runtime checks for spatial memory bugs and are safe from temporal memory bugs.

To do this, ERASAN performs the following three steps: (1) identifying raw pointers, (2) extracting unsafe memory accesses, and (3) conducting selective check instrumentation. Initially, ERASAN identifies instructions that directly handle raw pointers (e.g., dereferencing raw pointers or assigning an address to a raw pointer). Unfortunately, at the LLVM-IR code, the type information of raw pointers is no longer available (i.e., difficult to distinguish from other instructions such as Rust's safe references), so raw pointer identification is performed at the MIR code for precise analysis. During the MIR analysis phase, ERASAN inserts additional annotation into every instruction that directly handles raw pointers to accurately identify raw pointers in the subsequent analysis phase (e.g., LLVM IR pass). This allows for a more precise analysis in the following analysis stages. In the next step, ERASAN conducts points-to analysis (to identify possible targets each raw pointer can point to) and value-flow analysis to extract all memory accesses (e.g., references) that are in alias relation with the extracted raw pointer set. These identified raw pointers and references in alias relation, potentially related to memory bugs (e.g., accessing a freed object that is freed by a raw pointer), are candidates for the sanitizer check instrumentation to detect memory bugs. However, for memory accesses excluding raw pointer memory accesses, Rust conducts spatial memory bug checks. Therefore, inserting additional check instrumentation is unnecessary if an identified memory access is not via a raw pointer and is unrelated to temporal memory safety bugs. For this, ERASAN identifies the free (e.g., drop and return) locations of all objects and then finds all memory access instructions (i.e., ERASAN's target memory access set) that come after the freeing code, which are potentially related to temporal memory safety bugs. Lastly, ERASAN selectively applies ASan's memory access check instrumentation to the identified memory access set.

Our evaluation shows that ERASAN significantly reduces ASan's performance overhead without any reduction in its bug-finding capability. More specifically, by removing an average of 90.03% of existing ASan checks, ERASAN can reduce ASan's performance overhead by an average of 239.05%. Additionally, ERASAN shows the same bug detection capability as ASan, successfully detecting the same set of reported Rust memory bugs.

This paper makes the following contributions:

- Conducting a broad and in-depth analysis of memory bugs found in the real-world programs, which allows us to taxonomize patterns of memory bugs in Rust programs. Our key finding is that only areas related to raw pointers, not all unsafe areas, are prone to have memory bugs.
- Designing and implementing ERASAN, which only instruments memory accesses in both safe and unsafe code regions affected by raw pointers not fully covered by Rust security checks.
- Evaluating ERASAN on various real-world Rust applications and showing that ERASAN significantly reduces ASan's performance overhead by 239.05% through the elimination of approximately 90.03% of existing ASan checks.

2. Background and Motivation

2.1. Security Rule of Rust

Rust is designed to guarantee memory safety by leveraging the following four main policies: ownership [17], borrowing [4], lifetime inference [14], and runtime bounds checking [5]. According to the ownership rule, each value

TABLE 1: Applicability of Rust memory bug related memory safety policies (R1:Ownership, R2:Borrow Check, R3:Lifetime Inference, R4: Bound Check in static-time, R5: Bound Check in run-time) to each capability.

Capabilities	Compile Time				Run time
	R1	R2	R3	R4	R5
Dereferencing a Raw Pointer	✗	✗	✗	✗	✗
Calling an Unsafe Function/Method	✓	✓	✓	✓	✓
Implementing an Unsafe Trait	✓	✓	✓	✓	✓
Other Operations	✓	✓	✓	✓	✓

must be owned by one entity during execution. The borrowing rule safely allows temporarily transferring the ownership of a value to an entity.

Additionally, these proactive policies in Rust help to prevent data races by ensuring that references cannot be used simultaneously as mutable (only one reference can exist). Lifetime inference ensures that references do not outlive the data they point to. For example, in the case of local variables stored in the stack, each local variable has a lifetime that defines its valid scope. When a function returns or a block ends (i.e., lifetime expires), these variables are removed from the stack, and their destructors [25] are called for cleanup. Additionally, in the case of dynamic allocations stored in a heap, this memory location is managed by ownership and lifetime inference. When the owner's lifetime ends (i.e., the owner goes out of scope), the data is automatically cleaned up by the `Drop` trait to prevent memory leaks or dangling pointers.

Due to these policies, a Rust program is guaranteed to have no temporal safety violations such as use-after-free, double-free, and data race bugs. Rust also provides compile and runtime checks for out-of-bounds access through bounds checking. More specifically, if the length of the memory allocation is determined at compile time, such as stack allocations, `rustc` checks whether the memory accesses a region within the valid memory range. Also, if the length is determined at runtime, such as heap allocations whose size cannot be statically determined, these memory accesses (e.g., indexing of a vector) are checked by the Rust runtime's bounds checking [5].

However, these strong Rust policies do not apply to all areas of Rust. In these areas where the policies are not fully applied, various memory bugs can occur. Therefore, it is important to identify these areas precisely and insert memory bug checks.

2.2. Address Sanitizer

Address Sanitizer (ASan) [49] is the most popular and widely used sanitizer. ASan detects memory bugs by maintaining the accessibility information of each byte in shadow memory. More specifically, in applications where ASan is applied, objects are marked as accessible in the shadow memory. They are surrounded by inaccessible red zones, which are marked as inaccessible in the shadow memory. Additionally, when an object is freed, the corresponding area in the shadow memory is marked as freed (internally using ASan's quarantine queue [1]). In the case of out-of-bounds memory bugs, ASan can detect when memory access goes to the red zone surrounding valid objects. Other memory bugs, like use-after-free or double-free bugs, can be detected when the application accesses memory marked as freed in the shadow memory.

```
1  fn main(){
2   /* Allocate the value */
3   let v1 = 23;
4
5   /* Safe Reference */
6   let s1 = &v1;
7
8   /* Access Safe Reference */
9   println!("{}", s1);
10
11  /* Raw Pointer */
12  let u1  = &v1 as *const i32;
13
14  /* Access Raw Pointer */
15  unsafe {println!("{}", *u1);}
16 }
```

```
17 define internal void @main() {
18 bb5:
19  ; Memory Access : s1
20  %2 = load i32, i32* %s1
21  ; ASan Instrumentation
22  call void @asan_load64(i32 %r)
23  br label %bb6
24 bb9:
25  ; Memory Access : *u1
26  %3 = load i32*, i32** %u1
27  ; ASan Instrumentation
28  call void @asan_load32(i32 %r)
29  br label %bb6
30  ...
31 }
```

Figure 1: An example of ASan's memory check instrumentation in Rust program.

Since the Rust compiler supports several sanitizers, including ASan (via `-Zsanitizer=address` flag) [18], applying ASan to Rust applications is easy and does not require any additional manual effort. However, after applying ASan to Rust, there is a considerable overhead problem (more than two times) caused by ASan. The majority of this high overhead comes from ASan's memory access checks, which generally contribute to 86% of ASan's overhead [57]. However, many of these memory access checks are superfluous, as many of these memory accesses are checked by Rust per its security policy. For example, in Figure 1, ASan inserts memory access safety check codes on Line 9, where Rust guarantees memory safety through its ownership and boundary policies (i.e., ASan instrument to LLVM-IR instruction on Line 22). Consequently, these unnecessary checks can significantly degrade the performance of Rust programs. Although several existing approaches [52, 53, 56, 57] have been proposed to identify ASan's unnecessary checks, these approaches do not consider the safety of memory accesses guaranteed by Rust. Therefore, some memory access checks identified as needed checks (but Rust's strong security policies can confirm memory safety) might be unnecessary. To address this issue more precisely, it is necessary to identify areas where Rust cannot ensure memory safety and correctly insert ASan's memory access checks in these areas.

3. Real-World Rust Memory Bugs Analysis

In this section, we introduce (§3.1) how Rust's security policies are applied within the Rust unsafe and explain its implication on potential memory bugs in Rust programs. Furthermore, through large-scale analysis of real-world Rust memory bugs, we explain (§3.2) how improper uses of Rust's unsafe areas could lead to memory bugs.

TABLE 2: Analysis of all 131 memory bugs in RustSec from 2016 to 2023

Memory Bug Type	Detected by Rust		Total
	Yes	No	
Buffer-Overflow (BOF)	12	49	61
Use-After-Free (UAF)	0	44	44
Double-Free (DF)	0	26	26
Total	12	119	131

3.1. Memory Safety Implications of Unsafe Rust

Rust security policies are often too restrictive when implementing low-level abstractions or interacting with programs written in foreign programming languages. These restrictions can be selectively bypassed by using the unsafe keyword [26]. In Rust, the unsafe keyword signals a transition to an unsafe domain, allowing certain unsafe operations that bypass Rust's safety policies. Within these unsafe domains, developers obtain the following five specific capabilities [26]:

(i) Dereferencing a raw pointer
(ii) Calling an unsafe function/method
(iii) Implementing an unsafe trait
(iv) Accessing fields of unions
(v) Accessing/modifying a mutable static variable

However, these operations, allowed in the unsafe domain, do not directly lead to memory bugs. We find that the first capability, (i) *dereferencing a raw pointer*, is the only capability that, when misused, can directly cause memory bugs. These raw pointers are not guaranteed to point to in-bounds, valid memory, nor be `non-NULL` by Rust. Also, there is no restriction to using raw pointers to create multiple immutable and mutable references, which are the well-known root causes of temporal memory bugs.

