LiteRSan: Lightweight Memory Safety Via Rust-specific Program Analysis and Selective Instrumentation

Tianrou Xia
The Pennsylvania State University
tzx17@psu.edu

Kaiming Huang
The Pennsylvania State University
kzh529@psu.edu

Dongyeon Yu
UNIST
dy3199@unist.ac.kr

Yuseok Jeon
Korea University
ys_jeon@korea.ac.kr

Jie Zhou
The George Washington University
jie.zhou@gwu.edu

Dinghao Wu
The Pennsylvania State University
dinghao@psu.edu

Taegyu Kim
The Pennsylvania State University
tgkim@psu.edu

ABSTRACT

Rust is a memory-safe language, and its strong safety guarantees combined with high performance have been attracting widespread adoption in systems programming and security-critical applications. However, Rust permits the use of *unsafe* code, which bypasses compiler-enforced safety checks and can introduce memory vulnerabilities. A widely adopted approach for detecting memory safety bugs in Rust is Address Sanitizer (ASan). Optimized versions, such as ERASan and RustSan, have been proposed to selectively apply security checks in order to reduce performance overhead. However, these tools still incur significant performance and memory overhead and fail to detect many classes of memory safety vulnerabilities due to the inherent limitations of ASan.

In this paper, we present LiteRSan, a novel memory safety sanitizer that addresses the limitations of prior approaches. By leveraging Rust's unique ownership model, LiteRSan performs Rust-specific static analysis that is aware of pointer lifetimes to identify risky pointers. It then selectively instruments risky pointers to enforce only the necessary spatial or temporal memory safety checks. Consequently, LiteRSan introduces significantly lower runtime overhead (18.84% versus 152.05% and 183.50%) and negligible memory overhead (0.81% versus 739.27% and 861.98%) compared with existing ASan-based sanitizers while being capable of detecting memory safety bugs that prior techniques miss.

1 INTRODUCTION

Memory-safe programming languages have emerged as a promising approach [74] to mitigate prevalent memory safety vulnerabilities, which account for 70%–80% of all software vulnerabilities [36, 46, 73]. Among these languages, Rust [22] stands out by enforcing strong compile-time safety guarantees. Its advanced type system detects security issues early and helps confine additional costs to protect safety-critical operations. Studies show that, aside from these checks, Rust's performance can closely match that of C/C++ [81]. Consequently, Rust has rapidly gained adoption in security-critical and performance-sensitive domains [10, 37, 66].

Despite its robust safety guarantees, Rust's type system is not flawless. It can be too restrictive, preventing the expressiveness required for low-level systems programming, or it may introduce prohibitive runtime overhead in performance-critical code paths.

Consequently, Rust permits *unsafe* code, such as raw pointer dereferences or calling external C library functions [2, 12, 48], enabling developers to bypass Rust's memory safety checks. Nevertheless, the use of unsafe Rust code reintroduces memory safety vulnerabilities, such as buffer overflows and Use-After-Free (UAF) bugs, undermining Rust's foundational memory-safety benefits [39, 40, 48, 79].

Various detection and mitigation mechanisms have been proposed to address memory safety challenges introduced by unsafe Rust. Static analysis tools, such as Rudra [4], MirChecker [28], and SafeDrop [9], have successfully identified many real-world vulnerabilities in Rust programs. However, these tools typically suffer from high false positives (e.g., Rudra reports approximately 89% false positives [4]), and have limited capability in detecting diverse bug types [8, 38]. Memory isolation techniques, such as XRust [31], TRust [5] and PKRUSafe [21], provide runtime protection by restricting unsafe code's access to memory objects exclusively used by safe code. Nonetheless, these approaches target only subsets of memory objects and primarily focus on spatial memory errors (e.g., buffer overflows) while neglecting temporal memory errors such as UAF. Rust fuzzing frameworks [60–62, 80] have also emerged to detect memory safety vulnerabilities. However, it is well-known that the probabilistic nature of the fuzzing approach results in challenges of systematical detection of memory errors [6].

Researchers have also developed tools based on Address Sanitizer (ASan) [65]—a compiler-based memory error detector—to reveal memory safety vulnerabilities in Rust. Compared to static analysis (limited bug detection capability), memory isolation (partial protection), and fuzzing (probabilistic by nature), ASan-based approaches provide deterministic dynamic validation of *every* memory access. Notably, ERASan [38] and RustSan [8] have advanced this area by optimizing away redundant checks for memory accesses already instrumented by the Rust compiler, thereby significantly reducing ASan's runtime overhead by 71.4% and 62.3%, respectively.

Despite these advances, existing ASan-based tools still do not fully align with Rust's native safety guarantees. Although ERASan and RustSan remove certain checks already enforced by Rust's type system, their reliance on traditional C/C++ pointer analyses (i.e., SVF [69]) leads to significant over-approximation of unsafe pointers, as such analyses are not integrated with Rust's ownership and borrowing semantics [25]. As a result, both tools introduce superfluous checks for memory accesses that are already guaranteed to be safe, imposing unnecessary runtime overhead. In addition, the

static analysis time of ERASan and RustSan is prohibitively high, increasing compilation time by 1,635.35% and 1,193.31% per our measurements, due to their reliance on SVF, which is particularly expensive for large programs [21]. Furthermore, ASan suffers from inherent limitations in bug detection. Its red zones can be bypassed by overflows that exceed the boundaries, and its shadow memory mechanism may fail to detect UAF bugs when freed memory is reallocated post-quarantine, which makes dangling pointers to the original object undetectable. These gaps cause ASan-based tools to provide incomplete bug coverage despite significant overhead.

To address these limitations, our goal is to design a memory error detection mechanism tailored to Rust's inherent safety guarantees while addressing the loopholes introduced by unsafe code and the weaknesses of existing detection frameworks. Specifically, we strive to (1) identify pointers that truly pose spatial or temporal risks by incorporating a Rust-specific static analysis, eliminating extraneous checks on pointers that are either statically-proven safe or protected with compiler-inserted checks, (2) maintain complete coverage and precision in detecting **all** classes of memory errors, including spatial and temporal errors without using heavyweight ASan-based approaches, and (3) minimize overhead by integrating Rust's ownership and borrowing rules into both static analysis and enforcing selective instrumentation for lightweight runtime checks.

Achieving these three objectives requires addressing three major challenges: (1) Rust's allowance of raw pointers within otherwise safe code complicates standard pointer analysis, as many may-alias inferences valid in the C/C++ context break under Rust's stricter ownership model, (2) bridging static checks and runtime validation demands a lightweight metadata design that captures Rust memory safety model, and (3) avoiding expensive and coarse-grained ASan-based runtime checks. To address these challenges, we developed LiteRSan (**Lite**-**R**ust-**San**itizer), which deploys a Rust-specific static analysis to pinpoint *truly risky* pointers and selectively instrument them with minimal metadata to detect both spatial and temporal memory errors at runtime. This synergy of compile-time insights and targeted runtime checks enables comprehensive and accurate memory error detection with minimal overhead across 28 widely used Rust benchmarks: only 18.84% runtime, 0.81% memory and 97.21% compile-time overhead. In contrast, ERASan incurs 152.05% runtime, 739.27% memory, and 1,635.35% compile-time overhead, while RustSan incurs 183.50%, 861.98%, and 1,193.31%.

In summary, we make the following contributions:

- **Rust-specific Taint Analysis:** We introduce a Rust-specific static analysis scheme that identifies risky pointers by integrating Rust's ownership and borrowing semantics rather than defaulting to generic pointer analysis.
- **Lightweight Metadata Inference and Runtime Checks:** We design a compact metadata mechanism for runtime validation of spatial and temporal safety, removing the heavyweight components (e.g., red zones and shadow memory) of classic sanitizers.
- **Comprehensive and Efficient Bug Detection:** Our approach, LiteRSan, systematically detects spatial and temporal memory errors in Rust. Compared to prior Rust sanitizers, LiteRSan offers complete coverage and higher accuracy in detecting bugs while minimizing compile-time, runtime, and memory overhead compared with existing ASan-based tools.

2 BACKGROUND

In this section, we explain the background on Rust's safety guarantees and root causes of memory safety violations (§2.1). We also briefly discuss pointer analyses and AddressSanitizer (ASan) [65], which are commonly used by existing Rust memory safety tools, along with their limitations (§2.2 and §2.3). Finally, we present a motivating example to illustrate the redundant checks applied by prior ASan-based sanitizers (§2.4).

2.1 Rust's Memory Safety Guarantee

Spatial memory safety. Rust prevents out-of-bounds memory accesses by disallowing explicit pointer arithmetic on references and by internally maintaining spatial metadata (e.g., capacity and length) for containers, such as vector [22]. At compile time, the compiler either verifies the safety of a memory dereference or inserts runtime checks to detect any out-of-bounds access. However, Rust also permits *unsafe* code regions in which developers can manipulate raw pointers directly. These unsafe constructs bypass the compiler's spatial checks and can lead to memory safety violations, making them the root cause of out-of-bounds errors [2, 38, 48].

Temporal memory safety. Through its ownership and borrowing model, Rust ensures that each memory object has a single owner and that all borrowed references remain valid only as long as that owner is in scope [22]. This prevents use-after-free and double free by enforcing deallocation once the owner goes out of scope. Nevertheless, *unsafe* code regions allow the creation and handling of raw pointers in ways that can violate the ownership rules, enabling temporal errors if these pointers outlive their underlying objects. As with spatial safety, these unsafe constructs are the primary source of temporal memory safety violations.

2.2 Pointer Analyses in Rust

Prior work on identifying unsafe pointers (i.e., those that may lead to memory errors) in Rust typically relies on classical alias analysis [15]. In C/C++ contexts, such analysis often produces over-approximation: when it cannot disprove aliasing between two pointers, the analysis labels them as *may-alias* [11, 16]. This approach does not account for Rust's ownership and borrowing rules, where each object has a unique owner and references are strictly managed. Consequently, it leads to unnecessary and even higher false positives in Rust context compared with C/C++, as many pointers flagged are actually safely managed by the Rust compiler.

A prominent example is SVF [69], a state-of-the-art pointer alias analysis tool, used by both ERASan [38] and RustSan [8]. While SVF supports sophisticated inter-procedural and context-sensitive analyses, it suffers from two key drawbacks when applied to Rust. First, SVF's alias analysis significantly over-approximates due to its inability to leverage Rust's strict ownership semantics, resulting in the imprecise identification of unsafe pointers. Second, SVF incurs substantial analysis time, particularly on medium to large code bases [21, 27]. While powerful in theory, SVF introduces considerable computational overhead and scalability issues, making it impractical for large-scale analysis pipelines.

2.3 Address Sanitizer and Its Limitations

Address Sanitizer (ASan) [65] is a widely adopted tool for detecting memory safety violations at runtime, including both spatial errors and temporal errors. Its practicality and effectiveness have led to broad integration across major compilers and use in projects written in C, C++, and Rust. ASan instruments each memory access instruction with runtime checks to validate its legitimacy.

ASan detects memory errors using three core mechanisms: *red zones*, *shadow memory*, and *quarantine*. However, ASan's red zones are limited in size, as large overflows that bypass ASan's red zones evade detection. Meanwhile, once a memory region is freed and subsequently reallocated, the shadow memory is updated for the new allocation, erasing the evidence of original dangling pointers. Even with the quarantine mechanism that temporarily delays the reuse of freed memory regions by placing them in a quarantine pool, this protection is short-lived. Thus, ASan may miss temporal violations if a quanrantined region is reallocated while dangling pointers to the region are still in scope.

In addition to its incomplete coverage, ASan introduces substantial runtime and memory overhead. Typical performance slowdown ranges from 2–3×, and the red zone and shadow memory can cause the overall memory overhead to grow by several times. These limitations highlight the need for a more precise and lightweight memory safety mechanism. Ideally, such a mechanism would avoid red zones and shadow memory while preserving strong detection capabilities for both spatial and temporal safety violations.

2.4 Motivating Example

While Rust enforces memory safety for most memory accesses (§2.1), severe errors (e.g., buffer overflows and UAF) can still occur when unsafe code is used. Listing 1 shows an example of a common scenario in web applications (e.g., Servo [66]).

```
1  struct Cache {
2      ptr: Option<*mut u8>,
3  }
4
5  impl Cache {
6      fn save(&mut self, ptr: *mut u8) {
7          self.ptr = Some(ptr);}
8
9      fn load(&self) -> Box<String> {
10         unsafe {
11             Box::from_raw(self.ptr.unwrap())}}
12 }
13
14 fn main() {
15     let mut cache = Cache { ptr: None };
16     let token = Box::from("session-token");
17     println!("Session_token:_{}", token);
18     {
19         let local_token = token;
20         cache.save(local_token.as_ptr() as *mut u8);
21         // local_token goes out of scope here.
22     }
23     let stale_token = cache.load(); // Dangling pointer
24     println!("Stale_session_token:_{}", stale_token); // UAF
25 }
```

Listing 1: Use-after-free by caching a raw pointer after ownership transfer.

