RUDRA: Finding Memory Safety Bugs in Rust at the Ecosystem Scale

Yechan Bae Youngsuk Kim Ammar Askar Jungwon Lim Taesoo Kim
Georgia Institute of Technology

Abstract

Rust is a promising system programming language that guarantees memory safety at compile time. To support diverse requirements for system software such as accessing low-level hardware, Rust allows programmers to perform operations that are not protected by the Rust compiler with the `unsafe` keyword. However, Rust's safety guarantee relies on the soundness of all `unsafe` code in the program as well as the standard and external libraries, making it hard to reason about their correctness. In other words, a single bug in any `unsafe` code breaks the whole program's safety guarantee.

In this paper, we introduce RUDRA, a program that analyzes and reports potential memory safety bugs in `unsafe` Rust. Since a bug in `unsafe` code threatens the foundation of Rust's safety guarantee, our primary focus is to scale our analysis to all the packages hosted in the Rust package registry. RUDRA can scan the entire registry (43k packages) in 6.5 hours and identified 264 previously unknown memory safety bugs—leading to 76 CVEs and 112 RustSec advisories being filed, which represent 51.6% of memory safety bugs reported to RustSec since 2016. The new bugs RUDRA found are non-trivial, subtle, and often made by Rust experts: two in the Rust standard library, one in the official `futures` library, and one in the Rust compiler. RUDRA is open-source, and part of its algorithm is integrated into the official Rust linter.

CCS Concepts: • **Theory of computation → Program analysis**; • **Security and privacy → Software and application security**.

Keywords: Rust, Memory-safety, Program analysis

1 Introduction

Rust is an emerging programming language for system software. As a system language like `C` or `C++`, its primary concern is to enable native performance and to allow programmers complete control of resource management. Unlike traditional system languages, however, Rust enables these features in a memory-safe way by default. This unique paradigm which provides both safety and performance, makes Rust appealing for developing system software. Rust has started to receive major adoption in conventional system software such as operating systems [22, 33, 36, 47], embedded systems [26], web frameworks, [21] and web browsers [59], where both security and performance are indispensable.

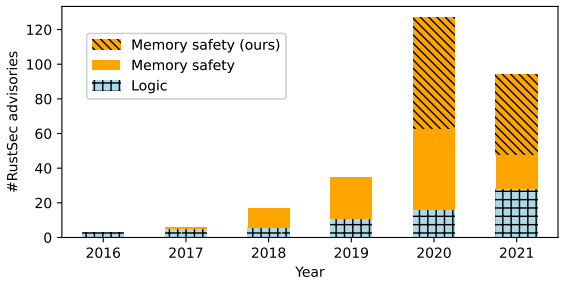

Figure 1. RUDRA found 264 new memory safety bugs in the Rust ecosystem. They received 112 RustSec advisories, which represent 51.6% of the memory safety bugs reported to the official Rust security advisory database, RustSec [39].

The key idea of Rust's memory safety is to validate the *ownership* of memory at compile time, where the compiler validates the access and the lifetime of memory-allocated objects (or values). Simply put, each value in Rust has an owner variable, and the memory used for the value is immediately reclaimed when the owner variable goes out of scope. Rust's ownership system is often viewed as similar in concept to substructural type systems [61, 62] but supports a novel concept of *borrowing* that allows the creation of shared or mutable references to values. The compiler's borrow checker provides two guarantees: 1) references cannot outlive their owner variables, preventing use-after-free (UAF) vulnerabilities and 2) both shared and mutable references are never present at the same time, eliminating the possibility of concurrent read and update to the value (Figure 3).

Unfortunately, such safety rules are often too restrictive in certain system software that requires low-level hardware access (*e.g.*, accessing raw pointers) or hamper performance and temporarily need to be bypassed (*e.g.*, creating uninitialized objects). Since these requirements cannot be addressed by safe Rust but are essential to system software, Rust introduces the concept of `unsafe`, in which the duty of the

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483570

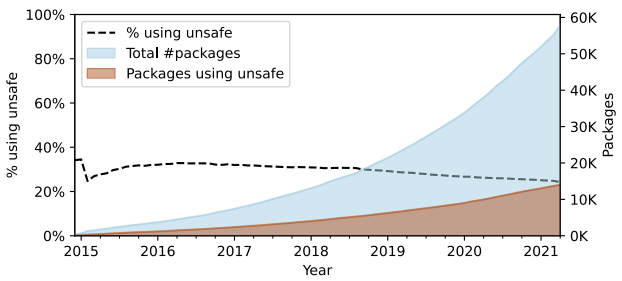

Figure 2. Although the number of packages grows exponentially, the percentage of packages using `unsafe` code remains consistently around 25-30%, similar to other reports [16, 31].

compiler's safety check is temporarily delegated to the programmer. Although `unsafe` is an opt-in feature in Rust, most system software like OSes or standard libraries cannot be implemented without it, and 25-30% of Rust packages directly use `unsafe` in their code for various reasons [16, 31].

The soundness of `unsafe` Rust code is critical to the memory safety of the whole program and, alas, is difficult to reason about. Some people naively believe that using `unsafe` code sparsely or exercising extreme caution in reviewing the source code can avoid such problems. However, reasoning about soundness is subtle and error-prone for the following three reasons: 1) a soundness bug transitively breaks Rust's safety boundaries, meaning that all external code including standard libraries need to first be sound; 2) safe and `unsafe` code located distantly are often interdependent [4, 11]; and 3) all non-visible code paths inserted by the compiler need to be reasoned about correctly by the programmer (*e.g.*, reclaiming objects). Since a single soundness bug breaks the safety guarantee of the entire Rust program, the Rust community considers such a bug security-critical and essential to the foundation of Rust's safety guarantee [45].

Many existing research works have contributed to building a foundation of soundness in the Rust ecosystem. There has been a large body of research projects in formalizing Rust's type system and operational semantics [27, 43, 44, 51, 63, 64], in verifying its correctness [17, 46], and in model checking [20, 25, 60]. These are all important steps toward making a sound, theoretical foundation but are not yet practical enough to scale to the entire ecosystem. Similarly, dynamic approaches exist, such as Miri, that detect certain classes of undefined behavior by interpreting the compiler IR [53] and fuzzing that performs random testing [18]. Unfortunately, these dynamic approaches cannot be easily adopted at a large scale because they require extensive computational resources or non-trivial amounts of development effort—2.7% of packages in the registry support fuzzing. As a new language, Rust is rapidly gaining popularity, but the number of packages using `unsafe` is also keeping pace (Figure 2). It is thus important to devise practical algorithms that can proactively assure the memory safety of all packages in the registry.

```rust
fn ownership_and_borrowing() -> &u32 {
    // creates a `Vec`, a heap allocated buffer
    let vec = vec![1, 2, 3];

    // creates a reference to the first value with borrowing
    let first_val = &vec[0];

    // Ownership: `Vec` is automatically reclaimed
    // when its owner `vec` goes out of the scope.
    //
    // Borrowing: compile error; Rust prevents `first_val`
    // to outlive `vec` by tracking variable lifetimes.
    return first_val;
}

fn aliasing_xor_mutability() {
    let mut vec = vec![1, 2, 3];

    // exclusive mutable borrowing
    let mut_ref = &mut vec;

    // shared read-only borrowing
    let shared_ref1 = &vec;
    let shared_ref2 = &vec;
    println!("{}", shared_ref1[0]);
    println!("{}", shared_ref2[0]);

    // Aliasing xor Mutability: compile error;
    // Rust invalidates `mut_ref` when `shared_ref1` is
    // used since they cannot coexist at the same time.
    mut_ref.push(4);
}
```

Figure 3. Example code that shows Rust's core concepts: (1) ownership, (2) borrowing, and (3) aliasing xor mutability

In this paper, we present three important bug patterns in `unsafe` Rust code and introduce a tool called RUDRA that can quickly recognize error-prone parts of `unsafe` code. It can scan the entire 43k packages in the Rust package registry (`crates.io`) in 6.5 hours and has found 264 previously unknown memory safety bugs—leading to 76 CVE records being filed to the CVE database, as well as 112 advisories to the official Rust security advisory database, RustSec [39]. This is an unprecedented number of memory safety bugs in the Rust ecosystem. As of September 2021, these bugs represent 39.0% of all bugs and 51.6% of memory safety bugs reported to RustSec since 2016 (Figure 1). These bugs are subtle and non-trivial, found even in code written and extensively reviewed by Rust experts: two in the standard library, one in the official `futures` library, and one in the Rust compiler.

We make three key contributions:

- **Scalable algorithms.** We identified three bug patterns in `unsafe` Rust and devised two new algorithms that can discover them. We implemented the algorithms as RUDRA, a static analyzer that can scale to all the programs in the Rust package registry.
- **New bugs.** RUDRA found 264 new memory safety bugs in the Rust ecosystem. They represent more than half (51.6%) of the memory safety bugs known to RustSec since 2016.
- **Open source.** RUDRA is open-sourced at https://github.com/sslab-gatech/Rudra, and part of its core algorithm is integrated into the official Rust linter, Clippy [52].

2 Background

2.1 The Foundation: Safe Rust

Rust is a memory-safe programming language [54]. It provides strong guarantees at compile time with three core ideas: ownership, borrowing, and aliasing xor mutability (Figure 3).

Ownership. Each value in Rust has an owner variable that determines the lifespan of the value. A value is initialized when the owner variable is created, and the memory associated with the value is automatically reclaimed when its owner goes out of scope. The Rust compiler tracks the lifetime of each value via the ownership system and inserts the required reclamation routines (`drop()`), similar to the Resource-Acquisition-Is-Initialization (RAII) pattern in other languages.

Borrowing. Rust allows a value to be borrowed (*i.e.*, creating a reference to it) during the lifetime of the owner variable. With borrowing, a value can be read or updated without changing the ownership of the value. Rust's type system ensures that traditional memory safety issues like use-after-free or dangling pointers cannot happen by disallowing references that outlive the owner variable.

