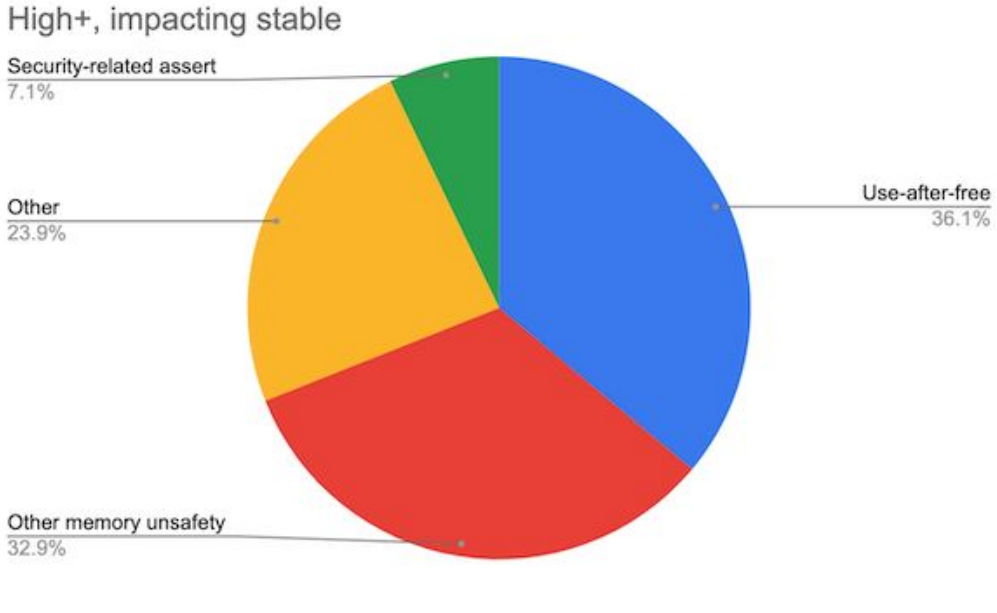
Counterexamples in Safe Rust

Muhammad Hassnain (mhassnain@ucdavis.edu)
Caleb Stanford (cdstanford@udavis.edu)

Memory Safety



FEBRUARY 26, 2024

Press Release: Future Software Should Be Memory Safe

Leaders in Industry Support White House Call to Address Root Cause of Many of the Worst Cyber Attacks

Enter's Rust



- Code written in Rust guarantees:
  - Runtime performance of traditional system languages like C/C++
  - Memory Safety
    - No Dangling Pointers
    - No Garbage Collector
    - No Use after free or out of bounds
  - Type Safety
- Rust allows certain operations using "unsafe" keyword
- Rust compiler cannot guarantee safety in "unsafe" blocks

Is Rust really memory safe?

- Unsafe → the responsibility lies on the programmer
- Safe → Let's take a look!

Contributions

- 5 Attack Patterns → code patterns that break memory safety.
- Evaluation of these patterns on Rust Ecosystem.
  - Behaviour of Existing Program Analysis and Verification tools on attack patterns.
  - Prevalence of Attack Patterns in Real-World Code.
  - Prevalence of Attack Patterns in Rust CVEs.

Attack Patterns

Code patterns that can break memory safety.

- Filesystem Access
- Command Execution
- Compiler Unsoundness
- Build-time Effects
- Environment Variables

Filesystem Access

Filesystem Access

When file paths are improperly handled or there is insufficient sanitization, the integrity of Rust's safety mechanisms can be compromised.

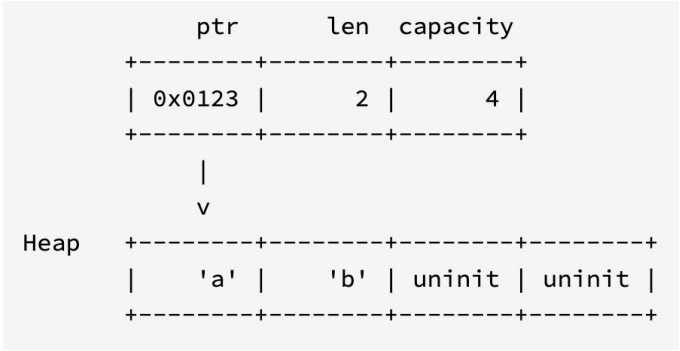
```rust
pub fn write_to_memory(x: *const i32, value: usize) -> Result<()> {
    let mut file = OpenOptions::new().write(true).open("/proc/self/mem")?;

    // 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(())
}
```

