deepSURF: Detecting Memory Safety Vulnerabilities in Rust Through Fuzzing LLM-Augmented Harnesses

Georgios Androutsopoulos
Purdue University
gandrout@purdue.edu
West Lafayette, Indiana, USA

Antonio Bianchi
Purdue University
antoniob@purdue.edu
West Lafayette, Indiana, USA

***Abstract*—Although Rust ensures memory safety by default, it also permits the use of unsafe code, which can introduce memory safety vulnerabilities if misused. Unfortunately, existing tools for detecting memory bugs in Rust typically exhibit limited detection capabilities, inadequately handle Rust-specific types, or rely heavily on manual intervention.**

To address these limitations, we present deepSURF, a tool that integrates static analysis with Large Language Model (LLM)-guided fuzzing harness generation to effectively identify memory safety vulnerabilities in Rust libraries, specifically targeting unsafe code. deepSURF introduces a novel approach for handling generics by substituting them with custom types and generating tailored implementations for the required traits, enabling the fuzzer to simulate user-defined behaviors within the fuzzed library. Additionally, deepSURF employs LLMs to augment fuzzing harnesses dynamically, facilitating exploration of complex API interactions and significantly increasing the likelihood of exposing memory safety vulnerabilities. We evaluated deepSURF on 27 real-world Rust crates, successfully rediscovering 20 known memory safety bugs and uncovering 6 previously unknown vulnerabilities, demonstrating clear improvements over state-of-the-art tools.

1. Introduction

Rust's focus on prioritizing memory safety while maintaining high performance makes it a compelling competitor in the systems programming space. Unlike traditional low-level languages like C/C++, which rely on manual memory management and are prone to common memory safety bugs such as buffer overflows, Rust's unique set of safety rules eliminates these risks as early as during compilation. This feature has enabled its adoption in mainstream operating systems [21], [35], while its public registry of over 180k crates[1] highlights its growing popularity [12].

There are scenarios in systems programming where Rust's safety checks can be too restrictive. To handle such cases, developers can use the `unsafe` keyword to mark code blocks or functions where the compiler's safety checks are suspended. This explicitly signals that the enclosed code may violate Rust's memory safety guarantees [8].

Previous studies have shown that approximately one in four Rust projects use unsafe mode [2]. Developers often rely on it to implement features that are not feasible in Rust's safe mode. In other cases, unsafe Rust is used to pursue better performance or due to the complexity of writing code that the Rust compiler can verify as being safe [19], [29].

Unsafe Rust undermines the language's memory safety guarantees, introducing memory safety vulnerabilities. Microsoft and Google report that over 70% of critical bugs stem from memory safety issues [34], [47], leading agencies such as CISA to advocate for memory-safe programming [14], [27]. To date, around 27% of Rust bugs in the RustSec Advisory Database [46] are related to memory corruption, resulting from incorrect use of unsafe Rust [30], [49]. This significant percentage has motivated research into static [7], [31], [33] and dynamic [5], [13], [36] analysis tools to detect such issues.

Static analyzers such as Rudra [7] and Yuga [31] aim to identify certain bug patterns in Rust source code. Although these tools successfully detect memory safety bugs, significant human involvement is required to filter their output (with high false positive rates) and to create proof-of-concept (PoC) test cases to confirm each bug. In contrast, dynamic analysis approaches, such as fuzzing, due to their context-sensitive analysis tend to have lower false positives and can uncover a broader range of bugs [5], [36], [39]. Although fuzzers are effective at detecting memory safety violations, their utility is often limited in the Rust ecosystem, where most code is provided as libraries rather than standalone binaries that can be directly tested by the fuzzer [23]. Consequently, before fuzzing Rust libraries, it is necessary to generate appropriate harnesses that encapsulate the library's functionality and convert the fuzzer's input into well-typed, valid data types expected by the target functions.

Harness generation in Rust is tedious and time-consuming due to its complex type system and syntax. To ease this burden, several tools use static analysis to automate harness generation. However, despite improving test coverage, these tools struggle to detect memory safety bugs. RULF [23] and FRIES [51] lack support for traits, and

1. In Rust, a crate is a unit of code that can be a library or a binary.

RuMono [54] does not handle closures—both crucial for modeling buggy user-defined behavior. RPG [50] targets all unsafe code, including explicitly unsafe APIs, resulting in false positives. Additionally, these tools use heuristics and API dependency analysis to generate API call sequences. While the sequences can be syntactically valid, their API calls are often not semantically related and fail to reflect realistic usage patterns. Due to these limitations, although these tools have discovered bugs in various Rust crates, none of the reported issues have involved memory safety.

More recently, the rise of Large Language Models (LLMs) for code generation has prompted their use in automating test generation [10], [11], [22], [48]. As unit test generators, LLMs tend to focus on components they infer as important—guided largely by the user's prompt. The effectiveness of this process strongly depends on prompt quality [52], a challenge amplified in Rust due to its expressive and strict type system, which complicates the generation of valid code without precise context. To address this, researchers augment prompts with static analysis metadata and break down the generation task into smaller, focused sub-tasks. RUG [11] follows this approach, using ChatGPT-4 and static analysis to generate unit tests that capture complex trait relationships. While RUG's high-quality test code improves coverage, its evaluation has not demonstrated discovery of memory safety bugs.

In summary, the limitations of existing tools, include their insufficient approach to targeting unsafe code and relevant API call sequences, as well as limited support for Rust types such as generics requiring custom implementations and closures.

To address these limitations, we present deepSURF, a tool that combines static analysis and LLMs to automatically generate harnesses targeting unsafe code with the goal of uncovering memory corruption vulnerabilities. deepSURF uses the Rust compiler's analysis to identify APIs that can reach unsafe code and generates initial harnesses accordingly. It then employs DeepSeek-R1 [15] to augment these harnesses with complex API call sequences, enabling the fuzzer to explore richer execution paths and increasing the chances of exposing memory corruption in multi-step interactions. deepSURF also introduces a novel approach to handling generics by generating custom types and trait implementations. This allows the fuzzer to simulate execution of user-defined code within the library's context, further increasing the likelihood of exposing memory bugs. For unsafe traits, where implementations must uphold safety guarantees, deepSURF prompts the LLM to decide candidate types within the library that satisfy the required bounds.

deepSURF employs fuzzing to test the generated harnesses, shifting the burden of discovering bug-triggering conditions from the developer to the fuzzer and eliminating manual effort. To reduce false positives, it generates harnesses that invoke safe APIs capable of reaching unsafe code and configures the fuzzer to exclude crashes unrelated to memory corruption—a common issue in Rust fuzzing [39].

We evaluate deepSURF on 27 crates with real-world memory safety vulnerabilities and detect 26 bugs via fuzzing. Notably, six of these were previously-unknown and have been disclosed to the respective library authors. Our key contributions are summarized below.

- **Targeting Memory Safety Bugs.** We generate LLM-augmented harnesses that exercise semantically related API call sequences capable of reaching unsafe code. Their interaction with unsafe code can lead to memory safety violations, which our approach exposes through fuzzing.
- **Innovative Handling of Generics.** We introduce a novel method for generating harnesses for APIs that require generic type arguments, including substitution with custom types and implementation of required traits. This approach simulates the insertion and execution of potentially buggy user-defined code within the Rust library's context.
- **Enhanced Data Type Support.** Our approach includes the generation of Rust-specific function arguments, such as closures and containers of complex or generic types, unlocking the limitations of previous work and extending the range of functions that can be fuzzed.
- **LLM-Guided Search Space Reduction.** We use LLMs to guide decisions in large search spaces by prioritizing potentially vulnerable API call sequences and selecting among multiple candidates for generics implementing unsafe traits and for complex types requiring instantiation.
- **deepSURF Tool for Automatic Detection of Memory Safety Bugs.** We implement deepSURF, a tool that combines LLM-augmented harness generation with fuzzing to detect memory safety bugs in Rust libraries. Compared to existing approaches, deepSURF shows significantly improved bug-finding capability, identifying 26 memory bugs—20 previously-known and six newly-discovered.

2. Background

Rust: Memory-Safe by Design. The Rust programming language enforces strong protections against memory violations through a set of strict rules imposed by the Rust compiler (`rustc`) that developers must follow. These rules ensure memory safety by identifying potential issues either at compile time, causing the compilation to fail, or at runtime, leading to controlled program termination. In Rust, every memory object is tied to a single owner (variable). Ownership can be transferred between variables or temporarily borrowed through references, all in a controlled manner [8]. Additionally, Rust introduces the concept of lifetimes [8], which ensure that references remain valid for as long as the referenced memory object exists and is accessible. Thanks to lifetime rules, memory bugs such as use-after-frees and double-frees are impossible in safe Rust. Finally, Rust performs bounds checking at compile or runtime, blocking out-of-bounds memory access.

Unsafe Rust: Superpowers with High Risk. Systems programming often requires low-level operations, such as interfacing with native C/C++ libraries—tasks that cannot be performed using "safe" Rust alone. To support these use cases, Rust provides unsafe Rust, which relaxes safety rules to give developers low-level control at the cost of reintroducing the risk of memory bugs.

Developers switch to unsafe Rust using the `unsafe` keyword to begin an unsafe code block. Functions containing unsafe blocks can be marked as either safe or unsafe. If a function is explicitly marked as unsafe, its caller is responsible for ensuring the function is not invoked with arguments that could cause memory violations. In contrast, if an unsafe code block is enclosed in a safe function, the function should act as a safe wrapper for the internal unsafe operations. Additionally, Rust defines unsafe traits for which the compiler does not enforce safety checks. Unsafe code may rely on their implementations, and the responsibility for ensuring safety lies with the developer who implements them—whether a library author or a user [8].