Aliasing xor mutability. There are two types of borrowing: 1) shared borrowing for read access and 2) exclusive mutable borrowing for write access. The Rust compiler ensures that both shared and mutable references are never present at the same time. This means that concurrent reads and writes are fundamentally impossible in Rust, eliminating the possibility of conventional race conditions and memory safety bugs like accessing invalid references (*e.g.*, iterator invalidation in `C++`). This property also guides programmers to confine mutability, which can help prevent other logic bugs.

2.2 The Necessity: Unsafe Rust

Unfortunately, Rust's safety rules are often too restrictive to model low-level hardware behaviors that are required for system software. Rust can neither perform memory-mapped I/O in an OS kernel (*i.e.*, accessing memory through raw pointers) nor invoke a system call, as the Rust language does not understand their semantics to conclude their safety. Rust relies on the axiomatic foundation provided by the authors of `unsafe` code to incorporate these operations under Rust's safety model.

In Rust, developers can declare their own axioms that are beyond Rust's type system with the `unsafe` keyword. It is called `unsafe` because the Rust compiler cannot check the safety of the provided code and assumes the provided `unsafe` code is sound and bug-free. As a result, a single bug in `unsafe` code, regardless of whether it is from the developer's own code or from a library, can subvert the safety guarantee of the entire Rust program. In this regard, `unsafe` should be used sparsely and with extreme caution across any Rust code.

Interestingly, the use of `unsafe` is much more commonplace than ideal—it has been reported that 25-30% of Rust packages utilize `unsafe` in their code [16, 31]. In principle, `unsafe` usage should be limited *internally* to packages that provide high-level abstractions and data structures. For instance, the Rust standard library, `std`, uses `unsafe` to implement containers that support dynamic buffer allocations (*e.g.*, `Vec`), smart pointers that extend Rust's default ownership model (*e.g.*, `Rc`), synchronization primitives (*e.g.*, `Mutex`), and OS abstractions (*e.g.*, `File`). However, despite the addition of new features to the standard library and mature packages offering safe APIs around `unsafe` primitives, the ratio of packages utilizing `unsafe` is declining very slowly (Figure 2).

```rust
// The caller needs an `unsafe` keyword to call this function.
// The caller is responsible for providing a correct index.
unsafe fn get_unchecked(index: usize) -> Output { ... }

// The API author guarantees that this function is safe to call
// with any `index` value (e.g., by checking the array bound).
fn get(index: usize) -> Option<Output> { unsafe { ... } }
```

Figure 4. Unsafe Rust code can be directly exposed or encapsulated, which determines the responsibility of safety bugs.

Encapsulated unsafety. The `unsafe` keyword introduces an interesting design domain: a way to communicate the safety of APIs. Rust developers have two choices when building a high-level abstraction with `unsafe`. The internal `unsafe` code can be directly exposed to the API users or can be *encapsulated* with a safe API (Figure 4). It is considered more idiomatic to hide such unsafety in user-facing APIs. When an API is defined safe, it is assuring that it conforms to Rust's safety rule that no input can trigger a memory safety bug and its internal unsafety is properly guarded.

Responsibility for safety bugs. The separation of safe and `unsafe` definitions makes it possible to distinguish who is responsible for a safety bug: "No matter what, safe Rust can't cause undefined behavior" [58]. In other words, it is always a safe API's responsibility to ensure that any valid input does not lead to a memory safety violation in encapsulated `unsafe` code. This is in stark contrast to `C` or `C++` where it is the user's responsibility to correctly obey the intended usage of the API. For example, no one would fault `printf()` in `libc` if the API call causes a segmentation fault when provided an incorrect pointer, yet this exact problem has led to a popular class of memory safety issues: the format-string vulnerability. In Rust, `println!()` never causes a segmentation fault no matter how it is used. Moreover, if a valid input does cause a segmentation fault in safe API, it is considered the API developer's fault.

2.3 Defining Memory Safety Bugs in Rust

There are two types of `unsafe` definitions in Rust: `unsafe` functions and `unsafe` traits. An `unsafe` function requires the caller to uphold certain properties when calling the function. For instance, `slice::get_unchecked()` requires the caller to provide a correct index because it does not perform bounds checking. There are `unsafe` intrinsic functions that are part of the language, and violating their safety invariant leads to

```rust
fn double_drop<T>(mut val: T) {
    unsafe { ptr::drop_in_place(&mut val); }
    drop(val);
}

double_drop(123); // no memory-safety violation when `T=u32`
double_drop(vec![1, 2, 3]); // double-free when `T=Vec<u32>`
```

Figure 5. Generic function is considered to have a memory safety bug if any of its instantiation has a memory safety bug.

a memory safety violation. Traits are similar to interfaces in other languages. They declare a list of expected methods on a type, and a type implements a trait by providing the required methods. An unsafe trait requires additional semantic guarantees from the implementer because other `unsafe` code may rely on its correctness. For instance, an iterator that implements the `unsafe` `TrustedLen` trait must provide a correct value for `size_hint()`.

With `unsafe` definitions and Rust's core safety statement, we clarify the definition of memory safety bugs in Rust along with the necessary terminology. Our goal is to provide concise definitions, not the full operational semantics of Rust [44, 51, 63, 64], so that we have common ground to describe certain behaviors of Rust programs.

Definition 2.1. A *type* and a *value* are defined in a conventional manner [56]. A type is considered as a set of values.

Definition 2.2. For a type T, safe-value(T) is defined as values that can be safely created. For instance, Rust's string is internally represented as a byte array, but it can only contain UTF-8 encoded values when created via safe APIs.

Definition 2.3. A function F takes a value of type $arg(F)$ and returns a value of type $ret(F)$. We consider a function that takes multiple arguments as if it takes a tuple of values.

Definition 2.4. A function F has a memory safety bug if $\exists v \in$ safe-value($arg(F)$) such that calling $F(v)$ triggers a memory safety violation or generates a return value $v_{ret} \notin$ safe-value($ret(F)$).

Example 2.5. Consider a function that overwrites the length field of a `Vec` with `usize::MAX`. This does not cause an immediate memory safety violation, since it is just an integer field write. However, it will break other safe code and should be considered a bug. Hence, a function that generates a non-safe-value is also considered to have a memory safety bug. In this example, a vector with an incorrect length (*i.e.*, `usize::MAX`) is a value but not a safe-value of `Vec`.

Definition 2.6. For a generic function Λ, pred(Λ) is defined as a set of types that satisfies the type predicate [55] of Λ. Given a type $T \in$ pred(Λ), resolve(Λ, T) instantiates a generic function Λ to a concrete function F.

Definition 2.7. A generic function Λ has a memory safety bug if it can be instantiated to a function that has a memory safety bug, i.e., $\exists T \in$ pred(Λ) such that $F =$ resolve(Λ, T) has a memory safety bug.

Example 2.8. Consider a function that accepts a single argument of generic type `T` and drops it (*i.e.*, calls its destructor) twice (Figure 5). This function does not cause a memory safety bug with integer types because dropping an integer is no-op. However, calling this function with allocating types (*e.g.*, `Vec`) leads to a security-critical double-free bug. Thus, this generic function is considered to have a memory safety bug because it has an instantiation that causes a memory safety bug. The correctness of a generic function depends on its type predicate. The function would not have a memory safety bug if it specified a type predicate `T: Copy`, since `Copy` types cannot have a destructor (all integer types are `Copy`).

Rust uses two unsafe traits, `Send` and `Sync`, to encode memory safety with multiple threads, *i.e.*, thread safety. The Rust compiler automatically implements them for simple user-defined types. However, types that interact with `unsafe` code often require manual implementations of `Send` and `Sync`.

Definition 2.9. An ownership of a type that implements the `Send` trait can be sent to another thread. A `Send` implementation has a memory safety bug if it is implemented on a type whose ownership cannot be transferred to another thread.

Definition 2.10. A type that implements the `Sync` trait can be accessed concurrently from different threads through shared references. A `Sync` implementation has a memory safety bug if it is implemented on a type that defines a non-thread-safe method that takes a shared `self` reference, `&self`.

Example 2.11. The basic reference-counted smart pointer `Rc<T>` is neither `Send` nor `Sync` because it can be cloned through a shared reference, which modifies its counter in a non-thread-safe way. On the other hand, the atomic version `Arc<T>` is `Send` and `Sync` if the inner type `T` is `Send` and `Sync`.

3 Pitfalls of Unsafe Rust

It is commonly thought that memory safety bugs in Rust are infrequent for the following three reasons. First, unsafe Rust is explicit so it stands out in the code. Unsafe Rust can only be used in a block or a function marked with the `unsafe` keyword, which signals the developer and the reviewer to thoroughly check the required safety invariants [31]. Second, it is a common practice to keep `unsafe` blocks simple, short, and self-contained to alleviate the burden of manually reasoning about their soundness [16]. Third, the vast majority of Rust applications can be implemented without using `unsafe` at all. More than 70% of Rust packages are implemented without using `unsafe` [16, 31].

Despite these hopeful beliefs, implementing safe abstraction using `unsafe` in Rust is error-prone and subtly difficult, requiring different coding practices than writing safe Rust code. We conducted a qualitative analysis of known Rust vulnerabilities, as well as an audit of popular Rust packages, and identified three non-trivial root causes of such bugs in `unsafe` code, explained as follows.

```
1  // CVE-2020-36317: a panic safety bug in String::retain()
2  pub fn retain<F>(&mut self, mut f: F)
3      where F: FnMut(char) -> bool
4  {
5      let len = self.len();
6      let mut del_bytes = 0;
7      let mut idx = 0;
8
9  +   unsafe { self.vec.set_len(0); }
10     while idx < len {
11         let ch = unsafe {
12             self.get_unchecked(idx..len).chars().next().unwrap()
13         };
14         let ch_len = ch.len_utf8();
15
16         // self is left in an inconsistent state if f() panics
17 *       if !f(ch) {
18             del_bytes += ch_len;
19         } else if del_bytes > 0 {
20             unsafe {
21                 ptr::copy(self.vec.as_ptr().add(idx),
22                       self.vec.as_mut_ptr().add(idx - del_bytes),
23                       ch_len);
24             }
25         }
26         idx += ch_len; // point idx to the next char
27     }
28 +   unsafe { self.vec.set_len(len - del_bytes); }
29 }
30
31 // PoC: creates a non-utf-8 string in the unwinding path
32 "0è0".to_string().retain(|_| {
33      match the_number_of_invocation() {
34         1 => false,
35         2 => true,
36         _ => panic!(),
37     }
38 });
```

Figure 6. An example of a panic safety bug, fix, and PoC in the Rust standard library that RUDRA found (CVE-2020-36317 [9]). It was independently fixed, but the latest stable version was still vulnerable when RUDRA discovered it.