Filesystem Access



https://doc.rust-lang.org/std/vec/struct.Vec.html

```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();

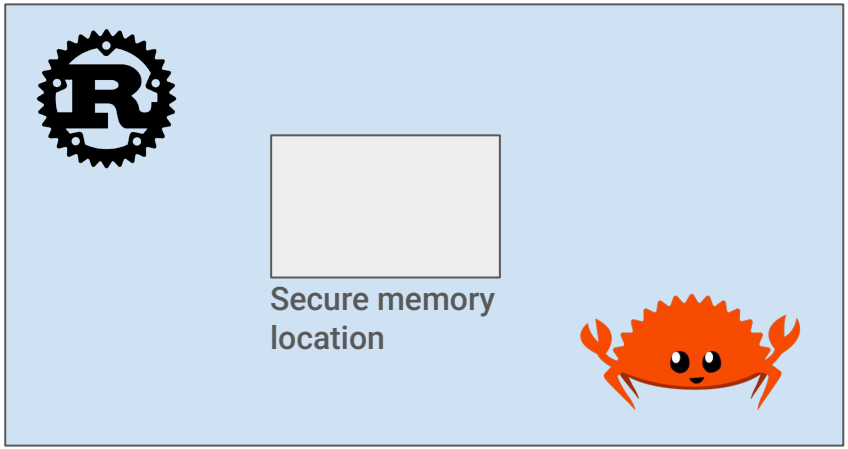
    // Write the provided element into the calculated index position
    file.write_all(&element.to_ne_bytes()).unwrap();

    // 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]);
}
```

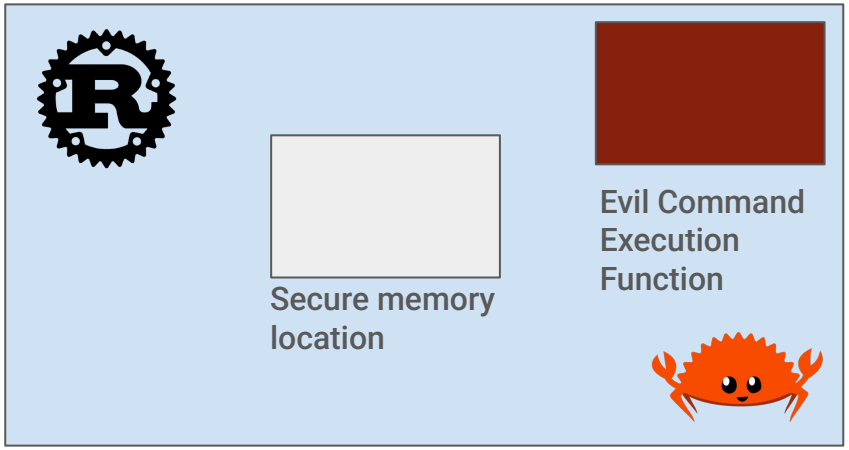
Figure 2: Function to read/write out-of-bounds in safe Rust, using the Linux-specific path `/proc/self/mem`.

