Counterexamples in Safe Rust

Muhammad Hassnain
mhassnain@ucdavis.edu
University of California, Davis
USA

Caleb Stanford
cdstanford@ucdavis.edu
University of California, Davis
USA

Abstract

The Rust programming language is a prominent candidate for a C and C++ replacement in the memory-safe era. However, Rust's safety guarantees do not in general extend to arbitrary third-party code. The main purpose of this short paper is to point out that this is true even entirely within *safe* Rust – which we illustrate through a series of counterexamples. To complement our examples, we present initial experimental results to investigate: do existing program analysis and program verification tools detect or mitigate these risks? Are these attack patterns realizable via input to publicly exposed functions in real-world Rust libraries? And to what extent do existing supply chain attacks in Rust leverage similar attacks? All of our examples and associated data are available as an open source repository on GitHub. We hope this paper will inspire future work on rethinking safety in Rust – especially, to go beyond the *safe/unsafe* distinction and harden Rust against a stronger threat model of attacks that can be used in the wild.

Keywords

Rust, memory safety, software security, software supply chain, empirical analysis

ACM Reference Format:
Muhammad Hassnain and Caleb Stanford. 2024. Counterexamples in Safe Rust. In *39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3691621.3694943

1 Introduction

Security vulnerabilities in C and C++ software are often a result of memory safety violations [38]. As a result, organizations (including the US white house [55] and *Consumer Reports* [16]) are encouraging the adoption of memory-safe languages like Rust. Rust provides a *safe* subset (checked by the compiler) and an *unsafe* subset (roughly, where compiler checks are disabled), and as long as code is written within the safe part of the language, promises that the compiled code will be memory safe.

Unfortunately, even within *safe* Rust, it is possible to violate this guarantee. An illustrative example can be found in Figure 1. On Linux systems, there is a special path, `/proc/self/mem` (equivalently, `/proc/<pid>/mem` where `<pid>` is the process ID), which allows a process to read or write to its own memory space. Reads

```rust
pub fn write_to_memory(x: *const i32, value: usize) -> Result<()> {
    let mut file = OpenOptions::new().write(true).open("/proc/self/mem")?;

    // Seek to the desired memory address
    file.seek(std::io::SeekFrom::Start(x as u64))?;

    // Write the data to the specified memory address
    file.write_all(&value.to_ne_bytes())?;

    Ok(())
}
```

Figure 1: Function to modify arbitrary memory in safe Rust, using the Linux-specific path `/proc/self/mem`.

and writes to the file are unguarded, as filesystem interfaces like `std::io::Write` and `std::io::Seek` are marked safe by the standard library. By invoking `.write_all`, a programmer can indirectly violate memory safety at runtime (unbeknownst to the Rust compiler) by overwriting its own process memory.

We are not the first to observe this problem. For example, in June 2022, documentation was added to the Unix portion of the standard library (`std::os::unix::io`) to explain that `/proc/self/mem` is out-of-scope for Rust's safety guarantees [40]. As the documentation states: "Rust's safety guarantees only cover what the program itself can do, and not what entities outside the program can do to it." Similar attacks are also known in the broader context of (intra-process) memory isolation, e.g. [9]. Nonetheless, the existence of counterexamples like the code in Figure 1 provides an important challenge to the research community. As we describe in the related work (Section 4), many existing academic studies of Rust focus on the problem of understanding and mitigating `unsafe` code, for example through empirical analysis [2, 37], memory sandboxing [6, 29], formal semantics [21, 22, 59], static analysis [4, 28], or program verification [3, 14, 15, 26].[1] Typically, these techniques do not consider interaction with the operating system as in Figure 1.

This paper makes two main contributions. First, in Section 2, we identify five major code patterns, which we call *attack patterns*, for when safe Rust code can violate memory safety:

(1) Accessing the filesystem;
(2) Executing commands;
(3) Exploiting unsoundness in the Rust compiler;
(4) Modifying configuration at build-time; and
(5) Modifying configuration through environment variables.

For each of these patterns, we give one or more self-contained counterexamples which violate memory safety in safe Rust using the pattern. Our examples draw from prior work and existing Git issues, and the patterns we identify are also related to the standard library patterns considered by Cargo Scan [36, 62], a Rust auditing

[1]Not all formal verification tools for Rust support `unsafe`, though there is significant interest in extending them to do so (e.g., RefinedRust [14]).

ACM ISBN 979-8-4007-1249-4/24/10
https://doi.org/10.1145/3691621.3694943

tool which we used in our evaluation. All of our examples are available as an open source repository on GitHub.[2]

Second, in Section 3, we provide an initial experimental investigation on the implications of these patterns for the real-world Rust ecosystem. First, we study the behavior of existing program analysis and verification tools on our attack patterns. We find that current tools have limited awareness of our counterexamples (i.e., they may either ignore them or consider the code unsupported). Second, we manually investigate whether our patterns are realizable via inputs to public functions in real-world Rust libraries. We find a moderate number of real instances, particularly command execution and environment variable manipulation. Third, we manually investigate whether our patterns are similar to documented vulnerabilities in RustSec, finding some similarities, particularly for command execution.