Fuzzing. Fuzzing is one of the most effective methods for detecting memory bugs. Fuzzers generate diverse inputs and run programs against them to uncover vulnerabilities. Greybox fuzzers such as AFL++ [20] are especially efficient, using coverage feedback to explore different code paths. Their bug-finding capability can be further enhanced with sanitizers. AddressSanitizer (ASan) [38], for example, detects subtle memory corruption bugs—such as small buffer overflows—that may not crash the program. However, sanitizers introduce performance overhead, which can slow down fuzzing. In Rust, the `afl.rs` [5] crate integrates the functionality of AFL++ and automates advanced features such as CmpLog [6] and persistent mode [4].

Rust Libraries and Harness Generation. Rust code is organized into crates, which may be either binaries or libraries [8]. Binary crates are standalone executables that must define a `main` function as the program's entry point. Library crates, however, typically lack a `main` function and instead provide reusable functionality by exposing an API of public functions for users to call. The Rust ecosystem primarily consists of library crates, posing challenges for automated testing tools like fuzzers [23]. While binaries can be fuzzed directly, libraries require the generation of harnesses—test cases that invoke the library's API using input provided by the fuzzer. These harnesses are compiled into executables, allowing the fuzzer to explore the library's functionality and uncover bugs.

LLM-Based Harness Generation. Large Language Models (LLMs) have shown strong capabilities in code generation by learning from large-scale code corpora [3], [9], [25]. Their ability to synthesize semantically meaningful code has motivated security researchers to use LLMs to generate fuzz harness (sometimes referred to as fuzz drivers or fuzz targets). This approach is particularly effective when bugs are triggered by chained API sequences, as LLMs can generate interactions beyond typical usage patterns, enabling deeper testing and uncovering more intricate bugs [16], [28].

The effectiveness of LLM-based harness generation heavily depends on prompt quality [42], [53]. Yet, even with sufficient documentation and examples, LLMs often struggle to generate valid harnesses for complex targets like the Linux kernel [24] or Rust libraries [11], [48]. In Rust specifically, the combination of intricate trait relationships and a highly expressive type system poses major challenges. These complexities often prevent the automatic creation of valid harnesses, as accurately reasoning about such features is better suited to traditional static analysis techniques.

Definitions. To aid in understanding the rest of the paper, we propose the following terminology:

- *Unsafe Block (UB):* A code segment enclosed within the `unsafe` keyword, allowing operations that bypass the language's safety checks.
- *Unsafe Function (**UF**):* A function explicitly marked with the `unsafe` keyword in its signature. Any operation within this function can bypass Rust's safety checks. A *UF* can only be called within a *UB* or another *UF*.
- *Safe Function (**SF**):* A function that is not *UF*.
- *Unsafe Encapsulating Function (**UEF**):* A *SF* that contains a *UB*. *UEFs* are designed to encapsulate internal unsafe code within a safe wrapper, preventing its callers from being affected by unsafe behavior.
- *Unsafe Reaching Function (**URF**):* A *SF* that is not a *UEF* itself but can reach unsafe code indirectly by calling other *UEFs* or *URFs*.
- *Unsafe API (**UAPI**):* A publicly accessible *UF* of a Rust library.
- *Unsafe Reaching API (**URAPI**):* A publicly accessible function of a Rust library that is either a *UEF* or *URF*.
- *URAPI Coverage:* The *URAPI* coverage of a set of harnesses is the ratio of *URAPIs* directly called in the harnesses to the total number of *URAPIs* in the library.

Memory Safety Bugs in Rust Libraries. The use of unsafe Rust can introduce memory safety bugs due to incorrect handling of unsafe code [30], [49]. These bugs are similar to those found in other systems programming languages, including use-after-free, double-free, and buffer overflows. However, in Rust, not all memory violations constitute genuine memory bugs.

In a library crate, functions not explicitly marked as `unsafe` are expected to be safe to call under all circumstances—even if they contain internal unsafe code. If a user triggers memory corruption by interacting only with such safe functions, this constitutes a genuine memory bug: the library has failed to properly encapsulate its unsafe behavior [7], [43]. In contrast, if corruption results from incorrect use of a *UAPI*, the fault lies with the user for violating the safety contract [8], [32]. The same principle applies to unsafe traits. When users implement unsafe traits defined by a library, they are responsible for ensuring correctness. However, if the library itself provides implementations of unsafe traits for use by its clients, it must guarantee that these implementations do not expose unsafe behavior.

Thus, identifying genuine memory safety bugs in Rust libraries requires harnesses that target *URAPIs* and rely on library-defined implementations of unsafe traits. Conversely, harnesses that directly invoke *UAPIs* or implement unsafe traits without adhering to their safety contracts can lead to false positives.

3. Challenges of Fuzzing Rust Libraries

Fuzzing Rust's ecosystem, which is primarily composed of libraries, requires generating harnesses that invoke library

APIs with arguments derived from the fuzzer's input. While C functions often accept arguments constructed directly from raw bytes, Rust's expressive type system and language-specific features, such as traits, demand more sophisticated handling.

Furthermore, although Rust confines potential memory vulnerabilities to unsafe code—unlike C, where all code is vulnerable—this introduces unique challenges in identifying and reaching that code. Unsafe code in Rust is governed by safety contracts that developers are expected to uphold. A memory violation caused by misusing a *UAPI* or incorrectly implementing an unsafe trait is typically attributed to the user for violating these contracts and does not constitute a true memory bug. Additionally, Rust fuzzers [5], [36] may classify panics and assertion failures as bugs, even when intentionally inserted by library developers. While such failures may indicate logical errors, they do not represent exploitable memory corruption and should be filtered out.

Generating harnesses to expose memory vulnerabilities in Rust requires overcoming several key challenges. Based on our analysis of real-world bugs in Rust libraries, in this section, we explain these key challenges and illustrate them with concrete examples.

```rust
impl UnsafeSer for StructA {
    fn process(&self) -> Vec<u8> {
        unsafe { /* Unsafe operations */ }
    }
}

impl SafeSer for StructA {
    fn process(&self) -> Vec<u8> { ... }
}

pub unsafe fn unsafe_read(addr: usize) -> u8 {
    /* Unsafe operations */
}

pub fn unsafe_ser<T: UnsafeSer>(data: T) -> Vec<u8> {
    data.process()
}

pub fn safe_ser<T: SafeSer>(data: T) -> Vec<u8> {
    data.process()
}
```

Listing 1: Callee target resolution influenced by trait bounds.

Challenge 1 (C1): Targeting Unsafe Reaching APIs and Memory Corruption Vulnerabilities. The use of unsafe code can cause memory violations. As discussed in Section 2, for a memory violation to be considered a true memory bug, it must occur when the users interact solely with the library's safe APIs. Thus, identifying the *URAPIs* of a library is essential, but challenging.

A first challenge lies in determining which functions are actually part of a library's public API, as this often requires analyzing more than just the file where a function is defined. In Rust, a function marked with the `pub` keyword may still be inaccessible to users if it resides in a private module [44]. Additionally, public re-exports of private modules and feature-gated code—which can alter the visible API surface across builds—introduce further complexity.

Even after identifying the exported APIs, determining which ones are *URAPIs* is non-trivial. Treating every API that reaches unsafe code as a *URAPI* can lead to false positives during fuzzing, especially if *UAPIs* are invoked directly in the harness. Avoiding this requires a more targeted control-flow analysis of the library. This analysis presents additional challenges. As shown in Listing 1, both `unsafe_ser` and `safe_ser` call `process`, but the method invoked depends on the trait bound—`UnsafeSer` or `SafeSer`. Resolving the correct callee requires analyzing the caller's type context, which is further complicated by the use of dynamic dispatch [8]. Finally, beyond false positives from misuse of *UAPIs*, fuzzers may also classify crashes from developer panics and assertions as bugs [39]. To avoid this, the fuzzer must be configured to ignore non-memory corruption bugs.

To address these challenges, deepSURF performs dedicated analysis to determine unsafe code reachability, identify *URAPIs*, and generate fuzz harnesses that target them. These harnesses are then tested using a specially configured fuzzer that ignores crashes caused by panics or assertions that do not indicate memory corruption. In the case of Listing 1, deepSURF generates a harness only for the *URAPI* `unsafe_ser`, and not for the *UAPI* `unsafe_read` or the function `safe_ser` that does not reach unsafe code.

```rust
impl<T> TooDee<T> {
  pub fn with_capacity(cap: usize) -> TooDee<T>
  pub fn init(c: usize, r: usize, val: T) -> TooDee<T>
  pub fn from_vec(c: usize, r: usize, v: Vec<T>) -> TooDee<T>
  pub fn insert_row<I>(&mut self, ...)
}
```

Listing 2: `TooDee` constructors and *URAPI* `insert_row`.

Challenge 2 (C2): Supporting Complex Types. Rust supports the definition of complex types through structs and enums. Unlike in C, where such types can often be instantiated directly from raw bytes, Rust's visibility rules typically require using specialized functions to construct instances [45]. However, Rust does not have constructors as a built-in language construct. Instead, any library API function with the appropriate inputs and output can act as a constructor for a complex type.

Identifying APIs that serve as constructors for a given complex type requires type analysis and type matching, tasks made non-trivial by Rust's expressive type system. Even after identifying compatible constructors, selecting which ones to include in a harness remains a challenging decision. The selection is often heuristic-driven and constrained by the number of harnesses to generate, or whether a constructor has already been used in another harness [23], [50]. Our experiments show that constructor choice significantly affects both the likelihood and speed of bug discovery. However, it is not possible to statically determine which constructor will be more effective in triggering a bug. Listing 2 shows three constructors for the `TooDee` object from the `toodee` crate, along with the *URAPI* function `insert_row`. The choice of constructor has a substantial impact on the time required to trigger the double-free bug in `insert_row`: objects created with `init` trigger the bug within seconds, whereas those created using `from_vec` may take over five hours. Interestingly, empty objects constructed with `with_capacity` do not trigger the bug at all.