3.1 Panic Safety

Rust provides a feature called `panic` which is used to signal that the current program has reached an unrecoverable state. When a `panic` happens, Rust unwinds the active call stack, releases resources held by the current thread by invoking the destructors of the variables, and transfers the control flow to the `panic` handler. This reclamation logic ensures that no program resources are leaked when panic happens, which is important for long-living multi-threaded programs.

Although such reclamation logic is helpful for writing safe Rust code, its interaction with `unsafe` code is error-prone and often causes non-trivial memory safety bugs. It is common for encapsulated `unsafe` code to temporarily create a lifetime-bypassed object that bypasses Rust's ownership system (*e.g.*, extending object lifetime, creating uninitialized variables) and fix up the introduced inconsistency later. If a `panic` happens in between the bypass and its fix-up, the destructors of the variable will run without realizing that the variable is in an inconsistent state, resulting in memory safety issues similar to uninitialized uses or double frees in `C/C++`. Such a bug is called a panic safety bug.

A panic safety bug is difficult to reason about because unwinding paths are automatically inserted by the compiler and invisible to programmers. The developer needs to manually reason about the consistency of stack variables for every invisible unwinding path to prevent a panic safety bug, which is an unusual task for Rust programmers. Ironically, a feature designed for easier resource management in safe Rust requires intensive manual reasoning when used in `unsafe` context. Panic safety bugs are similar in concept to exception safety bugs in other programming languages like `C++`. It is notoriously difficult to write generic exception-safe code [15, 24], and exception-based error handling is not allowed in complex system software like web browsers for this particular reason [37]. Due to their subtlety, panic safety violations have caused several memory safety bugs in popular Rust packages [5–7] and the Rust standard library [1, 9, 12, 14].

Definition 3.1. A function F has a panic safety bug if it drops a value v of type T such that $v \notin$ safe-value(T) during unwinding and causes a memory safety violation.

Bug example. Figure 6 shows a panic safety bug in `String::retain()` from the Rust standard library that RUDRA found (CVE-2020-36317 [9]). It filters characters in a string with a caller-provided closure but can leave the string as non-UTF-8 encoded when `f()` panics (line 17). The standard library assumes that all strings are UTF-8 encoded, and using the nonconforming string can lead to memory safety violations. This was fixed by overwriting the length of the string to zero before running the loop (line 9) and restoring it later (line 28), so that the string is left empty if `f()` panics.

3.2 Higher-order Safety Invariant

A Rust function should execute safely for *all safe inputs*; from the data types of its arguments, generic type parameters as well as user-provided closures. In other words, a safe function is not allowed to assume anything more than the safety invariants provided by the Rust compiler. For example, the `sort` function in Rust must not trigger any undefined behavior even when a user-provided comparator does not respect total ordering, unlike the `sort` function in `C++` that can cause a segmentation fault with an incompatible comparator.

Often, the only safety invariant that the Rust type system provides for higher-order types is the correctness of their type signature [42]. However, common mistakes are made with incorrect assumptions of ① logical consistency (*e.g.*, respects total ordering), ② purity (*e.g.*, always returns the same value for the same input), ③ and semantic restrictions (*e.g.*, only writes to the argument because it may contain uninitialized bytes) on a caller-provided function. `unsafe` code must check these properties by itself or specify the correct bound (*e.g.*, with an `unsafe` trait) so that the obligations of these checks can be at the caller's side.

It is worth emphasizing that it is fairly difficult and error-prone to enforce a higher-order invariant under Rust's type system. One notable example is passing an uninitialized buffer to a caller-provided `Read` implementation. `Read` is commonly

```
1  // CVE-2020-36323: a higher-order invariant bug in join()
2  fn join_generic_copy<B, T, S>(slice: &[S], sep: &[T]) -> Vec<T>
3      where T: Copy, B: AsRef<[T]> + ?Sized, S: Borrow<B>
4  {
5      let mut iter = slice.iter();
6
7      // `slice` is converted for the first time
8      // during the buffer size calculation.
9  *   let len = ...;
10     let mut result = Vec::with_capacity(len);
11     ...
12     unsafe {
13         let pos = result.len();
14         let target = result.get_unchecked_mut(pos..len);
15
16         // `slice` is converted for the second time in macro
17         // while copying the rest of the components.
18 *       spezialize_for_lengths!(sep, target, iter;
19 *           0, 1, 2, 3, 4);
20
21         // Indicate that the vector is initialized
22         result.set_len(len);
23     }
24     result
25 }
26
27 // PoC: a benign join() can trigger a memory safety issue
28 impl Borrow<str> for InconsistentBorrow {
29     fn borrow(&self) -> &str {
30         if self.is_first_time() {
31             "123456"
32         } else {
33             "0"
34         }
35     }
36 }
37
38 let arr: [InconsistentBorrow; 3] = Default::default();
39 arr.join("-");
```

Figure 7. A missing check of the higher-order invariant introduces a time-of-check to time-of-use bug in the Rust standard library (`join()` for `[Borrow<str>]`). RUDRA found this previously unknown bug (CVE-2020-36323 [10]).

expected to read data from one source (*e.g.*, a file) and write into the provided buffer. However, it is perfectly valid to read the buffer under Rust's type system. This leads to undefined behavior if the buffer contains uninitialized memory. Unfortunately, many Rust programmers provide an uninitialized buffer to a caller-provided function as performance optimization without realizing the inherent unsoundness. Due to its prevalence and subtlety, the Rust standard library now explicitly calls out that invoking `read()` with an uninitialized buffer is unsound behavior [57].

Definition 3.2. A higher-order invariant bug is a memory safety bug in a generic function that is caused by incorrectly assuming a higher-order invariant that is not guaranteed by the type system. A generic function Λ with a higher-order invariant bug incorrectly assumes certain properties (*e.g.*, purity) on its higher-order type parameter (*e.g.*, closure) although pred(Λ) does not guarantee it.

Bug example. Figure 7 shows a time-of-check to time-of-use bug caused by a missing check of the higher-order invariant. This bug in the `join()` function for `[Borrow<str>]` [9] was discovered by RUDRA in the Rust standard library. This function creates a joined vector by alternating a slice component and a separator. `&S` (the type contained in the slice) is converted to `&[T]` (the separator's type) twice in this function. The first conversion occurs during the length calculation (line 9) and the second conversion happens during the buffer copy *inside a macro* (line 18-19). A string with uninitialized bytes is returned if the slice code returns different results for the two conversions. This example shows the benefit of a tool-assisted approach. A trait method call that is inside the macro is not immediately visible, so the bug was missed during the code review process. The bug was fixed by setting `result`'s length to the number of written bytes instead of speculative `len`.

Type	Description	+Send only if	+Sync only if
`Vec<T>`	Owning container	`T: Send`	`T: Sync`
`&mut T`	Exclusive reference	`T: Send`	`T: Sync`
`&T`	Aliased reference	`T: Sync`	`T: Sync`
`RefCell<T>`	Internal mutability	`T: Send`	-
`Mutex<T>`	RAII mutex	`T: Send`	`T: Send`
`MutexGuard<T>`	Mutex guard	-	`T: Sync`
`RwLock<T>`	RAII rwlock	`T: Send`	`T: Send+Sync`
`Rc<T>`	Reference counter	-	-
`Arc<T>`	Atomic reference counter	`T: Send+Sync`	`T: Send+Sync`

Table 1. The propagation rule of various types from the Rust standard library for `Send`/`Sync` traits. The rule becomes complicated when non-trivial sharing is involved.

3.3 Propagating Send/Sync in Generic Types

Rust's thread safety is governed by two `unsafe` traits, `Send` and `Sync`. `Send` is used to indicate a type that can be sent to other threads and `Sync` is used to indicate a type that can be referenced concurrently by multiple threads. `Send` and `Sync` are derived traits, which means that the compiler will automatically implement `Send` and `Sync` on a type if all of its fields are `Send` and `Sync`, respectively. However, developers need to manually implement `Send` and `Sync` on synchronization primitives like locks and types that contain fields with unknown thread safety (*e.g.*, a raw pointer).

The `Send` and `Sync` rules become complex as the implementation bound becomes *conditional* when generic types are involved (see Table 1). One simple example is a container type, `Vec<T>`, that is `Send` only if the inner type `T` is `Send` and is `Sync` only if the inner type `T` is `Sync`. The logic quickly becomes non-intuitive and error-prone for types that provide non-trivial sharing like `Mutex` and `RwLock` (see Table 1). Inspired by type variance in subtyping relations, we call this subtle relation between the `Send`/`Sync` of a generic type and the `Send`/`Sync` of the inner types the `Send`/`Sync` variance.