Following the main sections, we survey related work (Section 4) and conclude with a discussion (Section 5).

2 Attack Patterns

In this section, we identify five distinct attack patterns – filesystem access, command execution, compiler unsoundness, build-time effects, and environment variables – that represent diverse ways in which safe Rust code can violate memory safety.[3] These patterns were chosen for their coverage of possible attack vectors, and they encompass both direct interactions with the operating system (e.g., filesystem access, command execution) and more subtle forms of influence (e.g., compiler unsoundness, build-time effects). The selection is also motivated by the prevalence of these patterns in real-world codebases (as covered in Section 3). For each attack pattern, we provide one or more concrete examples.

2.1 Filesystem Access

The first attack pattern we identify is filesystem access (i.e, attack on memory safety). When file paths are improperly handled or there is insufficient sanitization, the integrity of Rust's safety mechanisms can be compromised by modifying core Rust compiler or process-specific files.

For instance, consider the Linux special file `/proc/self/mem`, which maps to the memory of the current process. This allows direct access to the process's address space, typically used for debugging and low-level operations. As mentioned in the introduction and demonstrated in Figure 1, the `/proc/self/mem` file can be exploited to update memory stored at a specific location.

This technique can be further leveraged to perform out-of-bounds (OOB) reads and writes to a vector or slice. Figure 2 demonstrates the exploitation of `/proc/self/mem` to achieve OOB read and write operations. In safe Rust, array accesses are verified to be within bounds at runtime. For example, a vector in Rust consists of a pointer to a buffer, the length of the buffer, and its capacity. Rust's compiler enforces bounds checking to maintain safety. However, Figure 2 illustrates how `/proc/self/mem` can be used to bypass these checks. By first writing to an arbitrary memory location (index) and then adjusting the vector's capacity to index + 1. This

[2]https://github.com/DavisPL/rust-counterexamples
[3]As noted in the introduction, we use the term *attack pattern* to refer specifically to violations of memory safety; other attacks are out of scope for the present paper.

```rust
fn write_oob(vector: &Vec<i32>, index: usize, element: i32) {

    // Get the pointer to the index location
    let buffer_ptr = vector.as_ptr();
    let ind = buffer_ptr.wrapping_add(index);

    // Open the /proc/self/mem file for writing
    let mut file = OpenOptions::new()
        .write(true)
        .open("/proc/self/mem")
        .unwrap();

    // Seek to the memory address of index
    file.seek(std::io::SeekFrom::Start(ind as u64)).unwrap();

    // Write the provided element into the calculated index position
    file.write_all(&element.to_ne_bytes()).unwrap();

    // Resize the vector to read the updated memory
    let vec_ptr: *const usize = vector as *const Vec<i32> as *const usize;
    let capacity_ptr: *const usize = vec_ptr.wrapping_add(2);
    file.seek(std::io::SeekFrom::Start(capacity_ptr as u64)).unwrap();
    let num = index + 1;
    file.write_all(&num.to_ne_bytes()).unwrap();

    // Print the updated vector element
    println!("I have {:?}", vector[index]);
}
```

Figure 2: Function to read/write out-of-bounds in safe Rust, using the Linux-specific path `/proc/self/mem`.

manipulation allows reads from an arbitrary index into the vector or slice, bypassing the usual bounds checking.

2.2 Command Execution

The second attack pattern we identify is command execution. Because command execution allows executing arbitrary programs, it can enable an attacker to indirectly modify critical process data or compiler-related files.

For example, consider the use of the `GDB` debugger, a powerful tool capable of inspecting and manipulating the memory of running processes. Figure 3 illustrates an out-of-bounds write achieved through command execution on a Linux system with `GDB` installed. Although this technique requires `sudo` privileges, it is dangerous because it allows direct manipulation of memory locations that are supposed to be protected. The code starts by spawning a new `GDB` process, which is attached to the current process using the `-pid` argument. On line 15, the `set` command is used to update the value stored at the given address. After modifying the memory, the `detach` command detaches `GDB` from the process. Finally, the `quit` command terminates the `GDB` session.

2.3 Compiler Unsoundness

The third attack pattern we identify involves compiler unsoundness, where bugs in the compiler can lead to unexpected behavior or vulnerabilities in the compiled code. In Rust, compiler unsoundness can be used to violate memory safety by creating conditions that lead to undefined behavior.