In Listing 1, the string "session-token" is a heap object allocated at line 16. A smart pointer, token, points to and owns this object. At line 19, ownership is transferred: a new smart pointer, local_token, takes the ownership of the heap object. Lines 20 and 7 define a raw pointer, self.ptr, derived from local_token. This raw pointer does not take the ownership of the string, so the owner remains local_token. local_token goes out of scope at the end of line 21, causing the object it owns to be deallocated. The raw pointer self.ptr then becomes dangling. At line 23, a new smart pointer, stale_token, is created from the dangling raw pointer, and it also becomes dangling. Then, all subsequent dereferences of the two dangling pointers are UAF (e.g., the one at line 24).

ASan instruments all memory accesses, which can be redundant, incurring high performance and memory overhead without guaranteeing comprehensive memory safety.[1] For the example in Listing 1, no spatial memory safety check is necessary. For temporal memory safety, the dereference of token (line 17) does not require safety instrumentation, as Rust's ownership model ensures its validity.

To address this deficiency, prior work—namely, ERASan [38] and RustSan [8]—improves ASan's performance by selectively instrumenting only raw pointers, or pointers in unsafe code, and their aliases. However, for the example discussed here, conventional alias analysis would identify all pointers in Listing 1 as aliases to the raw pointer self.ptr in unsafe code, resulting in redundant checks inserted to the dereference site of a safe pointer (e.g., line 17). This redundancy stems from insufficient consideration of Rust's memory safety guarantees, leading to over-approximating and instrumenting safe pointer dereferences.

2.5 Ideal Memory Error Detection for Rust

Ideally, a Rust memory safety sanitizer should: (1) leverage Rust's memory safety model to precisely differentiate safe pointers (e.g., token) from unsafe ones (e.g., self.ptr); (2) selectively instrument only unsafe pointers, avoiding redundant checks on Rust-guaranteed safe pointers; and (3) provide comprehensive, accurate, and efficient detection of all memory error classes. Such an approach narrows safety checks to only unsafe operations, incurring minimal overhead while ensuring comprehensive detection coverage.

3 THREAT MODEL AND CHALLENGES

In this section, we introduce the threat model and the challenges to be addressed through the design and implementation of LiteRSan.

3.1 Threat Model

We assume that memory errors, including both spatial (e.g., out-of-bound read/write) and temporal (e.g., UAF and double-free) errors, are possible in Rust programs. Our goal is to detect all such memory errors. While directly-linked C/C++ libraries may also contain memory errors, we focus on Rust source code and neither analyze nor harden such external libraries. We also assume that no extra memory safety defenses are deployed beyond Rust's built-in safety support. Memory leaks are out of scope, as they are generally

---

[1]As evaluated in MSET [76], ASan failed to detect around 50% of C/C++ memory errors in their constructed benchmark. We believe the rationale would be similar for Rust, as ASan is not aware of Rust's memory safety model.
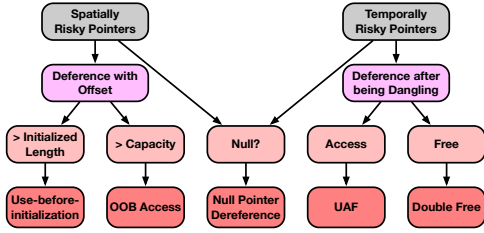
Figure 1: Memory safety bug patterns in Rust programs.

not classified as memory safety violations [42, 43, 65, 70]. Figure 1 shows the complete set of the bug patterns that LiteRSan covers, as ASan-based tools [8, 38] do in Rust programs.

Non-memory-safety errors, such as concurrency bugs and logic errors, are outside the protection scope of LiteRSan. In addition, LiteRSan is not designed to detect type conversion bugs. Notably, Rust's std::mem::transmute() [53] allows converting the type of an object to any other type. LiteRSan does not address errors caused by misusing this dangerous API. However, LiteRSan can detect type confusion bugs that arise from temporal errors, such as UAF. As mentioned above, LiteRSan does not target external C/C++ libraries. Therefore, cross-language attacks [35] that propagate exploitation from components written in unsafe languages (e.g., C/C++) are out of scope and can be addressed by existing works [50].

3.2 Challenges

As discussed in §1 and §2, existing memory error detection approaches are both incomplete and inefficient. Static analyzers [4, 28] often produce a high number of false positives, while ASan-based techniques [8, 38] incur significant performance overhead and still fail to detect many bugs. We observed that the shortcomings of ASan-based tools largely stem from analyzing Rust in LLVM IR [26]—a language-independent, low-level compiler intermediate representation, using generic pointer analysis [69] without accounting for Rust's unique memory safety guarantees.

To propose our approach, we first introduce the key concept of *risky pointer*, which will be used throughout the rest of this paper.

Definition 3.1. A **risky pointer** is a pointer whose dereferences may violate memory safety. Such a pointer is *spatially risky* if it bypasses Rust's spatial enforcements, or *temporally risky* if it may outlive its referenced object.

Detailed explanations of spatially and temporally risky pointers are presented in §5.2. Note that (1) a pointer may be both spatially and temporally risky; (2) A raw pointer becomes risky *only when it is exposed in unsafe code* (Definition 3.2); and (3) Rust's native smart pointers may also be risky. For example, constructing multiple smart pointers from a raw pointer may violate Rust's ownership rules, rendering these smart pointers risky and potentially causing UAF bugs that elude compiler checks.

Definition 3.2. An **exposed raw pointer** is a raw pointer *directly used in unsafe code*, bypassing Rust's safety guarantees.

We identify three key challenges in building an efficient and comprehensive memory safety sanitizer tailored to Rust.

- **C1: Leveraging Rust's unique type system to precisely identify risky pointers.** Program analysis for Rust in prior

work [5, 8, 31, 38] does not utilize Rust's ownership and borrowing semantics, significantly over-approximating risky pointers. A refined approach should integrate Rust's intrinsic memory safety model to more precisely identify risky pointers.

- **C2: Managing lightweight safety metadata for runtime checks.** Relying on a coarse-grained protection scheme like ASan's shadow memory and red zones [65] is expensive and imprecise. Tailoring compact yet fine-grained metadata that incorporates Rust's memory safety guarantees enables more efficient and accurate runtime error detection. Additionally, because raw pointers lack spatial metadata (i.e., bounds information), LiteRSan must infer and maintain their metadata to enable runtime validation.

- **C3: Minimizing overhead while ensuring coverage.** As Rust's memory safety model already protects a substantial amount of memory accesses, additional checks are only needed for those involving risky pointers. The challenge is to minimize cost while maintaining accuracy and comprehensiveness by (1) selectively instrumenting only the truly risky pointers based on their specific risk types and (2) enforcing an efficient runtime check mechanism rather than incomplete and inefficient ASan-style checks.

By addressing these challenges, LiteRSan complements Rust's inherent memory safety guarantees with precise instrumentation to achieve comprehensive and low-overhead runtime safety checks, closing the gap left by prior work [8, 38].

4 OVERVIEW

To address the three key challenges described in §3.2, we propose a Rust-specific static analysis to identify risky pointers. Coupling it with a metadata-based runtime checking mechanism, we develop our prototype system, LiteRSan. Figure 2 illustrates the main components and the overall workflow of LiteRSan.

Stage 1 conducts Rust-specific static analysis to addresses **C1**. LiteRSan first pre-processes the target Rust program using reachability analysis to narrow the analysis scope to potentially reachable functions (§5.1). Within this scope, LiteRSan identifies both *Spatially Risky Pointers* (§5.3) and *Temporally Risky Pointers* (§5.4). Since the misuse of exposed raw pointers is the primary cause of memory safety violations in Rust, LiteRSan begins by identifying them. It annotates the instructions involving raw pointers during Mid-level IR (MIR) [57] to LLVM IR code generation, and analyzes the definitions and uses of these pointers in annotated instructions to identify exposed raw pointers, which are classified as both spatially and temporally risky because they are exempt from Rust's compile-time safety enforcement. To identify additional temporally risky pointers, LiteRSan performs *lifetime-aware* taint analysis starting from exposed raw pointers. This is necessary because exposed raw pointers can propagate temporal risks to other pointers referencing the same memory object. In contrast, spatial risks do not propagate if raw pointer arithmetic and dereference is instrumented with bounds checking. Additionally, LiteRSan identifies risky pointers used in unsafe APIs that may cause memory safety violations.

Stage 2 constructs lightweight spatial and temporal metadata to address **C2** (§6.1 §6.2), enabling efficient runtime validation. For
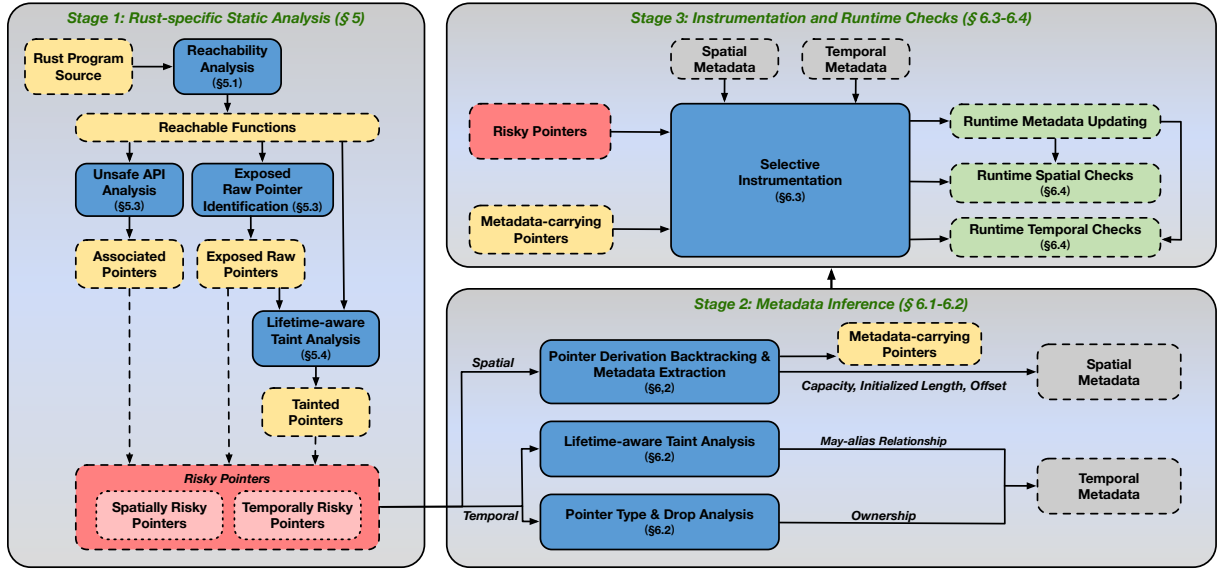
Figure 2: LITERSAN overview. LITERSAN consists of three stages. Each addresses one of the primary challenges in enabling efficient and comprehensive sanitizer checks. The output of each stage serves as the input to the next.

spatially risky pointers, LITERSAN maintains three pieces of metadata: capacity, initialized length and offset. When spatial metadata is unavailable at a pointer's definition site (e.g., an exposed raw pointer derived from other pointers), LITERSAN backtracks pointer derivations (Definition 5.1) to extract metadata from the object's allocation site. This process also identifies *metadata-carrying pointers*, which are responsible for transmitting spatial metadata at runtime.

For temporally risky pointers, the risk arises from shared access to the same memory object. Once the object is deallocated by its owner or via an unsafe API, all referencing pointers that remain in scope become dangling. To address this, LITERSAN maintains may-alias relationships and ownership information as temporal metadata. May-aliases are established during Stage 1 via taint analysis, as pointers tainted by the same exposed raw pointer reference the same object. Among these, LITERSAN analyzes pointer types and `Drop` implementations [59] to identify owners.

Stage 3 addresses **C3** by selectively instrumenting identified risky pointers and metadata-carrying pointers (§6.3), thereby minimizing runtime overhead while preserving the comprehensive coverage of runtime checks. Leveraging spatial and temporal metadata collected in Stage 2 and updating it during execution, LITERSAN performs accurate and comprehensive detection of both spatial and temporal memory errors (§6.4). The complete set of memory safety bugs detectable by LITERSAN is summarized in Figure 1.