Manual `Send`/`Sync` implementations are not only difficult to correctly implement, but also make code maintenance fragile. `Send` and `Sync` are type level properties that guarantee the thread safety of *all possible APIs on that type*. A developer who is not aware of the manual `Send`/`Sync` implementation may add a new API that is not thread-safe and silently introduce a soundness bug without any unsafe code. The complexity around this rule leads to safety violations even in the Rust standard library [3].

```
1  // CVE-2020-35905: incorrect uses of Send/Sync on Rust's futures
2  pub struct MappedMutexGuard<'a, T: ?Sized, U: ?Sized> {
3      mutex: &'a Mutex<T>,
4      value: *mut U,
5  +   _marker: PhantomData<&'a mut U>,
6  }
7
8  impl<'a, T: ?Sized> MutexGuard<'a, T> {
9      pub fn map<U: ?Sized, F>(this: Self, f: F)
10         -> MappedMutexGuard<'a, T, U>
11     where F: FnOnce(&mut T) -> &mut U {
12         let mutex = this.mutex;
13         let value = f(unsafe { &mut *this.mutex.value.get() });
14         mem::forget(this);
15 -       MappedMutexGuard { mutex, value }
16 +       MappedMutexGuard { mutex, value, _marker: PhantomData }
17     }
18 }
19
20 - unsafe impl<T: ?Sized + Send, U: ?Sized> Send
21 + unsafe impl<T: ?Sized + Send, U: ?Sized + Send> Send
22     for MappedMutexGuard<'_, T, U> {}
23 - unsafe impl<T: ?Sized + Sync, U: ?Sized> Sync
24 + unsafe impl<T: ?Sized + Sync, U: ?Sized + Sync> Sync
25     for MappedMutexGuard<'_, T, U> {}
26
27 // PoC: this safe Rust code allows race on reference counter
28 * MutexGuard::map(guard, |_| Box::leak(Box::new(Rc::new(true))));
```

Figure 8. An incorrect Sync/Send trait bound for a generic type parameter in Rust's official `futures` library, which breaks the thread safety guarantee. RUDRA found this previously unknown bug [8].

Definition 3.3. A generic type that takes a type parameter `T` has a Send/Sync variance (SV) bug if it specifies an incorrect bound on the inner type `T` when implementing Send/Sync.

Bug example. Figure 8 shows an incorrect use of Send/Sync marker traits in the official `futures` library that RUDRA found (CVE-2020-35905 [8]). This bug results in a data race in safe Rust code. A `MappedMutexGuard` that dereferences to type `U` is created from a `MutexGuard` that dereferences to type `T` by applying a closure that converts `&mut T` to `&mut U` (line 13). `MappedMutexGuard`'s Send and Sync have a trait bound only to the type parameter `T` but not for the type parameter `U` (line 20 and 23). This definition turns out to be unsafe because it allows sharing a reference of a `MappedMutexGuard` even when the closure's return type `U` is not thread-safe. The bug was fixed by adding a proper bound to the type parameter `U` (line 21 and 24). Note that the `map()` and Send/Sync implementations are not adjacent to each other in the source code, making it difficult to notice this bug in code reviews.

4 Design

RUDRA implements two static analysis algorithms that can detect three bug patterns in `unsafe` code (§3), as described in Figure 9. It has three important design goals:

- **Generic type awareness.** RUDRA should be able to reason about generic types without knowing the concrete forms of their type parameters. This means that low-level analysis (*e.g.*, using LLVM IR) is not an option. Low-level representations only contain a specific instantiation of generic code, and Rust's high-level abstractions such as trait variance do not exist at these levels. Instead, RUDRA implements algorithms by combining two internal IRs of the Rust compiler, namely, HIR and MIR.
- **Scalability.** As our primary goal is to check all the packages in the Rust package registry, it is critical to strike a balance between the precision of analysis and the execution time—expensive whole-program analyses and dynamic analyses like fuzzing are not feasible options for RUDRA. In addition, RUDRA aims to be a push-button solution that requires no manual annotation and effort from the original package developers.
- **Adjustable precision.** With the limited computation available to each package, it is impossible to formulate analyses with no false positives. RUDRA provides an option to adjust the false positive rates based on the goal and the available time budget; RUDRA can be used for both scanning the package registry (fewer false positives) or as part of the development process (tolerant to more false positives).

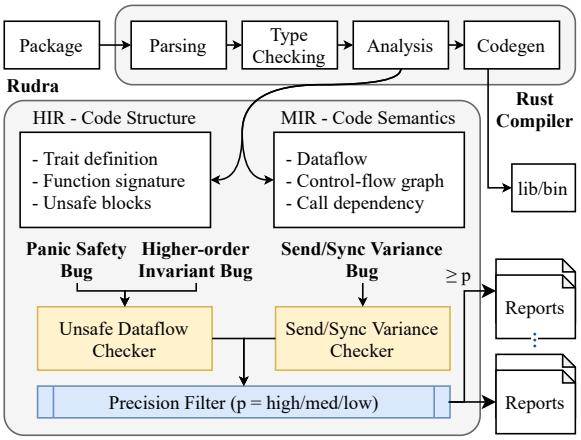

Figure 9. Overview of RUDRA's design.

Overview. RUDRA accepts a Rust package as input and produces a comprehensive report of two analysis algorithms, namely, the unsafe dataflow checker and the Send/Sync variance checker, that can identify three `unsafe` bug patterns (§3). We implemented RUDRA as a custom Rust compiler driver. RUDRA hijacks the Rust compiler after type checking and runs our analysis using the compiler's internal data. We also have an adapter to Rust's package manager and an executor to download and run RUDRA on published Rust packages.

4.1 Hybrid Analysis with HIR and MIR

RUDRA uses two Rust compiler IRs when implementing analysis algorithms: the HIR and the MIR. The first IR is the High-Level IR (HIR) generated from the AST. HIR contains the IDs of each definition inside the target program (*e.g.*, functions, trait implementations) as well as their associated expressions. HIR keeps the original code structure in its expressions. The second IR is the Mid-Level IR (MIR) generated by lowering

Algorithm 1: Checking unsafe dataflow

```
// contains impl. items, trait items, free functions from HIR
body_set := Set<BodyId>;
foreach body_id in body_set do
    body := compiler.getHIR(body_id);
    if not is_unsafe(body) then
        continue
    body ← compiler.getMIR(body_id);
    graph := Graph(body.basic_blocks);
    foreach block in body.basic_blocks do
        if block.terminator.isStaticCall() then
            call := block.terminator.asStaticCall();
            if is_life_bypassing_func(call) then
                graph.mark_bypass_type(block.id, call);
            else if compiler.resolve(call, ∅) fails[1] then
                graph.add_sink(block.id);
    graph.propagate_taint();
    taint := {};
    foreach sink in graph.sinks() do
        taint ← taint ∪ graph.get_taint(sink);
    if taint ≠ ∅ then
        report_potential_violation(body_id);
```

the HIR. MIR focuses on the semantic information. It has a much simpler and more analyzer-friendly structure than HIR, but it also lacks some important non-semantic information that RUDRA needs, such as the locations of `unsafe` blocks, which are removed after type checking. The HIR and the MIR provide a generic representation, meaning that a generic function or a generic type remains as a single definition. RUDRA cannot use the IRs from later stages (*e.g.*, LLVM IR) because they only provide a single instantiation of a generic function. Using them will make RUDRA miss bugs in generic functions such as the bug in Figure 4.

RUDRA implements a hybrid analysis that uses both HIR and MIR. It uses HIR to quickly collect interesting code regions using structural information available in HIR. Specifically, it collects function declarations and trait implementations inside the package with their declared safety. It also records if a function contains any `unsafe` blocks if the function is defined as a safe function. Then, RUDRA uses MIR to reason about code semantics. RUDRA implements a coarse-grained dataflow analysis on the control-flow graph provided as MIR expressions. It is worth emphasizing that mixed usage of multiple IR levels is unconventional but is required for RUDRA's goal of scaling the analysis to the entire ecosystem. Since HIR and MIR are Rust-specific IRs, traditional analysis algorithms and tools such as dataflow analysis, declaration collectors, or even an error-reporting system, which are readily available in the lower-level LLVM infrastructure, were reimplemented for RUDRA.

[1] RUDRA uses the Rust compiler's instance resolution API with an empty type context to determine if a generic function is resolvable or not.

4.2 Algorithm: Unsafe Dataflow Checker (UD)

The unsafe dataflow checker (Algorithm 1) examines the dataflows in functions that handle lifetime-bypassed values. It uses coarse-grained taint tracking to identify panic safety bugs (§3.1) and higher-order invariant bugs (§3.2). Simply put, the analysis algorithm checks if there exists a dataflow that starts from a lifetime bypass to a suspicious function call—a function that might `panic()` or a function that is provided as a higher-order parameter by the caller.

The algorithm models six classes of lifetime bypasses:

- *uninitialized*: creating uninitialized values
- *duplicate*: duplicating the lifetime of objects (*e.g.*, with `mem::read()`)
- *write*: overwriting the memory of a value
- *copy*: `memcpy()`-like buffer copy
- *transmute*: reinterpreting a type and its lifetime
- *ptr-to-ref*: converting a pointer to a reference

The key challenge of the unsafe dataflow algorithm is to find program locations that might `panic()` or where higher-order safety invariants are implicitly assumed. At first glance, it may seem that detecting panic safety bugs needs precise analysis to determine if a `panic()` can happen at a given program point, and detecting the higher-order invariant bugs requires accurate reasoning of the semantic correctness of a given trait implementation. However, soundly determining them without pre/post conditions is undecidable, and RUDRA makes a deliberate approximation with a concept of an unresolvable generic function to meet its performance and scalability goal.

RUDRA, at the MIR layer, uses an unresolvable generic function call as an approximation of a potential panic site or a location where higher-order invariants are implicitly assumed. An unresolvable function is a function whose definitions cannot be found without precise type parameters. For example, `<reader as Read>::read()` is one such function and, unlike `Vec<T>::push()` where one `push()` implementation exists for all possible inner types `T`, there can be no implementation found without knowing the exact type of `reader`. As unresolvable generic functions are implemented and provided by the caller, it is invalid for the callee to assume that the functions do not `panic()` or always satisfy the implicitly assumed semantic requirements. We observed that dataflows that contain unresolvable generic functions are not only analyzer-friendly but also where Rust programmers tend to make mistakes, perhaps because speculating about the unknown function's behavior is more difficult than reasoning about a concrete implementation.