The following two capabilities (i.e., (ii) and (iii)) pertain to denoting the safety of a function, method, or trait. Calling unsafe functions or methods can lead to memory bugs, but the root cause is the improper use of raw pointers (the first capability mentioned before) within the called functions or methods. A trait is marked unsafe when Rust is unable to guarantee that certain invariants hold. Violations of these invariants, however, generally result in logical errors rather than memory bugs. Memory safety invariants could possibly be compromised by using unsafe traits as well. Still, the root causes of these memory bugs are also the misuse of raw pointers within one of its methods. The last two capabilities (i.e., (iv) and (v)) do not directly cause memory bugs either; the former can directly cause type confusion only, and the latter causes data races, which lead to misinterpretation of union and global variables, respectively. Accessing union fields and static (i.e., global) variables are still within the bounds (therefore, no spatial memory bugs) of the underlying objects, and while these objects are valid (no temporal memory bugs).

We analyze the types of safety policies of Rust that are removed when using each capability and summarize the result in Table 1. This analysis clearly shows that the capability of *raw pointer dereferencing* is the only operation that can directly cause memory safety violations. Therefore, despite the unsafe label in the function or method, a large portion of these functions and methods are protected by the Rust safety policies, as shown in Table 1.

```
1  /* RUSTSEC-2020-0097: UAF due to a raw pointer without unsafe
   ↪  block */
2  #![forbid(unsafe_code)]
3
4  use xcb::base::Error;
5
6  fn main() {
7      let mut v1: Vec<i8> = vec![1, 2, 3, 0];
8      let _ = Error {
9          ptr: v1.as_mut_ptr(); // a raw pointer
10     };
11
12     // use-after-free in v1
13     v1[0] = 123;
14 }
```

Figure 2: An example of UAF (RUSTSEC-2020-0097) in Rust *xcb* crate

Furthermore, to detect all possible memory bugs in Rust, an unsafe Rust-based analysis is not enough, and an analysis based on raw pointers is needed. For example, in the real-world memory bug (RUSTSEC-2020-0097) [33] found in the `xcb` crate, as shown in Figure 2, an UAF bug can occur without using unsafe Rust with `![forbid(unsafe_code)]` (line 2). More specifically, this bug occurs when the `v1` vector in the `Error` structure is deallocated due to the lifetime rule, and then an attempt to access (line 13) deallocated `v1` vector from a safe Rust. As demonstrated in the above example, without the use of unsafe areas, memory bugs can arise in Rust through raw pointers that are initialized in safe Rust. Therefore, to more accurately identify memory bugs in Rust, it is necessary to check all code related to raw pointers. In the following section (§3.2), based on the analysis of all Rust bugs listed in the RustSec Advisory Database [20], we will explain in detail how these raw pointers can lead to various memory bugs.

3.2. Real-World Rust Memory Bug Patterns

To check the correctness of our research direction (i.e., raw pointer-based analysis), we analyze all 581 Rust bug reports in the RustSec Advisory Database [20] over seven years (from 2016 to 2023-12). RustSec Advisory-DB contains all Rust-related vulnerabilities and is a superset of Rust program CVEs. Among them, as shown in Table 2, we identify 131 (around 22%, 131 out of 581) memory bugs (i.e., BOF, UAF, and DF bugs) that ASan can detect. The remaining types of bugs are mainly related to maintenance, crypto-failed, logic, and concurrency bugs. Among 131 temporal and spatial memory bugs, we find that only 9% (12 out of 131) can be detected (all these bugs cause the program panic) by Rust's

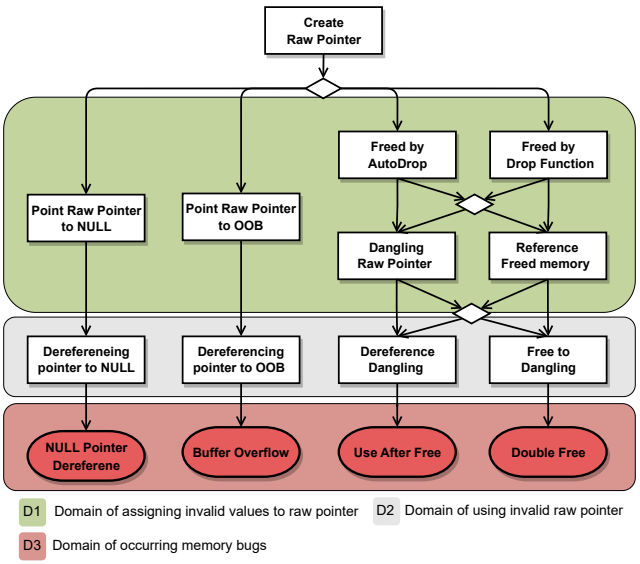

Figure 3: Rust memory bug patterns. Domain means the behaviors related to each domain description at the bottom of the figure.

```rust
// RUSTSEC-2019-0016: UAF due to a lifetime error

fn gen_Vec(tmp_str: String) -> Vec<u8> {
    let mut s = tmp_str;
    let ptr = s.as_mut_ptr(); // a raw pointer
    unsafe {
        let vec = Vec::from_raw_parts(ptr, s.len(), s.len());
        // Patched: mem::forget() increase lifetime
        // mem::forget(s);
        return vec;
    }
}

fn main() {
    let mesg = String::from("Hello");
    let mut x = gen_Vec(mesg);
    x[0] = '1' as u8; // use-after-free
}
```

Figure 4: An Example of UAF (RUSTSEC-2019-0016) in Rust *Isahc* crate

security policies. In most cases of spatial memory bugs (49 out of 61), Rust does not perform any checks on memory access through raw pointers, while other memory accesses through other pointer types (e.g., reference) are checked at compile time or runtime. All bugs not detected by Rust are related to incorrect memory access through these raw pointers. In the case of temporal memory bugs, if some objects are forcibly freed (i.e., break Rust's lifetime rules) through deallocation functions, Rust is unable to prevent (cannot detect these temporal memory bugs) accessing these freed objects through various types of pointers (e.g., references or raw pointers) in both Unsafe and Safe Rust. Therefore, to accurately detect temporal memory bugs, it is necessary to check raw pointers and all other pointers that have an aliasing relationship with raw pointers. This is because other aliased pointers can access objects freed by a raw pointer. Detailed examples and explanations are provided in the Use After Free part (in this subsection).

Based on our analysis of these real-world memory bugs, we summarize Rust memory bug patterns and illustrate these patterns. As shown in Figure 3, all memory bugs begin with the creation of a raw pointer. Following this, an invalid value (e.g., `NULL` value) is assigned to the raw pointer, and if this value is used inappropriately (e.g., dereferencing raw pointer memory), it occurs memory bugs. Detailed explanations for each memory bug type are as follows.

Buffer Overflow. A Buffer Overflow (BOF) bug occurs when a pointer dereferences out-of-bound of an allocated object. `rustc` prevents out-of-bound access by statically and dynamically checking memory access. However, as mentioned in §3.1, Rust does not perform any static or dynamic bound checks for the raw pointer's memory access. This means that the raw pointer can access the allocated memory object without any restrictions. Specifically, BOF bugs in Rust occur through the following three steps: (1) creating a raw pointer, (2) assigning an invalid address (pointing out-of-bound object) to the raw pointer, and (3) dereferencing the raw pointer pointing to out-of-bound. Therefore, to address these BOF bugs that are not covered by Rust, we need to check all raw pointer dereferencing, except any safe reference or smart pointer (safe references or smart pointers have metadata about their designated object and are subject to rigorous bound checks based on length and capacity).

NULL Pointer Dereference. A NULL Pointer Dereference (NPD) bug occurs when an application accesses memory through a pointer that points to a `NULL`. In Rust, only a raw pointer can have this `NULL` value through specific APIs (e.g., `ptr::null( )` or `ptr::null_unchecked( )`), while other safe references are not allowed [16] to have a `NULL` value. Therefore, the NPD bug can only occur by dereferencing a raw pointer that points to a `NULL` value. This NPD bug in Rust occurs through the following three steps: (1) creating a raw pointer, (2) assigning a `NULL` value to the raw pointer, and (3) dereferencing the raw pointer pointing to `NULL`. To detect NPD, we only need to check all dereferencing raw pointers that could cause NPD, as other pointers or references are not allowed to point to `NULL` by the Rust safety rule.