5 RUST-SPECIFIC STATIC ANALYSIS

In this section, we present LITERSAN's Rust-specific static analysis, which addresses the challenge of identifying risky pointers (C1) discussed in §3.2. We begin by defining the scope of the analysis in §5.1 and introducing the definition of risky pointers in §5.2. We then describe our approach to identifying spatially and temporally risky pointers in §5.3 and §5.4, respectively, and conclude this section by discussing soundness and precision in §5.5.

5.1 Static Analysis Scope Restriction

LITERSAN restricts its static analysis to *reachable functions*, motivated by the structure of Rust programs, which often include deeply nested library code, much of which is dead code (i.e., unreachable from the program entry point[2]). To exclude such dead code from LITERSAN's analysis, LITERSAN performs *reachability analysis* to conservatively identify and analyze only potentially reachable functions which will be executed at runtime.

Specifically, starting from the program's entry point, LITERSAN identifies and enqueues both directly called functions and address-taken functions (i.e., potential indirect call targets [33, 78]) for analysis. For each function in the queue, LITERSAN recursively discovers and further enqueues the function's callees and address-taken functions, thereby restricting its analysis scope to functions potentially reachable during execution. By limiting analysis to reachable functions, LITERSAN focuses on identifying risky pointers within this scope that may lead to memory safety violations.

5.2 Risky Pointer Definition

Within LITERSAN's restricted analysis scope, most pointers are safe thanks to Rust's native safety guarantees for safe code, as discussed in §2.1. However, a subset of pointers remains unprotected and may still violate memory safety. We refer to these as *risky pointers* (Definition 3.1), and LITERSAN focuses its safety checks exclusively on them. For fine-grained analysis and instrumentation, we further classify risky pointers into *spatially risky* and *temporally risky* categories, corresponding to potential violations of spatial and temporal memory safety, respectively.

Spatially risky pointers include (1) exposed raw pointers and (2) smart pointers used in certain unsafe APIs. Unlike encapsulated raw pointers within smart pointers, which enforce Rust's safety

---

[2]The entry point is typically the `main` function. For library crates compiled with built-in benchmarks, benchmarking functions compiled and included in LLVM IR are treated as entry points as well.

guarantees, exposed raw pointers are spatially unsafe because arbitrary pointer arithmetic is permitted on them, which may result in invalid pointers whose dereferences are out-of-bound and not checked. Moreover, Rust's standard libraries (e.g. std) provide unsafe APIs that may subvert bounds checking if misused [3, 20]. When a pointer is used in conjunction with such unsafe APIs, it is considered spatially risky.

Temporally risky pointers include (1) exposed raw pointers and (2) any *valid* (i.e., in-scope according to Rust's scoping rules [58]) pointers that reference the same memory object as an exposed raw pointer. Exposed raw pointers are temporally unsafe because they are exempt from Rust's ownership rules; once the referenced object is deallocated, such raw pointers become dangling. Furthermore, as illustrated in §2.4, if a smart pointer is constructed from an exposed raw pointer and takes ownership of an object that already has an owner, multiple owners will coexist. Deallocating the object through one owner leaves the others dangling. Invalid pointers whose lifetimes have ended (e.g., token in Listing 1) are excluded from temporally risky pointers, since any use of them is prevented by Rust compiler.

5.3 Spatially Risky Pointer Identification

Exposed raw pointers. The misuse of exposed raw pointers is a primary cause of memory safety bugs in Rust programs [38]. To identify them, LiteRSan first tracks all raw pointers via LLVM IR metadata annotation during the MIR-to-LLVM IR lowering phase, followed by a fine-grained filtering to determine which are exposed raw pointers, as outlined in §4.

Based on ERASan [38]'s approach, LiteRSan attaches custom LLVM metadata [32] to IR instructions by modifying the codegen-ssa and codegen-llvm components of the rustc compiler. To determine the locations of annotations, LiteRSan performs a type-matching analysis during the MIR-to-LLVM IR lowering phase. Specifically, it analyzes the types of program variables and expressions in the MIR (i.e., Rust's mid-level representation) to identify those involving raw pointers (e.g., *const T). If a value is of raw pointer type, the corresponding LLVM IR instruction is tagged with !rawptr. Additionally, instructions originating from unsafe code are marked with !unsafe. This analysis allows LiteRSan to propagate type information from Rust's MIR and identify the instructions relevant to raw pointers in the resulting LLVM IR.

After annotating the LLVM IR, LiteRSan analyzes instructions tagged with !rawptr to filter out encapsulated raw pointers and identify only exposed raw pointers. These annotated instructions either define or use raw pointers. For each definition, LiteRSan checks whether this raw pointer is used within unsafe code by examining the presence of the !unsafe metadata. If so, it is identified as an exposed raw pointer. For each use, LiteRSan similarly checks whether it occurs in unsafe code and, if so, traces back to the corresponding definition site to identify the exposed raw pointer. In short, only raw pointers that are directly used within unsafe code are identified for subsequent lifetime-aware taint analysis. This filtering is crucial because encapsulated raw pointers within safe abstractions (e.g., smart pointer creation) are never directly dereferenced and therefore do not pose risks. Through this analysis, LiteRSan accurately identifies only exposed raw pointers

that may cause memory safety bugs, which are considered as *risky pointers*—both spatially risky and temporally risky (see §5.2).

Unsafe APIs. As Rust's standard libraries (e.g., std) provide unsafe APIs that may cause spatial safety violations [3, 20], LiteRSan analyzes these APIs to extend its protection. In general, an API may be unsafe because (1) it directly uses exposed raw pointers or (2) it subverts Rust's bounds checks when misused. LiteRSan can handle type (1) APIs by identifying underlying exposed raw pointers using the method discussed above and marking them as spatially risky.

Type (2) APIs are more challenging to address. A notable example is vec::set_len() [56], which can alter a vector's length to an arbitrary value, potentially resulting in out-of-bounds accesses that bypass the compiler's spatial safety checks. Automatically and comprehensively identifying such APIs would requires analyzing and *understanding* all library code, which is an undecidable problem [49]. Therefore, we manually examined Rust's standard libraries and identified nine APIs that may circumvent bounds checks (Appendix B). LiteRSan marks the pointers involved in these APIs as spatially risky, updates their metadata, and inserts runtime checks accordingly. For example, to detect out-of-bounds accesses potentially caused by vec::set_len(), LiteRSan retrieves the capacity of the vector at its definition site and inserts a spatial check at the API's call site to verify whether the new length (i.e., the argument to vec::set_len()) exceeds the legal capacity.

5.4 Temporally Risky Pointer Identification

To detect temporal memory safety violations, LiteRSan must go beyond merely identifying exposed raw pointers (as discussed in §5.3). It must also detect any valid pointer that might reference the same memory object as an exposed raw pointer. This may sound similar to finding all may-alias pointers to this memory object; however, the key difference is that any may-alias smart pointers that have been *moved* or gone out of scope should be excluded (e.g., token after line 19 in Listing 1).

One approach to identifying temporally risky pointers is to perform alias analysis on each exposed raw pointer, find the complete set of pointers that refer to the same memory object, and then filter out aliased pointers that are invalid. However, existing alias analysis frameworks for LLVM (e.g. SVF [69]) conservatively mark two pointers as may-alias whenever they cannot prove that the pointers are not aliases, which leads to over-approximated alias sets even before accounting for the additional over-approximation introduced by ignoring Rust's safety guarantees. Furthermore, whole-program alias analysis is generally highly expensive for large programs [21]. Therefore, we develop a new *Rust-specific, inter-procedural, flow-sensitive, lifetime-aware taint analysis* to **directly** identify temporally risky pointers without relying on traditional alias analysis.

Illustrative example. We reuse the example in Listing 1 to briefly illustrate the workflow of LiteRSan's taint analysis. In this example, an exposed raw pointer, self.ptr is defined at line 20 through an existing smart pointer (local_token), which owns a heap object. Here, the exposed raw pointer self.ptr serves as a *taint source*. LiteRSan's taint analysis performs two key operations to identify temporally risky pointers to the object pointed by self.ptr:

- **Backward propagation** traces the ownership transfer preceding the definition of the taint source. This includes identifying the smart pointer (`local_token`) whose ownership is transferred from `token`. Since `token` goes out of scope before `self.ptr` is defined, the analysis terminates backward propagation at `local_token`'s definition site (line 19) without tainting `token`.
- **Forward propagation** tracks pointers derived from the taint source and taints them. In this example, a new pointer `stale_token` is derived from `self.ptr` (line 23), making it be tainted and identified as temporally risky.

In short, to capture all temporal safety violations, the taint analysis must consider both the preceding ownership history of any object referenced by an exposed raw pointer (i.e., backward) and all subsequent pointers derived from that raw pointer (i.e., forward).

Exposed raw pointer classification. To distinguish the exposed raw pointers (taint sources) that require different taint analysis propagation directions, we classify them as follow.

- **Type 1 (T1) raw pointers:** exposed raw pointers created by referencing an object already owned by an existing smart pointer (e.g., via `Vec::as_ptr()` [55]). As shown in Listing 1, `self.ptr` is derived from a valid smart pointer `local_token`. The creation of such raw pointers implies existing ownership. Consequently, T1 raw pointers require both backward taint analysis (to trace the ownership history of the referenced object) and forward taint analysis (to track subsequent pointer derivations).
- **Type 2 (T2) raw pointers:** exposed raw pointers created to reference a newly allocated memory object. Since there is no preexisting ownership chain to consider, T2 raw pointers require only forward taint analysis.

After locating these definitions, LITERSAN performs an inter-procedural taint analysis starting from the definition site of each exposed raw pointer (i.e., taint source) and propagating on only forward or both directions according to the class of raw pointers.

Lifetime-aware taint analysis. LITERSAN performs a combination of backward and forward taint analysis, both of which track pointer derivation instructions (Definition 5.1), augmented with lifetime-aware propagation that respects Rust's ownership rules, to identify temporally risky pointers.

Definition 5.1. A ***pointer derivation instruction*** is any operation that produces a new pointer value from an existing one, by one of the following:

- Assignment (direct copy of a pointer),
- Computation (arithmetic or type conversion),
- Memory propagation (store/load through objects), or
- Inter-procedural transfer (via function calls or returns).

- **Backward taint analysis:** Starting from each taint source (i.e., the definition site of a T1 pointer), the analysis traces backward along the derivation chain to the pointers from which T1 is derived. These pointers share the same referenced object with T1, and their definition sites are marked as taint sinks. This propagation halts if ownership is transferred, as any further use of the original owners is disallowed by the Rust compiler, ensuring that the original owners are free from temporal memory safety violations. In particular, if a pointer is invalidated before

the exposed raw pointer is defined—through function return for stack pointers or through explicit `drop` for heap pointers—it is considered safe and excluded from tainted pointers, as it can no longer contribute to temporal safety violations.
- **Forward taint analysis:** Starting from each taint source (i.e., the definition site of a T1 or T2 pointer), the analysis propagates taint to all pointers derived from it. Any pointer derived from a tainted pointer is likewise marked as tainted. This forward propagation continues until no further pointer derivations exist.

Inter-procedural analysis. Taint propagates inter-procedurally through function calls and returns. For direct calls, forward propagation flows from actual arguments at the call site to the corresponding formal parameters of the callee, and from the callee's return value to the variable receiving it in the caller. Backward propagation flows in the reverse direction and terminates at pointers that have been invalidated, such as those that are out of scope or have transferred ownership.

For indirect calls, LITERSAN conservatively resolves potential call targets using a type-based analysis [45, 75]. This approach matches function signatures (i.e., function prototypes) at indirect call sites with those of address-taken functions. Although more advanced multi-layer type analysis techniques [33, 78] can improve precision, they introduce additional static analysis overhead, while the precision gain in Rust is limited. This is because Rust programs typically rely less on dynamic dispatch [13, 34, 41] than programs in other languages, and the multi-layered structural patterns common in C/C++ are less prevalent in Rust. Once potential callees are identified, taint is propagated in the same manner as for direct calls.

LITERSAN adopts a worklist-based algorithm [44] adapted for inter-procedural taint analysis, as presented in Algorithm 1. First, it caches pointer derivations whose sources originate from other functions, either directly (e.g., formal parameters) or indirectly (e.g., intermediate variables derived from formal parameters), and are therefore *unresolved* within the current function context (lines 4–10). After completing the initial pass over all functions, it performs a depth-first search over the cached derivations to exhaustively propagate taint (lines 11–25). This two-step process ensures that all transitive taint relationships are resolved and that all potential temporally risky pointers are identified. In addition, LITERSAN also addresses temporally risky pointers involved with unsafe APIs, similar to the approach described in §5.3.