To address this, deepSURF leverages `rustc`'s type analysis to identify and extract constructor APIs for complex

type arguments. These constructor candidates are passed to the integrated LLM, which uses its semantic understanding to select a heterogeneous subset. The resulting LLM-augmented harness allows the fuzzer to dynamically choose among these constructors based on its input, prioritizing those more likely to trigger bugs.

Challenge 3 (C3): Handling Generics Types. Rust supports generic data types, enabling reusability through type parameters. Generic types often have trait bounds specifying the behavior a type must implement. When an API expects a generic type, the harness must substitute it with a concrete type that satisfies the required trait bounds. This substitution is complex, as trait definitions may be scattered across the target or external libraries and some bounds may depend on supertraits [8]. Such complexity requires robust static analysis to extract complete trait information.

To address this challenge, deepSURF performs trait analysis at compile time to collect all required bounds for generic arguments. During harness generation, it uses this information to generate custom types with corresponding trait implementations that substitute the generic arguments. These custom types allow the fuzzer to control traits behavior based on its input. For generics bounded by unsafe traits, deepSURF prompts the integrated LLM to identify compatible library-defined types for substitution [8], thereby avoiding violations of safety contracts due to incorrect custom implementations (see Section 2).

```rust
struct MyRead(());
impl Read for MyRead {
  fn read(&mut self, _: &mut [u8]) -> Result<usize> {
    Ok(131313) // Return always the same big number
  }
}
fn main() {
  let mut hashes = BlockHashes::empty(32);
  let diff = hashes.diff_and_update(MyRead(()));
}
```

Listing 3: PoC for the RUSTSEC-2021-0094 BOF bug.

Challenge 4 (C4): Simulating Interaction with User-Defined Code. Rust libraries often allow users to define custom behavior through closures and traits. Closures are anonymous functions that can capture variables from their surrounding scope and are typically passed as arguments to enable dynamic behavior. Traits, by contrast, define required functions that describe a behavior types must implement. Users can create custom types with specific trait implementations, allowing libraries to adapt their functionality.

Our analysis of real-world memory safety bugs shows that library developers often fail to fully account for the range of behaviors that user-defined trait implementations or closures may exhibit—especially when interacting with unsafe code. For example, if a user-defined closure panics inside an unsafe block of the library, the Rust runtime will unwind the stack and invokes destructors for all live variables. If the unsafe code has duplicated ownership of any of these variables without proper safeguards, this can lead to double-free bug.

For instance, an incorrect custom implementation of the `read` function from the `Read` trait can trigger a buffer overflow in the `rdiff` crate (Listing 3). In this case, the vulnerable *URAPI* `diff_and_update` takes a generic argument bounded by the `Read` trait. During execution, it creates a `Window` object with vector lengths determined by the user-provided `read` function, using unsafe blocks to set these lengths without validation (Listing 4). If the developer supplies a `Read` implementation that reports reading more bytes than the buffer can hold, the library will incorrectly allocate vectors with lengths exceeding their capacity, leading to memory corruption.

To simulate the execution of user-defined code within the library's context, deepSURF generates custom functions that substitute trait implementations or closures, adhering to the correct syntax for each case. These functions are designed so that their behavior is driven by the fuzzer, allowing them to mimic user-defined logic that may panic or return edge-case values within the expected return type—conditions that can trigger bugs in the tested library.

```rust
impl<R: Read> Window<R> {
  pub fn new(mut r: R, b_sz: usize) -> Result<Window<R>>{
    let mut back = vec![0; b_sz];
    let size = r.read(back.as_mut_slice())?;
    unsafe { back.set_len(size); }
    // Similar unsafe length set for front
    Ok(Window {
      front,
      back,
      ...
```

Listing 4: Unsafe length setting in `Window` constructor.

Challenge 5 (C5): Supporting Sequences. Although some bugs can be triggered by fuzzing a single API, others require exercising sequences of API calls, where each call sets up conditions for a more complex bug. For example, Listing 5 shows a POC for a heap buffer overflow bug due to an off-by-one error in the unsafe code of `remove` of the `simple-slab` crate. Notably, this bug is only triggered after inserting at least two items into a `Slab` object before calling `remove`.

The three APIs (`with_capacity`, `insert`, and `remove`), encapsulate unsafe code and are therefore identified as *URAPIs*. However, approaches that fuzz each *URAPI* in isolation fail to expose the bug described above. This example highlights that targeting a single *URAPI* is often insufficient; instead, harnesses must be designed to fuzz a target API within meaningful sequences of related API calls. For instance, as shown in the example, if we want to target a method called `remove` (removing elements from a collection), it is reasonable to first call one or multiple times the method `insert` on the same collection.

Existing methods extract fixed sequences of APIs using static analysis and dependency graphs, aiming to maximize API coverage. While these sequences are often syntactically valid, they may yield API combinations that cannot represent realistic usage scenarios. To address this limitation, deepSURF leverages LLMs to generate harnesses that embed *URAPIs* within semantically relevant sequences of other API calls. For example, in the case of the `simple-slab` crate, if the LLM is provided with appropriate documentation, it can infer that `Slab` implements a list-like data structure and, accordingly, group operations such as insertion and removal within the same harness.

Although LLMs are effective at generating semantically coherent API groupings, they are less suited for exploring code paths or identifying paths with high coverage or a greater likelihood of exposing bugs [11]. To address this, deepSURF prompts the LLM to generate harnesses in which both the length and composition of API sequences are parameterized by the fuzzer's input. This enables the fuzzer to dynamically compose sequences—skipping, repeating, or prioritizing API calls—based on runtime feedback.

```
struct StructA(String);

fn main() {
    let mut slab = Slab::with_capacity(2);
    slab.insert(StructA("Hi".to_string()));
    slab.insert(StructA("Bye".to_string()));
    slab.remove(0); // Crashes due to HBOF
}
```

Listing 5: PoC for the RUSTSEC-2020-0039 BOF bug.

Existing Solutions. As shown in Table 1, state-of-the-art Rust fuzzing tools face key limitations in addressing the explained challenges, hindering their ability to uncover memory safety bugs.

None of the existing tools configure the fuzzer to ignore non-memory corruption bugs, leading to false positives when crashes caused by panics or assertions are reported. Only RPG and RUG employ strategies to target unsafe code, but they may also directly fuzz *UAPIs* without preserving their safety invariants, which can also result in false positives. While all four tools support complex type arguments, some struggle when these types are nested within container types such as vectors. Although several approaches support generic types, none supports user-defined code simulation through custom trait implementations.

Regarding **C5**, RUG generates sequences of APIs needed to construct arguments for each target function but does not embed the target function itself within broader sequences of semantically related APIs. Other tools use static analysis to build API dependency graphs and extract sequences by maximizing API coverage, prioritizing unsafe functions, or mimicking real-world usage patterns. However, these approaches impose fixed sequences that often miss "semantic" connections between related APIs (e.g., `insert` and `remove`) and may combine unrelated or already well-tested APIs. This limits the fuzzer's ability to dynamically explore alternative interactions that could be more effective at exposing bugs.

In summary, as we will show in Section 5, while these tools improve test coverage and have identified some bugs, they have not yet demonstrated effectiveness in uncovering memory corruption vulnerabilities.

4. deepSURF Design

The design of our tool is illustrated in Figure 1. deepSURF consists of three key components: **Static Analysis**, **Harness Generation**, and **Dynamic Analysis**. The tool processes a Rust library as input and attempts to detect memory safety vulnerabilities affecting it.

Tool	C1	C2	C3	C4	C5
RULF	✗	◐	✗	✗	◐
RPG	◐	◐	✓	✗	◐
FRIES	✗	◐	✗	✗	◐
RuMono	✗	✓	✓	✗	◐
RUG	◐	✓	✓	✗	◐
deepSURF	✓	✓	✓	✓	✓

TABLE 1: Tools comparison against the challenges (C1-C5). Symbols indicate: full (✓), partial (◐) or no (✗) support.

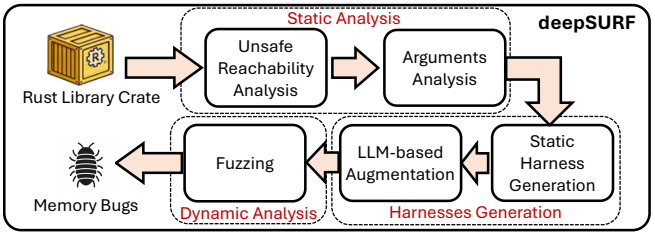


Figure 1: The workflow of deepSURF.

4.1. Unsafe Reachability Analysis

First of all, deepSURF needs to identify all *URAPIs* in the provided Rust library, since these are the functions that can potentially trigger the execution of unsafe code. This process consists of two steps: (a) detecting all locations of unsafe code within a library and (b) finding publicly accessible safe functions that can reach those locations.

4.1.1. Identifying Unsafe Usage and UEFs. In a Rust library, unsafe code can be found in unsafe blocks and in unsafe functions (i.e., code blocks and functions marked by the keyword `unsafe`). deepSURF focuses specifically on unsafe code that is reachable through safe functions (*SFs*), as directly calling *UFs* may lead to false positives (see Section 2). Therefore, deepSURF identifies *UEFs* within the library, which act as transition points from safe to unsafe code. To achieve this, deepSURF utilizes `rustc`'s unsafety checking pass to detect all occurrences of unsafe blocks. It then leverages Rust's High-Level Intermediate Representation (HIR) to identify safe functions containing these unsafe blocks, forming the set of *UEFs* in the library.

4.1.2. Detecting Public Entry Points to *UEFs*. The next step for deepSURF is to identify all safe API functions in the library that can reach the previously-extracted *UEFs*. We focus on safe APIs capable of reaching *UEFs* since they serve as public entry points for the harness to access unsafe code when the fuzzer provides appropriate inputs. To identify call paths between safe APIs and *UEFs*, deepSURF needs to build a Control Flow Graph (CFG) for the input library through the following steps.