Use After Free. Use-After-Free (UAF) can occur when an object is deallocated either through automatic memory deallocation via Rust lifetime rule or manually calling drop function (e.g., `dealloc` or `libc::free()`), leading raw pointers, safe pointers (like references), or smart pointers to become dangling pointers. Dereferencing these dangling pointers can trigger UAF bugs. More specifically, Rust cannot guarantee that raw pointers always point to valid memory areas, allowing them to unrestrictedly point to objects already pointed by other pointers like references (i.e., breaking Rust's ownership rule). If this object is later deallocated through lifetime and automatic drop, the raw pointer becomes a

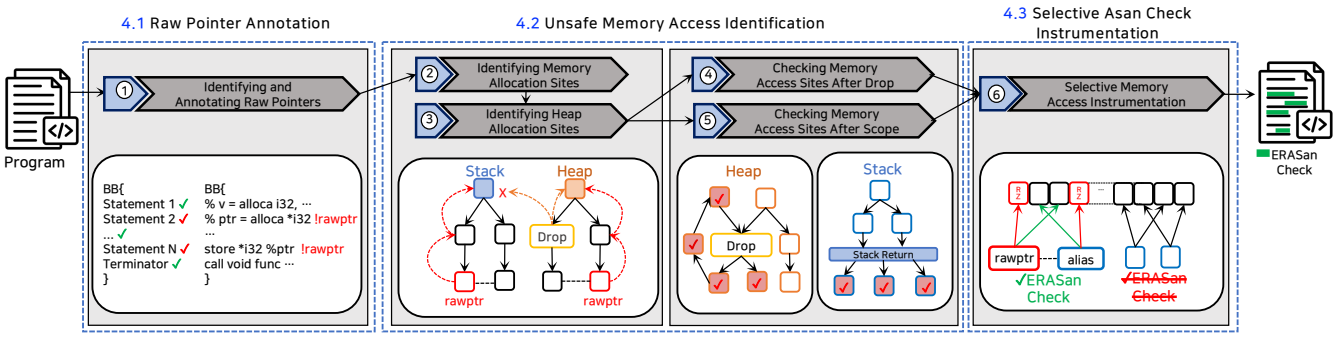

Figure 5: Overview of ERASAN. The small black boxes indicate the memory accesses and the small red boxes present the memory accesses by the raw pointer. The small boxes filled with red represent the memory accesses checked by ERASAN. The arrows indicate the direction of static analysis.

dangling state. Dereferencing this dangling pointer can lead to UAF bugs. Additionally, objects can be forcibly deallocated by the drop function (with raw pointer), and accessing these deallocated memories through references or smart pointers (now dangling pointers) in safe and unsafe areas can also cause UAF bugs.

For example, Figure 4 shows an example of a real-world UAF bug [32] caused by raw pointers. The `gen_Vec` function creates a stack object (named `s`) that includes a pointer to a `String` type heap object (line 4) and generates a raw pointer pointing to it (line 5). Subsequently, the `Vec::from_raw_parts` function is used to create a new vector (line 7). Internally, the address of `String s` saved in the raw pointer (`ptr`) is assigned to the pointer within the newly allocated vector (`vec`). Now, the pointer inside the `vec` vector can also point to `s`. When the `gen_Vec` function returns the vector `vec`, the `String` type object `s` is automatically deallocated according to Rust's lifetime rule. Following this, the returned vector `vec` is assigned to `x` in the `main` function. Then, memory access through `x` (line 17) within the safe Rust occurs UAF memory bugs. In principle, it should be protected by the Rust compiler. However, the use of raw pointers bypasses Rust's security checks, leading to a situation where access deallocated memory through `x`, which is perceived as safe and consequently leads to UAF.

As shown in Figure 3, triggering UAF in Rust requires the following three steps: (1) creating a raw pointer, (2) deallocating memory by automatic drop or calling drop function, and (3) dereferencing these dangling pointers or references. Note that while UAF can occur through incorrectly using raw pointers, dangling pointer deference (i.e., ASan's UAF checkpoint) can happen in various places (e.g., reference in safe Rust) during dangling pointer dereferencing. Therefore, it is important to check all locations where dangling pointers can be dereferenced.

Double Free. The overall procedure for the Double Free (DF) bug in Rust is quite similar to that of UAF. The main difference is attempting to deallocate it again after the target object is deallocated through a raw pointer or other pointers (e.g., reference). These DF bugs can occur through an additional free by manually calling the drop function or automatic drops when a reference pointing to the memory, freed by a raw pointer, reaches the end of its lifetime. Consequently, triggering DF bugs in Rust requires three steps: (1) creating a raw pointer, (2) deallocating memory by automatic drop or by calling the drop function, and (3) deallocating again freed memory by automatic drop or calling drop function.

4. ERASAN Design

ERASAN performs selective instrumentation to memory accesses in both safe and unsafe code areas, where Rust cannot guarantee safety, to significantly reduce the runtime overhead of ASan. The primary challenge is to identify as many safe memory access sites as possible to maximize performance gain during memory access validation check at runtime, while fully preserving ASan's memory bug detection capabilities. Figure 5 illustrates the overall architecture of ERASAN. ERASAN has the following three main components:

(i) Raw Pointer Annotation (§4.1): ERASAN performs type-matching analysis to identify every raw pointer at the MIR level and annotates the LLVM IR instructions. Since the raw pointer is a unique attribute that exists only up to the MIR level, ERASAN passes this information for later static analysis.

(ii) Unsafe Memory Access Identification (§4.2): ERASAN performs static analysis to identify potentially unsafe memory accesses. This is the core part of ERASAN, which identifies all potentially unsafe memory access sites according to our bug patterns (§3.2).

(iii) Selective ASan Check Instrumentation (§4.3): ERASAN only instruments memory access check instructions at potentially unsafe memory access sites identified in the previous identification phase.

4.1. Raw Pointer Annotation

ERASAN annotates all LLVM IR instructions corresponding to raw pointers. For this, since this information is only available at Rust's IR levels, such as HIR [13] and MIR [15] (not at the LLVM IR level), ERASAN transfers the raw

pointer information through annotations to the LLVM IR, which is translated from MIR.

To annotate raw pointers, ERASAN performs a type-matching analysis during the translation phase from MIR to LLVM IR. More specifically, the MIR represents a control-flow graph composed of basic blocks, each containing a series of statements ending with a terminator. To handle all these MIR instructions and find raw pointer instruction, ERASAN checks for raw pointer types within all statements and terminators of MIR. Since we need to annotate all codes related to raw pointers (e.g., dereferencing raw pointer, passing the raw pointer the arguments, or return values of function callee), ERASAN parses each statement and terminators of MIR to find raw pointer type information existing at a more granular level (e.g., place and rvalue). The newly generated LLVM IR code, annotated with raw pointer information, is utilized in the next points-to and value-flow analyses stage.

4.2. Unsafe Memory Access Sites Identification

ERASAN proposes two static analysis algorithms that can identify always-safe memory allocation sites and potentially unsafe memory access sites in the LLVM-IR. These algorithms performs as follows:

- Compute Raw Pointer Points-to Set. As mentioned in section §3.2, raw pointer can lead to memory bugs not only through direct dereferencing but also through all pointers that alias with them. To accurately identify all pointers in an alias relationship with raw pointers, ERASAN first needs to determine all object locations that these raw pointers can access. This initial step is crucial for the subsequent extraction of aliased pointers. For this purpose, ERASAN utilizes conservative points-to analysis to compute the *may-points-to* set for each raw pointer identified in the raw pointer annotation phase (§4.1).
- Differentiate Allocation Sites. ERASAN categorizes object allocation sites into stack and heap types. This differentiation is essential because the mechanisms of potential memory bugs in Rust vary depending on whether they occur from stack or heap object. To accurately differentiate between these memory allocation sites, ERASAN performs identifies object types at allocation sites (§4.2.1).
- Identify Unsafe Memory Access Sites. ERASAN extracts vulnerable memory access sites associated with pointers (alias pointers with raw pointers) computed by points-to analysis. ERASAN employs optimization techniques to selectively track only memory access sites that are actually vulnerable to memory bugs. This approach will be discussed in detail in section (§4.2.2).

4.2.1. Identify Object Type at Allocation Site. ERASAN conducts analysis (Algorithm 1 in §A) to determine the types of objects (i.e., stack, heap, or global) allocated at memory allocation sites accessible by raw pointers. This analysis mainly consists of two parts: (1) identifying all accessible memory allocation sites and (2) identifying heap allocation sites.

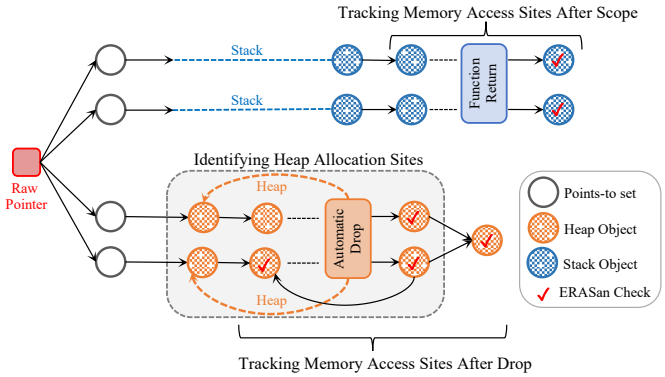

Figure 6: Identifying Heap Allocation Sites and Alias Pointer Dereference Sites

Identifying Memory Allocation Sites. ERASAN utilizes Andersen points-to analysis [27], available in SVF [51], in a context- and flow-insensitive mode to identify all the memory allocation sites that can be pointed to by raw pointers. Then, starting from each raw pointer, ERASAN iteratively traverses the statements backward, using the Sparse Value Flow Graph (SVFG) up to these memory allocation sites. In the SVFG, each node represents a pointer, and each edge represents the instructions' operations performed on these pointers. ERASAN is able to identify memory allocation sites during traversal if it encounters an `Addr` SVFG node that can represent these sites (e.g., `alloca` instruction).

Identifying Heap Allocation Sites. The key challenge is accurately classifying memory allocation sites as either stack or heap. Rearding heap variables, Rust uses wrapper functions such as `exchange_malloc()` for memory allocation [45], rather than directly invoking the `malloc` function. However, a significant challenge arises as substantial stack allocation sites (i.e., `alloca` instructions) are also linked to these heap wrapper functions. This makes it difficult to distinguish between stack and heap allocation sites accurately [30].