5.5 Soundness and Precision

LITERSAN's Rust-specific static analysis is sound in identifying both spatially and temporally risky pointers. It begins with a conservative reachability analysis (§5.1) that includes all functions that may execute at runtime, ensuring all risky pointers that can trigger memory errors at runtime are within the analysis scope.

To identify spatially risky pointers, LITERSAN begins by conservatively annotating all instructions involving raw pointers. This over-approximation ensures that all raw pointers are initially captured. The subsequent analysis prunes false positives by leveraging the fact that exposed raw pointers that bypass Rust's safety guarantees can only be used within unsafe code. Thus, LITERSAN excludes encapsulated raw pointers that cannot be directly accessed and are protected by Rust memory safety rules. This pruning maintains

Algorithm 1: Inter Procedural Taint Propagation

Input: F — set of functions; R — set of initially tainted raw pointers
Output: *TaintedSets* — mapping from each taint source to its tainted pointers

1 **function** InterProcTaintPropagation(F, R)
2 $\quad$ Initialize *TaintedSets* to map each $r \in R$ to $\{r\}$
3 $\quad$ Initialize *WorkList* $\leftarrow \emptyset$
4 $\quad$ **for** *each unresolved pointer derivation D in F* **do**
5 $\quad\quad$ $(src, dst) \leftarrow$ ExtractSourceAndDestination(D)
6 $\quad\quad$ Add D to *WorkList*
7 $\quad\quad$ **if** *src is tainted* **then**
8 $\quad\quad\quad$ Add dst to the same tainted set as src
9 $\quad\quad$ **else if** *dst is tainted* **and** *no ownership transfer in D* **then**
10 $\quad\quad\quad$ Add src to the same tainted set as dst
11 $\quad$ **for** *each tainted pointer t in TaintedSets* **do**
12 $\quad\quad$ Initialize *Visited* $\leftarrow \emptyset$
13 $\quad\quad$ Initialize *Stack* $\leftarrow \{t\}$
14 $\quad\quad$ **while** *Stack is not empty* **do**
15 $\quad\quad\quad$ $p \leftarrow$ pop an element from *Stack*
16 $\quad\quad\quad$ **if** $p \notin$ *Visited* **then**
17 $\quad\quad\quad\quad$ Add p to *Visited*
18 $\quad\quad\quad\quad$ **for** *each unresolved pointer derivation D in WorkList* **do**
19 $\quad\quad\quad\quad\quad$ $(src, dst) \leftarrow$ ExtractSourceAndDestination(D)
20 $\quad\quad\quad\quad\quad$ **if** $src = p$ **then**
21 $\quad\quad\quad\quad\quad\quad$ Add dst to the same tainted set as t
22 $\quad\quad\quad\quad\quad\quad$ Push dst onto *Stack*
23 $\quad\quad\quad\quad\quad$ **else if** $dst = p$ **and** *no ownership transfer in D* **then**
24 $\quad\quad\quad\quad\quad\quad$ Add src to the same tainted set as t
25 $\quad\quad\quad\quad\quad\quad$ Push src onto *Stack*
26 $\quad$ **return** *TaintedSets*

---

soundness while improving precision. Additionally, LiteRSan identifies and handles each unsafe API individually, based on a thorough review of the Rust standard library.

For temporally risky pointers, LiteRSan employs a lifetime-aware taint analysis. Soundness in identifying such pointers is ensured by three key elements. First, the taint and pointer-derivation analyses described above are sufficient to capture all potential may-alias relationships in this context, as Rust's ownership and borrowing rules ensure that aliasing can only occur through explicit and syntactically visible pointer derivations [19]. Second, this guarantee naturally extends to loops because the LLVM IR contains a fixed and finite set of pointer derivation instructions. LiteRSan tracks each such instruction within a loop, analyzing only static derivation relationships rather than mutable program states (e.g., ranges, offsets, or sizes). As a result, every pointer along the derivation chain that may reference the same object as the taint source is captured, regardless of the complexity of the loop body or control flow. Third, LiteRSan employs a type-based analysis to conservatively resolve indirect calls, ensuring that all relevant pointer derivations are included. Together, these three elements guarantee that LiteRSan marks all valid alias pointers as temporally risky.

While the analysis is sound by design, potential false negatives may arise in practice due to implementation limitations, such as compiler optimizations or missing IR from dynamically linked code. The approach may also introduce some over-approximation, for example, by analyzing derivations that never occur during actual execution. However, this imprecision is significantly reduced compared to prior work [8, 38] relying on traditional points-to analysis, which is unaware of pointer lifetimes. In contrast, LiteRSan excludes pointers that are invalid at the time of exposed raw pointer

creation or not derived from tainted sources, as these are protected by Rust's compile-time safety guarantees and are not susceptible to temporal memory safety violations.

6 LIGHTWEIGHT RUNTIME CHECKS

In this section, we present the lightweight runtime checks of LiteR-San. LiteRSan uses compact memory safety metadata in place of red zones and shadow memory to address Challenge C2 and adopts a selective instrumentation strategy to address Challenge C3 (see §3.2). We describe the metadata structures in §6.1 and the metadata inference approach in §6.2. §6.3 details our selective instrumentation strategy, and §6.4 explains how the instrumented checks uses memory safety metadata to perform runtime validation.

6.1 Metadata Structure

For each risky pointer, LiteRSan maintains *spatial metadata* for spatially risky pointers and *temporal metadata* for temporally risky pointers. This metadata is inferred during static analysis, propagated at runtime through instrumentation, and stored in dedicated data structures rather than embedded directly in the pointer representation (e.g., fat pointers). At runtime, the metadata is dynamically updated and used to detect memory safety violations.

Spatial metadata. For spatially risky pointers, LiteRSan tracks three key attributes that are necessary and sufficient to enforce spatial memory safety in Rust:
- **Capacity:** the maximum number of elements allowed in a referenced memory object. For pointers referencing scalar-type objects (e.g., integers), the capacity is set to 1.
- **Initialized length:** the number of elements that have been initialized within a referenced memory region.
- **Offset:** the index within an object that the pointer references.

LiteRSan maintains a map from each spatially risky pointer to its spatial metadata, which consists of the attributes listed above; this metadata map is stored separately from the pointers themselves.

Temporal metadata. For temporally risky pointers, LiteRSan tracks the following information as temporal metadata:
- **May-alias relationships:** pointers that may reference the same memory object are grouped to represent their potential aliasing.
- **Ownership:** the owner(s) of the referenced objects.

LiteRSan uses **taint source raw pointers** to link associate temporally risky pointers with their temporal metadata via two maps: a *reverse map* that links each temporally risky pointer back to its originating taint source, and a *forward map* that links each taint source to the corresponding metadata which it shares with its tainted pointers. This dual-mapping design avoids redundant metadata storage for multiple pointers derived from the same source while accurately associating each temporally risky pointer with its temporal metadata.

To record may-alias relationships, LiteRSan groups all temporally risky pointers that may reference the same memory object into a *pointer set*. This pointer set integrates with LiteRSan's taint analysis, as may-alias pointers identified by LiteRSan share a common taint source and can be grouped during taint analysis without additional effort. When an exposed raw pointer is derived from another,

their pointer sets are merged to ensure a complete representation of may-alias relationships.

Ownership is a critical component of temporal metadata, as the owners are the pointers responsible for deallocating the referenced objects. As a result, owners must be tracked at runtime to update the temporal state (i.e., dangling or valid) of all pointers in the same pointer set. To record the owners, LiteRSan maintains a dedicated *owner set*, which is a subset of the pointer set.

6.2 Metadata Inference

Metadata inference is performed during static analysis. LiteRSan infers each risky pointer's memory safety metadata at its definition site and instruments the code to receive and maintain the inferred metadata, enabling runtime validation.

Spatial metadata inference. To infer spatial metadata for spatially risky pointers, LiteRSan analyzes each spatially risky pointer's definition site and, if necessary, traces pointer derivations back to the allocation site of the memory object, where the spatial metadata is defined. Specifically, LiteRSan backtracks along the pointer-derivation chain to locate the corresponding *root* pointer, which is the first pointer that references the memory object. LiteR-San then extracts spatial metadata from the root pointer's definition site, where the memory allocation appears as an operand. Depending on how the memory region is referenced, metadata extraction falls into two distinct cases:

- In the **direct case**, the root pointer references a memory object whose spatial metadata can be directly extracted. This occurs when the referenced object is a basic container provided by the Rust standard library, such as vectors (Vec<T>) and arrays ([T; N]), which manage contiguous memory regions. In this case, LiteRSan directly obtains spatial metadata from the container's fields. For example, the length and capacity fields in Vec<T> directly provide *Initialized Length* and *Capacity* (§6.1).
- In the **indirect case**, the root pointer refers to a memory object indirectly via abstractions such as Box<T> or Rc<T>, which do not explicitly carry spatial metadata. In this case, LiteRSan backtracks to the definition site of the underlying T-typed object to infer and extract spatial metadata.

Additionally, when the definition site of any exposed raw pointer involves pointer arithmetic, LiteRSan computes the resultant *Offset*. Another category of spatially risky pointers, smart pointers associated with unsafe APIs, is relatively uncommon. Thus, LiteR-San handles these cases individually, applying sanitizer checks based on the semantics of each unsafe API, as discussed in §5.3.

Once spatial metadata is extracted from a root pointer, it is transmitted to the spatially risky pointers along the pointer-derivation chain. The root pointer, along with the intermediate pointers on this chain, is referred to as *metadata-carrying pointers*. Note that metadata-carrying pointers are not necessarily risky themselves but are tracked to enable accurate metadata propagation.

To enable runtime checking, LiteRSan propagates statically inferred spatial metadata to spatially risky pointers through inserted instrumentation. This process involves two cases. First, if the risky pointer is a root pointer, the metadata is available at its definition site; therefore, instrumenting its definition site suffices. Second, if the risky pointer is a derived pointer, metadata must be passed along the derivation chain. In this case, LiteRSan instruments the definition sites of all metadata-carrying pointers along the chain to ensure proper metadata propagation to the derived pointer.

Temporal metadata inference. For temporal metadata, the may-alias relationships and the taint-source raw pointers are inferred during the identification of temporally risky pointers, through lifetime-aware taint analysis, as described in §5.4. The owner(s) of the referenced memory objects are inferred by analyzing the definition site of each tainted pointer based on Rust's ownership model. Specifically, owners are smart pointer types (e.g., Box and Rc) that manage the lifetime of a memory object and are responsible for its deallocation. LiteRSan detects owners by analyzing pointer types and determining whether they are associated with memory deallocation, typically indicated by their Drop implementation [59].

Robustness and completeness guarantee. LiteRSan adopts a hybrid strategy to infer spatial and temporal metadata, ensuring that metadata is accurately and reliably extracted.

Spatial metadata is inferred for (i) exposed raw pointers and (ii) smart pointers associated with unsafe APIs. For exposed raw pointers, LiteRSan extracts initial metadata from referenced objects via static analysis and passes it to runtime functions through instrumentation. This method is **robust** because Rust's ownership and borrowing rules guarantee aliasing occurs only through explicit, visible derivations [19], making metadata fully traceable. It is also **complete** since static analysis extracts only initial metadata, while runtime instrumentation ensures precise, immediate updates. During execution, root pointers are initialized before derived pointers, enabling accurate metadata transmission and preventing stale values. For smart pointers associated with unsafe APIs, LiteRSan addresses each case individually based on its semantics. The small number of such APIs, combined with tailored handling, ensures **robustness** and **completeness**.

Temporal metadata consists of (i) may-alias relationships and (ii) object owners. May-alias relationships are inferred through lifetime-aware taint analysis, proven sound in §5.5. Owners are identified based on pointer types and Drop usage, following Rust's ownership rules. As both types and Drop usage are statically deterministic, this approach guarantees both **completeness** and **robustness**.

6.3 Selective Instrumentation

LiteRSan performs selective instrumentation by applying only the runtime checks necessary to each identified risky pointer. To support these checks, it also instruments the program to propagate statically inferred metadata and manage it at runtime. This section introduces the five classes of instrumentation used in LiteRSan and how they are selectively applied according to the risky pointer types and program context.

Instrumentation types. LiteRSan defines five instrumentation classes (**I1**–**I5**) to initialize and update metadata during execution, and perform runtime memory safety checks based on the metadata.

- **I1**: Pointer activation (metadata initialization).
- **I2**: Spatial metadata update.
- **I3**: Pointer deactivation (temporal metadata update).
- **I4**: Spatial safety checks.
- **I5**: Temporal safety checks.