Function Call Resolution. First, deepSURF analyzes the calling relationships between functions in the library by leveraging the Mid-Level Intermediate Representation (MIR) of each function to record all caller-callee pairs. It identifies three types of function calls: (a) direct calls with a single target, (b) direct calls with multiple targets, and (c)

indirect calls through function pointers. deepSURF does not perform pointer analysis, so type (c) is not supported. Direct calls with single target are resolved at compilation time using the context in which the call occurs. deepSURF extracts this information from `rustc` to record the corresponding caller-callee pairs. Regarding calls with multiple potential targets at compile time, which require dynamic dispatch, deepSURF handles such cases by over-approximating and recording a caller-callee pair for each possible callee.

CFG Construction and *URAPIs* Extraction. After collecting all caller-callee pairs, deepSURF constructs the CFG involving all functions of the library. Next, it uses the unique identifiers assigned during HIR analysis to locate the *UEFs* within the CFG. Starting from each *UEF*, deepSURF performs a reverse traversal of the graph using breadth-first search (BFS), moving from callees to callers until it reaches functions with no further callers. During this traversal, any function that is both public and safe is added to the set of *URAPIs*. By the end of this process, the *URAPIs* form a subset of the library's public API: safe functions that can reach unsafe code. This set becomes the fuzzing target, addressing **C1** (see Section 3).

4.2. Arguments Analysis

Rust supports a diverse range of types, from primitive types such as integers to common types found in other programming languages, such as structs and generics, as well as Rust-specific types like closures. deepSURF is designed to handle this variety to enable effective fuzzing of *URAPIs* with diverse inputs, addressing the limitations of state-of-the-art tools in supporting different argument types. A complete list of the data types that deepSURF can generate using fuzzer's input is provided in Appendix A.

4.2.1. Complex Types Support. Rust uses complex types such as structs and enums to define new types within a library. Since deepSURF aims to support a wide range of *URAPIs*, including those requiring complex type arguments, it must identify ways to generate instances of these types to invoke the respective *URAPIs* during testing.

Instantiation of Complex Types in Rust. Structs group related data into a single complex type. Although structs can be initialized by directly assigning values to their public fields, this approach is discouraged as it bypasses encapsulation. Instead, struct fields in Rust are by default private, and constructor functions are used for validated initialization. Similarly, enums define a type that can represent one of several variants, each of which can optionally hold data. The variants of a public enum can be directly assigned, or constructors can be used for initialization.

Identifying Constructors of Complex Types. Since constructors are not a built-in language construct in Rust, deepSURF identifies candidate constructors for structs and enums by searching for functions that meet specific criteria. We qualify a function as a candidate constructor for a complex type if it satisfies all of the following: (a) it is public, (b) none of its input arguments are of the same type as the target complex type (nor contain it as an inner type), and (c) its return type matches the target type—either directly, wrapped (e.g., in an `Option`), or as a field within a tuple. To verify condition (a), deepSURF uses the visibility analysis provided by `rustc`. For (b) and (c), it recursively analyzes input and output types (see §4.2.3), leveraging unique type identifiers from the HIR and internal structures of `rustc`'s type checker for precise type matching. In the case of enums, deepSURF also collects and inspects their variants, analyzing any inner data types. Collectively, these steps help address **C2** (see Section 3).

4.2.2. Generic Types Support. Based on our analysis of real-world memory safety bug PoCs, nearly 90% of the involved APIs use generic arguments, most bounded by traits. Additionally, we observed that many of these bugs stem from two factors: (a) substituting these generics with user-defined types, and (b) providing faulty custom trait implementations. This highlights the prevalence of generic arguments in Rust libraries and their connection to memory bugs when misused. As deepSURF aims to generate harnesses that expose memory safety vulnerabilities, supporting generic arguments and trait bounds is essential.

Collecting Trait Bounds. When handling APIs with generic type arguments, deepSURF must collect their trait bounds to ensure valid substitutions. Trait bounds define the traits a concrete type must implement to substitute the generic argument. These bounds may also include supertraits—traits a type must implement as a prerequisite to the current trait. deepSURF utilizes metadata of the trait analysis performed by `rustc` to collect the trait (and supertrait) bounds for each generic argument.

Collecting Trait Functions and Associated Types. Traits in Rust define required behavior through trait functions that a type must implement to satisfy the trait. Some traits also declare associated types—types tied to the trait and used within its context. Like generic parameters, associated types can have trait bounds and must be substituted with concrete types during harness generation. Additionally, APIs may impose constraints that link associated types to generic parameters, creating dependencies between them.

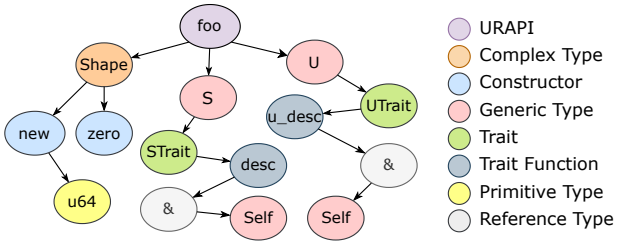
To support generic type substitution required by **C3**, deepSURF collects detailed information about all relevant traits—including their trait functions and associated types—by leveraging `rustc`'s APIs. In Rust, traits with multiple functions often include default implementations, meaning only a subset must be explicitly implemented for a type. deepSURF records both the required trait functions and any default ones that can be overridden by user-defined implementations. This information is also essential for addressing **C4**, guiding the substitution of generics and associated types with custom types and implementations of the required and optionally overridden trait functions.

4.2.3. Recursive Type Analysis. To invoke *URAPIs*, the harness must construct argument types from the fuzzer's byte stream input. While trivial for primitives, this is more complex for structs, generic types, and container types (e.g.,

vectors). To ensure correct invocation of library functions, deepSURF performs recursive type analysis on function arguments during static analysis and leverages `rustc`'s type-checking system for precise type matching. For generic and complex types, deepSURF analyzes trait function or constructor arguments, respectively. For container types, it recursively analyzes inner types. The pseudocode for this recursive analysis is provided in Appendix A.

```
pub struct Shape{ sid: u64 }
impl Shape {
  pub fn new(sid: u64) -> Self { Self { sid } }
  pub fn zero() -> Self{ Self{ sid: 0 } }
  pub fn foo<S: STrait, U: UTrait>(&self, i1: S, i2: U) {
    unsafe { ... }
  }
}
pub trait STrait{ fn desc(&self) -> String; }
pub unsafe trait UTrait{ unsafe fn u_desc(&self) -> String; }
impl STrait for Shape{
  fn desc(&self) -> String { ... }
}
unsafe impl UTrait for Shape{
  unsafe fn u_desc(&self) -> String { ... }
}
```

(a) Example library `foo`.



(b) deepSURF's dependency tree for *URAPI* `foo`.

```
struct CustomTy0(String);
struct CustomTy1(String);

impl STrait for CustomTy0 {
  fn desc(&self) -> String {
    if _to_u8(fz_data)%2 == 0{ panic!("INTENTIONAL PANIC!") }
    return _to_string(fz_data);
  }
}
unsafe impl UTrait for CustomTy1 {/*Similar to STrait*/}

fn main (){
  fuzz!(|fz_data: &[u8]| {
    let t0 = foo::Shape::zero();
    let t1 = _to_string(fz_data);
    let t2 = CustomTy0(t1);
    /* Similar steps with the above for building t3*/
    &t0.foo(t2, t3);
  });
}
```

(c) Statically generated harness for the *URAPI* `foo`.

Figure 2: deepSURF's static harness generation for `foo`.

4.3. Static Harness Generation

After completing the static analysis phase, deepSURF uses the collected information to generate fuzz harnesses. It streamlines this process for each *URAPI* by constructing a dependency tree that captures the relationships among the function's arguments and their components. After building the dependency tree, deepSURF generates harnesses by performing a Depth-First Search (DFS) from the leaves up to the root (*URAPI*), progressively constructing each argument along the way.

Example. In Figure 2a, we present the example library `foo`. We focus on harness generation for the *URAPI* `foo`; a simplified version of its dependency tree appears in Figure 2b. The tree's root is the target `foo`, with child nodes for its argument types (`Shape`, `S`, `U`). These nodes connect to their corresponding constructors or trait bounds.

Handling of Multiple Candidate Constructors. Dependency trees with multiple constructors per complex type are split into separate dependency trees, where each complex type is tied to a single constructor. To mitigate the exponential growth of possible trees, the number of constructors considered during this step is configurable.

Generating Arguments. For primitive type nodes, deepSURF directly converts the fuzzer's input bytes into the required types using helper functions. For generic nodes, it substitutes all occurrences of the generic parameter with custom concrete types. For complex types, it first generates the constructor arguments, which uses to invoke the constructors. Listing 2c shows a harness generated for the `foo` function: the complex type `Shape` is instantiated via its `zero` constructor (line 14), while the generic parameters `S` and `U` are replaced with `CustomTy0` and `CustomTy1`, respectively—custom structs that enable trait implementation.

Generating Custom Functions. To address **C4**, deepSURF generates custom implementations for all required traits. Given the trait names and trait function signatures, it synthesizes their bodies that produce return values of the expected types, using the fuzzer's input. At this stage, it also implements unsafe traits, which temporarily introduces unsafe code into the harness—a practice we should avoid to prevent false positives (see Section 2). However, this unsafe code is removed during the LLM augmentation stage. Following a similar process, deepSURF generates custom functions to substitute closures. As shown in Listing 2c, deepSURF generates a custom implementation of the `desc` method for the trait `STrait`. To simulate user-defined behavior, the generated `desc` function uses the fuzzer's input to either trigger a panic—potentially exposing panic-safety bugs—or return any value of the expected type.