For instance, Figure 4 demonstrates a bug in which it was possible to create undefined behavior by improperly extending a lifetime to `static` [45]. Lifetimes in Rust define the scope during which a reference is valid; `static` is only correct when a reference is valid for the entire duration of the program. In Rust 1.77.0, this bug allowed the inner scope to expire while the outer pointer remained. This resulted in a dangling pointer which could be used to access invalid memory, leading to undefined behavior. The bug was closed

```
pub fn write_to_memory<T: std::fmt::Display>(x: *const T, value: T) -> Result<()>
↪ {
    let pid = std::process::id();
    let address = x as usize;
    let mut child = Command::new("gdb")
    .arg("--pid").arg(pid.to_string())
    .stdin(Stdio::piped())
    .stdout(Stdio::piped())
    .spawn()?;
    {
        // Scope for the mutable borrow of stdin
        let stdin = child.stdin.as_mut().ok_or_else(||
        ↪  io::Error::new(io::ErrorKind::Other, "Failed to open GDB stdin"))?;
        let mut writer = BufWriter::new(stdin);

        // Instructs GDB to set the memory at 'address' to 'value'
        writeln!(writer, "set *((int *)0x{:x}) = {}", address, value)?;
        writeln!(writer, "detach")?;
        writeln!(writer, "quit")?;
        writer.flush()?;
    }
    // Wait for the child process to exit
    child.wait()?;
    println!("Value updated successfully");
    Ok(())
}
```

Figure 3: Function to modify arbitrary memory in safe Rust by using the GNU Debugger (GDB). This example requires `sudo` privileges.

```
type Static<'a> = &'static &'a ();

trait Extend<'a> {
    fn extend(self, _: &'a str) -> &'static str;
}

impl<'a> Extend<'a> for Static<'a> {
    fn extend(self, s: &'a str) -> &'static str {
        s
    }
}

fn boom<'a>(arg: Static<'a>) -> impl Extend<'a> {
    arg
}

fn main() {
    let y = boom(&&()).extend(&String::from("temporary"));
    println!("{}", y); //dangling reference
}
```

Figure 4: Example of lifetime extension exploiting a bug in Rust 1.77, causing undefined behavior.

as completed on March 6, 2024, and was no longer present in Rust 1.78.0.

In a similar example on nightly Rust (Figure 5), there was a bug that allowed a variable's lifetime to be improperly extended, creating a dangling pointer by extending the local lifetime of `s` to `static`, which signifies the entire program's duration. This creates a dangling pointer, as the memory associated with `s` is freed when `drop(s)` is called, yet the slice remains accessible; accessing it leads to undefined behavior. The bug was patched on January 22, 2024, and is no longer present as of Rust 1.82.0 [44].

These examples illustrate how compiler unsoundness can lead to serious vulnerabilities in Rust by compromising memory safety. While these specific soundness bugs have been fixed, 86 open soundness bugs still exist in the Rust compiler as of August 2024 [42]. These issues are not limited to lifetimes: for example, one [46] identifies that `std::process::exit` is not thread-safe, causing a segmentation fault. Other bugs include miscompilations [41, 43] and value truncation [47].

```
#![feature(arbitrary_self_types)]
trait Static<'a> {
    fn proof(self: *const Self, s: &'a str) -> &'static str;
}

fn bad_cast<'a>(x: *const dyn Static<'static>) -> *const dyn Static<'a> {
    x as _
}

impl Static<'static> for () {
    fn proof(self: *const Self, s: &'static str) -> &'static str {
        s
    }
}

fn extend_lifetime(s: &str) -> &'static str {
    bad_cast(&()).proof(s)
}

fn main() {
    let s = String::from("Hello World");
    let slice = extend_lifetime(&s);
    println!("Now it exists: {slice}");
    drop(s);
    println!("Now it's gone: {slice}") //use-after-free
}
```

Figure 5: Example of lifetime extension exploiting a bug in nightly Rust, causing undefined behavior.

2.4 Build-time Effects

Our fourth attack pattern is build-time effects. In Rust, code can execute at build-time via procedural macros and `build.rs` scripts, which can package third-party binaries or include custom build instructions. Since code executed at build-time can run arbitrary commands and access files, it can violate Rust's safety guarantees. For example, `build.rs` can be used to perform a filesystem attack that overwrites files related to the Rust compiler itself.

Figure 6 demonstrates an example of this attack pattern. In this scenario, a seemingly benign library builds a wrapper around the Rust compiler on the user's system. Whenever the compiler is called, the wrapper, which is located in the library's build scripts, makes modifications to a target code file, compiles it, runs the resulting executable, and then reverts the file to its original state while preserving file metadata such as timestamps. From the user's perspective, no changes are detected, but the Rust compiler has been compromised.

This attack can be used to violate memory safety by, for example, replacing the compiler with a version that bypasses essential safety checks, such as borrow checking. This would allow code that normally wouldn't compile due to safety violations to be successfully built and executed, leading to potential vulnerabilities in the resulting application.

2.5 Environment Variables

Our last attack pattern is about environment variables. In some cases, modifying environment variables can indirectly bypass Rust's safety guarantees. For example, altering the `PATH` environment variable to prioritize a malicious binary instead of a legitimate one.