I1 type. I1 instrumentation is inserted at the definition sites of identified spatially and temporally risky pointers, and metadata-carrying pointers (including root pointers). It receives statically inferred metadata and marks each pointer as active when being triggered at runtime, by registering the pointer with its associated spatial or temporal metadata.

For *spatially risky pointers* and their derivation sources (i.e., root and metadata-carrying pointers), I1 establishes a runtime mapping between each pointer and its spatial metadata. Root pointers are initialized with metadata directly from static analysis, while derived pointers inherit metadata from their source pointer. For *temporally risky pointers*, I1 maintains a mapping from each taint-source raw pointer to associated temporal metadata and a reverse mapping from each tainted pointer to its taint source, as discussed in §6.1. Depending on whether the definition site corresponds to a taint source or a tainted pointer, I1 creates or updates these mappings.

I2 type. LiteRSan employs I2 to update spatial metadata at runtime in two scenarios: (1) when the offset of a spatially risky pointer or metadata-carrying pointer is modified (e.g., through pointer arithmetic), and (2) when the underlying memory object is modified (e.g., through container operations, such as push() and pop()).

For (1), I2 updates the *Offset* field in its associated spatial metadata. For (2), I2 updates and synchronizes the *Initialized length* and/or *Capacity* field(s) for both the pointer performing the modification and all preceding pointers in the derivation chain. This is because those pointers all reference the same memory object.

I3 type. I3 instrumentation is inserted at deallocation sites, including function returns for stack-allocated objects and explicit drop operations for heap-allocated objects, to deactivate invalidated pointers. If an owner deallocates the memory object, or if the last owner is invalidated (e.g., via mem::forget()), I3 queries the reverse map to identify the taint-source raw pointer and updates corresponding spatial metadata, marking the taint-source raw pointer along with all pointers in its pointer set as dangling.

I4 and I5 types. LiteRSan applies **I4** at pointer arithmetic and dereference sites of spatially risky pointers, and **I5** at dereference and deallocation sites of temporally risky pointers, to detect spatial and temporal memory safety violations, respectively. The detection mechanisms for I4 and I5 are detailed in §6.4.

Instrumentation strategy. LiteRSan employs a selective instrumentation strategy that inserts only the necessary code at each instrumentation site. This approach ensures that metadata remains up-to-date and that memory safety violations are detected efficiently. Table 1 summarizes the instrumentation strategy.

For *all three pointer types*, LiteRSan inserts **I1** at their definition sites to register the pointers along with their associated spatial or temporal metadata at runtime, making them activated. This metadata is later used to initialize derived pointers or to validate memory safety at runtime.

For *spatially risky pointers*, which may cause spatial memory safety violations, LiteRSan selectively inserts **I2** and **I4**. I2 is placed at pointer arithmetic operations (e.g., add(), offset()) and at container modifier operations (e.g., unsafe API set_len()) to update spatial metadata instantly at runtime, enabling I4 to precisely perform spatial memory safety validation. **I4** is inserted at pointer

Pointer Type	Definition	Dereference	Pointer Arithmetic	Container Modifier	Deallocation
Spatially Risky Pointers	I1	I4	I2, I4	I2	-
Temporally Risky Pointers	I1	I5	-	-	I5, I3
Metadata-Carrying Pointers	I1	-	I2	I2	-

Table 1: Selective instrumentation strategy. Each class of instrumentation is applied based on the type of pointer and the type of operation, ensuring that only the necessary code is inserted at each instrumentation site. At the pointer arithmetic of a spatially risky pointer, I2 is before I4. At the deallocation site of a temporally risky pointer, I5 is before I3.

arithmetic and dereference sites to detect spatial violations using the maintained spatial metadata. Importantly, I2 is placed before I4 at pointer arithmetic sites, ensuring that any invalid pointer arithmetic is detected immediately using the up-to-date metadata.

For *temporally risky pointers*, which can result in temporal memory safety violations, LiteRSan selectively applies **I3** and **I5**. **I3** is inserted at deallocation sites (e.g., drop()) to update temporal metadata, ensuring that the temporal validity state of each pointer is accurately maintained. **I5** is applied at dereference and deallocation sites to detect temporal errors, such as use-after-free and double-free. At deallocation sites, I5 is placed before I3 to prevent I3 from prematurely marking the pointer as invalid and causing erroneous double-free reports.

For *metadata-carrying pointers*, which serve only to transmit spatial metadata (see §6.2), LiteRSan selectively applies **I2** at pointer arithmetics and container modifiers (e.g., Vec::push()), ensuring that spatial metadata is instantly updated and accurately propagated to spatially risky pointers.

6.4 Runtime Check

LiteRSan maintains and updates memory safety metadata through I1–I3, and leverages this metadata at runtime to detect spatial and temporal violations via I4 and I5, respectively. Figure 1 (see §3.1) summarizes the complete set of memory errors that LiteRSan detects and illustrates how I4 and I5 perform runtime checks.

For all risky pointers, LiteRSan first performs null checks at dereferences. For spatially risky pointers, I4 compares the pointer's *Offset* against its *Initialized Length* and *Capacity*. An access is alarmed as a use-before-initialization if the offset exceeds the initialized length, or as an out-of-bounds access if it exceeds the capacity. For temporally risky pointers, I5 consults the temporal metadata maintained by I3 to determine whether the pointer is dangling. A dereference of a dangling pointer triggers a use-after-free alarm, while a deallocation of a dangling pointer raises a double-free alarm.

Benefits of our strategy. The benefits of LiteRSan are to impose lower runtime and memory overhead while providing more comprehensive detection coverage in comparison with ASan-based approaches [8, 38]. Specifically, LiteRSan selectively instruments only the pointers that may potentially violate memory safety and inserts only necessary checks for them. These pointers are only a subset of the pointers that existing ASan-based techniques [8, 38] check. Moreover, LiteRSan can detect memory safety bugs that

existing ASan-based approaches may miss by maintaining the fine-grained spatial and temporal metadata. This metadata is compact and lightweight, contributing further to runtime efficiency.

7 IMPLEMENTATION

We implement LiteRSan on top of LLVM-14. It takes the program's LLVM bitcode as an input, performs static analysis, applies selective instrumentation, and generates instrumented LLVM bitcode.

The input bitcode is generated from Rust programs using a customized version of `rustc-1.64-nightly`. This compiler is extended to support metadata annotation during the MIR-to-LLVM IR lowering phase. Following ERASan's [38] annotation mechanism, we modify the `codegen-llvm` and `codegen-ssa` to insert LLVM metadata on instructions involving raw pointers. These annotations enable LiteRSan to identify raw pointers during static analysis, as described in §5.3. On the other hand, the output bitcode includes inserted calls to runtime functions (i.e., I1-I5 in §6.3). At runtime, the instrumented code is invoked to update metadata instantly and detect memory safety violations as the program executes.

8 EVALUATION

We evaluate LiteRSan in comparison with the two state-of-the-art Rust sanitizers, ERASan [38] and RustSan [8], as follows: runtime overhead (§8.1), memory overhead (§8.2), compilation overhead (§8.3), and bug detection capability (§8.4).

Experiment setup. All experiments were conducted on a server with an Intel Xeon Gold 6230 CPU, 80 cores, and 754 GB RAM, running Ubuntu 24.04. The benchmarks are first compiled to unoptimized LLVM IR, allowing LiteRSan and comparison tools to analyze program semantics and insert instrumentation before optimizations may alter or remove metadata. Each inserted check is tied to its corresponding memory operation and realized as a runtime function call, ensuring that subsequent LLVM optimizations do not eliminate it. The instrumented IR is then compiled with the default -O3 pipeline, reflecting realistic deployment conditions. This staging preserves analysis precision while ensuring that performance measurements correspond to practical compilation, runtime, and memory costs (further explained in Appendix E).

Benchmarks. We evaluated LiteRSan on 28 benchmarks: 26 most frequently downloaded Rust crates from `crates.io`[3] and two real-world applications (`servo` and `ripgrep`). For each benchmark, we compile and execute both the baseline versions (without instrumentation) and the instrumented versions produced by each sanitizer 20 times. The average compilation time, execution time, and memory usage are used to compute the respective overheads. For the benchmarks shared with ERASan, we use its experiment setup [23]. Thus, we use the same test cases to ensure a fair comparison. For the remaining benchmarks, we use their native test suites.

Ablation study. To decompose LiteRSan's overhead, we developed a variant Semi-LiteRSan, which uses LiteRSan's static analysis to identify risky pointers and selectively instruments runtime

---

[3]`crates.io` is Rust's official package registry. Each crate is implemented entirely in Rust and compiled as an independent unit, functioning as either a library or an executable given the benchmark input.

checks, but employs ASan's runtime checking instead of LiteRSan's metadata-based approach. Comparing Semi-LiteRSan with LiteRSan isolates the benefit of lightweight metadata while comparing Semi-LiteRSan with RustSan and ERASan highlights the benefit of our precise risky pointer identification, as RustSan and ERASan use ASan's runtime checking mechanism.

8.1 Runtime Overhead

The **Runtime Overhead** columns in Table 2 show the runtime overhead introduced by LiteRSan, ERASan, and RustSan. Across all benchmarks, LiteRSan consistently exhibits the lowest runtime overhead. By geometric mean, LiteRSan incurs a runtime overhead of only 18.84%, significantly lower than ERASan's 152.05% and RustSan's 183.50%, presenting reductions of 87.61% and 89.73%, respectively.

LiteRSan achieves significantly lower runtime overhead for two reasons. First, it uses lifetime-aware taint analysis to precisely identify risky pointers, greatly reducing unnecessary instrumentation. In contrast, comparison tools rely on points-to analysis, conservatively treating all aliases of pointers in unsafe regions (RustSan) or raw pointers (ERASan) as risky, resulting in redundant instrumentation for pointers whose safety is already guaranteed by the Rust compiler. As shown in the **Pointer Count** columns of Table 2, LiteRSan identifies greatly fewer risky pointers than the aliased counts. Second, LiteRSan employs the lightweight metadata-based runtime mechanism in place of the heavyweight ASan checks.

To quantify the contributions of these two components discussed above, we conduct an ablation study using Semi-LiteRSan, a variant of LiteRSan introduced earlier in this section. Semi-LiteRSan incurs 70.04% runtime overhead by geometric mean, demonstrating that the use of our metadata-based checking mechanism in LiteRSan reduces overhead by 73.10%. Compared to ERASan and RustSan, Semi-LiteRSan achieves reductions of 53.94% and 61.83%, highlighting the benefit of Rust-specific static analysis. We show the detailed results in Table 5 in Appendix C.

8.2 Memory Overhead

We measured memory overhead using the Linux `time` command [30], which reports the peak resident set size (max RSS) of the process. This metric captures the maximum amount of memory consumption during execution, which is widely adopted as a realistic measure of memory overhead.

In terms of memory usage, LiteRSan demonstrates a substantial advantage over both comparison tools. As shown in the **Memory Overhead** columns of Table 2, LiteRSan introduces trivial memory overhead across all benchmarks, with a geometric mean of only 0.81%. In contrast, ERASan and RustSan incur significantly higher memory overhead, reaching 739.27% and 861.98%, respectively.