4.4. LLM-based Augmentation

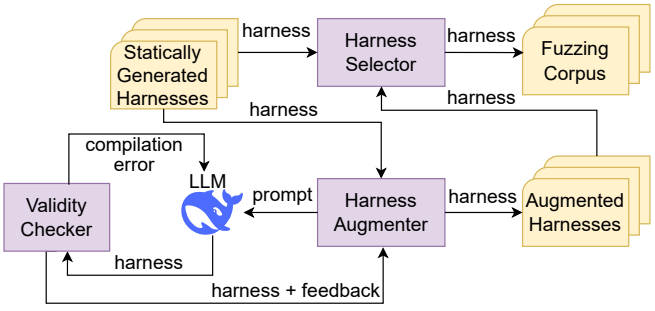


Figure 3: deepSURF's LLM harness augmentation.

The harnesses generated in the previous stage represent a foundation for targeting the respective *URAPIs*, but they do not fully address all the challenges we explained in Section 1. Specifically, they lack support for initializing complex types using multiple constructors (**C2**), fail to substitute generic parameters with unsafe trait bounds using library-defined concrete types (**C3**), and do not incorporate semantically related API call sequences that involve the targeted *URAPI* (**C5**). These limitations are addressed in the current stage using LLMs.

Figure 3 outlines the steps deepSURF performs in this stage. In summary, the *Harness Augmenter* attempts to use an LLM to improve statically-generated harnesses, while the *Harness Selector* optimally combines statically-generated and LLM-generated harnesses.

Prompting the Model. The *Harness Augmenter* selects a statically generated harness from the *Statically Generated Harnesses* set, targeting a specific *URAPI*, and uses it to prompt the LLM for an augmented version. When available, we prioritize fetching statically generated harnesses that compile successfully. If none exists for a given *URAPI*, we fall back to non-compilable ones, which, based on our experiments, often still provide sufficient structure for the LLM to produce corrected and functional augmented versions.

Each prompt includes: (a) the fetched harness, (b) the targeted *URAPI*, (c) metadata from static analysis, (d) relevant documentation, and (e) augmentation instructions. The static metadata includes available constructors for complex types and the list of *URAPIs* defined in the library. Documentation—extracted via `rustdoc`—is included when input token limits allow. The instructions guide the model on how to augment the harness and outline side effects to avoid.

To fully address **C2**, we instruct the model to analyze the list of compatible constructors, select a heterogeneous subset it deems relevant, and modify the harness to support instantiating complex types using any of these constructors. The augmented harness includes a switch statement driven by the fuzzer's input to dynamically select among them.

As a final step in addressing **C3**, we prompt the model to replace custom types used for generic parameters bounded by unsafe traits with compatible, library-defined types that already implement the required traits. Importantly, we direct the model to preserve custom types wherever possible, removing only those associated with unsafe traits. This is crucial, as custom types are often paired with valuable custom trait implementations that simulate user-defined code execution (**C4**), so we aim to preserve as many of them as possible—removing only those associated with unsafe traits.

Finally, to address **C5**, we instruct the model to identify a set of semantically related APIs compatible with the targeted *URAPI*, enabling the construction of meaningful API call sequences. We also request that the generated harnesses use the fuzzer's input to dynamically determine the ordering of these sequences rather than relying on fixed orderings.

Validating Augmented Harnesses. After receiving the augmented harness (response) from the LLM, the *Validity Checker* uses the Rust compiler as an oracle to determine whether the harness compiles. If it does, the harness is added to the *Augmented Harnesses* set, and the *Harness Augmenter* is notified of the successful augmentation. In this case, the feedback to the *Harness Augmenter* also includes a list of all *URAPIs* invoked in the augmented harness. This list is then used to decide future harness selection for augmentation.

However, the harness may fail to compile due to errors introduced during augmentation or inherited from the original statically generated version. In such cases, the *Validity Checker* retries augmentation by updating the prompt: it replaces the previous harness with the latest version and appends the relevant compiler error message before re-prompting the model. This process continues for a configurable number of tries. If all tries fail, the failure is recorded, no augmented harness is stored, and the *Harness Augmenter* proceeds to the next *URAPI* awaiting augmentation.

Harnesses Selection for LLM Augmentation and Fuzzing Corpus Generation. deepSURF's goal is to create a corpus of fuzzing harnesses that targets unsafe code (**C1**) and captures diverse and complex behaviors within the fuzzed crate. However, generating too many harnesses risks dispersing the fuzzer's efforts and reducing its effectiveness. Furthermore, statically-generated and LLM-generated harnesses often reach different code paths in the fuzzed code; thus, an appropriate combination of LLM-generated and statically-generated harnesses should be included in the final *Fuzzing Corpus*. For these reasons, we apply specific *harness selection policies* both in the *Harness Augmenter* and in the *Harness Selector* to choose which *Statically Generated Harnesses* to augment and which harnesses to include in the final *Fuzzing Corpus*.

Specifically, following the *Harness Augmenter*'s *harness selection policy* we skip augmenting a statically generated harness for *URAPI A* if another already augmented harness invokes *URAPI A*. This arises when the LLM augments a harness for *URAPI B* by including a call to *URAPI A* (e.g., when addressing **C2** or **C5**). However, *Statically Generated Harnesses* containing custom implementations are always augmented. This decision is based on our observation that such implementations, which simulate user-defined code (**C4**), cannot be reliably generated by the LLM alone. In contrast, when a statically generated harness with these implementations is provided to the LLM, the LLM can often successfully augment it while preserving its custom logic.

Regarding the *Harness Selector*, we apply the following *harness selection policy*. First, we include in the final *Fuzzing Corpus* all compilable LLM-augmented harnesses, as these address all **C1**–**C5**. For *URAPIs* whose augmentation fails after the maximum number of attempts, we check the set of *Statically Generated Harnesses* for compilable options and add up to four harnesses if available. Additionally, for each *URAPI* involving custom functionality, we include up to four compilable *Statically Generated Harnesses*, regardless of whether augmentation succeeded (**C4**).

Appendix B and Appendix C evaluate alternative harness augmentation and harness selection policies, and they provide further justification for the chosen policies.

4.5. Fuzzing for Memory Bugs.

deepSURF dynamically tests the generated harnesses using AFL++, delegating the task of finding bug-triggering conditions to the fuzzer. While effective at detecting memory bugs in Rust, fuzzing can produce false positives from panics or asserts inserted by developers to enforce invariants or handle errors [8], [39]. To prevent such cases from being treated as crashes, deepSURF modifies `afl.rs` and enables ASan [38] to detect non-crashing memory violations. Its output is then used to filter out false positives caused by large memory allocations. These measures ensure that fuzzing focuses exclusively on memory corruption vulnerabilities, fully addressing **C1**.

5. Evaluation

In this section, we evaluate deepSURF by addressing the following three research questions:

RQ1: Can deepSURF automatically generate harnesses that uncover memory safety bugs through fuzzing?

RQ2: How does deepSURF compare to state-of-the-art Rust fuzzing tools in detecting memory safety bugs?

RQ3: How do the key components of deepSURF contribute to its bug-finding capability and unsafe code coverage?

5.1. Experimental Setup

To evaluate deepSURF's effectiveness in detecting memory bugs and to compare it against state-of-the-art approaches, we use the ERASAN dataset. ERASAN is a Rust-specific sanitizer, and its dataset includes 27[2] library crates from the RustSec repository [46], each containing real-world memory safety issues [37]. We chose this dataset because it is sufficiently large, includes well-documented memory corruption bugs across diverse libraries, and has been used in prior work [30], making it a heterogeneous and unbiased foundation for deepSURF's evaluation.

We ran our experiments on a machine equipped with AMD EPYC 7B13 CPUs (112 cores, 2.2GHz) and 224GB of memory, running Ubuntu 24.04. In all experiments, we used DeepSeek-R1 with its default settings (temperature and top-p = 1.0) and set the maximum number of prompt retry attempts to 6. For static harness generation, we considered up to 4 constructors per complex type, we selected dependency trees using seeded random sampling and attempted to compile up to 50.

5.2. Evaluation Results

RQ1: deepSURF's Bug-Finding Capability. In this experiment, we evaluate deepSURF on ERASAN's dataset to determine its ability to automatically generate harnesses that uncover existing or new memory safety bugs. We fuzz each of the generated harnesses for 24 hours using two AFL++ threads working together on the same target: one with ASan and the other with CmpLog [20], as suggested by [1].