ERASAN distinguishes heap allocation sites based on Rust's drop traits. Unlike C/C++, which can allocate or free heap memory directly, Rust manages heap memory automatically to prevent memory bugs [28, 31, 45, 55]. Particularly, Rust's heap management scheme automatically deallocates heap objects when they go out of scope, effectively preventing the creation of dangling pointers, which are a common cause of UAF bugs. This management approach indicates that for all heap objects in Rust, automatic drop functions (i.e., `drop_in_place, box_free`) are integrated during the compilation process. Therefore, if a pointer flows into a drop function, it should be considered as associated with a heap allocation site. This approach can be more precise compared to analyzing Rust's heap wrapper functions (i.e., `exchange_malloc()`), as a large number of pointers, many of which are not related to heap allocation sites, flow into these wrappers and can obscure their association with

heap allocation. However, as drop functions are explicitly associated with heap allocation pointers, focusing on these functions simplifies the static analysis process and enhances accuracy by reducing complexity.

Based on this Rust's heap management scheme, ERASAN performs a top-down traversal of the SVFG, as shown in Figure 6, to identify heap allocation sites. ERASAN starts with the allocation sites identified from the points-to analysis, tracing the flow of pointers within the SVFG. It then examines if this pointer flows into Rust's drop functions. If a pointer is found to lead into a drop function, it is identified as a heap allocation site. Note that any allocation site not identified as heap is automatically classified as stack allocation site. For global, related UAF cannot occur, and all spatial memory safety bugs occur through raw pointers. ERASAN ensures that all memory accesses via raw pointers are checked, allowing for the detection of memory bugs related to all global objects, so global objects (and its allocation sites) are not considered separately.

4.2.2. Identify Unsafe Memory Access Sites. ERASAN performs using the memory allocation sites information computed in the previous step (Algorithm 1 in §A), focusing on tracking related memory access sites. Here, alias pointers refer to the pointers that have an aliasing relationship with any raw pointers. The analysis is done in two steps, as depicted in Figure 6.

Checking Memory Access Sites After Drop. As explained earlier in §3.2, Rust's safe memory accesses can cause UAF and DF if they alias with raw pointers. So, all alias pointers of raw pointers should be considered potentially unsafe. But from our Rust memory bug patterns, alias-pointers (safe references) of raw pointers are vulnerable only when they are used after drop functions. It means that alias-pointers' memory access before the drop function will be protected by Rust's safety rules. More precisely, temporal memory bugs do not occur, and spatial memory bugs are checked by Rust, except for memory access through a raw pointer. So, ERASAN only tracks alias-pointers after drop operations when it comes to heap objects.

To this end, ERASAN performs value-flow analysis starting from the heap memory allocation sites identified during the previous analysis (§4.2.1). This involves continuously searching the SVFG in a forward direction to find the heap's drop function. When a drop function is found, it searches for memory access operations (e.g., load and store) following the drop function and marks the corresponding instructions as heap alias pointer accesses. By doing so, ERASAN can reduce unnecessary overheads caused by checking safe memory accesses using alias pointers.

Checking Memory Access Sites After Scope. Similar to the case of heap, only a few alias pointers point to stack objects should be considered as potentially unsafe memory access sites. If out-of-scope access to the stack is performed in the safe Rust, memory bugs can be prevented through Rust's lifetime inference rules. However, if a raw pointer points to the stack memory area, the stack object is no longer subject to Rust's lifetime inference rules, and subsequent memory accesses in the safe Rust through an alias pointer can potentially lead to memory bugs.

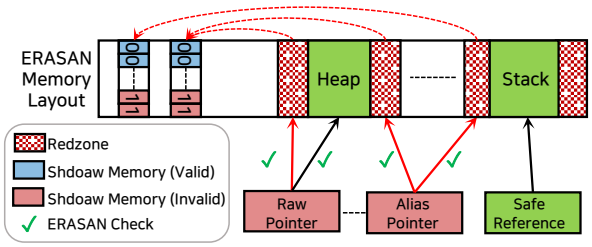

Figure 7: The memory layout of ERASAN with redzone and shadow memory

Based on this observation, ERASAN performs stack after scope access analysis to track all alias pointer dereferences after stack objects are cleaned up. It conducts value flow analysis in the same manner as heap after drop access (§4.2.2). However, unlike the heap, the stack in Rust does not incorporate an automatic drop function; instead, stack objects are cleaned up at the end of the function's scope. Reflecting on this, ERASAN assesses whether a stack object has been cleaned up by examining the lifetime of the function that contains the stack object.

4.3. Selective ASan Check Instrumentation

To check validity of memory access, ERASAN utilizes ASan's compile-time instrumentation and its runtime verification logic. Based on the set of potentially unsafe memory accesses identified through our MIR-level and LLVM-IR-level analysis (§4.1 and §4.2), ERASAN selectively inserts ASan's instrumentation. To enhance efficiency through selective instrumentation while perserving bug detection capabilities, ERASAN only removes instrumentation from safe memory access sites while keeping all the instrumentation that maintains ASan's full metadata in shadow memory. Figure 7 shows the memory layout of ERASAN with redzone and shadow memory. ERASAN places a redzone before and after all objects regardless of whether they are on the heap or stack, maintaining full metadata in shadow memory. On the other hand, ERASAN selectively instruments memory accesses, depending on unsafe memory access identification. For safe dereferences, ERASAN removes the sanitizer checks for higher performance.

We note that ASan's custom heap allocator, which inserts redzones between the allocated objects and initializes shadow memory for them, generates an overhead of 9.6%, according to a prior study [57]. To accomplish our goal of balancing detection performance and accuracy, ERASAN follows the same allocation scheme as ASan, aiming to preserve full detection capabilities of memory bugs (e.g., particularly from dereferencing random-value raw pointers). We take this minor overhead caused by metadata management in exchange for a broader vulnerability detection scope.

5. Implementation

In this section, we describe the main implementation details of three parts: modification of rustc to perform metadata annotation (§5.1), SVF [51] implementation of our static analysis (§5.2), and ERASAN's LLVM Pass (§5.3).

5.1. Annotating Raw Pointers

ERASAN automatically annotates all the created raw pointers with our own raw pointer metadata (i.e., `!rawptr`) using our type matching analysis for all the statements and terminators during code generation phase [6]. ERASAN not only checks raw pointer type matching but also checks raw pointer operations such as dereferencing or used as function arguments and return values. We implement the automatic identification of raw pointers by modifying the source codes of `codegen-ssa` and `codegen-llvm` within `rustc-1.64-nightly`. We automatically find all LLVM IR instructions related to raw pointers with our raw pointer metadata, with no additional manual effort.

Annotating Raw Pointers in Statements. The statement (i.e., `StatementKind` [22]) includes `assign` or `deinit` and consists of two components: `place` which indicates memory location, and `rvalue` which represents value production. The type information of `RawPtr` indicating raw pointer can be verified through the variable to indicate the type (e.g., `ty`) in both `place` and `rvalue`. In order to annotate all instructions related to raw pointer operations, we implement the analysis in code to transfer statements to LLVM IR instruction (i.e., `codegen-statement`), covering granular level information such as `projectElem` of `Deref` type to indicate dereferencing.

Annotating Raw Pointers in Terminators. A terminator is a statement with potentially multiple successors and is always at the end of a block in MIR. It is a `Terminatorkind` [23] that includes `Call`, `Assert`, and other various subtypes. It means that several structures such as `Call` (function calls like `as_mut_ptr()`) and `Assert` and each type has various sub-components containing `place` or `rvalue`. We implement our type analysis in `rustc` to transfer the terminator to instruction (i.e., `codegen-terminator`). To analyze terminators, ERASAN checks the type of all terminators and checks the subcomponents for the existence of the `RawPtr` type (e.g., call evaluated for `dest`, which is a function return value, and `args`, which is a function argument value). From type matching analysis for every terminator, ERASAN extracts the terminator containing information about `RawPtr` and injects metadata into the LLVM IR instructions generated from that terminator.

5.2. Identifying Unsafe Memory Access Sites

We implement an unsafe memory access sites identification algorithm based on the SVF-2.4 version, using context-insensitive and flow-insensitive Andersen's points-to analysis [27, 51]. It shows tolerable false positives when calculating unsafe memory access sites while satisfying the scalability that a sanitizer should have. The reason why ERASAN established a context-insensitive and flow-insensitive approach during the LLVM-IR analysis is that it should not miss any check instrumentation for vulnerable memory access sites.

Modifications to SVF. Rust programs frequently contain `InsertValueInst` and `ExtractValueInst` instructions, which are used to embed a value into a struct field or to retrieve a value from it. However, SVF currently does not support these instructions. Previously, when such instructions were encountered, they were processed as a black hole, and points-to analysis of such instructions is impossible. To escape these holes, we followed the same manner [30] for supporting LLVM instructions such as `InsertValueInst` and `ExtractValueInst` instructions to SVF.