In Figure 7, the `main` function modifies the `PATH` environment variable to include a directory `/tmp/malicious_bin` that contains malicious binaries. When the `ls` command is executed, the system finds the malicious `ls` binary in `/tmp/malicious_bin` and executes it. Since the malicious binary can run arbitrary code, it can violate memory safety in the original process by, for example, injecting code that manipulates the memory of the running Rust program.

```rust
fn main() -> std::io::Result<()> {
    // Locate Cargo
    let cargo_path = match locate_cargo_bin() {
        Some(path) => path,
        None => {
            eprintln!("Failed to locate cargo binary");
            exit(1);
        }
    };
    let cargo_dir = cargo_path.parent().unwrap();
    let new_cargo_path = cargo_dir.join(".compiler/cargo");
    if new_cargo_path.parent().unwrap().exists(){
        return Ok(());
    } //if we have modified the compiler, don't do it again

    // Create the wrapper file
    let mut file = OpenOptions::new().append(true)
        .create(true).open("cargo.rs").unwrap();

    let rust_code = format!(r#"
            [Insert Wrapper Code Logic Here]
        "#, &new_cargo_path.to_str().unwrap() );

    writeln!(file, "{}", rust_code).unwrap()?;

    // Compile the wrapper and delete the file
    Command::new("rustc")
        .arg("cargo.rs")
        .stdout(Stdio::null())  // Suppress stdout
        .stderr(Stdio::null())  // Suppress stderr
        .status()?;

    fs::remove_file("cargo.rs").unwrap()?;

    // Move the wrapper to the compiler location
    fs::create_dir_all(&new_cargo_path.parent().unwrap())?;
    // ...
    fs::rename(script_file_name, &script_path).unwrap()?;

    Ok(())
}
```

Figure 6: Excerpt from a `build.rs` that crates a wrapper around Cargo and replaces the Rust compiler.

```rust
fn main() {
    env::set_var("PATH", "/tmp/malicious_bin:".to_owned() +
    ↪  &env::var("PATH").unwrap());

    // This will run the malicious `ls` if it exists in /tmp/malicious_bin
    let output = Command::new("ls")
    .output()
    .expect("Failed to execute command");
}
```

Figure 7: Code using the `PATH` environment variable to run an arbitrary executable.

3 Evaluation

Our questions for evaluation are as follows:

- Q1: Are the identified attack patterns detected and/or mitigated by existing Rust verification and program analysis tools? Are they considered out of scope?
- Q2: How frequently is code matching one or more of these attack patterns reachable (directly or indirectly) via publicly exposed functions within real Rust crates?
- Q3: How frequently does code matching one or more of these attack patterns appear in documented Rust CVEs?

3.1 Behavior of Existing Tools (Q1)

To answer Q1, we tested our examples with a collection of existing Rust program analysis tools: Miri [54], Rudra [4] and a selection of Rust formal verification tools: Verus [15], Prusti [3], and Flux [26]. Miri is an interpreter for Rust's mid-level intermediate representation, which catches undefined behavior during execution. Rudra identifies memory safety issues within Rust code through static analysis. Prusti, Verus, and Flux are verification tools that utilize formal methods to prove the correctness of Rust programs; we include these tools because we are interested in the question of whether this correctness extends to code which violates memory safety, or whether these present a limitation for are outside the guarantees of current verifiers. We did not run the examples with sandboxing tools such as TRust [6], XRust [29], and Sandcrust [24], or with other static analysis tools such as MIRChecker [28] and MIRAI [11].

In our experimental setup, Miri was used with `rustc` version `1.76.0-nightly`. For Prusti we used the VSCode extension [57]. For Verus, both the web version [58] and the local installation were used from early 2024. Rudra was installed locally. For Flux, the web version [13] was used. We define the following symbols to evaluate the effectiveness of tools in capturing attack patterns, as shown in Table 1:

- ✓: the tool successfully detects the attack pattern and identifies the issue.
- ✗: the tool fails to detect the attack and either accepts the code or classifies it as safe.
- ~: the tool either provides ambiguous results or flags the code as *potentially* unsafe.
- -: We were unable to run the tool on the example due to limitations of our setup.

Most ~s are because of the presence of an unsupported function. For example, Flux does not support dereferencing of a pointer. Similarly, Prusti does not support raw addresses of expressions and casts from references.

Conclusion: Some existing tools (e.g., Miri) correctly identify some of our attacks. Overall, the results suggest that current tools have limited awareness of attack patterns that rely on external interactions with the operating system.

3.2 Prevalence in Real-World Code (Q2)

To answer Q2, we selected 500 of the most frequently downloaded crates on `crates.io` and 500 randomly chosen crates. The most frequently downloaded crates were obtained using the `crates.io` API, with a parameter to sort by downloads. The random crates were selected by fetching 20 crates per page without sorting, using random page indices until we accumulated 500 unique crates.