Crate	#Detected Memory Bugs (New Bugs)	URAPI Coverage (#Harnesses)			
	deepSURF	deepSURF	RUG	RPG	RULF
algorithmica	1	100% (4)	100% (20)	0% (1)	0% (1)
arc-swap	0	55.6% (7)	22.2% (12)	0% (0)	0% (0)
tokio	0	69.2% (7)	—	0% (0)	0% (0)
secp256k1	0	96.5% (92)	—	31.4% (56)	20.9% (35)
bumpalo	0	96.4% (48)	64.3% (32)	35.7% (29)	35.7% (9)
toodee	4 (1)	90% (34)	28.3% (65)	0% (0)	0% (0)
nano_arena	0	100% (1)	0% (15)	—	0% (0)
stack_dst	2	57.1% (8)	0% (3)	0% (0)	0% (0)
slice_deque	5 (4)	95.3% (316)	64.7% (67)	—	0% (0)
lru	0	80.8% (18)	0% (0)	0% (0)	0% (0)
rusqlite	0	84.2% (108)	0% (0)	2% (39)	1% (3)
stackvector	1	82.5% (44)	0% (0)	0% (0)	0% (0)
insert_many	1	100% (3)	0% (0)	0% (0)	0% (0)
smallvec-0.6.6	1	87.5% (52)	0% (0)	0% (0)	0% (0)
smallvec-1.6.0	1	84.7% (56)	0% (0)	0% (0)	0% (0)
futures-task-0.3.3	0	78.3% (8)	—	0% (0)	0% (0)
futures-task-0.3.5	0	68.2% (11)	—	0% (0)	0% (0)
simple-slab	2	100% (3)	75% (6)	0% (0)	0% (0)
ordnung	2 (1)	97.1% (69)	45.7% (18)	—	0% (0)
cbox	1	66.7% (11)	0% (2)	0% (0)	0% (0)
string-intern	0	100% (2)	33.3% (8)	33.3% (1)	33.3% (1)
http	0	86.4% (109)	48% (178)	—	—
qwutils	1	100% (10)	80% (143)	0% (0)	0% (0)
endian_trait	1	100% (9)	100% (60)	0% (0)	0% (0)
pnet_packet	1	100% (42)	—	!	0% (21)
rdiff	1	100% (4)	0% (23)	0% (3)	0% (2)
through	1	100% (4)	50% (1)	0% (0)	0% (0)
Total	**26 (6)**	**87.3% (1080)**	**21.8%(653)**	**4% (129)**	**3% (72)**

TABLE 2: Bug-finding capability of deepSURF and comparison of *URAPI Coverage* and number of compilable generated harnesses with other Rust fuzzing tools. — signifies cases where the tool crashes and ! denotes cases where harness generation did not finish within 24 hours.

deepSURF generates harnesses that detect 26 memory bugs by fuzzing, as detailed in the second column of Table 2. These include double-free (DF), buffer overflow (BOF), arbitrary memory accesses (SEGV), and use-after-free (UAF) violations. Out of the 26 found memory corruption vulnerabilities, 20 are previously-known which deepSURF detects automatically without any human involvement. In addition, deepSURF discovers 6 new memory bugs in three crates.

We have disclosed all the newly-discovered bugs to their affected developers. After our reporting, the bug in `toodee` has been patched and is pending RustSec ID assignment and the bugs in the maintained fork[3] of `slice-deque` are pending patches.

Many of the memory bugs detected by deepSURF involve intricate scenarios and advanced Rust features that pose challenges for automated tools. For example, triggering 12 bugs requires custom function implementations that panic or return unexpected values. Others, such as those in

2. The `xcb` crate is excluded due to its dependency on external software.

3. The crate `slice-deque` has been unmaintained since 2020, but the bugs we found also affect its maintained fork, `slice-ring-buffer`.

`slice-deque` and `simple-slab`, are triggered only by specific API call sequences. These bug-triggering patterns are typically found only in human-written PoCs, yet deepSURF generated harnesses containing them automatically.

deepSURF misses 11 reported memory bugs across the remaining 11 crates. These require language features beyond its current support, including async programming, multi-threading, and sequences of function calls that involve not only library-defined APIs but also other internal operations like unexpected calls to `std::mem::forget`.

RQ1: deepSURF identified 26 memory safety bugs (including 6 previously-unknown) by fuzzing the harnesses that it automatically generated.

RQ2: Comparison with State-of-the-Art Tools. We evaluated existing Rust fuzzing tools on our dataset to compare their bug-finding capabilities with deepSURF. Specifically, we ran RUG [11], RPG [50] and RULF [23] to generate harnesses for the libraries in our dataset and fuzzed these targets to compare their findings with deepSURF. For this comparison, the other tools were tested with their original settings. We could not test RuMono [54] and FRIES [51], as their code was not publicly available at the time of writing.

None of the other tools were able to automatically expose any of the bugs that deepSURF uncovered in our dataset. Moreover, as shown in Table 2, deepSURF achieved a high overall *URAPI Coverage* (see Section 2) of 87.3%, while RUG, RPG, and RULF achieved just 21.8%, 4%, and 3% respectively. This significant difference highlights deepSURF's focus on fuzzing unsafe code reachable through APIs, along with its broad support for Rust data types, enabling it to cover a substantially larger set of potentially vulnerable APIs.

RUG was the best-performing tool among the existing ones, generating harnesses of relatively high quality. While it failed to detect any memory corruption vulnerabilities when tested with its default fuzzing engine (libFuzzer [26]), we observed that it could potentially detect one of the bugs in the `toodee` crate if used with AFL++. In contrast, RULF and RPG exhibit limited argument-type support, preventing them from fully addressing **C2** and **C3**. As a result, their ability to fuzz complex libraries with diverse API argument types is limited.

None of the existing tools fully addresses **C4** (support for custom implementations simulating user-defined logic) or **C5** (support for generating semantically-related complex API sequences), which are critical for detecting complex memory corruption vulnerabilities. Moreover, all three tools failed to properly address **C1**, since they produced false positives due to crashes unrelated to actual memory safety bugs either by directly invoking unsafe code without preserving safety invariants or due to incorrect handling of panics and assertions. Finally, RUG and RPG failed to run on five crates, and RULF on one, due to crashes, timeouts, or errors caused by unimplemented features or implementation errors.

RQ2: deepSURF outperforms state-of-the-art Rust fuzzing tools in detecting memory corruption vulnerabilities by supporting complex sequences of APIs with diverse and complex argument types, enabling it to detect 26 memory safety bugs that other tools fail to uncover.

RQ3: Effectiveness of deepSURF's Components. We performed an ablation study to assess the impact of the three core components of deepSURF: (1) static analysis for static harness generation, (2) LLM-based augmentation, and (3) the *harness selection policies*. We fuzz three crates from our dataset, which require diverse harness characteristics to reveal memory bugs. We compared the bug-finding capability and unsafe code coverage deepSURF achieves in these crates against four configurations. *deepSURF-static* disables LLM-based augmentation, while *deepSURF-llm* omits static harness generation. Details regarding all the tested configurations, along with further discussion, are provided in Appendix C.

Our results show that deepSURF achieves the highest bug-finding capability—detecting seven memory corruption vulnerabilities—and the highest unsafe code coverage (86.2%) when combining static analysis with LLM-based augmentation. In contrast, *deepSURF-llm* and *deepSURF-static* detected only two and three bugs, with unsafe code coverage of 72.2% and 32.7%, respectively. Moreover, the default *harness selection policy* (see Section 4.4) detected one additional bug compared to the second-best configuration.

RQ3: deepSURF performs best when combining static analysis with LLM integration, as each component contributes uniquely and synergistically to its effectiveness.

6. Related Work

Several approaches have been proposed by researchers to identify bugs in Rust code. Static analysis tools such as Rudra [7] and Yuga [31] scan Rust source code to identify specific bug patterns and have uncovered numerous memory safety issues [17], [18]. However, these tools are limited to detecting specific bug types, suffer from a high false positive rate, and require manual effort to validate results and develop PoCs.

Dynamic analysis, which tests programs by executing them, has also been used in this domain. Specifically, sanitizers like ASan [38] instrument code to detect bugs during execution but introduce performance overhead. ERASAN [30] mitigates this overhead by leveraging safety checks already performed by `rustc`, but it cannot detect bugs on its own and must be integrated with fuzzers or manually written test cases. Grey-box fuzzers such as AFL++ [20] and libFuzzer [26] use coverage feedback to discover more bugs and are available in Rust via specific crates [5], [36]. However, since Rust code is typically distributed as libraries, it must first be harnessed. SyRust [41] synthesizes well-typed API call sequences by modeling Rust's type system, but its scalability and lack of input mutation limit its

testing power. CrabTree [40] extends SyRust with support for traits, closures, and integrates fuzzing, using feedback to guide test synthesis toward sequences that improve coverage and introduce new types. However, its effectiveness depends heavily on user-provided input templates, reducing automation. In contrast, deepSURF fully automates harness generation, eliminating the need for developers to understand library internals.

Some works have aimed to automate Rust fuzz harness generation. RULF [23] models API relationships with a dependency graph to generate harnesses. FRIES [51] improves RULF's scalability by fuzzing API sequences that reflect real-world usage. RPG [50] and RuMono [54] aim to maximize code coverage by supporting generics and traits, with RPG also targeting unsafe code. However, none has detected memory corruption bugs. RUG [11] combines LLMs and fuzzing to generate high-quality unit tests but primarily focuses on improving test coverage and is not well-suited for detecting memory safety bugs. deepSURF overcomes these limitations by targeting key challenges that, based on our analysis of real-world Rust vulnerabilities, underlie memory violations.

7. Conclusion

Fuzzing Rust for memory bugs is challenging due to its complex type system and the need for harness generation. In this work, we present deepSURF, a tool that automatically generates and fuzzes LLM-augmented harnesses to expose memory bugs in Rust libraries. deepSURF targets code with potential memory corruption risks, introduces novel approaches to support Rust data types and leverages LLMs to support complex sequences of semantically related APIs. Evaluated on 27 crates, deepSURF detected 26 memory corruption vulnerabilities, including 6 previously-unknown, outperforming existing tools.

References

[1] AFL++ Developers. Fuzzing in depth. https://aflplus.plus/docs/fuzzing_in_depth/, 2023. Accessed: 2025-06-02.

[2] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[4] Rust Fuzzing Authority. Add persistent mode to afl.rs. https://github.com/rust-fuzz/afl.rs/pull/137, 2018. GitHub pull request.

[5] Rust Fuzzing Authority. afl.rs: Fuzzing rust code with american fuzzy lop. https://github.com/rust-fuzz/afl.rs, 2022. Accessed: 2024-12-07.

[6] Rust Fuzzing Authority. Add cmplog support to afl.rs. https://github.com/rust-fuzz/afl.rs/pull/392, 2023. GitHub pull request.

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

[8] The Rust Programming Language Book. The rust programming language. https://doc.rust-lang.org/book. Online documentation.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, and Greg Brockman et al. Evaluating large language models trained on code, 2021.

[10] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 572–576, New York, NY, USA, 2024. Association for Computing Machinery.