These results highlight the contribution of LiteRSan's lightweight metadata, which eliminates the substantial memory overhead imposed by shadow memory and red zone mechanisms in ASan-based tools. Specifically, by comparing LiteRSan with Semi-LiteRSan (443.90% memory overhead), we can observe a 99.82% reduction attributed to the metadata-based runtime checking mechanism. Additionally, reduced instrumentation also contributes to memory savings. Semi-LiteRSan achieves 39.95% and 48.50% lower

Benchmark	LOC	Pointer Count			Compilation Overhead (%)			Runtime Overhead (%)			Memory Overhead (%)		
		Expo-raw	Risky	Aliased	LiteRSan	ERASan	RustSan	LiteRSan	ERASan	RustSan	LiteRSan	ERASan	RustSan
base64	7,025	1,787	14,320	131,242	174.17	SE	6,260.03	35.21	-	431.28	3.56	-	5,271.84
byteorder	3,411	95	355	5,391	66.52	614.35	674.93	1.72	53.36	76.37	0.03	357.73	406.27
bytes(buf)	5,867	88	376	2,904	47.94	636.77	748.59	28.39	137.90	154.33	2.68	86.27	98.58
bytes(bytes)	5,867	91	411	2,089	45.25	625.32	682.33	25.57	166.97	169.62	2.15	2,218.86	2,143.63
bytes(mut)	5,867	102	484	2,267	46.19	627.09	755.26	27.82	157.28	165.48	2.19	5,376.09	5,339.28
indexmap	8,693	386	2,214	32,132	103.59	5,807.79	2,310.17	23.06	287.14	293.65	1.78	1,754.14	2,090.46
itoa	613	9	32	291	47.76	334.31	414.24	20.34	116.11	131.05	1.43	65.71	69.83
memchr	1,139	50	185	3,133	86.14	514.86	560.28	13.18	212.39	217.74	1.96	49.61	61.56
num-integer	2,383	570	3,095	8,449	186.57	1,359.64	1,512.04	1.17	5.59	8.34	0.02	524.07	674.62
ryu	3,443	17	82	2,247	64.03	868.87	931.11	17.71	63.80	70.89	0.28	81.69	85.22
semver	2,483	24	81	839	42.34	390.24	451.72	4.83	317.21	388.52	1.88	6,832.92	7,683.53
smallvec	2,912	59	278	982	59.21	422.05	489.63	13.34	134.53	152.33	1.14	4,370.14	4,518.99
strsim-rs	1,102	109	431	1,015	58.25	452.52	528.97	1.06	380.51	389.72	1.36	5,568.87	5,729.60
uuid(format)	4,971	15	50	62,149	207.82	9,230.63	2,669.96	40.62	362.41	411.04	0.15	875.22	879.71
uuid(parse)	4,971	15	50	62,091	202.32	9,038.49	2,684.37	37.31	338.06	402.53	0.15	1,065.03	1,094.13
bat	53,517	2,567	25,546	138,726	167.91	25,020.17	4,826.05	321.11	894.97	931.36	4.96	4,619.51	5,017.39
crossbeam-utils	31,246	64	290	2,227	104.06	586.35	639.52	1.28	116.09	136.17	0.05	57.85	58.97
hashbrown	10,384	51	383	6,596	72.16	1,227.80	1,182.08	9.32	58.65	69.45	0.37	6,613.42	6,814.56
hyper	20,952	1,824	15,201	114,269	184.98	20,920.41	5,127.40	43.57	278.16	297.23	3.69	2,681.30	2,966.32
rand(generators)	15,220	27	78	2,619	84.27	776.41	835.54	31.85	26.49	31.64	0.26	85.18	88.64
rand(misc)	15,220	66	258	2,527	95.93	1,079.36	866.17	7.94	10.12	23.47	0.22	1,457.08	1,654.97
regex	65,417	294	3,896	6,383	128.70	1,463.51	923.09	38.54	831.87	867.34	1.83	8,191.68	8,574.25
ripgrep	33,226	1,864	18,734	-	142.07	SEGV	SEGV	304.03	-	-	4.77	-	-
syn	58,884	1,088	23,034	186,730	123.84	25,397.17	1,499.46	72.29	583.25	618.92	1.65	327.11	343.74
tokio	69,875	1,482	19,375	74,697	149.02	18,607.29	4,965.73	53.35	563.24	593.22	1.33	1,343.17	1,504.36
unicode	172,875	95	363	1,054	54.95	549.73	677.47	8.89	47.96	62.37	0.86	56.83	63.38
url	40,595	353	2,266	34,368	191.89	1,618.63	1,521.94	21.06	612.75	686.41	1.37	312.35	356.88
servo	11.26 M	1.27 M	14.63 M	-	186.13	TO	TO	86.58	-	-	2.82	-	-
GeoMean	-	-	-	-	97.21	1,635.35	1,193.31	18.84	152.05	183.50	0.81	739.27	861.98

Table 2: Pointer counts and overhead comparison. Benchmarks are grouped by scale. Pointer counts report exposed raw pointers (Expo-raw) and risky pointers identified by LiteRSan, along with raw pointers plus aliases (Aliased) identified by traditional points-to analysis. Overheads are shown for LiteRSan, ERASan, and RustSan. Nonapplicable results are listed as -. SE indicates silent exit, SEGV indicates segmentation fault, and TO indicates a compilation timeout.

memory overhead than ERASan and RustSan, respectively. We describe their details in Table 5 in Appendix C.

8.3 Compilation Overhead

Compilation overhead refers to the additional compilation time introduced by a sanitizer's static analysis and instrumentation compared to the baseline build. As shown in the **Compilation Overhead** columns in Table 2, LiteRSan consistently incurs significantly lower overhead. By geometric mean, LiteRSan produces 97.21% overhead, compared to 1,635.35% for ERASan and 1,193.31% for RustSan, presenting reductions of 94.06% and 91.85%, respectively. Both comparison tools failed to complete compilation for servo within a 24-hour timeout, and encountered a segmentation fault when analyzing ripgrep due to SVF errors.

LiteRSan achieves lower compilation overhead than RustSan and ERASan because both of them rely on SVF, which is a heavyweight points-to analyzer, as discussed in §2.2. In contrast, LiteRSan adopts a lightweight static analysis tailored to Rust. Despite its low cost, this analysis is sufficient to identify all spatially and temporally risky pointers, enabling selective instrumentation with modest compilation overhead.

8.4 Security Evaluation

In addition to performance improvement, LiteRSan provides a more comprehensive memory error detection coverage than ERASan and RustSan, both of which share the same capability as ASan. Therefore, we show the bug detection capability of LiteRSan and compare it only with ASan (whose detailed approach and limitations are discussed in §2.3).

We analyzed bugs reported by RustSec—the Rust Security Advisory Database [77]—over the past two years, focusing on the cases where bug root causes (i.e., PoCs) are publicly available for validation. We list memory safety bugs in our scope (discussed in §3.1) in Table 3. LiteRSan successfully detects all of 20 bugs, whereas ASan fails to identify two out-of-bounds access bugs, one use-before-initialization bug, and one use-after-free bug. These cases occur in Rust-specific contexts (illustrated as case studies in §8.4.1 and §8.4.2). Because ASan was originally designed for C/C++, it effectively detects conventional memory safety violations but lacks the ability to handle Rust-specific memory safety rules and check per-pointer spatial and temporal memory safety. In contrast, LiteRSan incorporates Rust's memory safety rules in its static analysis and enforces per-pointer spatial and temporal memory safety checks, enabling the detection of such missing bugs. We illustrate one spatial memory safety bug in §8.4.1 and one temporal memory safety bug in §8.4.2.

8.4.1 Case Study 1. RUSTSEC-2023-0056 [63] is an out-of-bounds access vulnerability in the vm-memory crate [52]. In this crate, get_slice is a trait method intended to return a smart pointer-like abstraction, VolatileSlice, over a slice, but it lacks a default implementation. If a user implements this method incorrectly, for example, by miscomputing the offset or count, the internal pointer in the returned VolatileSlice may reference memory outside the intended region, potentially leading to out-of-bounds access.

Several methods in VolatileMemory trait, such as get_ref and get_array_ref, invoke get_slice without proper bounds checking, thereby raising potential memory safety violations. Listing 2

RUSTSEC ID	Type	Class	ASan	LITERSAN
RUSTSEC-2023-0021	NPD	Null-pointer deref	✓	✓
RUSTSEC-2023-0024	NPD	Null-pointer deref	✓	✓
RUSTSEC-2023-0038	OOB	Spatial	✓	✓
RUSTSEC-2023-0039	OOB	Spatial	✓	✓
RUSTSEC-2023-0056	OOB	Spatial	✗	✓
RUSTSEC-2024-0002	OOB	Spatial	✗	✓
RUSTSEC-2025-0003	OOB	Spatial	✓	✓
RUSTSEC-2025-0005	OOB	Spatial	✓	✓
RUSTSEC-2025-0018	OOB	Spatial	✓	✓
RUSTSEC-2023-0045	UBI	Spatial	✓	✓
RUSTSEC-2023-0087	UBI	Spatial	✗	✓
RUSTSEC-2024-0018	UBI	Spatial	✓	✓
RUSTSEC-2024-0374	UBI	Spatial	✓	✓
RUSTSEC-2024-0400	UBI	Spatial	✓	✓
RUSTSEC-2023-0010	DF	Temporal	✓	✓
RUSTSEC-2023-0078	UAF	Temporal	✗	✓
RUSTSEC-2024-0007	UAF	Temporal	✓	✓
RUSTSEC-2024-0017	UAF	Temporal	✓	✓
RUSTSEC-2025-0016	UAF	Temporal	✓	✓
RUSTSEC-2025-0022	UAF	Temporal	✓	✓

Table 3: Bug detection capability of ASan and LITERSAN. Listed are the 20 most recent memory safety vulnerabilities in RustSec, grouped by bug class. Additional results on older vulnerabilities are provided in Appendix A.

illustrates this issue using `get_atomic_ref` as an example. In line 6, `get_slice` is invoked to wrap an allocated memory region with a requested size of `size_of::<T>()` bytes. In line 9, the internal pointer of the returned `VolatileSlice` (i.e., `slice.addr`) is cast and dereferenced without verifying whether the underlying memory actually aligns with the requested bounds. If `get_slice` returns a region smaller than the requested region, any dereference beyond the actual region results in an out-of-bounds access.

```
1  fn get_slice(&self, offset: usize, count: usize)
2                      -> Result<VolatileSlice<BS<Self::B>>>;
3
4  fn get_atomic_ref<T: AtomicInteger>(&self, offset: usize)
5                                      -> Result<&T> {
6      let slice = self.get_slice(offset, size_of::<T>())?;
7      slice.check_alignment(align_of::<T>())?;
8
9      unsafe { Ok(&*(slice.addr as *const T)) }
10 }
```
Listing 2: Potential OOB in `get_atomic_ref`.

According to our experiment, ASan cannot detect this bug because it only places red zones around memory objects. However, in this case, the pointer returned by `get_slice` may point to a valid memory object, but beyond the actual valid bound, which is within this object. As a result, *invalid* accesses beyond the actual bound but within the larger allocated object remain undetected by ASan, since no red zones are placed at the logical boundary returned by `get_slice`. In contrast, LITERSAN tracks memory safety metadata for each pointer at its definition site. This allows LITERSAN to precisely extract the actual bound of `slice.addr` and perform spatial memory safety checks, detecting potential out-of-bounds access.

8.4.2 Case Study 2. RUSTSEC-2023-0078 [64] is a use-after-free vulnerability reported in the `tracing` crate [51]. As shown in Listing 3, the vulnerability originates from the improper use of `mem::forget` in line 4, where the exclusive owner of the underlying memory object is forgotten. While `mem::forget` prevents the object's destructor from being called, the Rust compiler considers the object to be logically invalid after its owner is forgotten. The

memory region may subsequently be reused by the compiler, making any future access to the original object via existing pointers a use-after-free violation.

```
1  pub fn into_inner(self) -> T {
2      let span: *const Span = &self.span;
3      let inner: *const ManuallyDrop<T> = &self.inner;
4      mem::forget(self);
5
6      let _span = unsafe { span.read() };
7      let inner = unsafe { inner.read() };
8      ManuallyDrop::into_inner(inner)
9  }
```
Listing 3: Potential UAF in `Instrumented::into_inner`.

This vulnerability stems from a violation of Rust's ownership model rather than traditional heap misuse found in C/C++. Because the memory is never explicitly freed, ASan does not update its shadow memory to mark the region as invalid, thus fails to detect the temporal safety violations. Covering this type of vulnerability in ASan is fundamentally challenging as ASan is unaware of Rust's ownership semantics. To detect such bugs, ASan would need to determine whether an object still has a valid owner at every program point, which requires a significant change in the underlying design of ASan. In contrast, LITERSAN is designed with ownership awareness. It tracks ownership and marks the pointers referencing the same object as dangling when the last owner is dropped. Any subsequent dereferences of the dangling pointers are flagged as use-after-free. This allows LITERSAN to detect ownership-related memory safety violations that lie beyond ASan's capabilities.

9 DISCUSSION

Type conversion bugs. We consider type conversion bugs, such as those introduced via unsafe APIs like `transmute()` [54], out of scope, as it is widely accepted as orthogonal to spatial and temporal memory safety. The same view is shared by many prior works [7, 14, 47]. Type conversion bugs stem from reinterpreting one type as another, which can break safety invariants without violating spatial bounds or temporal validity. As a result, LITERSAN may not be able to detect them if they do not violate spatial bounds or temporal validity. State-of-the-art ASan-based tools [8, 38] also share the same problem [67]. One way to address this problem is to integrate type confusion bug detection techniques [7]. But it is worth noting that LITERSAN is able to detect such type confusion bugs if they stem from memory errors such as UAF.