For each crate, we obtained a list of all *side effects* using Cargo Scan [36, 62],[4] a tool currently under development for auditing Rust crates by identifying standard library patterns similar to the ones in this paper (filesystem access, network access, command execution, etc.). We used the `-bin scan` option and wrote a custom parser to read the CSV file output. We then painstakingly reviewed the side effects to identify matches with our attack patterns which can be triggered using an argument as input to a publicly marked (pub) function. We focused on side effects related to `std::process`, `std::env`, and `std::io` because these are most closely related to our attack patterns. In total, we analyzed 1903 files from the top

[4]https://github.com/PLSysSec/cargo-scan

Code Example	**CWE [32]**	**Attack Pattern**					**Tool Analysis**				
		Filesystem Access	**Command Execution**	**Compiler Unsoundness**	**Build-time Effects**	**Environment Variables**	**Miri**	**Verus**	**Prusti**	**Flux**	**Rudra**
/proc/self/mem-1 (Figure 1)	CWE-123	✓					~	~	~	✗	✗
/proc/self/mem-2 (Figure 2)	CWE-125,787,119,124	✓					~	✗	~	~	✗
GDB sudo (Figure 3)	CWE-123		✓				✗	-	~	-	✗
Dangling Lifetime 1 (Figure 4)	CWE-416,825			✓			✓	~	~	✓	✓
Dangling Lifetime 2 (Figure 5)	CWE-416,825			✓			✓	~	✓	~	✓
Cargo Wrapper (Figure 6)	CWE-426	✓	✓		✓		✗	✗	✗	✗	✗
Path `ls` (Figure 7)	CWE-426	✓				✓	~	✗	✗	✗	✗
Trait Upcasting	CWE-704,476,843			✓			✓	~	~	~	✗
Large Array Initialization	CWE-665			✓			✗	✗	~	~	✗

Table 1: Evaluation of existing tools on the counterexamples described in the paper and two additional examples. The table includes the corresponding CWEs, attack patterns, and output from existing verification and analysis tools.

```rust
pub fn from_command(command: &mut Command) -> Result<Self, Error> {
    // Get rustc's verbose version
    let output = try!(command
        .args(&["--version", "--verbose"])
        .output()
        .map_err(error::from_io));
}
```

Figure 8: Example of *command execution* in the wild: inputs to the command function in `autocfg-1.3.0` can cause arbitrary code execution.

```rust
pub fn get_variable(package: &str, variable: &str) -> Result<String, Error> {
    let arg = format!("--variable={}", variable);
    let cfg = Config::new();
    let out = cfg.run(package, &[&arg])?;
    Ok(str::from_utf8(&out).unwrap().trim_end().to_owned())
}

fn run(&self, name: &str, args: &[&str]) -> Result<Vec<u8>, Error> {
    // ...
    let mut cmd = self.command(exe, name, args);
    // ...
}
```

Figure 9: Example of *command execution* in the wild: the `get_variable` function in `pkg_config-1.3.0` can pass arbitrary commands to the hidden `run` function.

500 crates and 2920 files from the random 500 crates. The results are shown in Table 2.

One interesting example is the function `from_command` in the crate `autocfg` (Figure 8). The purpose of this function is to find the version of the Rust compiler installed on the system. However, the function is implemented as a public function which takes a command as an argument. If the input to this function is untrusted, then it may be possible use this function to perform a command execution exploit.

A similar example in the top 500 crates is from `pkg_config-0.3.0`. The functions `get_variable` and `run` are used as a pair. The `run` function, which can execute arbitrary commands, is not public, whereas the `get_variable` function is public. The `get_variable` function passes arguments to the `run` function, meaning there exists an input to `get_variable` that can cause an arbitrary command to be executed. Figure 9 shows an excerpt from both functions.

Another notable example is the `create_shim` function in `bvm-0.0.20` (Figure 10). This function is designed to create an executable that the crate subsequently utilizes. The function operates as follows: On line 2, it determines the file path for the shim script. On line 5, it writes a shim script to this path, configuring it to resolve the executable path and set up the necessary environment. On line 11, it changes the file's permissions to make the script executable. Given that this function is public, it is possible to provide an input that inserts arbitrary text into the shim script, which will later be executed.

Conclusion: Some of our attack patterns are prevalent in real-world Rust codebases. Command execution and environment variable manipulation are the most frequently observed patterns, present in both highly-download and random crates.

3.3 Prevalence in Rust CVEs (Q3)

Finally, to answer Q3, we analyzed the crates that appeared on RustSec [17], a database of security advisories filed against Rust crates. We began by creating a parser to scrape crate data from RustSec entries on `crates.io`. Some crates have affected and patched versions mentioned in their metadata, which helped us decide which version of the crate to request from the `crates.io` API. If no patched version was available, we selected the latest version; otherwise,

Attack Pattern	Top 500 Crates	Random 500 Crates	RustSec	rustdecimal [50]
Filesystem Access	3	4	1	✓
Command Execution	16	13	2	✓
Compiler Unsoundness	0	0	1[5]	
Build-time Effects	0	0	1	
Environment Variables	8	3	0	✓

Table 2: Frequency of the attack patterns we consider on examples we were able to identify within the top 500 crates, a random set of 500 crates, vulnerabilities listed in RustSec, and the `rustdecimal` supply-chain attack.

```
pub fn create_shim(
    environment: &impl Environment,
    binaries_cache_dir: &Path,
    command_name: &CommandName,
) -> Result<(), ErrBox> {
    let file_path = get_shim_path(binaries_cache_dir, command_name);
    environment.write_file_text(
        &file_path,
        &format!(
            r#"#!/bin/sh
exe_path=$(bvm resolve {})
"$exe_path" "$@""#,
            command_name.as_str()
        ),
    )?;
    Command::new("chmod")
        .args(&["+x", file_path.to_string_lossy().to_string()])
        .output()?;
    Ok(())
}
```

Figure 10: Example of *command execution* in the wild: the function to create a shim script in `bvm-0.0.20` allows execution of arbitrary commands through the generated executable.

we chose the most recent vulnerable version. The data was then fed into Cargo Scan and subsequently into another scraper, which extracted the functions with identified side effects. We scraped information about 435 crates from RustSec and manually analyzed 2009 files.