[11] Xiang Cheng, Fan Sang, Yizhuo Zhai, Xiaokuan Zhang, and Taesoo Kim. Rug: Turbo llm for rust unit test generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 634–634. IEEE Computer Society, 2025.

[12] The Rust Community. crates.io: The rust community's crate registry. https://crates.io/. Accessed: December 6, 2024.

[13] Miri Contributors. Miri: An interpreter for rust's mid-level intermediate representation. https://github.com/rust-lang/miri, 2015. Accessed: December 7, 2024.

[14] Cybersecurity and Infrastructure Security Agency (CISA). The case for memory safe programming: Roadmaps for safer software. https://www.cisa.gov/case-memory-safe-roadmaps, 2023. Accessed: 2024-12-05.

[15] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, and Peiyi Wang et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[16] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 423–435, New York, NY, USA, 2023. Association for Computing Machinery.

[17] Rudra Developers. Rudra-poc. https://github.com/sslab-gatech/Rudra-PoC?tab=readme-ov-file. Accessed: 2025-01-07.

[18] Yuga Developers. Yuga repository. https://github.com/vnrst/Yuga, 2024. Accessed: 2025-06-03.

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

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

[21] Rust for Linux Contributors. Rust-for-linux/linux. https://github.com/Rust-for-Linux/linux, dec 2019. Accessed: December 5, 2024.

[22] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. Utopia: Automatic generation of fuzz driver using unit tests. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2676–2692, 2023.

[23] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. Rulf: rust library fuzzing via api dependency graph traversal. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 581–592. IEEE Press, 2022.

[24] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and Quan Zhang. When fuzzing meets llms: Challenges and opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 492–496, New York, NY, USA, 2024. Association for Computing Machinery.

[25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[26] LLVM Project. Libfuzzer — a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2024. https://llvm.org/docs/LibFuzzer.html.

[27] Bob Lord. The urgent need for memory-safe software products. https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products, 2023. Accessed: 2024-12-05.

[28] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3793–3807, New York, NY, USA, 2024. Association for Computing Machinery.

[29] Ian McCormack, Tomas Dougan, Sam Estep, Hanan Hibshi, Jonathan Aldrich, and Joshua Sunshine. A mixed-methods study on the implications of unsafe rust for interoperation, encapsulation, and tooling. https://arxiv.org/abs/2404.02230, 2024.

[30] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. Erasan: Efficient rust address sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4053–4068, 2024.

[31] Vikram Nitin, Anne Mulhern, Sanjay Arora, and Baishakhi Ray. Yuga: Automatically detecting lifetime annotation bugs in the rust language. *IEEE Trans. Softw. Eng.*, 50(10):2602–2613, August 2024.

[32] Alex Ozdemir. Unsafe in rust: The abstraction safety contract and public escape. https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-escape/, 2022. Accessed: 2025-06-04.

[33] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 763–779, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Alex Rebert and Christoph Kern. Secure by design: Google's perspective on memory safety. Technical report, Google Security Engineering, 2024.

[35] The Register. Microsoft is rewriting core windows libraries in rust. https://www.theregister.com/2023/04/27/microsoft_windows_rust/, 2023. Accessed: December 5, 2024.

[36] Rust Fuzzing Authority. cargo-fuzz: A cargo subcommand for fuzzing with libfuzzer! https://github.com/rust-fuzz/cargo-fuzz/. Accessed: 2025-01-07.

[37] S2-Lab. Erasan pocs. https://github.com/S2-Lab/ERASan/tree/main/poc. Accessed: 2025-01-10.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[39] Diane B. Stephens, Kawkab Aldoshan, and Mustakimur Rahman Khandaker. Understanding the Challenges in Detecting Vulnerabilities of Rust Applications . In *2024 IEEE Secure Development Conference (SecDev)*, pages 54–63, Los Alamitos, CA, USA, October 2024. IEEE Computer Society.

[40] Yoshiki Takashima, Chanhee Cho, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. Crabtree: Rust api test synthesis guided by coverage and type. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[41] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 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, page 899–913, New York, NY, USA, 2021. Association for Computing Machinery.

[42] OSS-Fuzz Team. Fuzz target generation using llms. https://google.github.io/oss-fuzz/research/llms/target_generation/. Accessed: 2025-05-28.

[43] The Rust Reference. Behavior considered undefined. https://doc.rust-lang.org/reference/behavior-considered-undefined.html. Accessed: 2024-12-21.

[44] The Rust Reference. Visibility and privacy. https://doc.rust-lang.org/reference/visibility-and-privacy.html. Accessed: 2025-05-26.

[45] The Rust Team. Rust by example - struct visibility. https://doc.rust-lang.org/rust-by-example/mod/struct_visibility.html. Accessed: 2025-05-26.

[46] The RustSec Advisory Database. Rustsec: A security advisory database for rust. https://rustsec.org. Accessed: 2024-12-05.

[47] Gavin Thomas. A proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/, 2019. Accessed: 2024-12-05.

[48] C. Zhang X. Wu, N. Cheriere and D. Narayanan. Rustgen: An augmentation approach for generating compilable rust code with large language models. In *ICML 2023 Workshop on Deployment Challenges for Generative AI*, 2023.

[49] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. Memory-safety challenge considered solved? an in-depth study with all rust cves. *ACM Trans. Softw. Eng. Methodol.*, 31(1), September 2021.

[50] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. 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*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Xizhe Yin, Yang Feng, Qingkai Shi, Zixi Liu, Hongwang Liu, and Baowen Xu. Fries: Fuzzing rust library interactions via efficient ecosystem-guided target generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1137–1148, New York, NY, USA, 2024. Association for Computing Machinery.

[52] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[53] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. How effective are they? exploring large language model based fuzz driver generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1223–1235, New York, NY, USA, 2024. Association for Computing Machinery.

[54] Yehong Zhang, Jun Wu, and Hui Xu. Rumono: Fuzz driver synthesis for rust generic apis. *ACM Trans. Softw. Eng. Methodol.*, December 2024. Just Accepted.

Appendix A.
Data Types Supported by deepSURF and Type Analysis

Category	Examples
1. Primitive Types	Integers (e.g., `i32`, `usize`), Booleans (`bool`), Floating-point numbers (`f64`), Characters (`char`)
2. Standard Library Types	Strings (`String`), Vectors (`Vec<T>`), Options (`Option<T>`), Results (`Result<T, E>`), Boxes (`Box<T>`)
3. Slices	Common Slices (`&[T]`), String Slices (`&str`)
4. Compound Types	Arrays (`[T; N]`), Tuples (`(T1, T2, ..., Tn)`)
5. Complex Types	Structs, Enums
6. Reference and Pointer Types	References (`&T`, `&mut T`), Raw Pointers (`*const T`, `*mut T`)
7. Generic Types	Type parameters (e.g., `G`) serve as placeholders for compatible concrete types and may require traits (`G: Trait1 + Trait2 + ... + Traitn`)
8. Rust-Specific Types	Closures, Trait-associated Types, Dynamic Trait Objects (`dyn Trait`)

TABLE 3: Rust data types supported by deepSURF. `N` refers to an integer; `T`, `T1`, `T2`, ..., `Tn` refer to types from categories 1 to 7; `Trait`, `Trait1`, `Trait2`, ..., `Traitn` define required behaviors; and `E` represents an error type.

Table 3 presents the full list of Rust data types supported by deepSURF, while Algorithm 1 outlines the recursive analysis of function arguments performed by the tool. This analysis enables deepSURF to extract *URAPI* argument types and identify appropriate generation strategies. By marking already-analyzed functions, deepSURF avoids infinite recursion and efficiently records and reuses type information.

Algorithm 1 Recursive Type Analysis

```
1: function ANALYZE_ARGS(fn)
2:     for all arg ∈ fn.args do
3:         ANALYZE_ARG(arg)
4:     end for
5: end function
6: function ANALYZE_ARG(arg)
7:     atype ← GET_ARG_TYPE(arg)
8:     if IS_GENERIC(atype) then
9:         for all fn ∈ GET_TRAIT_FNS(arg) do
10:            MARK(fn); ANALYZE_ARGS(fn)
11:        end for
12:    end if
13:    if IS_COMPLEX(atype) then
14:        for all ctor ∈ GET_CONSTRUCTORS(arg) do
15:            MARK(ctor); ANALYZE_ARGS(ctor)
16:        end for
17:    end if
18:    if IS_CONTAINER(atype) then
19:        ANALYZE_ARG(GET_INNER_ARG(arg))
20:    end if
21:    return atype
22: end function
```

Appendix B.
Harnesses Selector Policies

When experimenting with LLM-based augmentation, we observed that, despite prompting the LLM against it, the LLM often removes statically-generated custom functionality during augmentation. However, such custom logic can be crucial for exposing memory bugs during fuzzing since it allows simulation of user-defined code execution (**C4**). To avoid missing bugs that may be triggered only by statically generated harnesses, the *harness selection policy* of the *Harness Selector* includes up to four of the corresponding statically generated harnesses in the *Fuzzing Corpus* for all *URAPIs* with custom functionality.

Regarding the selection of up to four harnesses per *URAPI*, we experimented with *deepSURF-static* (see Appendix C) generating different numbers of harnesses by selecting varying numbers of dependency trees. Specifically, we tested configurations generating up to one (1fh), two (2fh), and four (4fh) harnesses per *URAPI*, and fuzzed them for six hours. Table 4 summarizes the vulnerabilities detected in each case. Results show that 4fh achieved the best results. For this reason, we select up to four statically generated harnesses per *URAPI* for inclusion in the *Fuzzing Corpus* when the *URAPI* involves custom functionality or when LLM-based augmentation fails.