5.3. Selective ASan Check Instrumentation

We implement the ERASAN pass based on ASan in LLVM 14.0.6 [2], instrumenting unsafe memory accesses only, as identified by ERASAN's static analysis. Memory access instructions are constantly checked by instrumentation to ensure that they access in-bounds, valid objects in memory. Also, ERASAN follows the same memory allocation instrumentation as ASan. This approach incurs an inevitable overhead for red zone insertions and shadow memory management. However, the impact of these overheads on a Rust program's performance is not significant (i.e., about 9.6% overhead described in §4.3). By maintaining full metadata in a shadow memory, we can fully protect all objects in memory from unexpected, unsafe raw pointer operations.

6. Evaluation

In this section, we evaluate ERASAN on five aspects: unnecessary check reduction (§6.2), runtime overhead (§6.3), compile time overhead (§6.4), bug detection capability (§6.5), and comparison with ASan-- (§6.6).

6.1. Evaluation Setup

Evaluation Environment. All evaluations are conducted on a machine equipped with a 10-core Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz, 64GB DDR4 RAM, and running Ubuntu 22.04.1 LTS (GNU/Linux 6.2.0). We conducted all experiments five times and reported the average number.

Baseline Configurations. To precisely evaluate the efficiency of each proposed technique of ERASAN, we set the baseline to ASan and implement three additional modes besides ERASAN: ERASAN-unsafe, ERASAN-rawptr, and ERASAN-thread. The ASan is a native address sanitizer (i.e., unmodified version). ERASAN-unsafe conducts an unsafe block-based static analysis approach (not raw pointer-based). This mode instruments every memory access reachable from unsafe blocks in the Rust program. It is because existing works such as XRUST [45] assume that only all unsafe related codes are vulnerable. For this approach, we

TABLE 3: The comparison of unnecessary check reduction and runtime overhead between ASan, ERASAN-unsafe, ERASAN-rawptr, and ERASAN against our selected benchmarks. Inst (#) presents the total llvm-ir instructions of the benchmark program. Check (#) indicates the number of instrumented instructions. The Over. (%) represents runtime overhead over native. Redu. (%) presents the reduction percentage compared to ASan.

Benchmark	ASan	ERASAN-unsafe		ERASAN-rawptr		ERASAN-thread		ERASAN	
	Check. (#)	Check. (#)	Redu. (%)	Check. (#)	Redu. (%)	Check. (#)	Redu. (%)	Check. (#)	Redu. (%)
syn	260,805	214,566	17.71	155,985	40.17	90,648	65.24	85,235	67.31
rand (generator)	3,729	1,623	56.47	656	82.41	6	99.83	6	99.83
rand (misc)	2,343	1,482	37.74	280	88.05	6	99.83	6	99.74
crossbeam	2,086	766	63.28	496	76.22	179	91.42	179	91.42
itoa	1,167	659	43.53	210	82.01	0	100.0	0	100.0
base64	55,049	46,466	15.59	25,385	53.88	6,878	87.51	5,918	89.25
regex	9,442	7,738	18.04	5,292	43.95	894	90.53	894	90.53
memchr	203,592	190,066	6.644	125,216	38.49	59,459	73.09	26,434	88.03
hashbrown	14,518	11,923	17.87	5,381	62.94	1,396	90.38	1,188	91.82
smallvec	2,244	1,505	32.93	499	77.76	36	98.39	36	98.39
ryu	2,676	2,228	16.74	422	84.23	238	91.11	46	98.28
semver	1,056	759	28.13	250	76.32	30	97.16	30	97.16
strsim	974	897	7.906	452	57.19	44	95.48	34	96.78
bytes (bytes)	4,456	3,511	21.21	419	90.59	133	97.02	133	97.02
bytes (buf)	7,478	2,947	60.59	235	96.86	47	99.37	47	99.37
bytes-mut	5,095	4,138	18.78	722	85.83	85	98.33	85	98.33
indexmap	76,041	66,636	12.37	49,616	34.75	25,168	66.90	25,168	66.90
byteorder	5,242	2,126	59.44	906	82.72	49	99.07	49	99.07
num-integer	22,622	18,866	16.61	13,068	42.23	685	96.97	583	97.42
url	69,188	55,328	20.03	43,034	37.80	26,715	61.39	24,611	64.43
uuid (parse)	109,942	97,123	11.66	67,160	38.91	35,643	67.59	31,898	70.99
uuid (format)	110,558	97,637	11.68	67,141	39.27	35,657	67.75	31,926	71.12
unicode	1,762	1,700	3.518	1,162	34.05	41	97.67	41	97.67
Average	43,022	36,116	26.30% ↓	24,521	62.95% ↓	12,349	88.34% ↓	10,197	90.03% ↓

(a) Unnecessary Check Reduction

Benchmark	ASan	ERASAN-unsafe		ERASAN-rawptr		ERASAN-thread		ERASAN	
	Over. (%)	Over. (%)	Redu. (%)	Over. (%)	Redu. (%)	Over. (%)	Redu. (%)	Over. (%)	Redu. (%)
syn	471.14	407.02	64.12	328.40	142.74	302.58	168.56	200.26	270.88
rand (generator)	214.96	78.50	136.46	67.34	147.63	7.65	207.32	7.65	207.31
rand (misc)	58.93	37.22	21.71	10.96	47.97	0.21	58.73	0.21	58.73
crossbeam	45.71	12.35	33.36	2.49	43.22	0.39	45.31	0.39	45.31
itoa	660.27	94.69	565.57	31.76	628.51	21.04	639.22	21.04	639.23
base64	396.47	301.31	95.16	275.45	121.02	230.13	166.33	196.28	200.18
regex	530.32	476.63	53.69	449.37	80.96	306.19	224.13	306.19	224.13
memchr	458.05	225.06	232.99	51.73	406.32	44.42	413.63	31.47	426.58
hashbrown	360.73	332.77	27.96	280.97	79.76	73.01	287.73	65.85	294.87
smallvec	490.26	434.75	55.51	401.39	88.86	354.54	135.72	354.54	135.72
ryu	252.81	197.87	54.94	124.42	128.39	50.15	202.66	46.16	206.65
semver	583.76	555.95	27.81	397.19	186.56	252.88	330.86	252.88	330.87
strsim	275.73	262.22	13.51	97.42	178.32	38.87	330.88	34.39	241.33
bytes (bytes)	150.87	133.02	17.88	73.85	77.05	49.82	101.08	49.82	101.08
bytes (buf)	327.41	63.11	264.28	61.09	266.31	59.39	268.01	59.39	268.01
bytes-mut	134.96	106.59	28.37	98.54	36.42	55.95	79.02	59.95	79.02
indexmap	378.12	331.07	47.04	320.51	57.61	143.77	234.34	143.77	234.34
byteorder	330.15	72.80	257.34	54.59	275.56	15.22	314.93	15.22	314.93
num-integer	173.48	147.36	26.12	84.34	89.13	29.97	143.51	1.05	172.43
url	348.21	337.69	10.52	332.44	15.77	264.74	83.48	221.91	126.29
uuid (parse)	531.49	499.86	31.63	53.33	478.16	46.16	485.33	40.42	491.08
uuid (format)	242.72	170.05	72.69	105.47	137.25	67.31	175.40	55.47	187.25
unicode	288.13	287.89	0.25	253.96	34.18	46.32	241.82	46.32	241.82
Average	334.98%	241.99%	92.99% ↓	172.04%	162.94%↓	106.86%	228.12%↓	95.94%	239.05%↓

(b) Runtime Overhead

insert unsafe block metadata at MIR code and perform reachability analysis to insert ASan's checks in all blocks that are reachable from these unsafe blocks. ERASAN-rawptr checks all memory accesses through all raw pointers and their aliased pointers. This mode turns off our optimization approaches, which distinguish between stack and heap objects for safe check elimination (tracking-after-drop optimization). Note that ERASAN cannot properly support multi-threaded applications with this tracking-after-drop optimization since the order of allocation and deallocation in codes executed by threads cannot be guaranteed. To support multi-threaded applications, we introduce ERASAN-rawptr mode, which checks all problematic memory accesses without ERASAN's tracking-after-drop optimizations. However, this ERASAN-rawptr mode does not provide the best runtime performance compared to the ERASAN mode. We also introduce ERASAN-thread mode that only applies tracking-after-drop optimization to functions unrelated to threads. In this ERASAN-thread mode, since codes executed in threads have `closure` symbols (LLVM IR level), this mode identified these codes marked with `closure` symbols and did not apply ERASAN's tracking-after-drop optimization to these functions.

Evaluation Target Collection and Selection. For our evaluation, we utilize the 23 benchmark tests from real-world crates in crates.io [7], the Rust library repository. To ensure fair selection, we select libraries that support benchmark or unit tests after sorting all crates according to their all-time downloads and stars since no widely accepted benchmark is available for Rust, unlike SPEC CPU2017 [21] for C/C++. However, we exclude some crates if the target crate does not include benchmarks or provide a reasonable test workload (e.g., conducts very short and few tests). All selected crates and benchmarks are well-maintained and updated frequently in their GitHub repository. To evaluate the effectiveness of ERASAN, we use these benchmarks to measure unneeded check reduction, the runtime overhead after eliminating checks, compile-time overhead during static analysis, and comparison with existing address sanitizer optimization approaches.