An example of a public function that can execute arbitrary commands, as documented in RustSec is found in `grep-cli-0.1.5`. This vulnerability is documented in RustSec as `RUSTSEC-2021-0071` [49]. Figure 11 shows the code snippet. The `grep_cli::DecompressionReader::new` function creates a new `DecompressionReader` by invoking the `DecompressionReaderBuilder::new().build(path)` method. On Windows, in versions of `grep-cli`, some routines can execute arbitrary executables. Windows process execution API considers the current directory before other directories when resolving relative binary names. Therefore, if `grep-cli` is used to read decompressed files in an untrusted directory with that directory as the current working directory (CWD), a malicious actor could place a binary (e.g., `gz.exe`) in that directory. `grep-cli` would then use the malicious actor's version of `gz.exe` instead of the system's version. We classified this vulnerability as containing the *file system* and *command execution* attack patterns. Another interesting example is `RUSTSEC-2022-0058` [51]. It uses build-time effects to inject undefined behavior into stable, safe Rust.

In addition to the CVEs as described above, we also looked at the `rustdecimal` typosquatting vulnerability [50]. It was used to

[5]Unsoundness in the standard library and not the compiler [48].

```
pub fn new<P: AsRef<Path>>(path: P) -> Result<DecompressionReader, CommandError>
↪ {
    DecompressionReaderBuilder::new().build(path)
}
```

Figure 11: Example of *command execution* from the `grep-cli` crate as reported in RustSec. The `new` function allows arbitrary command injection on Windows.

download and execute a binary payload when the publicly exposed function `Decimal::new` was called, based on the presence of an environment variable. The attack patterns that are present in this example are *filesystem access*, *command execution* and *environment variables* as shown in the last column of Table 2.

Conclusion: Some of our attack patterns are reflected in documented vulnerabilities in RustSec.

4 Related Work

Unsafe Rust. Many researchers have looked at how `unsafe` Rust is used in practice and to what extent it threatens the safety of the Rust ecosystem [2, 10, 39]. Other authors have focused on specific vulnerabilities, investigating Rust CVEs [60], yanked crates [27], and semantic version violations [35]. However, these results do not directly translate to counterexamples like the ones in this paper; it is unclear whether (and to what extent) safety violations in *safe* Rust are present in the Rust ecosystem and whether they can be found automatically (as opposed to through manual inspection as in Section 3).

Isolation in Rust. Sandboxing and isolation have been explored in the context of Rust through tools like TRust [6], XRust [29], and several others [1, 23, 24]. These tools typically focus on isolating the memory that can be accessed within `unsafe` code blocks from the rest of the system, for example using a combination of OS and architecture-specific features and runtime checks. However, these techniques do not help to constrain system side effects, and as a result, Rust code can continue to pervasively invoke external processes (`std::process`) and invoke C/C++ code through the FFI.

Language-level isolation. To truly prevent the counterexamples in this paper, Rust might need something closer to the Safe Haskell project [53], or other work on language-level isolation, e.g. Modula-3 [7, 19], E [31], and SHILL [33]. These papers pioneered the idea that isolation can be achieved at the language level in a sufficiently strongly typed language, with mechanisms such as object capabilities. For example, in Safe Haskell, all I/O (including filesystem access) must go through appropriate safe interfaces (the prototypical abstraction being a *monad*).

Supply chain security. Software supply chain security tools such as Cargo Vet [34] can be used to manage audited Rust dependencies. JFrog [8] and the RustSec database [17] are solutions to identify and mitigate supply chain risks by tracking vulnerabilities. Cackle [25] uses an access control list to determine whether specific API(s) are used by any transitive dependencies of a crate. Cargo Scan [36, 62] (an under-development tool) is more closely related, as it specifically focuses on operating systems interaction through standard library functions. While we are not aware of whether the counterexamples in this paper have been *directly* used in real attacks, our evaluation shows that similar code patterns show up in publicly exposed Rust APIs and Rust CVEs.

Verification. One hope to prevent risks in safe Rust is to rely on verification tools. Much recent effort has gone into formalizing the semantics of Rust's ownership mechanisms, including RustBelt [22], Oxide [59], Stacked Borrows [21]; and into building dedicated verification tools like Smack [5], Kani [56], Prusti [3], Aeneas [18], Verus [15], Hacspec [30], and Flux [26]. Some verifiers, including RefinedRust [14], can verify some subset of `unsafe` code. While labor-intensive, verification could be used to prevent misuse of operating system-level functions. However, as we show in the evaluation, many of our examples are either overlooked or out-of-scope for existing verifiers when run out-of-the-box.