Adjustable precision. RUDRA's approximation of each lifetime bypass has different precision. RUDRA only detects uninitialized values (*e.g.*, `Vec::set_len()` to extend a `Vec`) in the high precision setting because a single function call leads to a lifetime bypass in such cases. In the medium precision setting, RUDRA additionally detects the lifetime bypass of

Algorithm 2: Checking Send/Sync variance

```
foreach trait_impl in local_trait_impls() do
    if impl_trait = Send then
        foreach param in trait_impl.generic_params do
            if param ∉ self.phantom_params
            and ¬(param :> Send) then
                report_potential_violation(trait_impl);
    else if impl_trait = Sync then
        self := trait_impl.self_ty();
        reqs := hashmap() // a set of necessary bounds
        foreach param in trait_impl.generic_params do
            if param ∈ self.phantom_params then
                continue;
            foreach api in self.safe_self_ref_apis() do
                if api.moves(param) then
                    reqs[param].add('Send);
                if api.exposes_ref(param) then
                    reqs[param].add('Sync);
        foreach param in trait_impl.generic_params do
            if param :> Send then
                reqs[param].remove('Send)
            if param :> Sync then
                reqs[param].remove('Sync)
        if {'Send, 'Sync} ∩(∪reqs[..]) ≠ ∅ then
            report_potential_violation(trait_impl);
```

values using `read()`, `write()`, and `copy()`. These bypasses are more difficult to reason about because they are often used with pointer arithmetic. Finally, RUDRA detects lifetime forging with `transmute()` or raw pointer casting in the low precision setting.

4.3 Algorithm: Send/Sync Variance Checker (SV)

The Send/Sync variance checker (Algorithm 2) estimates the necessary minimum set of Send/Sync bounds for each Algebraic Data Type (ADT) based on the associated API signatures. If the ADT does not contain the necessary bounds, it reports that Send/Sync might be incorrectly implemented. One might be able to accurately model such usages by performing inter-procedural and flow-sensitive analysis to verify the thread safety at an arbitrary program point. RUDRA intentionally avoids these complex and performance-intensive approaches to meet its scalability goal.

The key idea of the Send/Sync variance checker is to determine if an ADT requires Send, Sync, or both, based on a set of effective heuristics using the type definition and the associated API signatures:

Given an `ADT` with a generic parameter `T`,

- **+Send.** If there exists an API that moves `T` (*i.e.*, either taking as input the owned `T` or returning the owned `T`) but none of its APIs exposes `&T` (*i.e.*, returning `&T`), then `T:Send` is the minimum necessary condition. For `ADT:Sync`, it is important to check the exposure of `&T` because it allows threads to concurrently access `T`. For `ADT:Send`, `T:Send` is the minimum necessary condition regardless of its API. Moving an ADT (holding ownership of `T`) to another thread moves `T` to another thread.
- **+Sync.** If there exists an API that exposes `&T` but none of its APIs move the owned `T`, then `T:Sync` is the minimum necessary condition for `ADT:Sync`.
- **+Send/+Sync.** If there exists an API that exposes `&T` and that moves the owned `T`, then `T:Send+Sync` is the minimum necessary condition for `ADT:Sync`.
- **None.** If there is no API that exposes `&T` or moves the owned `T`, it is not possible to verify the thread safety of the Send/Sync markers from the API signatures and it places no minimum necessary condition for `ADT:Sync`.

Note that these rules are not applied to generic parameters `T` placed within `PhantomData<T>`—this is a zero-sized marker type that allows the binding of `T` to an ADT but does not actually own `T`. This helps us avoid several false positives where a generic parameter is used only as a type-level identifier.

Adjustable precision. On top of the baseline algorithm (described above) for inferring Send/Sync bound requirements, RUDRA uses additional heuristics to find more Send/Sync variance violations. In the high precision setting, RUDRA focuses on Send bounds which are less affected by custom synchronization than Sync bounds. It implements **+Send** analysis from the baseline algorithm to identify missing `ADT: Sync`, `T: Send` bound and analyzes the type structure to identify missing `ADT: Send, T: Send` bound. In the medium precision setting, RUDRA fully instruments the baseline algorithm while also reporting Sync impls with no Sync bounds on all of its generic parameters. In the low precision setting, RUDRA removes the `PhantomData`-filtering policy and reports Sync impls with no Sync bounds on any of its generic parameters.

5 Implementation

RUDRA is built on `rustc nightly-2020-08-26` in 4.3k lines of Rust code. The main analyzer, `rudra`, is implemented as a custom Rust compiler driver. It works as an unmodified Rust compiler when compiling dependencies and injects the analysis algorithms when compiling the target package. RUDRA adjusts the precision filter based on an environment variable. It provides tight integration with the official Rust package manager, `cargo`, so that an entire package can be checked with one command, `cargo rudra`. It also provides `rudra-runner`, which downloads and analyzes all packages from the official package registry, `crates.io`.

6 Evaluation

Our evaluation attempts to answer the following questions:

- How effectively can our approach detect new memory safety bugs at the ecosystem scale? (§6.1)
- How does it compare to other approaches? (§6.2)
- What lessons can be learned from running RUDRA on Rust-based OSes? (§6.3)

Package	Location	Tests[1]	LoC	#unsafe	Alg	Description	L[2]	Bug ID[3]
std	str.rs mod.rs	U / -	61k	2k	UD	The `join` method can return uninitialized memory when string length changes. `read_to_string` and `read_to_end` methods overflow the heap and read past the provided buffer.	3y 2y	C20-36323 C21-28875
rustc	worker_local.rs	U / -	348k	2k	SV	`WorkerLocal` used in parallel compilation can cause data races.	3y	rust#81425
smallvec	lib.rs	U / F	2k	55	UD	Buffer overflow in `insert_many` allows writing elements past a vector's size.	3y	R21-0003 C21-25900
futures	mutex.rs	U / -	5k	84	SV	`MappedMutexGuard` can cause data races, violating Rust memory safety guarantees in multi-threaded applications.	1y	R20-0059 C20-35905
lock_api	rwlock.rs	U / -	2k	146	SV	Multiple RAII objects used to represent acquired locks allow for data races. Types that should be accessible by only one thread at a time are allowed to be used concurrently, leading to violations of Rust's memory safety guarantees.	3y	R20-0070 C20-35910 C20-35911 C20-35912
im	focus.rs	U / F	13k	23	SV	`TreeFocus`, an iterator over tree structure, can cause data races when sent across threads.	2y	R20-0096 C20-36204
rocket_http	formatter.rs	U / -	4k	16	UD	A use-after-free is possible for the string buffer in the `Formatter` struct on panic.	3y	R21-0044 C21-29935
slice-deque	lib.rs	U / F	6k	89	UD	`drain_filter` can double-free elements with certain predicate functions.	3y	R21-0047 C21-29938
generator	gen_impl.rs	U / -	2k	72	SV	`Generators` can be sent across threads leading to data races.	4y	R20-0151
glium	mod.rs	U / -	39k	4k	UD	`Content` passes uninitialized memory to safe functions.	6y	glium#1907
ash	util.rs	U / -	89k	2k	UD	`read_spv` returns uninitialized bytes when reading incompletely.	2y	R21-0090
atom	lib.rs	U / -	600	25	SV	`Atom<T>` can be instantiated with any `T`, allowing data races for non-thread safe types when used concurrently.	2y	R20-0044 C20-35897
metrics-util	bucket.rs	U / -	3k	13	SV	`AtomicBucket<T>` can cause data races.	2y	R21-0113
libp2p-deflate	lib.rs	U / -	200	1	UD	`DeflateOutput` passes uninitialized memory to safe Rust.	2y	R20-0123
model	lib.rs	U / -	200	3	SV	`Shared` bypasses concurrency safety without being marked `unsafe`.	2y	R20-0140
claxon	metadata.rs	U / F	3k	5	UD	`metadata::read` methods return uninitialized memory.	6y	claxon#26
stackvector	lib.rs	U / -	1k	32	UD	`StackVector` trusts an iterator's length bounds which can lead to writing out of bounds.	2y	R21-0048 C21-29939
gfx-auxil	mod.rs	U / -	100	1	UD	`read_spirv` passes uninitialized memory to safe Rust.	2y	R21-0091
futures-intrusive	mutex.rs	U / -	9k	120	SV	`GenericMutexGuard`, an RAII object representing an acquired `Mutex` lock, allows data races.	2y	R20-0072 C20-35915
calamine	cfb.rs	U / -	6k	3	UD	`Sectors::get` trusts the size in a file header, exposing uninitialized when a malicious file is used.	4y	R21-0015 C21-26951
atomic-option	lib.rs	- / -	91	5	SV	`AtomicOption<T>` can be used with any type, leading to data races with non-thread safe types.	6y	R20-0113 C20-36219
glsl-layout	array.rs	- / -	600	1	UD	`map_array` can double-drop elements in the list if the mapping function panics.	3y	R21-0005 C21-25902
internment	lib.rs	U / -	900	13	SV	Objects wrapped in `Intern<T>` could always be sent across threads, potentially causing data races.	3y	R21-0036 C21-28037
beef	generic.rs	U / -	900	23	SV	`Cow` allows usage of non-thread safe types concurrently.	1y	R20-0122
truetype	tape.rs	U / -	2k	2	UD	`take_bytes` passes an uninitialized memory buffer to a safe Rust function.	5y	R21-0029 C21-28030
rusb	device.rs	U / -	5k	78	SV	The `Device` trait lacks `Send` and `Sync` bounds; USB devices could cause races across threads.	5y	R20-0098 C20-36206
fil-ocl	event.rs	U / -	12k	174	UD	`EventList` can double-drop elements if the `Into` implementation of the element panics.	3y	R21-0011 C21-25908
toolshed	cell.rs	U / -	2k	23	SV	`CopyCell` allows data races with non-`Send` but `Copy`able types.	3y	R20-0136
lever	atomics.rs	U / -	3k	67	SV	`AtomicBox` allows data races with non-thread safe types.	1y	R20-0137
bite	read.rs	- / -	1k	44	UD	`read_framed_max` passes uninitialized memory to safe Rust.	4y	bite#1

[1]Contains unit tests with over 50% coverage (U) or fuzzing (F) suites. [2]Latent period in years. [3]C21/R21 stands for CVE-2021/RUSTSEC-2021.