Cross-language attacks. LITERSAN leverages Rust's ownership and borrowing semantics to infer memory safety metadata and enforce spatial and temporal safety. As a result, it does not guarantee the detection of memory safety violations originating from external code written in languages without such semantics, such as C/C++ libraries interfaced via FFI. Similar to ERASan and RustSan, LITERSAN does not cover cross-language memory safety violations, which are considered out of scope.

To address cross-language attacks, one potential direction is to integrate LITERSAN with existing isolation or sandboxing techniques [68, 71], to mitigate memory errors originating from external

code. Another direction is to extend the scope of LITERSAN to external libraries and enforce runtime checks at FFI boundaries. However, this requires a deep understanding of the semantics of each external API, which is difficult to generalize and automate. It also requires static analysis on C/C++ code, which lacks Rust's safety guarantees, making Rust-specific analysis inapplicable. Despite these challenges, exploring support for cross-language memory safety is a promising direction for future work.

10 RELATED WORK

Memory sanitizing. LITERSAN is closely related to ERASan [38] and RustSan [8], both of which are ASan-based [65] tools that detect memory safety errors at runtime. These tools rely on SVF [69] to identify pointers aliases with raw pointers [38] or used in unsafe code [8]. They retain ASan checks only for these potentially unsafe pointers to improve performance compared to ASan. Both tools provide the same memory safety guarantee as ASan.

In contrast, LITERSAN offers more comprehensive safety guarantees while incuring drastically lower runtime and memory overhead than ERASan and RustSan. Additionally, LITERSAN's compile time significantly outperforms that of ERASan and RustSan (§8), due to precise risky pointer identification for less instrumentation and lightweight runtime checks. SVF, in contrast, computes complete alias information for the whole program, thus, is computationally expensive and unscalable for large programs [21], resulting in high overhead in its client, such as ERASan and RustSan.

Memory isolation. Another major line of mitigations against unsafe code—including both unsafe Rust source code and external C/C++ libraries—is memory isolation for protecting safe Rust memory. Both XRust [31] and Trust [5] utilize SVF [69] to identify pointer dereferences of memory used by unsafe Rust code. XRust employs bounds checking, while Trust leverages Intel MPK [18] to enforce isolation between memory exclusively accessed by safe Rust code and memory accessed by unsafe code. Similarly, MetaSafe [20] protects smart pointer metadata (e.g., length of `String`) by storing them in a dedicated memory region and isolating that region using Intel MPK. PKRU-Safe [21] performs dynamic analysis to find unsafe memory accesses, motivated by concerns over SVF's performance, and also enforces isolation via Intel MPK. Sandcrust [24] restricts external C library code by running it in a separate process. Fidelius Charm [1] migrates target sensitive data to and from protected pages before and after invoking untrusted C libraries.

They intend to mitigate the impact of unsafe code but allow memory errors within it. In addition to preventing unsafe Rust code from affecting safe code, LITERSAN can detect memory safety bugs within unsafe code as well as "safe" code. Such bugs (e.g., UAF on a smart pointer) can arise due to issues originating from unsafe code (§5.4). Nevertheless, LITERSAN does not prevent directly-linked C/C++ library code from compromising Rust programs.

Bug detection via static analysis. Static analysis is also actively explored to detect bugs in Rust programs. Rudra [4] identifies three common memory safety bug patterns and performs static analysis based on these patterns. Similarly, MirChecker [28] also learns existing bug patterns and utilizes Abstract Interpretation techniques for bug detection. Both Rupair [17] and SafeDrop [9] employ data-flow

analysis: Rupair addresses buffer overflows while SafeDrop focuses on detecting invalid memory deallocations. FFIChecker [29] also targets memory management errors specifically caused by interactions of Rust and external libraries through the Foreign Function Interface (FFI). SyRust [72], on the other hand, uses program synthesis to generate test cases for testing Rust library APIs.

These tools effectively detect their respective types of bugs, but suffer from high false positive rates. For example, Rudra [4] reports the most bugs among these tools but has a false positive rate as high as 89%. In contrast, LITERSAN does not suffer from false positives—namely, each reported bug corresponds to a real memory safety violation, though LITERSAN may introduce redundant memory-safe checks due to the conservative nature of static analysis. Additionally, LITERSAN maintains lightweight analysis time (§8.3), whereas static analysis tools often impose prohibitive analysis time for the sake of broader code coverage.

11 CONCLUSION

Rust provides strong memory safety through its ownership semantics and type system. However, these guarantees can be undermined by the use of unsafe code, which reintroduces memory safety vulnerabilities. To detect such bugs, ASan-based tools are commonly used. Yet, even state-of-the-art sanitizers like ERASan and RustSan incur substantial performance and memory overhead, and still fail to catch certain memory safety violations.

Therefore, we propose a novel Rust memory sanitizer, LITERSAN, with lower overhead and more comprehensive and accurate memory error detection than ERASan and RustSan. We achieve this goal by precisely identifying risky pointers and selectively instrumenting those risky pointers to minimize overhead while ensuring higher detection coverage than ERASan and RustSan. As a result, LITERSAN imposes 18.84% runtime overhead, 97.21% compilation overhead, and 0.81% memory overhead, with geometric mean, while ERASan and RustSan, respectively, incur 152.05% and 183.50% runtime overhead, 1635.35% and 1193.31% compilation overhead, and 739.27% and 861.98% memory overhead. Furthermore, LITERSAN detects 55 memory safety vulnerabilities with 100% accuracy, unlike ASan-based approaches that miss four of them.

REFERENCES

[1] Hussain M. J. Almohri and David Evans. 2018. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY '18)*. Association for Computing Machinery, New York, NY, USA, 248–255. https://doi.org/10.1145/3176258.3176330

[2] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428204

[3] AWS. 2025. Verify the Safety of the Rust Standard Library. https://aws.amazon.com/blogs/opensource/verify-the-safety-of-the-rust-standard-library/. (2025).

[4] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. 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)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3477132.3483570

[5] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. 2023. TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6947–6964. https://www.usenix.org/conference/usenixsecurity23/presentation/bang

[6] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating residual risk in greybox fuzzing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 230–241. https://doi.org/10.1145/3468264.3468570

[7] Hung-Mao Chen, Xu He, Shu Wang, Xiaokuan Zhang, and Kun Sun. 2025. TypePulse: Detecting Type Confusion Bugs in Rust Programs. (2025). arXiv:cs.CR/2502.03271 https://arxiv.org/abs/2502.03271.

[8] Kyuwon Cho, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee. 2024. RustSan: Retrofitting AddressSanitizer for Efficient Sanitization of Rust. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3729–3746. https://www.usenix.org/conference/usenixsecurity24/presentation/cho-kyuwon

[9] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 82 (May 2023), 21 pages. https://doi.org/10.1145/3542948

[10] Jansens Dana. 2021. Supporting the Use of Rust in the Chromium Project. (2021). https://security.googleblog.com/2021/09/supporting-use-of-rust-in-chromium.html.

[11] Alain Deutsch. 1994. Interprocedural may-alias analysis for pointers: beyond k-limiting. *SIGPLAN Not.* 29, 6 (June 1994), 230–241. https://doi.org/10.1145/773473.178263

[12] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/3377811.3380413

[13] Krzysztof Grajek. 2024. Rust Static vs. Dynamic Dispatch. (2024). https://softwaremill.com/rust-static-vs-dynamic-dispatch/.

[14] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2016. TypeSan: Practical Type Confusion Detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 517–528. https://doi.org/10.1145/2976749.2978405

[15] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. Association for Computing Machinery, New York, NY, USA, 54–61. https://doi.org/10.1145/379605.379665

[16] Susan Horwitz. 1997. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.* 19, 1 (Jan. 1997), 1–6. https://doi.org/10.1145/239912.239913

[17] Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. 2021. Rupair: Towards Automatic Buffer Overflow Detection and Rectification for Rust. In *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 812–823. https://doi.org/10.1145/3485832.3485841

[18] Intel Corporation 2021. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. Order Number: 253665-075US.

[19] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages.

[20] Martin Kayondo, Inyoung Bang, Yeongjun Kwak, HyunGon Moon, and Yunheung Paek. 2024. MetaSafe: Compiling for Protecting Smart Pointer Metadata to Ensure Safe Rust Integrity. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3711–3728. https://www.usenix.org/conference/usenixsecurity24/presentation/kayondo

[21] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 17.

[22] Steve Klabnik and Carol Nichols. 2022. *The Rust Programming Language*. https://doc.rust-lang.org/stable/book/

[23] S2 Lab. 2025. Github : ERASan. (2025). https://github.com/S2-Lab/ERASan

[24] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS'17)*. Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/3144555.3144562

[25] The Rust Programming Language. 2024. Understanding Ownership. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html. (2024).

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO '04)*. IEEE Computer Society, Palo Alto, CA, 12. https://doi.org/10.1109/CGO.2004.1281665

[27] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. 2023. A Hybrid Alias Analysis and Its Application to Global Variable Protection in the Linux Kernel. In *32nd USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity23/presentation/li-guoren

[28] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2183–2196. https://doi.org/10.1145/3460120.3484541

[29] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. 2022. Detecting Cross-language Memory Management Issues in Rust. In *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*. Springer-Verlag, Berlin, Heidelberg, 680–700. https://doi.org/10.1007/978-3-031-17143-7_33

[30] Linux 2025. time(1) — Linux manual page. (2025). https://man7.org/linux/man-pages/man1/time.1.html.

[31] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing Unsafe Rust Programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/3377811.3380325

[32] LLVM. LLVM Language Reference Manual. (????). https://llvm.org/docs/LangRef.html https://llvm.org/docs/LangRef.html.

[33] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

[34] Giorgio Martinez. 2023. Unlocking Performance: Optimizing Rust's Dynamic Dispatch. (2023). https://medium.com/@giorgio.martinez1926/unlocking-performance-optimizing-rusts-dynamic-dispatch-600b57f78f99.

[35] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, USA.

[36] Microsoft. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. (2019).

[37] Shane Miller and Carl Lerche. 2022. Sustainability with Rust. (2022). https://aws.amazon.com/blogs/opensource/sustainability-with-rust/

[38] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. 2024. ERASan: Efficient Rust Address Sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4053–4068. https://doi.org/10.1109/SP54263.2024.00258

[39] MITRE 2018. CVE-2018-1000810. (2018). https://www.cve.org/CVERecord?id=CVE-2018-1000810.

[40] MITRE 2019. CVE-2019-16760. (2019). https://www.cve.org/CVERecord?id=CVE-2019-16760.

[41] Amit Nadiger. 2023. Dynamic & Static Dispatch in Rust. (2023). https://www.linkedin.com/pulse/dynamic-static-dispatch-rust-amit-nadiger/.

[42] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. In *2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[43] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler-Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. ACM, 31–40.

[44] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2005. *Principles of Program Analysis*. Springer.

[45] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2014). https://api.semanticscholar.org/CorpusID:2282679

[46] NSA-CSS. 2022. NSA Releases Guidance on How to Protect Against Software Memory Safety Issues. (2022).

[47] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. 2018. Mapping to Bits: Efficiently Detecting Type Confusion Errors. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. 518–528.

[48] Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. 2024. Understanding and Detecting Real-World Safety Issues in Rust. *IEEE Trans. Softw. Eng.* 50, 6 (March 2024), 1306–1324. https://doi.org/10.1109/TSE.2024.3380393

[49] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

[50] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *Annual Computer Security Applications Conference (ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 824–836. https://doi.org/10.1145/3485832.3485903

[51] Rust 2025. Crate tracing. (2025). https://crates.io/crates/tracing.

[52] Rust 2025. Crate vm-memory. (2025). https://crates.io/crates/vm-memory.

[53] Rust 2025. Function transmute. (2025). https://doc.rust-lang.org/std/mem/fn.transmute.html.

[54] Rust 2025. Function transmute. (2025). https://doc.rust-lang.org/std/mem/fn.transmute.html.

[55] Rust 2025. Method as_ptr. (2025). https://doc.rust-lang.org/std/vec/struct.Vec.html#method.as_ptr.

[56] Rust 2025. Method set_len. (2025). https://doc.rust-lang.org/std/vec/struct.Vec.html#method.set_len.

[57] Rust 2025. Rust Compiler Development Guide. (2025). https://rustc-dev-guide.rust-lang.org/mir/index.html.

[58] Rust 2025. Scoping rules. (2025). https://doc.rust-lang.org/rust-by-example/scope.html.

[59] Rust 2025. Trait Drop. (2025). https://doc.rust-lang.org/std/ops/trait.Drop.html.

[60] Rust-fuzz 2025. RUST Fuzzing : afl.rs. (2025). https://github.com/rust-fuzz/afl.rs.