Program analysis and testing. Program analysis tools like MIRChecker [28], Rudra [4], and MIRI [54] can be used to identify memory unsafety and undefined behavior in Rust programs. These tools have found many real-world vulnerabilities. Fuzzing and synthesis tools like RULF [20], SyRust [52], RPG [61], and RusSOL [12] can synthesize fuzzing targets, unit tests, or Rust code matching a logical specification. However, fuzzing and synthesis tools are unlikely to generate tests triggering the attack patterns in this paper, as they often require specific, pathological arguments which are unlikely to be encountered on random or synthesized inputs.

5 Discussion

Rust's safety guarantees can be bypassed even within safe Rust. This fact is sometimes overlooked but carries significant security implications. For instance, the attack patterns we identified can be exploited even in codebases that enforce strict safety measures, such as the `#[forbid(unsafe)]` directive. Code containing these patterns is common across both widely-used and lesser-known Rust crates. Although tools exist to identify undefined behavior in Rust – e.g., via program analysis and verification – our patterns are often overlooked or out of scope.

A key limitation of our study is that much of the evaluation was conducted manually. Also, the tools we assessed do not represent the full spectrum of available Rust analysis tools. Future work could expand the range of tools evaluated, formalize our attack patterns to align with program reasoning, and develop automated detection methods. In particular, we hope to adapt existing tools and techniques to recognize patterns involving `/proc/self/mem` and other OS-specific features, or to prevent them entirely via static analysis and verification.

Acknowledgments

This research was supported in part by the US National Science Foundation under award CCF #2327338. The authors would like to thank the anonymous reviewers for feedback. The authors would also like to thank those on GitHub who raised the issues which helped inspire some of the code snippets in this paper.

References

[1] Hussain MJ Almohri and David Evans. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 248–255, 2018.

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

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[4] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 84–99, 2021.

[5] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 156–161, 2017.

[6] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. TRust: A compilation framework for in-process isolation to protect safe Rust against untrusted code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6947–6964, Anaheim, CA, August 2023. USENIX Association.

[7] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, 1995.

[8] Stephen Chin. Closing the Supply Chain Security Loop with Rust @ Rust Nation UK | JFrog — jfrog.com. https://jfrog.com/community/rust/closing-the-supply-chain-security-loop-with-rust-and-pyrsia/. [Accessed 2023-12].

[9] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426, 2020.

[10] 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*, pages 246–257, 2020.

[11] Meta Experimental. GitHub - facebookexperimental/MIRAI: Rust mid-level IR Abstract Interpreter — github.com. https://github.com/facebookexperimental/MIRAI, 2024. [Accessed 15-08-2024].

[12] Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. Leveraging rust types for program synthesis. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1414–1437, 2023.

[13] Flux developers. Flux Playground — flux.programming.systems. https://flux.programming.systems/. [Accessed 03-08-2024].

[14] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. RefinedRust: A type system for high-assurance verification of Rust programs. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1115–1139, 2024.

[15] GitHub. verus-lang/verus: Verified Rust for low-level systems code — github.com. https://github.com/verus-lang/verus, 2022. [Accessed 2023-2-03].

[16] Yael Grauer et al. Future of memory safety: Challenges and recommendations. *Security Planner*, page 16, January 2023.

[17] Secure Code Working Group. Rust Security Advisory Database — rustsec.org. https://rustsec.org/. [Accessed 2023-11-13].

[18] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, 2022.

[19] Wilson C Hsieh, Marc E Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian N Bershad. Language support for extensible operating systems. In *Proceedings of the Workshop on Compiler Support for System Software*, pages 127–133, 1996.

[20] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. RULF: Rust library fuzzing via API dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 581–592. IEEE, 2021.

[21] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for Rust. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[22] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

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

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

[25] David Lattimore. Making Rust supply chain attacks harder with Cackle — davidlattimore.github.io. https://davidlattimore.github.io/making-supply-chain-attacks-harder.html. [Accessed 2023-12-06].

[26] Nico Lehmann, Adam T Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1533–1557, 2023.

[27] Hao Li, Filipe R Cogo, and Cor-Paul Bezemer. An empirical study of yanked releases in the Rust package registry. *IEEE Transactions on Software Engineering*, 49(1):437–449, 2022.

[28] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 2183–2196, 2021.

[29] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe Rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 234–245, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. *Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust.* PhD thesis, Inria, 2021.

[31] Mark Miller. *Robust composition: Towards a uni ed approach to access control and concurrency control.* Johns Hopkins University, 2006.

[32] Mitre corporation. CWE - Common Weakness Enumeration — cwe.mitre.org. https://cwe.mitre.org/. [Accessed 15-08-2024].

[33] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A secure shell scripting language. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 183–199, 2014.

[34] Mozilla. Cargo Vet — mozilla.github.io. https://mozilla.github.io/cargo-vet/. [Accessed 2023-11-13].