Table 2. Details of the new bugs found in the 30 most popular packages based on `crates.io` download numbers. RUDRA found memory-safety bugs from heavily tested packages—containing unit tests with extensive code coverage and fuzzers. The found bugs are non-trivial—they had existed for over three years on average.

Analyzer	Time†	Packages	Bugs	#RustSec	#CVE
UD	16.510 ms	83	122	54	46
SV	0.224 ms	63	142	58	30
Auditing	1 hour	19	46	17	25

† Average time taken to analyze one package.
UD/SV requires additional 33.7 sec for compilation.

Table 3. Summary of new memory-safety bugs found by RUDRA. The last row represents additional bugs found by code auditing during the pilot study and the bug reporting.

Precision		#Reports	#Bugs found		
			Visible	Internal	Total
UD	High	137	65 (47.4%)	8 (5.8%)	73 (53.3%)
	Med	434	119 (27.4%)	17 (3.9%)	136 (31.3%)
	Low	1,214	163 (13.4%)	31 (2.6%)	194 (16.0%)
SV	High	367	118 (32.2%)	60 (16.3%)	178 (48.5%)
	Med	793	181 (22.8%)	98 (12.4%)	279 (35.2%)
	Low	1,176	197 (16.8%)	111 (9.4%)	308 (26.2%)

Table 4. The total number of reports with varying precision and true bugs after scanning 43k packages (see §6.1).

Experimental setup. We ran all the following experiments on a machine with a 32-core AMD EPYC 7452 and 252 GB memory. The analysis session for each package was limited to one core for a fairer comparison and only the `rudra-runner` layer took advantage of the concurrency.

6.1 New Bugs Found by RUDRA

Applying to all packages. We downloaded and analyzed *all* 43k packages uploaded to `crates.io` (as of 2020-07-04). RUDRA took about 6.5 hours to scan all the packages on our machine: 15.7% (7k) did not compile with the `rustc` version RUDRA was based on, 4.6% (2k) did not produce any Rust code (*e.g.*, macro-only packages), and 1.8% (0.7k) did not have proper metadata (*e.g.*, depending on yanked packages), leaving us with 77.9% (33k) packages as analysis targets. It took 33.7 sec on average to analyze each package end-to-end. Among the total amount of time, RUDRA used 18.2 ms; the remaining time was spent in the Rust compiler. As a result, we generated 2,390 reports and inspected them all at a rough rate of 150 reports per man-hour. Most false positives were filtered out at a glance (in a few seconds) due to the precision level attached to them.

New bugs. We reported 264 previously unknown memory-safety bugs in 145 packages, resulting in 112 RustSec advisories and 76 CVEs (see Table 3 and Table 2). This is an unprecedented number of memory-safety bugs, constituting 51.6% of all memory-safety bugs in the Rust ecosystem since 2016 (see Figure 1). Also, the bugs RUDRA discovered are non-trivial: two higher-order invariant bugs in the Rust standard library, `std`, one SV bug in the Rust compiler, `rustc`, one SV bug in the official `futures` library, and several SV bugs in `lock_api`, a very popular lock abstraction library. These are bugs in code written and extensively reviewed by Rust experts. It is worth noting that the average latent time of the discovered bugs is over three years despite community efforts to manually audit `unsafe` code in Rust [38]. RUDRA was also able to re-discover two bugs in the Rust standard library that had been fixed, but their vulnerable versions were retained in some libraries. During the pilot study to identify common bug patterns and while auditing code from RUDRA reports, we manually found 46 additional bugs, resulting in 17 RustSec advisories and 25 CVEs, three of which are in the Rust standard library [12–14]. The list of all bugs are publicly available at https://github.com/sslab-gatech/Rudra-PoC.

Precision. In the high precision setting, the UD algorithm generated 137 reports (1 report per 309 packages) and found 73 bugs (53.3% precision). The SV algorithm generated 367 reports (1 report per 116 packages) and found 178 bugs (48.5% precision). When all bug patterns are turned on in the low precision setting, the UD algorithm generated 1,214 reports (1 report per 35 packages) and found 194 bugs (16.0% precision), and the SV algorithm generated 1,176 reports (1 report per 36 packages) and found 308 bugs (26.2% precision). RUDRA provides great improvement over a simple search for the `unsafe` keyword. A total of 330k functions encapsulate `unsafe` code in the Rust ecosystem, and the UD algorithm reduces this number to 137 in the high precision setting and 1,214 in the low precision setting. Table 4 shows the number of the bugs and the precision of each analysis in different precision settings. We separated the bugs further into two categories: visible bugs that affect users of the package and internal bugs that can only be triggered inside the same package. We provide the examples of false positives and negatives in the discussion section (§7.1).

Reporting. In addition to reporting bugs to the original maintainers of the package, we also reported bugs to the RustSec advisory database [39] and the CVE database. In total, 112 RustSec advisories and 76 CVE IDs have been assigned to the bugs found by RUDRA. As of September 2021, these bugs represent 39.0% of all bugs and 51.6% of memory safety bugs reported to RustSec since RustSec started tracking security bugs in 2016. When counting the total number of RustSec bugs, we excluded notices and unmaintained advisories as well as transitive advisories for C dependencies (*e.g.*, OpenSSL) because they do not represent a bug in the target Rust package. Figure 1 shows the number of bugs reported to RustSec advisory database each year, with RUDRA's contribution highlighted with hatches. In addition, 16 bugs reported in 2020 and 38 bugs reported in 2021 are currently pending to receive RustSec advisories because no fix is available for them. These bugs are either blocked by the maintainer's fix or `ReadBuf` RFC implementation [32] in the standard library.

New lints. From the bugs found by RUDRA, we were able to identify the most frequently misused Rust APIs. We ported

Package	Test Coverage	# Tests	Timeout	UB-A[1]	UB-SB[2]	Leak	Avg Memory[3]	Time Taken	Bug ID (Type)	Result
atom	76.2% (193 LoC)	16	0	0 (0)	3 (1)	5 (1)	372 MB	7 m	R20-0044 (SV)	0/2
beef	85.9% (440 LoC)	30	0	0 (0)	2 (1)	0 (0)	380 MB	5 m	R20-0122 (SV)	0/1
claxon	50.5% (1,941 Loc)	33	0	0 (0)	0 (0)	0 (0)	388 MB	7 m	GitHub #26 (UD)	0/2
futures	N/A	177	1	0 (0)	35 (4)	0 (0)	455 MB	28 h	R20-0059 (SV)	0/1
im	67.5% (7,135 LoC)	104	15	0 (0)	39 (7)	0 (0)	1345 MB	20 h	R20-0096 (SV)	0/2
toolshed	88.2% (1,186 LoC)	39	0	24 (1)	7 (2)	0 (0)	392 MB	14 m	R20-0136 (SV)	0/1

[1]Reference alignment issue. [2]Alias violation under Stacked Borrow model. [3]Average of the peak memory measured by `cgmemtime`.

Table 5. Summary of running unit tests with Miri. Test code LoC and test coverage were measured with grcov. Tests were run with a one-hour time limit for a single test case. The numbers in parenthesis are the deduplicated bug numbers.

Package	#H	Bug ID	Fuzzer	#execs	Result (FP)
claxon	4	GitHub #26	cargo-fuzz	12B	0/2 (0)
dnssector	5	GitHub #14	cargo-fuzz	29B	0/1 (4.4M)
im	3	R20-0096	cargo-fuzz	16B	0/2 (0)
smallvec	1	R21-0003	honggfuzz	0.9B	0/1 (0.6M)
slice-deque	1	R21-0047	afl	100k	0/1 (0)
tectonic	1	GitHub #752	cargo-fuzz	363k	0/1 (22k)

Table 6. Results of running provided fuzzing harnesses (marked #H) in each package with three sanitizers (A/M/TSAN) for 24 hours. None of the eight bugs found by RUDRA were discovered by the fuzzers, but a large number of false positives (marked FP) are reported by the fuzzers.

RUDRA's algorithms as lints to detect such misuses and integrated them into the official Rust linter, Clippy [52]. At the time of writing, two lints have been implemented: `uninit_vec` and `non_send_field_in_send_ty`. The `uninit_vec` lint detects a creation of an uninitialized `Vec`, which is commonly used with the `Read` trait and causes a higher-order invariant bug (§3.2). The `non_send_field_in_send_ty` lint implements a subset of **`+Send`** analysis of the SV algorithm that focuses on type definitions.

6.2 Comparison with Other Approaches

We compare RUDRA to two popular dynamic analysis approaches, fuzzing and Miri, as well as static analyzers that aim to detect the same classes of bugs in Rust.

Comparison with fuzzing. Fuzzing [18, 35], a dynamic approach that randomly mutates inputs for testing, is not an effective approach to find the classes of bugs RUDRA found. We selected six packages (see Table 6) that provide fuzzing harnesses and checked whether they could find the bugs RUDRA found. The fuzzing harnesses for `dnssector`, `im`, `slice-deque`, and `tectonic` did not test the buggy APIs. `claxon` and `smallvec`'s fuzzers stress the buggy APIs, but they failed to formulate a bug triggering input.

None of the fuzzers discovered bugs found by RUDRA. They suffer from the fundamental problem of dynamic testing; they can only test a single instantiation of generic code. `claxon` has a bug that provides uninitialized bytes to the caller provided `Read` implementation, but its fuzzer only tests the API with a `Read` implementation that does not read the uninitialized bytes. `smallvec` has a bug that requires an iterator with an unknown size, but its fuzzer only tests the API with a fixed-size iterator. Interestingly, fuzzers for three of the packages reported false positives. These were caused by compatibility issues with the sanitizers or due to incorrect handling of panics on malformed input. This indicates that some of these fuzzers are not actively used or maintained to find bugs continuously.