RUSTSEC ID	Crate	Bug Type	1fh	2fh	4fh
RUSTSEC-2021-0018	qwutils	DF	✓	✓	✓
RUSTSEC-2021-0028	toodee	HBOF	✓	✓	✓
RUSTSEC-2021-0028	toodee	DF	✗	✓	✓
*	toodee	HBOF	✗	✓	✓
*	slice_deque	DF	✓	✓	✓
*	slice_deque	DF	✓	✓	✓
RUSTSEC-2021-0047	slice_deque	DF	✗	✓	✓
RUSTSEC-2021-0094	rdiff	HBOF	✓	✓	✓
RUSTSEC-2021-0053	algorithmica	DF	✓	✓	✓
RUSTSEC-2021-0049	through	DF	✓	✓	✓
RUSTSEC-2021-0039	endian_trait	DF	✓	✓	✓
RUSTSEC-2020-0167	pnet_packet	HBOF	✓	✓	✓
RUSTSEC-2020-0005	cbox	SEGV	✓	✓	✓
RUSTSEC-2021-0042	insert_many	DF	✓	✓	✓
RUSTSEC-2020-0038	ordnung	DF	✗	✗	✓
*	ordnung	UAF	✗	✗	✓

TABLE 4: Bug detection capability of deepSURF without LLM-based harness augmentation. Each row corresponds to a unique bug. ✓ indicates detection via fuzzing, ✗ indicates failure to detect. * signifies a new bug discovered by deepSURF; otherwise, the assigned RustSec ID is provided.

Appendix C.
Results of the Ablation Study

To assess how deepSURF's core components contribute to its effectiveness in detecting memory safety bugs and fuzzing unsafe code reachable through a library's API, we conducted an ablation study targeting three key components: (1) static analysis for static harness generation, (2) LLM-based augmentation, and (3) the *harness selection policies*. We compared the full version of deepSURF against four alternative configurations, each disabling or modifying one of these components, as summarized in Table 5.

We evaluated these configurations on three crates from the ERASAN dataset that contain memory safety bugs requiring harnesses with diverse characteristics to be exposed. Since each configuration yields a different number of harnesses, and to ensure a fair comparison, we allocated the same total fuzzing time per configuration: 672 CPU-hours. Thus, the fuzzing time per harness is computed as:

$$\text{FuzzingTimePerHarness} = \frac{672}{\#\text{Harnesses}}$$

This approach gives more fuzzing time per harness to configurations that yield fewer harnesses. We fuzz each target using two AFL++ threads in a master-slave setup: one with ASan and the other with CmpLog.

To measure unsafe code coverage, we count code lines within unsafe blocks or unsafe functions that are reachable through safe APIs of the evaluated crates. We exclude empty lines, comments, and feature-gated code not compiled under the default feature set.

Configuration	Components		
	Static Harness Generation	LLM-based Augmentation	Harness Selection Policy
deepSURF	✓	✓	Skip *URAPIs* without custom functionality
deepSURF-static	✓	✗	—
deepSURF-llm	✗	✓	—
deepSURF-skip-all	✓	✓	Skip any *URAPI*
deepSURF-no-skip	✓	✓	Do not skip

TABLE 5: Configurations used in the ablation study. Each disables or modifies core components of deepSURF to evaluate their impact. Symbols indicate: ✓= enabled, ✗ = disabled, and — = not applicable.

C.1. Static vs. LLM-based Analysis

We distinguish the *deepSURF-static* and *deepSURF-llm* configurations to evaluate the individual contributions of static analysis and LLM integration to deepSURF's ability to fuzz unsafe code and uncover memory corruption bugs.

In *deepSURF-static*, LLM-based augmentation is disabled, and the *Fuzzing Corpus* consists solely of compilable harnesses produced during the static harness generation stage. In contrast, *deepSURF-llm* disables static analysis entirely—no initial harnesses are statically generated. Instead, the model is prompted to perform control-flow analysis to identify *URAPIs* and synthesize harnesses from scratch. In this case, we use a similar prompting strategy to the full deepSURF setup, but instead of seeding the LLM with a statically generated harness, we provide a simple template that specifies the expected fuzzer macro structure and includes examples for converting fuzzer bytes into basic Rust types (e.g., primitives and strings). Also, no static analysis metadata is included in the prompt. The results of the comparison are summarized in Table 6a.

Based on our results, deepSURF achieves the highest bug-finding capability (seven bugs) and the highest unsafe code coverage (86.72%) compared to the other two configurations, demonstrating that the combination of static analysis and LLM-based augmentation outperforms approaches that rely on only one of these components.

We observed that *deepSURF-llm* struggled to identify *URAPIs*, particularly in the case of `toodee`, where it returned APIs that could not reach unsafe code and failed to generate harnesses for 65% of the crate's *URAPIs*. Although it achieved high unsafe code coverage in `smallvec-1.6.0`, it did not trigger the bug, which required custom trait implementations—a feature supported by deepSURF through its static analysis. In contrast, for `simple-slab`, *deepSURF-llm* performed comparably to deepSURF, discovering both memory safety bugs. Upon inspection, we found that `simple-slab` has a relatively simple API (e.g., lacking complex trait relationships), and its bugs depend primarily on sequences of API calls, a feature supported in both deepSURF and *deepSURF-llm*.

On the other hand, *deepSURF-static* detected only three bugs in `toodee` and achieved much lower unsafe code coverage than the other configurations. Its success in `toodee` is attributed to static control flow analysis and support for custom trait implementations. However, it failed to find any bugs in `simple-slab` due to the lack of sequence support. Also, it could not find the bug in `smallvec-1.6.0`, where all *URAPIs* require generic arguments implementing the unsafe `Array` trait, feature supported only by the LLM-based augmentation. Consequently, it was unable to generate valid harnesses for `smallvec-1.6.0`.

C.2. Comparing Harness Policies

In Section 4.4, we described the operation of the *Harness Selector* and *Harness Augmenter*, along with their corresponding *harness selection policies*. These policies influence both the composition and size of the final *Fuzzing Corpus* and, consequently, affect deepSURF's bug-finding capability. In this section, we evaluate alternative *harness selection policies* for both components.

We consider two additional configurations: *deepSURF-skip-all* and *deepSURF-no-skip*. In *deepSURF-skip-all*, the *Harness Augmenter*'s *harness selection policy* is modified to skip augmentation for a statically generated harness if its target *URAPI* has already been invoked in a previously augmented harness—regardless of whether the current harness includes custom functionality. In contrast, *deepSURF-no-skip* disables the skip logic entirely, augmenting all statically generated harnesses. In both configurations, the *Harness Selector*'s *harness selection policy* still falls back to statically generated harnesses when LLM-based augmentation fails. However, unlike the default deepSURF configuration (see Section 4.4), neither configuration retrieves statically generated harnesses for *URAPIs* whose arguments support custom user-defined implementations.

The results of this comparison are presented in Table 6b. Among all *harness selection policies*, deepSURF's default policy achieves the highest bug-finding capability. The second-best configuration is *deepSURF-no-skip*, which

Crate	deepSURF			deepSURF-llm			deepSURF-static		
	Detected Bugs	Unsafe Coverage	URAPI Coverage (#Harnesses)	Detected Bugs	Unsafe Coverage	URAPI Coverage (#Harnesses)	Detected Bugs	Unsafe Coverage	URAPI Coverage (#Harnesses)
toodee	4	83.8%	90% (35)	0	47.2%	35% (9)	3	83.1%	81.7% (159)
simple-slab	2	100%	100% (3)	2	100%	100% (12)	0	50%	100% (14)
smallvec-1.6.0	1	86.8%	84.7% (56)	0	85.9%	66.7% (44)	0	0%	0% (0)
TOTAL	7	86.2%	87.9% (94)	2	72.2%	56.4% (65)	3	32.7%	40.7% (173)

(a) Impact of static analysis and LLM integration.

Crate	deepSURF			deepSURF-skip-all			deepSURF-no-skip		
	Detected Bugs	Unsafe Coverage	URAPI Coverage (#Harnesses)	Detected Bugs	Unsafe Coverage	URAPI Coverage (#Harnesses)	Detected Bugs	Unsafe Coverage	URAPI Coverage (#Harnesses)
toodee	4	83.8%	90% (35)	2	92.9%	91.7% (31)	3	95.8%	95% (49)
simple-slab	2	100%	100% (3)	2	100%	100% (3)	2	100%	100% (9)
smallvec-1.6.0	1	86.8%	84.7% (56)	0	81.5%	81.9% (22)	1	89.4%	84.7% (49)
TOTAL	7	86.2%	87.9% (94)	4	84.9%	87.1% (56)	6	89.1%	90% (107)

(b) Alternative Harness Selection Policies of the *Harness Augmenter*.

TABLE 6: Ablation study results across different deepSURF's configurations.

detects six bugs. The only missed bug is one that deepSURF identifies using a statically generated harness containing custom trait implementations. As discussed earlier, *deepSURF-no-skip* does not retrieve such harnesses from the static generation stage.

On the other hand, *deepSURF-skip-all* misses two additional bugs compared to *deepSURF-no-skip*. This occurs because it skips augmentation of statically generated harnesses that involve custom implementations, based solely on the fact that the corresponding *URAPI* has already been invoked in another augmented harness. The issue in these cases is that the LLM is unaware of the custom implementations derived from our tool's static analysis, as they are encoded only in the statically generated harnesses. As a result, the LLM calls the *URAPI* using its own heuristics, which may fail to expose the bug.

Finally, we observe that deepSURF achieves lower unsafe code coverage than *deepSURF-no-skip*, yet it discovers more bugs within the same set of *URAPIs*. This highlights that while high unsafe code coverage allows exploration of more potentially vulnerable regions of a Rust library, it does not necessarily result in finding more bugs. Our experiments show that bug-finding capability depends not only on the amount of unsafe code covered, but also on the context in which that code is exercised. This context is influenced by the types of arguments passed to the APIs and the specific sequences in which those APIs are invoked.