[61] Rust-fuzz 2025. RUST Fuzzing : cargo fuzz. (2025). https://github.com/rust-fuzz/cargo-fuzz.

[62] Rust-fuzz 2025. RUST Fuzzing : honggfuzz-rs. (2025). https://github.com/rust-fuzz/honggfuzz-rs.

[63] Rustsec 2023. RUSTSEC-2023-0056. (2023). https://rustsec.org/advisories/RUSTSEC-2023-0056.html.

[64] Rustsec 2023. RUSTSEC-2023-0078. (2023). https://rustsec.org/advisories/RUSTSEC-2023-0078.html.

[65] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference*. USENIX Association, Boston, MA, 309–318.

[66] Servo 2019. Servo: The Parallel Browser Engine Project. (2019). https://servo.org/.

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

[68] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijin Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA.

[69] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*. 265–266.

[70] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. 48–62. https://doi.org/10.1109/SP.2013.13

[71] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62.

[72] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2021. SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 899–913.

[73] Adrian Taylor, Andrew Whalley, Dana Jansens, and Nasko Oskov. 2021. An update on Memory Safety in Chrome. https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html. (2021).

[74] The White House. 2024. Press Release: Future Software Should Be Memory Safe. https://bidenwhitehouse.archives.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/. (2024).

[75] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*.

[76] Emanuel Vintila, Philipp Zieris, and Julian Horsch. 2025. Evaluating the Effectiveness of Memory Safety Sanitizers . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 88–88.

[77] Secure Code working group. 2025. RustSec Advisory Database. (2025). https://rustsec.org/.

[78] Tianrou Xia, Hong Hu, and Dinghao Wu. 2024. DEEPTYPE: Refining Indirect Call Targets with Strong Multi-layer Type Analysis. In *33rd USENIX Security Symposium (USENIX Security 24)*.

[79] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 2021. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 3 (sep 2021), 25 pages.

[80] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. 2024. RPG: Rust library fuzzing with pool-based fuzz target generation and generic support. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[81] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. 2023. Towards Understanding the Runtime Performance of Rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 140, 6 pages.

A CONTINUED BUG LIST

Table 4 reports the detection results of LITERSAN and ASan on earlier RustSec vulnerabilities, complementing Table 3. Together, these tables cover all publicly disclosed memory safety bugs reported in the RustSec Advisory Database to date. In total, the combined dataset includes 55 vulnerabilities: 21 use-after-free (UAF), 3 double-free (DF), 21 out-of-bounds accesses (OOB), 7 use-before-initialization (UBI), and 3 null-pointer dereference (NPD). They also cover all the vulnerabilities experimented by ERASan.

As a result, LITERSAN successfully detects all the listed bugs identified by ASan, demonstrating full coverage of ASan's detection capabilities on Rust memory safety bugs. Furthermore, as shown in Table 3, LITERSAN detects additional Rust-specific bugs, including UBI and certain safe-code out-of-bounds violations. ASan fails to capture due to its reliance on coarse-grained shadow memory and red-zone mechanisms, while LITERSAN deploys a metadata-based solution with respect to Rust's type model.

RUSTSEC ID	Type	Class	ASan	LITERSAN
RUSTSEC-2020-0061	NPD	Null-pointer deref	✓	✓
RUSTSEC-2023-0013	NPD	Null-pointer deref	✓	✓
RUSTSEC-2020-0039	OOB	Spatial	✓	✓
RUSTSEC-2020-0167	OOB	Spatial	✓	✓
RUSTSEC-2021-0003	OOB	Spatial	✓	✓
RUSTSEC-2021-0048	OOB	Spatial	✓	✓
RUSTSEC-2021-0094	OOB	Spatial	✓	✓
RUSTSEC-2023-0015	OOB	Spatial	✓	✓
RUSTSEC-2023-0016	OOB	Spatial	✓	✓
RUSTSEC-2023-0030	OOB	Spatial	✓	✓
RUSTSEC-2023-0032	OOB	Spatial	✓	✓
RUSTSEC-2019-0023	UAF	Temporal	✓	✓
RUSTSEC-2020-0005	UAF	Temporal	✓	✓
RUSTSEC-2020-0060	UAF	Temporal	✓	✓
RUSTSEC-2020-0091	UAF	Temporal	✓	✓
RUSTSEC-2020-0097	UAF	Temporal	✓	✓
RUSTSEC-2022-0070	UAF	Temporal	✓	✓
RUSTSEC-2022-0078	UAF	Temporal	✓	✓
RUSTSEC-2023-0005	UAF	Temporal	✓	✓
RUSTSEC-2023-0009	UAF	Temporal	✓	✓
RUSTSEC-2021-0031	UAF	Temporal	✓	✓
RUSTSEC-2021-0128	UAF	Temporal	✓	✓
RUSTSEC-2021-0130	UAF	Temporal	✓	✓
RUSTSEC-2019-0009	DF	Temporal	✓	✓
RUSTSEC-2019-0034	DF	Temporal	✓	✓
RUSTSEC-2020-0038	DF	Temporal	✓	✓
RUSTSEC-2021-0018	DF	Temporal	✓	✓
RUSTSEC-2021-0028	DF	Temporal	✓	✓
RUSTSEC-2021-0033	DF	Temporal	✓	✓
RUSTSEC-2021-0039	DF	Temporal	✓	✓
RUSTSEC-2021-0042	DF	Temporal	✓	✓
RUSTSEC-2021-0047	DF	Temporal	✓	✓
RUSTSEC-2021-0053	DF	Temporal	✓	✓

Table 4: Detection capability of ASan and LITERSAN on memory safety vulnerabilities, grouped by bug class. They are memory safety vulnerabilities discovered and registered in RustSec earlier than 20 memory safety vulnerabilities in Table 3.

B TYPE (2) UNSAFE APIS

We list the Type (2) unsafe APIs that may violate memory safety, within our scope, without involving raw pointers: `unchecked_add/sub/mul/neg`, `forward/backward_unchecked`, `unchecked_shl/shr`, and `set_len`. LITERSAN handles them by inserting bounds or validity checking at their call sites.

C ABLATION STUDY RESULTS

The **Pointer Count** column in Table 5 reports the number of risky pointers identified by LITERSAN, the total number of pointers guarded by ASan. Across all benchmarks, the number of risky pointers is substantially lower than that of ASan-guarded pointers, indicating that most pointers in Rust are guaranteed to be safe, ASan checks are excessively redundant.

Table 5 also presents the detailed runtime and memory overhead of SEMI-LITERSAN, for ablation study. These results allow for two key comparisons: (1) with LITERSAN, to quantify the impact of lightweight metadata-based runtime checking mechanism, and (2) with ASan-based tools, to evaluate the effectiveness of Rust-specific static analysis. Overall, both components significantly contribute to reducing runtime and memory overhead.

D COMPARISON OF LITERSAN AND ASAN

LITERSAN significantly reduces the number of instrumented pointers compared to ASan by leveraging precise Rust-specific static analysis to identify risky pointers. As shown in Table 5, LITERSAN instruments much less pointers than LITERSAN across all benchmarks, yielding great runtime and memory performance improvement.

Specifically, ASan incurs 359.90% runtime overhead, while SEMI-LITERSAN incurs 70.04%, indicating that our precise risky pointer identification and selective instrumentation reduce overhead by 80.54%. LITERSAN further lowers the overhead to 18.94%, demonstrating that replacing ASan's heavyweight shadow memory and red zones with our lightweight metadata yields an additional 73.10% reduction. As for memory usage, ASan introduces 3,282.12% overhead. SEMI-LITERSAN lowers it to 443.90%, through precise static analysis and selective instrumentation, while LITERSAN only presents negligible overhead of 0.81%, owing to the lightweight metadata-based run checks.

E BENCHMARK COMPILATION OPTIONS

To ensure both accurate static analysis and evaluation on realistic production situation, we adopt a staged compilation process. For each benchmark, the compiler first emits LLVM IR with inlining and LLVM prepopulate passes disabled so that MIR-derived annotations are preserved in the IR. At this stage, LITERSAN and the comparison tools (ERASan and RustSan) perform static analysis and insert their respective instrumentation. After instrumentation, compilation resumes with the standard optimization pipeline to produce optimized (i.e., `-O3`) executables.

This approach is essential rather than a shortcut. Running LLVM optimizations before analysis can replace or eliminate original instructions and drop critical metadata, leading to missed identification of risky operations. This metadata-preservation challenge is not unique to LITERSAN but equally affects ERASan and RustSan; analyzing directly on optimized IR would cause all these tools to miss protecting unsafe operations. By applying analysis on pre-optimized IR, we preserve full semantic information and ensure that risky pointers are precisely identified.

Benchmark	Pointer Count		Runtime Overhead (%)			Memory Overhead (%)		
	Risky	ASan-guarded	LITERSAN	SEMI-LITERSAN	ASan	LITERSAN	SEMI-LITERSAN	ASan
base64	14,320	1,075,072	35.21	89.27	624.71	3.56	3,015.06	10,723.03
byteorder	355	24,275	1.72	17.75	131.23	0.03	248.62	1,065.10
bytes(buf)	376	37,783	28.39	78.27	289.20	2.68	37.63	411.25
bytes(bytes)	411	18,354	25.57	79.32	292.98	2.15	1,044.84	21,368.49
bytes(mut)	484	26,796	27.82	79.64	295.37	2.19	3,433.34	64,467.47
indexmap	2,214	378,711	23.06	87.86	419.93	1.78	1,233.98	4,620.00
itoa	32	5,195	20.34	88.56	241.11	1.43	28.21	123.68
memchr	185	16,633	13.18	53.17	342.19	1.96	39.85	81.62
num-integer	3,095	75,990	1.17	5.77	35.12	0.02	416.44	769.33
ryu	82	9,769	17.71	52.62	101.45	0.28	66.26	92.20
semver	81	4,787	4.83	32.91	536.83	1.88	4,686.86	13,347.98
smallvec	278	13,736	13.34	53.71	284.08	1.14	2,863.90	78,376.98
strsim-rs	431	4,038	1.06	26.63	522.02	1.36	3,943.39	9,869.13
uuid(format)	50	730,713	40.62	75.66	481.02	0.15	157.72	1,008.17
uuid(parse)	50	731,856	37.31	96.46	467.23	0.15	166.83	1,239.21
bat	25,546	1,859,420	321.11	542.13	1,187.60	4.96	1,817.64	36,539.25
crossbeam-utils	290	5,695	1.28	16.19	187.15	0.05	41.91	548.96
hashbrown	383	67,106	9.32	35.51	124.61	0.37	4,067.88	28,530.65
hyper	15,201	737,542	43.57	84.70	323.03	3.69	1,752.87	35,747.78
rand(generators)	78	17,807	31.85	55.77	151.86	0.26	69.77	378.25
rand(misc)	258	9,632	7.94	30.49	131.42	0.22	1,050.56	8,025.48
regex	3,896	49,383	38.54	243.58	1,584.07	1.83	6,739.62	37,082.47
ripgrep	18,734	962,161	304.03	524.54	1,210.16	4.77	51.02	28,728.19
syn	23,034	1,770,205	72.29	278.64	1,390.21	1.65	33.68	609.24
tokio	19,375	728,064	53.35	108.46	914.47	1.33	769.19	1,843.51
unicode	363	7,715	8.89	42.59	157.94	0.86	41.17	333.48
url	2,266	255,893	21.06	85.74	937.53	1.37	206.59	791.42
servo	14.63 M	16,994,787	86.58	218.05	1,281.96	2.82	8,174.46	52,329.43
GeoMean	-	-	**18.84**	70.04	359.90	**0.81**	443.90	3,282.12

Table 5: Comparison of LITERSAN, SEMI-LITERSAN, and ASan. The table shows pointer counts (risky and ASan-guarded), runtime overhead, and memory overhead across benchmarks.

At the same time, compiling instrumented IR under -O3 guarantees that our evaluation reflects realistic deployment conditions. Most LLVM optimizations transform instructions in place rather than reordering them, so checks remain associated with the correct memory operations. Furthermore, because LITERSAN implements its checks as runtime calls with observable actual effects, subsequent LLVM optimizations do not remove them. Our experiments demonstrate this property in practice, as LITERSAN achieves 100% bug detection even when executing fully optimized binaries.

In summary, this staging is a necessary and fair methodology: it preserves metadata for accurate analysis, ensures that checks are preserved under aggressive optimization, and yields performance results under realistic deployment. Future improvements in preserving Rust-specific metadata across optimization could streamline this process, but the present design is the only viable way to ensure correctness across Rust memory safety sanitizers.