Comparison with Miri. Miri [43, 53] is an interpreter for Rust MIR that can detect certain classes of undefined behaviors during interpretation, such as alignment issues, alias violation, or memory leaks. Miri is similar to using sanitizers in fuzzing. It runs the executable with user input or unit tests to identify bugs. We ran Miri on six packages where RUDRA found memory safety bugs with all available tests in each package (see Table 5). Miri used 3.24× more memory on average compared to RUDRA and spent about 5 minutes to 20 hours of CPU time running all the tests in a single package. In comparison, RUDRA only spent 18.2 ms on average to scan a package. Miri did not find any of the nine bugs found by RUDRA because all unit tests explore the monomorphized forms of generic functions, similar to fuzzing. However, Miri found a few potential alignment issues and alias violations in some packages. This result indicates that Miri is complementary to RUDRA in terms of bug classes, but is not applicable to the ecosystem scale.

Comparison with other static analysis. Qin *et al*. [49] proposed two static analysis algorithms, namely, UAFDetector and DoubleLockDetector, to detect certain classes of memory/thread safety bugs in Rust programs. UAFDetector identified none of the 27 UAF bugs that the UD algorithm found in 16 different packages: 1) its flow-sensitive analysis visits the same basic block only once, missing panic safety bugs in partially iterated loops, and 2) it models almost all function calls as no-op or identity functions and fails to recover the alias information required to run the analysis. DoubleLockDetector is not a generic analyzer. It only targets the misuse of a specific third-party lock implementation, `parking_lot`'s `RwLock`. In addition, since it works at the LLVM IR layer, it fundamentally cannot find all the SV bugs RUDRA found.

OS	LoC	#unsafe	#Reports in each component				#Bugs
			Mutex	Syscall	Allocator	Total	
Redox	30k	709	0	4	0	4	0
rv6	7k	678	4	0	0	4	0
Theseus	40k	243	1	0	6	7	2
TockOS	10k	145	0	0	1	1	0

Table 7. The number of reports RUDRA emits for each Rust-based operating system kernel.

6.3 Analyzing Rust-based OSes

To understand the impact of unsafe Rust in Rust-based operating systems, we applied RUDRA to four Rust-based OSes: Redox [29], rv6 [30], Theseus [22], and TockOS [48]. Table 7 summarizes the analysis result. Although each kernel uses `unsafe` hundreds of times, the number of reports generated by RUDRA was small—one report per 5.4 kLoC—showing that it requires minimal effort to review the analysis results. This is because RUDRA focuses on bugs caused by misuse of generic types, but generic types are not very common in self-contained kernel code. In total, RUDRA found two internal soundness issues in Theseus OS: two safe public `deallocate()` APIs that unconditionally transmute the passed address to an allocation chunk. We discussed the issue with the developers and submitted a patch, which has been accepted upstream. Besides the issues that are directly found in Rust-based OSes, we observed an interesting implication of an isolation scheme built on Rust's soundness, which is gaining popularity in the system programming community [47, 48]. We believe Rust's safety rule does not provide enough guarantees for such designs yet; the detail of our observation is shared in the discussion section (§7.2).

7 Discussion

7.1 Understanding False Positives and Negatives

In this section, we discuss representative false positives and negatives of RUDRA to illustrate the scope of the bugs found by RUDRA. The most common cause of false positives in the UD algorithm is due to the imprecise modelling (*i.e.*, overapproximation) of lifetime bypasses. See Figure 10, a RUDRA report for the `few` package. The value `val` is bit-copied in line 6 and passed to a user-provided function `replace()` in line 7. If `replace()` panics, it would drop the duplicated value `old` and then the original value `val` while unwinding, leading to a double-free bug. However, a custom struct `ExitGuard` prevents this from happening. `ExitGuard` is dropped before `val` is dropped, and it stops the unwinding by aborting the program, preventing the second drop of the value. Such false positives can be eliminated with an interprocedural analysis that can look into `ExitGuard`'s implementation.

Similarly, the false positives of the SV algorithm is caused by the limitation of the type definition and the API signatures-based reasoning. The SV algorithm does not model complex

```
1  fn replace_with<T, F>(val: &mut T, replace: F)
2      where F: FnOnce(T) -> T {
3      let guard = ExitGuard;
4
5      unsafe {
6          let old = std::ptr::read(val);
7          let new = replace(old);
8          std::ptr::write(val, new);
9      }
10
11     std::mem::forget(guard);
12 }
```

Figure 10. A false positive example for the `few` package. The fixup routine in line 8 is not called if `replace()` in line 7 panics. However, `ExitGuard` prevents the panic safety issue.

```
1  unsafe impl<T> Send for Fragile<T> {}
2  unsafe impl<T> Sync for Fragile<T> {}
3  unsafe impl<T> Send for Sticky<T> {}
4  unsafe impl<T> Sync for Sticky<T> {}
5
6  // Sticky<T> has a similar guard
7  impl<T> Fragile<T> {
8      ...
9
10     pub fn get(&self) -> &T {
11         assert!(get_thread_id() == self.thread_id);
12         unsafe { &*self.value.as_ptr() }
13     }
14 }
```

Figure 11. A false positive example for the `fragile` package. The `Fragile` and `Sticky` structs do not specify any bound on type parameter `T`, but accesses to `T` are guarded with custom thread-aware execution.

program semantics such as thread-aware execution or manual synchronization. See Figure 11, a RUDRA report for the `fragile` package. The `Fragile` and `Sticky` struct implement `Send` and `Sync` without any bound on the type parameter `T`, which would have been definitive examples of a `Send`/`Sync` variance bug if they provide direct access to the inner value `T`. However, they have custom assertions that check the current thread ID before allowing the access. RUDRA's API signature-based logic generates false positive reports for them.

The false negatives originate from the similar limitations. We manually created the models for known `unsafe` functions in the standard library for the UD algorithm, but this set is not complete. The SV algorithm will miss `Send`/`Sync` bugs if the type's definition does not explicitly show the ownership, *e.g.*, when an owned value is stored as a universal pointer `*const ()`, a type similar to C++'s `void *`. In addition, both algorithms cannot detect any bugs caused by an interprocedural interaction. As these examples show, RUDRA's precision could be improved with a deeper semantic model and more comprehensive analyses. We believe there is a lot of room to improve in RUDRA's analysis framework. Developing a better analysis framework for Rust and integrating RUDRA's bug patterns with it is a promising future research direction.

7.2 Understanding Rust's Safety

Rust skeptics might claim that Rust does not provide better safety than C/C++ due to the existence of unsafe Rust.

However, our experience suggests that Rust's objective safety standard is a supreme improvement. It makes the communication cost of reporting and fixing safety bugs remarkably small, allowing programmers to spend more time on finding and fixing bugs. In C/C++, getting a confirmation from the maintainers whether certain behavior is a bug or an intended behavior is necessary in bug reporting, because there are no clear distinctions between an API misuse and a bug in the API itself. In contrast, Rust's safety rule provides an objective standard to determine whose fault a bug is. We have not received any replies from developers saying "this is intended behavior" while reporting hundreds of bugs found by RUDRA. Such distinction is not only beneficial to programmers but also to program analyzers. A program analyzer for C/C++ libraries needs to infer the intended usage of APIs [19, 40] when looking for bugs. In Rust, program analyzers can perform more aggressive optimizations because all safe usage of libraries are considered valid in the memory safety context. In fact, RUDRA's approximation of an unresolvable generic function panicking is based precisely on this guarantee.

On the other hand, our experience also suggests a limitation of Rust's safety: it is not (yet) practical to build a security mechanism solely based on Rust's safety guarantee. Examples of such design include running hostile arbitrary programs or device drivers while limiting their interaction to the predefined safe interfaces [47, 48]. Any memory safety bugs in the trust chain can transitively break the safety boundary of such systems. To demonstrate how a memory safety bug breaks trust boundaries built on Rust's soundness guarantee, we formulate a PoC against the Tock embedded operating system [48]. We exploited a bug we manually found in the Rust standard library's `Zip` iterator [13]. It took one man hour to create a capsule (an untrusted driver in TockOS) that allows arbitrary memory read/write of the private memory of other capsules[1]. Other long-lived type system bugs such as Rust issue #25860 [2] can be used for a similar exploitation.

8 Related work

Formal methods and verification. Rust, being a new programming language, has seen a lot of community effort into building formal foundations (*i.e.*, type system and operational semantics) with various design goals [27, 43, 44, 51, 63, 64], *e.g.*, proving the correctness of encapsulated unsafe code with an extensible semantic typing [44] and an aliasing model to validate raw pointers [43]. Being an early-stage language, much of the existing verification work for Rust focuses on transpiling Rust code or IR to existing verification frameworks: C [25, 60], Viper IR [17, 46], and LLVM IR [20]. These are promising directions in verifying memory safety or correctness at various layers, but, unlike RUDRA, it is fundamentally difficult to apply them to the entire ecosystem. Their scalability is limited by design: lack of generic type awareness, limited performance, or reliance on manual annotations.

Understanding unsafe Rust. As the soundness of `unsafe` is essential to Rust's safety guarantee, several attempts have been made to understand its uses (similar to Figure 2) and bug patterns from existing Rust projects and their CVEs [16, 31, 49, 65]. In RUDRA, we take this one step further to proactively discover `unsafe` bug patterns and automate their detection at a large scale. The unprecedented number of new memory safety bugs RUDRA found changes the perspective of these empirical studies; it is much more subtle and error-prone to write completely sound `unsafe` code, and it is even difficult for Rust experts and language designers.

Large-scale bug mining tools. There has been a growing trend of static analysis or bug mining tools that focus primarily on scaling their algorithms to a large scale [23, 28, 34, 41, 50], perhaps in response to the growing number of public code repositories like GitHub. Their main goal, similar to ours, is to enable a language agnostic, semantic-aware analysis, that can quickly scan and pattern match a large code base. Unfortunately, none of them officially support Rust yet, and it is unclear if the language agnostic IRs they provide can handle the specifics of the Rust language (*e.g.*, generic traits or macros) to find the `unsafe` bugs that RUDRA found.