6.2. Unnecessary Check Reduction

We evaluate how ERASAN effectively removes the ASan checks using all of our 23 target benchmarks. This evaluation measures the number of removed checks at compile time (not runtime). As shown in Table 3-(a), the full ERASAN eliminates 90.03% of sanitizer checks, while ERASAN-unsafe removes 26.30% of them, ERASAN-rawptr 62.95%, and ERASAN-thread 88.34%. Notably, ERASAN-unsafe, unsafe Rust-based static analysis approach overapproximates unsafe memory accesses. ERASAN achieves around 3.54 times fewer checks than ERASAN-unsafe (10,197 vs 36,116), which demonstrates the importance of precise raw pointer-based static analysis. When comparing ERASAN with ERASAN-rawptr, we find that ERASAN's optimization that tracks memory access sites after drop and scope heap to find safe accesses can remove 2.40 times more unnecessary checks (ERASAN's 10,197 compared to ERASAN-rawptr's 24,521). In particular, ERASAN removes all of the instrumented checks in several benchmark test suites (e.g., `itoa` and `smallvec`) since these benchmarks do not have raw pointers after drops inside their test set, meaning that these programs cannot have any temporal memory bugs at runtime. ERASAN removes 1.21 times more checks than ERASAN-thread does (ERASAN's 10,197 vs ERASAN-thread's 12,349) due to relatively few codes executing in threads.

6.3. Runtime Overhead

We now evaluate ERASAN's runtime overhead reduction due to reduced ASan's check instrumentation. In this evaluation, we repeatedly run each benchmark five times with the same workload provided by each benchmark and calculate the average execution time. Table 3-(b) compares the performance overhead of ERASAN with others. On average, after eliminating unnecessary check instrumentation, the performance overhead is reduced by 239.05% for ERASAN, 92.99% for ERASAN-unsafe, 162.94% for ERASAN-rawptr, and 228.12% for ERASAN-thread, compared to ASan's overhead. Thus, ERASAN is 3.49 times more efficient than ASan, 2.52 times more than ERASAN-unsafe, 1.79 times more than ERASAN-rawptr, and 1.11 times more than ERASAN-thread. The performance overhead of ERASAN can be reduced by up to 639.23% (for `itoa`), which is the highest checking overhead reduction case since ERASAN eliminates all unnecessary checks. Additionally, in this case, the measured maximum overhead of ASan is small (21.04%), and ERASAN's overhead has already reached 0%, making it impossible to reduce further.

ERASAN achieves 146.05% better performance improvement over ERASAN-unsafe (ERASAN's 95.94% vs. ERASAN-unsafe's 241.99%). To understand this performance gap, we investigate the impact of an unsafe block-based static analysis approach. For example, in the case of `unicode`, ERASAN-unsafe treats 98.9% (1,700 out of 1,762) of all memory access as reachable from unsafe blocks. In contrast, ERASAN traces only 41 memory accesses through raw pointers (and their aliases). These results reveal that unsafe block-based approaches can generally lead to over-approximate results and, therefore, high runtime checking overheads. In contrast, our raw pointer-based approach allows more precise and efficient memory access checks. As a result, ERASAN successfully eliminates sanitizer checks and improves the performance than ERASAN-unsafe.

We additionally study the effectiveness of our optimization techniques. ERASAN also shows 76.10% improvement over ERASAN-rawptr (ERASAN's 95.94% vs. ERASAN-rawptr's 172.04%). The `hashbrown` benchmark test leads to the most significant difference (around 215.12%) between these two modes; ERASAN can eliminate the 4,016 out of 5,147 checks through our optimization that sanitizes memory accesses considering the object stack and heap, reducing runtime cost. Therefore, ERASAN successfully removes safe heap and stack memory access checks that cannot produce any temporal memory bugs, thereby reducing run time performance overhead.

ERASAN only shows a 10.93% performance improvement over ERASAN-thread (ERASAN's 239.05% overhead reduction vs. ERASAN-thread's 228.12%). This is because the features of ERASAN and ERASAN-thread are almost identical; ERASAN-thread does not apply tracking-after-drop optimization codes executed by threads, which is a relatively small amount of code. However, in the `syn` benchmark test, ERASAN-thread generates 102.32% more overhead than ERASAN, which invokes the APIs (e.g., `thread::spawn` for creating a new thread) for building a multi-threaded program 953 times. For programs that are not threaded (e.g., `itoa` and `semver`), both ERASAN and ERASAN-thread have the same performance improvement. These results show that ERASAN-thread generates more overhead in multi-threaded programs to check all the unsafe memory accesses found in codes executed by threads.

TABLE 4: Comparison between three Native, ASan, and ERASAN compile-time overhead during benchmark tests. The Line (#) indicates the number of source code lines. The Time (µs) represents compile-time to micro-seconds. The Incre. (X) indicate the degree of increased compile-time compared to Native.

Benchmark	Program	Native	ASan		ERASAN	
	Line (#)	Time (µs)	Time (µs)	Incre. (X)	Time (µs)	Incre. (X)
syn	234,795	86,141	112,300	×1.30	641,181	×7.44
rand (generator)	20,859	1,393	2,058	×1.47	2,509	×1.80
rand (misc)	20,972	1,294	1,791	×1.38	4,041	×3.12
crossbeam	20,980	1,475	1,541	×1.04	4,775	×3.23
itoa	479	577	610	×1.05	976	×2.03
base64	7,120	9,716	27,305	×2.81	135,143	×13.81
regex	75,676	3,915	6,848	×1.74	25,769	×6.58
memchr	63,631	68,265	130,127	×1.91	302,887	×4.44
hashbrown	15,790	2,609	4,831	×1.85	9,203	×3.53
smallvec	3,688	852	1,361	×1.59	1,629	×1.91
ryu	3,930	917	1,478	×1.61	1,859	×2.02
semver	3,188	475	546	×1.14	861	×1.81
strsim	1,141	521	707	×1.36	1,004	×1.93
bytes (bytes)	9,936	1,432	2,180	×1.52	5,170	×3.61
bytes (buf)	10,082	1,173	2,123	×1.81	5,511	×4.69
bytes-mut	10,002	1,394	2,199	×1.58	5,239	×3.75
indexmap	11,166	19,441	40,387	×2.08	94,720	×4.87
byteorder	5,845	1,523	2,468	×1.62	3,309	×2.17
num-integer	2,939	5,130	8,321	×1.62	15,280	×2.98
url	26,864	11,917	26,016	×2.18	63.487	×5.32
uuid (parse)	7,745	18,634	39,103	×2.09	102,997	×5.53
uuid (format)	7,726	20,583	55,602	×2.70	114,980	×5.58
unicode	158,888	945	1,205	×1.28	1,430	×1.51
Average	31,454	9,579	10,527	×1.72 ↑	53,801	×4.07 ↑

ERASAN's effectiveness varies depending on each benchmark test. For example, in the case of `bytes (buf)`, ERASAN eliminates 38.78% more checks than ERASAN-unsafe does. However, ERASAN only reduces the 3.72% runtime overhead since only a small fraction of these checks are actually executed during runtime. In this case, we can further reduce the overhead by removing checks in hot paths, i.e., highly executed and, therefore, well-tested code paths [42, 54]. However, even if all or most of ASan's memory access checks were removed, runtime overheads could persist because of other sources of ASan's overheads, such as shadow memory initialization and teardown or memory poisoning. For instance, in the case of `smallvec`, even after removing 98.39% of checks, considerable overhead (354.54%) remains. This is mainly due to the overhead of poisoning objects and the shadow memory initialization and teardown at the beginning and end of the program's execution. While such overhead is typically a one-time cost and may not significantly impact long-running benchmarks, it may have a large impact on benchmarks performing repetitive short tests [39].

6.4. Compile-time Overhead

Since ERASAN proposes and utilizes several static analysis techniques, we measure the compile-time overhead incurred

TABLE 5: Detection capability of ASan and ERASAN on memory vulnerabilities from the *RustSec Advisory Database*. In cases where bug reports had both IDs present, we represented them using the RUSTSEC ID.

RUSTSEC/CVE ID	Crate	Bug Type	ASan	ERASAN
RUSTSEC-2023-0005	tokio	UAF	✔	✔
RUSTSEC-2022-0070	secp	UAF	✔	✔
RUSTSEC-2022-0078	bumpalo	UAF	✔	✔
RUSTSEC-2021-0018	qwutils	DF	✔	✔
RUSTSEC-2021-0028	toodee	DF	✔	✔
RUSTSEC-2021-0031	nano_arena	UAF	✔	✔
RUSTSEC-2021-0033	stack-dst	DF	✔	✔
RUSTSEC-2021-0047	slice-deque	DF	✔	✔
RUSTSEC-2021-0130	lru	UAF	✔	✔
RUSTSEC-2021-0128	rusqlite	UAF	✔	✔
RUSTSEC-2021-0094	rdiff	Heap Ovf.	✔	✔
RUSTSEC-2021-0053	algorithmica	DF	✔	✔
RUSTSEC-2021-0049	through	DF	✔	✔
RUSTSEC-2021-0048	stackvector	Stack Ovf.	✔	✔
RUSTSEC-2021-0042	insert_many	DF	✔	✔
RUSTSEC-2021-0039	endian_trait	DF	✔	✔
RUSTSEC-2021-0003	smallvec	Heap Ovf.	✔	✔
RUSTSEC-2020-0167	pnet_packet	Heap Ovf.	✔	✔
RUSTSEC-2020-0097	xcb	UAF	✔	✔
RUSTSEC-2020-0091	arc-swap	UAF	✔	✔
RUSTSEC-2020-0061	futures	NPD	✔	✔
RUSTSEC-2020-0060	futures	UAF	✔	✔
RUSTSEC-2020-0039	simple-slab	Heap Ovf.	✔	✔
RUSTSEC-2020-0038	ordnung	DF	✔	✔
RUSTSEC-2020-0005	cbox	UAF	✔	✔
RUSTSEC-2019-0023	string-interner	UAF	✔	✔
RUSTSEC-2019-0009	smallvec	DF	✔	✔
RUSTSEC-2019-0034	http	DF	✔	✔

by ERASAN's static analysis for each benchmark. Table 4 shows the compile-time overhead of Rust programs when using ERASAN compared to native (i.e., compile time without any sanitizer). ERASAN spends, on average, 4.07x more time building the program than the native, while ASan requires 1.72x more time. This is because ERASAN utilizes static analysis to track all raw pointers, to perform point-to and value-flow analysis, and to identify allocation types traversing the SVFG. Such cost can vary (e.g., from 13.81 to 1.51 times) depending on the size and complexity of the application. However, the static analysis of ERASAN is a one-time cost overhead that occurs only at compile time and makes runtime execution faster by removing sanitizer checks.