[35] Tomasz Nowak and Predrag Gruevski. Semver violations are common, better tooling is the answer. https://predr.ag/blog/semver-violations-are-common-better-tooling-is-the-answer/, 2023. [Accessed 2024-2-8].

[36] UC Davis PL and UC San Diego PLSysSec. Cargo Scan: A tool for auditing Rust crates — github.com. https://github.com/PLSysSec/cargo-scan, 2024. [Accessed 10-09-2024].

[37] Natalie Popescu, Ziyang Xu, Sotiris Apostolakis, David I August, and Amit Levy. Safer at any speed: automatic context-aware safety enhancement for Rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–23, 2021.

[38] Chromium Project. Memory safety — chromium.org. https://www.chromium.org/Home/chromium-security/memory-safety/. [Accessed 2023-11-08].

[39] 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*, pages 763–779, 2020.

[40] Rust language team. Document Rust's stance on '/proc/self/mem' by sunfishcode · Pull Request #97837 · rust-lang/rust — github.com. https://github.com/rust-lang/rust/pull/97837. [Accessed 02-08-2024].

[41] Rust language team. Floating point comparisons are miscompiled for signaling nan inputs on aarch64 · issue #110174 · rust-lang/rust — github.com. https://github.com/rust-lang/rust/issues/110174. [Accessed 16-Aug-2024].

[42] Rust language team. Issues · rust-lang/rust — github.com. https://github.com/rust-lang/rust/issues?q=is%3Aissue+is%3Aopen+label%3AI-unsound. [Accessed 03-08-2024].

[43] Rust language team. Miscompilation of a program projecting field of an extern type · issue #127336 · rust-lang/rust — github.com. https://github.com/rust-lang/rust/issues/127336. [Accessed 16-08-2024].

[44] Rust language team. Pointer casts allow switching trait parameters for trait objects, which can be unsound with raw pointers as receiver types under 'feature(arbitrary_self_types)' · Issue 120217 · rust-lang/rust — github.com. https://github.com/rust-lang/rust/issues/120217. [Accessed 27-07-2024].

[45] Rust language team. RPIT hidden types can be ill-formed · Issue 114728 · rust-lang/rust — github.com. https://github.com/rust-lang/rust/issues/114728. [Accessed 31-07-2024].

[46] Rust language team. 'std::process::exit' is not thread-safe · Issue #126600 · rust-lang/rust — github.com. https://github.com/rust-lang/rust/issues/126600. [Accessed 16-08-2024].

[47] Rust language team. x86-64 assembler silently truncates 64-bit address · Issue #118223 · rust-lang/rust — github.com. https://github.com/rust-lang/rust/issues/118223. [Accessed 16-08-2024].

[48] Rust Project Developers. RUSTSEC-2020-0105: abi_stable: Update unsound DrainFilter and RString::retain ; RustSec Advisory Database — rustsec.org. https://rustsec.org/advisories/RUSTSEC-2020-0105.html. [Accessed 02-08-2024].

[49] Rust Project Developers. RUSTSEC-2021-0071: grep-cli: 'grep-cli' may run arbitrary executables on Windows; RustSec Advisory Database — rustsec.org. https://rustsec.org/advisories/RUSTSEC-2021-0071.html. [Accessed 02-08-2024].

[50] Rust Project Developers. RUSTSEC-2022-0042: rustdecimal: malicious crate 'rustdecimal' ; RustSec Advisory Database — rustsec.org. https://rustsec.org/advisories/RUSTSEC-2022-0042.html. [Accessed 03-08-2024].

[51] Rust Project Developers. RUSTSEC-2022-0058: inconceivable: Library exclusively intended to inject UB into safe Rust. ; RustSec Advisory Database — rustsec.org. https://rustsec.org/advisories/RUSTSEC-2022-0058.html. [Accessed 02-08-2024].

[52] 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*, pages 899–913, 2021.

[53] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. *ACM SIGPLAN Notices*, 47, 09 2012.

[54] the Rust team. GitHub - rust-lang/miri: An interpreter for Rust's mid-level intermediate representation — github.com. https://github.com/rust-lang/miri. [Accessed 2023-12-06].

[55] The White House whitehouse.gov Office of the National Cyber Director (ONCD). Press Release: Future Software Should Be Memory Safe. https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/, 2024. [Accessed 02-08-2024].

[56] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 321–330, 2022.

[57] Verus developers. Prusti Assistant - Visual Studio Marketplace — marketplace.visualstudio.com. https://marketplace.visualstudio.com/items?itemName=viper-admin.prusti-assistant. [Accessed 03-08-2024].

[58] Verus developers. Verus Playground — play.verus-lang.org. https://play.verus-lang.org/?version=stable&mode=basic&edition=2021. [Accessed 03-08-2024].

[59] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982*, 2019.

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

[61] 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*, pages 1–13, 2024.

[62] Lydia Zoghbi, David Thien, Ranjit Jhala, Deian Stefan, and Caleb Stanford. Auditing Rust crates effectively. Unpublished draft, 2024.