9 Conclusion

It is commonly thought that memory safety bugs are infrequent in Rust. In this paper, we challenge this idea, presenting the hidden difficulties of writing unsafe Rust, and suggest three memory safety bug patterns in Rust. We implemented a static analyzer, RUDRA, to automatically find these bugs at the ecosystem scale. We found an unprecedented number of previously unknown memory safety bugs by using RUDRA. More importantly, these new bugs are non-trivial (*i.e.*, even made by the language designers) and unique (*i.e.*, not discoverable with existing approaches), providing a fresh view on the Rust language's safety landscape.

10 Acknowledgment

We thank Deokhwan Kim, Jiguo Song, the anonymous reviewers, and our shepherd Chris Hawblitzel, for their helpful feedback about the paper. This research was supported, in part, by Ford, the NSF award CNS-1563848 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE HR00112090034 and SocialCyber HR00112190087, ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google. The first author was partially supported by the Kwanjeong Educational Foundation scholarship.

[1] Proof-of-concept driver code can be found at: https://git.io/JzK0w

References

[1] 2015. CVE-2015-20001: BinaryHeap is not exception safe. Retrieved 2021-05-06 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-20001

[2] 2015. Rust Issue #25860: Implied bounds on nested references + variance = soundness hole. Retrieved 2021-09-26 from https://github.com/rust-lang/rust/issues/25860

[3] 2017. CVE-2017-20004: MutexGuard<Cell<i32>> must not be Sync. Retrieved 2021-05-06 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-20004

[4] 2018. CVE-2018-1000657: seg fault pushing on either side of a VecDeque. Retrieved 2021-05-06 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000657

[5] 2019. RUSTSEC-2019-0010: libflate: MultiDecoder::read() drops uninitialized memory of arbitrary type on panic in client code. Retrieved 2021-05-06 from https://rustsec.org/advisories/RUSTSEC-2019-0010.html

[6] 2019. RUSTSEC-2019-0011: memoffset: Flaw in offset_of and span_- of causes SIGILL, drops uninitialized memory of arbitrary type on panic in client code. Retrieved 2021-05-06 from https://rustsec.org/advisories/RUSTSEC-2019-0011.html

[7] 2019. RUSTSEC-2019-0022: portaudio-rs: Stream callback function is not unwind safe. Retrieved 2021-05-06 from https://rustsec.org/advisories/RUSTSEC-2019-0022.html

[8] 2020. CVE-2020-35905: MappedMutexGuard Send/Sync bound is unsound. Retrieved 2021-05-06 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-35905

[9] 2020. CVE-2020-36317: String::retain allows safely creating invalid (non-utf8) strings when abusing panic. Retrieved 2021-05-06 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-36317

[10] 2020. CVE-2020-36323: API soundness issue in join() implementation of [Borrow<str>]. Retrieved 2021-05-06 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-36323

[11] 2020. RUSTSEC-2020-0028: rocket: Clone implementation for LocalRequest is unsound. Retrieved 2021-05-06 from https://rustsec.org/advisories/RUSTSEC-2020-0028.html

[12] 2021. CVE-2021-28876: Panic safety issue in Zip specialization. Retrieved 2021-05-06 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28876

[13] 2021. CVE-2021-28879: Side effect handling in specialized zip implementation causes buffer overflow. Retrieved 2021-05-06 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28879

[14] 2021. CVE-2021-31162: Double free in Vec::from_iter specialization when drop panics. Retrieved 2021-05-06 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28879

[15] David Abrahams. 1998. Exception-safety in generic components. In *Generic Programming*. Vol. 1766. Springer, Dagstuhl Castle, Germany, 69–79.

[16] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust?. In *Proceedings of the 31st Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Everywhere, 136:1–136:27.

[17] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. In *Proceedings of the 30th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Athens, Greece, 147:1–147:30.

[18] Rust Fuzzing Authority. 2017. cargo-fuzz. Retrieved 2021-05-06 from https://github.com/rust-fuzz/cargo-fuzz

[19] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: fuzz driver generation at scale. In *Proceedings of the 18th European Software Engineering Conference (ESEC) / 27th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Tallinn, Estonia.

[20] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust Programs with SMACK. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis*. Los Angeles, CA, 528–535.

[21] Sergio Benitez. 2016. Rocket: A web framework for Rust (nightly) with a focus on ease-of-use, expressibility, and speed. Retrieved 2021-05-06 from https://rocket.rs/

[22] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, 1–19.

[23] Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA, 199–216.

[24] Tom Cargill. 1996. Exception handling: A false sense of security. In *C++ gems*. SIGS Publications, Inc., New York, NY, 423–431.

[25] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Vol. 2988. Barcelona, Spain, 168–176.

[26] Tokio Contributors. 2016. Tokio: A runtime for writing reliable, asynchronous, and slim applications with the Rust programming language. Retrieved 2021-05-06 from https://tokio.rs/

[27] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. In *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*. Cascais, Portugal, 34:1–34:29.

[28] DeepCode AG. 2020. DeepCode. Retrieved 2021-05-06 from https://www.deepcode.ai/

[29] Redox Developers. 2015. Redox Operating System. Retrieved 2021-05-06 from https://www.redox-os.org/

[30] Jeehoon Kang et al. 2018. rv6. Retrieved 2021-05-06 from https://www.redox-os.org/

[31] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. Seoul, South Korea, 246–257.

[32] Steven Fackler. 2020. RFC: Reading into uninitialized buffers. Retrieved 2021-05-06 from https://github.com/rust-lang/rfcs/blob/master/text/2930-read-buf.md

[33] Rust for Linux Team. 2021. Rust for Linux. Retrieved 2021-05-06 from https://github.com/Rust-for-Linux/linux

[34] GitHub, Inc. 2006. CodeQL. Retrieved 2021-05-06 from https://github.com/github/codeql

[35] Google. 2010. Honggfuzz. Retrieved 2021-05-06 from https://github.com/rust-fuzz/cargo-fuzz

[36] Google. 2021. Android Gabeldorsche Bluetooth Stack. Retrieved 2021-05-06 from https://android.googlesource.com/platform/system/bt/+/master/gd/rust/

[37] Google. 2021. Google C++ Style Guide. Retrieved 2021-05-06 from https://google.github.io/styleguide/cppguide.html

[38] Rust Secure Code Working Group. 2016. Rust Safety Dance. Retrieved 2021-05-06 from https://github.com/rust-secure-code/safety-dance

[39] Rust Secure Code Working Group. 2016. RustSec: The Rust Security Advisory Database. Retrieved 2021-05-06 from https://rustsec.org/

[40] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[41] Joern Contributors. 2019. Joern: The Bug Hunter's Workbench. Retrieved 2021-05-06 from https://joern.io/

[42] Ralf Jung. 2018. Two Kinds of Invariants: Safety and Validity. Retrieved 2021-05-06 from https://www.ralfj.de/blog/2018/08/22/two-kinds-of-invariants.html

[43] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked Borrows: An Aliasing Model for Rust. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL)*. New Orleans, LA, 41:1–41:32.

[44] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*. Paris, France, 66:1–66:34.

[45] Niko Matsakis. 2015. RFC: Semantic versioning for the language. Retrieved 2021-05-06 from https://github.com/rust-lang/rfcs/blob/master/text/1122-language-semver.md

[46] Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. St. Petersburg, FL, 41–62.

[47] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, 21–39.

[48] Filip Nilsson and Sebastian Lund. 2018. *Abstraction Layers and Energy Efficiency in TockOS, a Rust-based Runtime for the Internet of Things*. Master's thesis. Chalmers University of Technology, Gothenburg, Sweden.

[49] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, UK, 763–779.

[50] r2c. 2020. Semgrep. Retrieved 2021-05-06 from https://semgrep.dev/

[51] Eric Reed. 2015. *Patina: A formalization of the Rust programming language*. Master's thesis. University of Washington, Seattle, WA.

[52] The Rust Team. 2014. Clippy: A bunch of lints to catch common mistakes and improve your Rust code. Retrieved 2021-05-06 from https://github.com/rust-lang/rust-clippy

[53] The Rust Team. 2015. Miri: An interpreter for Rust's mid-level intermediate representation. Retrieved 2021-05-06 from https://github.com/rust-lang/miri

[54] The Rust Team. 2015. Rust: A language empowering everyone to build reliable and efficient software. Retrieved 2021-05-06 from https://www.rust-lang.org/

[55] The Rust Team. 2015. The Rust Reference - Trait and lifetime bounds. Retrieved 2021-05-06 from https://doc.rust-lang.org/reference/trait-bounds.html

[56] The Rust Team. 2015. The Rust Reference - Types. Retrieved 2021-05-06 from https://doc.rust-lang.org/reference/types.html

[57] The Rust Team. 2015. The Rust Standard Library Documentation - std::io::Read. Retrieved 2021-05-06 from https://doc.rust-lang.org/std/io/trait.Read.html

[58] The Rust Team. 2015. The Rustonomicon: The Dark Arts of Advanced and Unsafe Rust Programming. Retrieved 2021-05-06 from https://doc.rust-lang.org/nomicon/

[59] The Servo Team. 2012. Servo: The Servo Browser Engine. Retrieved 2021-05-06 from https://servo.org/

[60] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: A Bounded Verifier for Rust (N). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Lincoln, NE, 75–80.

[61] Philip Wadler. 1990. Linear Types can Change the World!. In *Programming concepts and methods*, Manfred Broy and Cliff B. Jones (Eds.). North-Holland, Sea of Galilee, Israel, 561.

[62] David Walker. 2004. Substructural type systems. In *Advanced topics in types and programming languages*, Benjamin C Pierce (Ed.). MIT press, Cambridge, PA, Chapter 1, 3–44.

[63] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 2018. KRust: A formal executable semantics of rust. In *Proceedings of the 2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. Guanzhou, China, 44–51.

[64] Aaron Weiss, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. 2019. Oxide: The essence of Rust. *CoRR* (2019). arXiv:1903.00982

[65] Hui Xu, Zhuangbin Chen, Mingshen Sun, and Yangfan Zhou. 2020. Memory-Safety Challenge Considered Solved? An Empirical Study with All Rust CVEs. *CoRR* abs/2003.03296 (2020). arXiv:2003.03296 https://arxiv.org/abs/2003.03296