6.5. Bug Detection Capability

ERASAN detects memory bugs based on the memory safety violation patterns outlined in §3.2. However, if there are corner cases for this pattern, ERASAN might produce false negatives. To validate our design and implementation, we evaluate ERASAN against real-world memory bugs, ensuring that ERASAN precisely detects various real-world memory bugs without any false negatives. Due to the absence of a Rust-specific dataset measuring bug detection capabilities comparable to the Juliet Test Suite [47] for C/C++, we created our own test set that can reliably reproduce memory bugs of different patterns. For this, we review all memory bug reports in the RustSec Advisory Database [20], selecting those that are reproducible (e.g., accompanied by a PoC). Also, we select reports detectable by ASan and undetectable by the Rust runtime. Cases where both ASan and the Rust runtime detected violations are excluded. We focus on the RustSec database, as it covers all Rust-related CVEs. In the end, we created a set of 28 bug detection test cases that can only be detected with ASan and not Rust. Table 5 presents ERASAN's detection capabilities. ERASAN successfully detects all memory bugs in the 28 test cases, confirming that these 28 real-world memory bugs are consistent with our predefined memory bug patterns and that our optimization techniques keep the detection capability of ASan.

TABLE 6: The comparison of removed check between ASan-- and ERASAN against our evaluation program set. Both represents the result of checks removed by both ASan-- and ERASAN. Unremoved indicates the number of checks that are not removed by either. Remv (#) indicates the number of removed checks by each mode. Redu (%) presents the removed percentage compared to ASan.

Benchmark	Only ASan--		Only ERASAN		Both		Unremoved	
	Remv. (#)	Redu. (%)	Remv. (#)	Redu. (%)	Remv. (#)	Redu. (%)	Remv. (#)	Redu. (%)
syn	27,038	10.37	118,387	45.39	57,183	21.93	58,197	22.31
rand (generator)	0	0	2,320	62.22	1,403	37.62	6	0.16
rand (misc)	0	0	1,247	53.22	1,090	46.52	6	0.25
crossbeam	40	1.92	1,334	63.95	573	27.47	139	6.66
itoa	0	0	839	71.89	328	28.11	0	0
base64	1,469	2.67	34,855	63.32	14,276	25.93	4,449	8.08
regex	252	2.67	6,663	70.57	1,885	19.96	642	6.79
memchr	18,785	8.49	108,703	49.18	57,598	26.06	35,932	16.26
hashbrown	305	2.10	7,862	54.14	5,471	37.68	883	6.08
smallvec	0	0	1,490	66.39	718	31.99	36	1.61
ryu	9	0.33	1,755	65.58	875	32.69	37	1.38
semver	17	1.61	649	61.46	377	35.70	13	1.23
strsim	4	0.41	611	62.73	329	33.78	30	3.08
bytes (bytes)	45	1.01	2,256	50.63	2,067	46.39	88	1.97
bytes (buf)	15	0.20	3,441	46.01	3,990	53.36	32	0.43
bytes-mut	27	0.52	2,680	52.60	2,330	45.73	58	1.14
indexmap	10,164	13.37	29,426	38.69	21,447	28.20	15,004	19.73
byteorder	11	0.21	3,482	66.43	1,711	32.64	38	0.72
num-integer	197	0.87	14,875	65.75	7,164	31.67	386	1.71
url	8,360	12.08	29,028	41.96	15,550	22.47	16,250	23.49
uuid (parse)	9,711	8.83	50,890	46.28	27,167	24.71	22,287	20.17
uuid (format)	9,712	8.78	51,161	46.27	27,471	24.85	22,214	20.09
unicode	6	0.34	1,120	63.56	601	34.11	35	1.98
Average	3,746	3.34%	20,655	56.88%	10,939	32.59%	7,680	7.19%

6.6. Comparison with ASan--

We evaluate how ERASAN efficiently removes the ASan checks compared to state-of-the-art, ASan-- [57]. For this comparison, we port ASan-- to the LLVM version used by ERASAN (from 12.0.0 to 14.0.6) since ASan-- is implemented based on LLVM 12.0.0. As shown in Table 6, the percentage of checks removed by both ASan-- and ERASAN from ASan's instrumentation is 32.59%. Moreover, ERASAN exclusively removes 56.88% of checks, while ASan-- removes only 3.34%. This difference originates from the fact that ASan-- can remove redundant or adjacent sanitizer checks, while ERASAN can remove checks that become redundant (or unneeded) when considering Rust safety rules. Consequently, these results show that ERASAN is an efficient sanitizer optimized for Rust, capable of safely

removing a significant number of checks that even ASan-- cannot remove. Additionally, since these two approaches are orthogonal, ERASAN can further eliminate 3.34% more unnecessary checks by applying ASan--'s optimization.

7. Discussion

Remaining ASan Overhead Reduction. Although most of ASan's overhead originates [57] from memory access checks and ERASAN removes these unnecessary checks, ERASAN does not completely reduce all ASan's overhead. To further reduce ASan's remaining overhead, we can avoid inserting redzones when objects are allocated. However, if some objects' redzones are removed and attackers illegally access these objects' surrounding areas through manipulated pointers, ASan cannot detect such attacks due to removed redzones. Therefore, we do not attempt to remove ASan's redzone insertion overhead. Moreover, when ASan is applied to repeated and short execution test environments (e.g., fuzzing), ASan incurs significant initialization and teardown overheads mainly due to ASan's heavy shadow memory. To mitigate this, we can utilize the existing FuZZan [39] approach to optimize ASan's data structure instead of heavy shadow memory to reduce these heavy initialization and teardown overheads.

8. Releated Work

Our study relates to existing works on protecting Rust programs against memory bugs and improving ASan performance by removing unnecessary sanitizer checks.

Memory Bug Detection for Rust. Several static analysis-based approaches [29, 31, 38, 43, 44] that can detect memory bugs in Rust have been proposed. Rudra [29] defines and detects three different types of memory bugs (e.g., panic safety bug, higher-order invariant bug, and send/sync variance bug). Then, Rudra conducts the static analysis to detect these bugs by utilizing their unsafe data-flow checker and send/sync variance checker. Rupair [38] conducts data-flow analysis to cover buffer overflow by tracking the relation between the definition and the usage site. SafeDrop [31] performs data-flow analysis to detect temporal memory bugs by using their modified Tarjan algorithm. In the case of MIRChecker [43], it aims to detect memory bugs related to runtime panics and lifetime corruption by utilizing static analysis and constraint-solving techniques. However, these static analysis-based approaches suffer from over-approximation analysis results that can lead to false positives. Additionally, these approaches have limited bug detection coverage, as they are generally designed to detect only specific predefined memory bug types. In contrast, ERASAN can detect various types of memory bugs, as it maintains the same bug detection capability as ASan. Additionally, unlike other static analyses, ERASAN leverages more precise information available at runtime for memory bug detection, resulting in significantly low false positives like Asan.

The isolation approaches [30, 40, 41, 45] generally focus on separating unsafe Rust from safe Rust to protect safe Rust. XRust [45] proposes a new memory allocator that splits memory into unsafe and safe regions and proposes efficient reference checking approaches between regions. Both TRUST [30] and PKRUsafe [40] utilize Intel MPK [48] hardware features to more effectively separate safe and unsafe Rust. Sandcrust [41] isolates Rust from foreign languages (e.g., C/C++) that cannot ensure safety. However, these approaches are unable to fundamentally remove memory bugs because they are isolation (not bug detection) approaches. Since memory bugs that occur even within unsafe areas can affect (e.g., control flow hijacking attack) the entire program operations, it is important to detect and remove memory bugs using bug detection approaches like ERASAN.

Dynamic analysis approaches [11, 49] are widely used to detect memory bugs. The MIRI interpreter [11] for Rust's mid-level intermediate representation (MIR) is designed to detect memory bugs. Address Sanitizer (ASan) [49] is the most widely used sanitizer [50] to detect memory bugs. ASan consists of a compiler instrumentation module to insert checking instrumentation and a runtime module to detect bugs during runtime. ASan can easily be applied to a target Rust program since ASan is integrated into the Rust compiler [24] and can detect various types of memory bugs. However, MIRI, an interpreter-based approach, suffers from high overhead and limited detection coverage. ASan also causes high overhead mainly due to unnecessary checks in safe memory access guaranteed by ERASAN. However, ERASAN significantly reduces ASan's overhead by eliminating these unnecessary checks while maintaining ASan's strengths, such as low false positives and broad detection coverage.

Address Sanitizer Optimization. Several existing approaches [54, 56, 57] have been proposed to reduce ASan's performance overhead through various strategies that eliminate unnecessary sanitizer checks. ASAP [54] profiles the program to identify "hot" code paths that are executed more frequently than others. ASAP eliminates checks in this hot code, as they are sufficiently tested and expensive, and maintains check codes in the "cold" code paths, which are executed less often and relatively less tested. SANRAZOR [56] also profiles the program, but SANRAZOR aims to identify and remove redundant checks. ASan-- [57] proposes several optimization approaches that can identify optimizable ASan checks, such as checks in loops and neighboring checks. However, existing works have not considered unnecessary checks for memory accesses in safe regions of Rust programs. ERASAN is optimized for the Rust environment based on our careful analysis of Rust safety rules. We also note that ERASAN can complement these works to reduce ASan's performance overhead further.

9. Conclusion

Rust programs frequently have memory bugs due to improper uses of unsafe Rust (more precisely, raw pointers). Although several approaches have been proposed to address this, these approaches suffer from high overhead and limited detection coverage issues. This paper proposes an efficient address

sanitizer design tailored for Rust, ERASAN, which has the same bug detection capability as ASan, yet is more efficient than prior work. For this, ERASAN only instruments memory accesses in both safe and unsafe code areas whose safety cannot be guaranteed by Rust. Our evaluation shows that ERASAN removes an average of 90.03% of existing ASan checks and significantly reduces ASan performance overhead by an average of 239.05% while showing the same bug-finding ability as ASan. The open-source version of ERASAN is available at https://github.com/S2-Lab/ERASan.

10. Acknowledgement

This work was supported by a Korea Internet & Security Agency (KISA) grant funded by the Korean government (PIPC) (No. 1781000003). This work was also supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-2021-0-01817) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation). We gratefully acknowledge their support.

References

[1] Address Sanitizer Algorithm. https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm.

[2] Address Sanitizer Pass. https://github.com/llvm/llvm-project/blob/llvmorg-14.0.6/llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp.

[3] Android Rust Introduction. https://source.android.com/docs/setup/build/rust/building-rust-modules/overview.

[4] Borrow Check Rule. https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html.

[5] Bound Check Rule. https://nnethercote.github.io/perf-book/bounds-checks.html.

[6] Code generation. https://rustc-dev-guide.rust-lang.org/backend/codegen.html.

[7] crate.io. https://crates.io/.

[8] Google joins the Rust Foundation. https://source.android.com/docs/setup/build/rust/building-rust-modules/overview.

[9] Introduction of Rust Programming Language. https://doc.rust-lang.org/book/ch00-00-introduction.html.

[10] Microsoft is rewriting core Windows libraries in Rust. https://www.theregister.com/2023/04/27/microsoft-windows-rust/.

[11] MIRI (MIR Interpreter). https://github.com/rust-lang/miri.

[12] Rust compiler development guide - sanitizers support. https://rustc-dev-guide.rust-lang.org/sanitizers.html.

[13] Rust high-level intermediate representation. https://rustc-dev-guide.rust-lang.org/hir.html.

[14] Rust Lifetime Rule. https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html.

[15] Rust MIR. https://rustc-dev-guide.rust-lang.org/mir/index.html.

[16] RUST NULL Value. https://doc.rust-lang.org/std/ptr/fn.null.html.

[17] RUST Ownership Rule. https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[18] Rust sanitizer guideline. https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html.

[19] Rust within the Linux Kernel. https://www.kernel.org/doc/html/next/rust/index.html.

[20] RustSec. https://rustsec.org.

[21] SPEC CPU 2017. https://www.spec.org/cpu2017/.

[22] StatementKind Type. https://doc.rust-lang.org/beta/nightlyrustc/rustc-middle/mir/syntax/enum.StatementKind.html.

[23] TerminatorKind. https://doc.rust-lang.org/beta/nightly-rustc/rustc-middle/mir/enum.TerminatorKind.html.

[24] The Description of RUST Compiler. https://github.com/rust-lang/rust.

[25] Unsafe Destructors. https://doc.rust-lang.org/nomicon/destructors.html.

[26] Unsafe Rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

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

[28] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[29] 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*, pages 84–99, 2021.

[30] 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 *32nd USENIX Security Symposium (USENIX Security 23). Baltimore, MD: USENIX Association*, 2023.

[31] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *arXiv preprint arXiv:2103.15420*, 2021.

[32] RustSec Advisory Database. Rustsec-2019-0016. https://rustsec.org/advisories/RUSTSEC-2019-0016.html.

[33] RustSec Advisory Database. Rustsec-2020-0097. https://rustsec.org/advisories/RUSTSEC-2020-0097.html.

[34] Dmitry Vyukov. Address/thread/memorysanitizer slaughtering c++ bugs. https://www.slideshare.net/sermp/sanitizer-cppcon-russia.

[35] Dmitry Vyukov. Syzbot. https://syzkaller.appspot.com/upstream.

[36] Google. Address sanitizer found bugs. https://github.com/google/sanitizers/wiki/AddressSanitizerFoundBugs.

[37] Google. Kernel address sanitizer (kasan), a fast memory error detector for the linux kernel. https://github.com/google/kasan/wiki.

[38] 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*, pages 812–823, 2021.

[39] 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.

[40] 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*, pages 132–148, 2022.

[41] 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*, pages 51–57, 2017.

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

[43] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Mirchecker: detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2183–2196, 2021.

[44] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Detecting cross-language memory management issues in rust. In *European Symposium on Research in Computer Security*, pages 680–700. Springer, 2022.

[45] 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.

[46] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[47] Nist. Software assurance reference dataset. https://samate.nist.gov/SARD/test-suites.

[48] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.

[49] 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.

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

[51] Yulei Sui and Jingling Xue. Svf: Interprocedrual static value-flow analysis in llvm. In *In Proceedings of the 25th International Conference on Compiler Construction, CC 2016*, pages 265–266. New York, NY, USA, 2016. Association for Computing Machinery, 2016.

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

[53] Vlad Tsyrklevich. Gwp-asan: Sampling heap memory error detection in-the-wild, 2021.

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

[55] 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.

[56] 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.

[57] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *Usenix Security Symposium*, 2022.

Appendix A.
Algorithm of ERASAN

Algorithm 1: The core algorithm of ERASAN

```
Input  : I, S_g
I: a given LLVM-IR instruction set;
S_g: a sparse value flow graph, SVFG;
A_h ⇐ a set of heap alloc sites accessible by rawptr;
A_s ⇐ a set of stack alloc sites accessible by rawptr;
A_r ⇐ a set of rawptr memory alloc sites;
1. Collects all instructions annotated with rawptr metadata
forall inst ∈ I do
    if inst.hasMetaData("!rawptr") then
        A_r ⇐ PointsToAnalysis(inst, S_g);
2. Perform Raw Pointer Allocation Sites Analysis
S_s ⇐ ∅; // a set of stack SVFGNode
S_h ⇐ ∅; // a set of heap SVFGNode
forall a ∈ A_r do
    if S_s.contains(a) then
        A_s.insert(a);
        return;
    else if S_h.contains(a) then
        A_h.insert(a);
        return;
    else
        if is_drop_traits(a, S_g) then
            A_s.insert(a);
            S_h.update(a.getDFSPath);
        else
            A_h.insert(a);
            S_s.update(a.getAllVisitedPaths);
3. Perform Raw Pointer Access Sites Analysis
3-1. Heap : Tracking Memory Access Sites After Drop
forall a_h ∈ A_h do
    S_h ← ValueFlowAnalysis(a_h, S_g);
    forall s_h ∈ S_h do
        if isAccessSite(s_h) and is_after_drop(s_h) then
            inst ← s_h.getLLVMInst();
            inst.setMetadata();
3-2. Stack : Tracking Memory Access Sites After Scope
forall a_s ∈ A_s do
    S_s ← ValueFlowAnalysis(a_s, S_g);
    forall s_s ∈ S_s do
        if isAccessSite(s_s) and is_after_scope(s_s) then
            inst ← s_s.getLLVMInst();
            inst.setMetadata();
```

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper introduces ERASAN, an efficient address sanitizer specifically designed for the Rust programming language. ERASAN aims to reduce unnecessary memory access checks by leveraging Rust's stringent safety features, such as ownership. The authors have carried out an extensive investigation of 581 memory bugs reported between 2016 and 2023, providing a detailed analysis of these issues. Based on the key insight that code related to raw pointers are prone to have memory bugs, ERASAN focuses on identifying raw pointers and employs static analysis to detect all memory accesses—both in safe and unsafe Rust code—that are performed using these pointers and therefore require security checks. The empirical evaluation demonstrates that ERASAN reduces ASAN's performance overhead by 61.6% and eliminates 93.6% instructions on average.

B.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) This work conducts a comprehensive analysis of Rust memory bugs in real-world applications, classifying common patterns of these defects.
2) Based on the key insight that memory bugs are solely associated with raw pointer manipulations, it designs and implements ERASAN, which authors pledge to release as open source.
3) The evaluation results demonstrate that ERASAN can substantially decrease the number of instrumentations required by ASAN, thereby enhancing performance during fuzz testing.