Command Execution

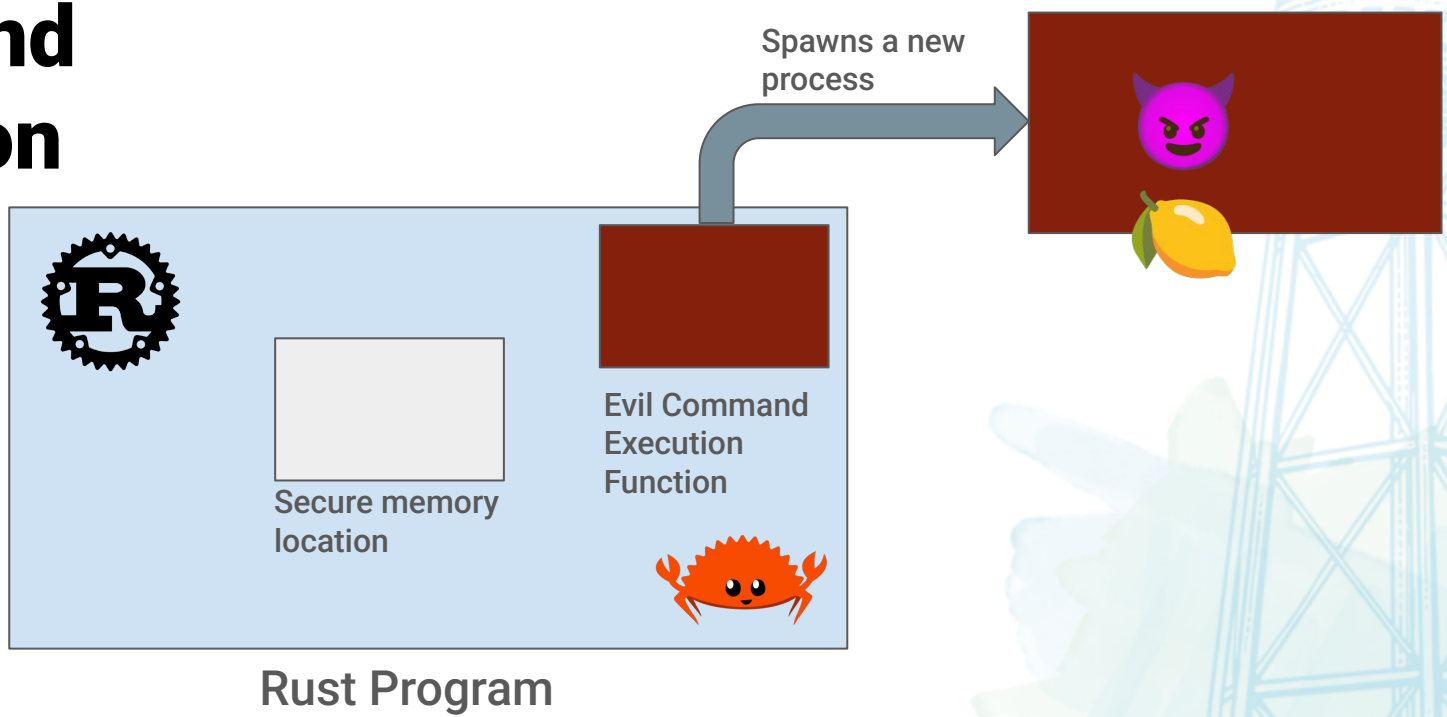


Rust Program

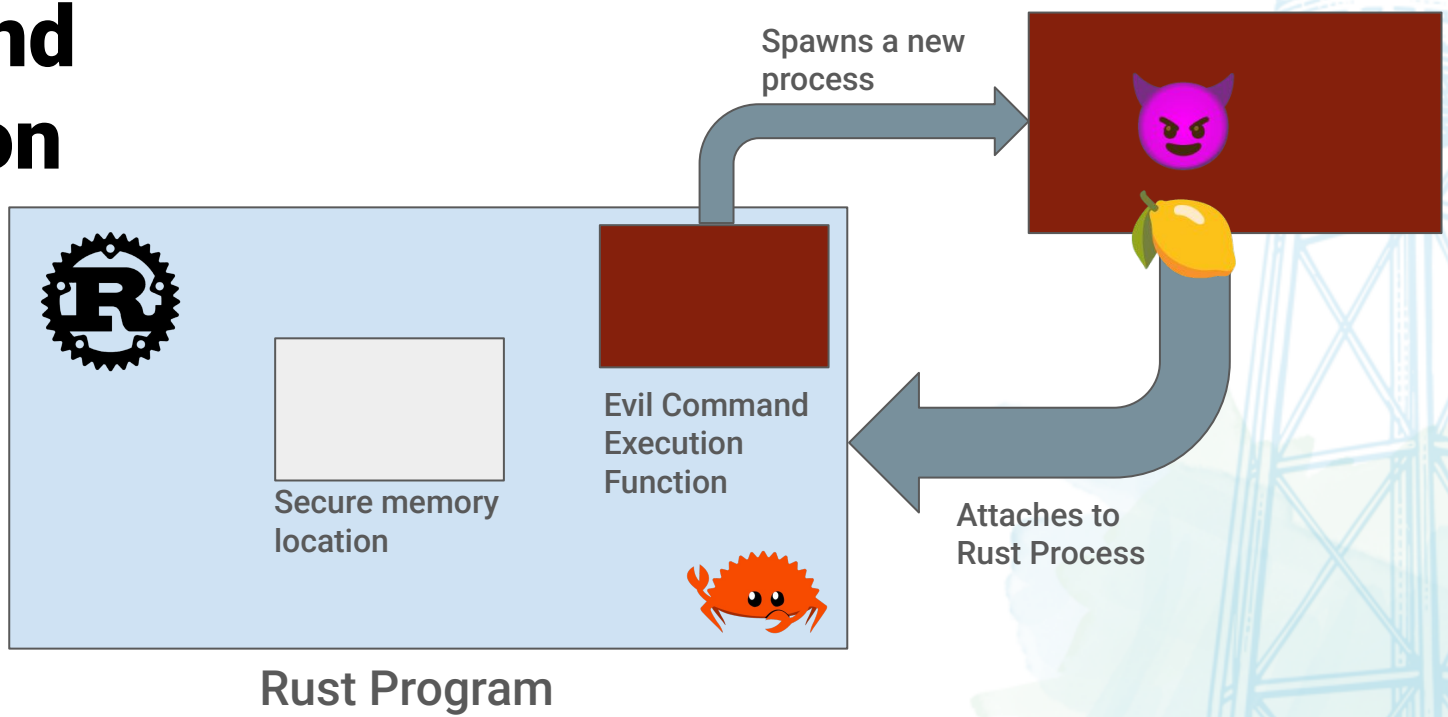
Command Execution



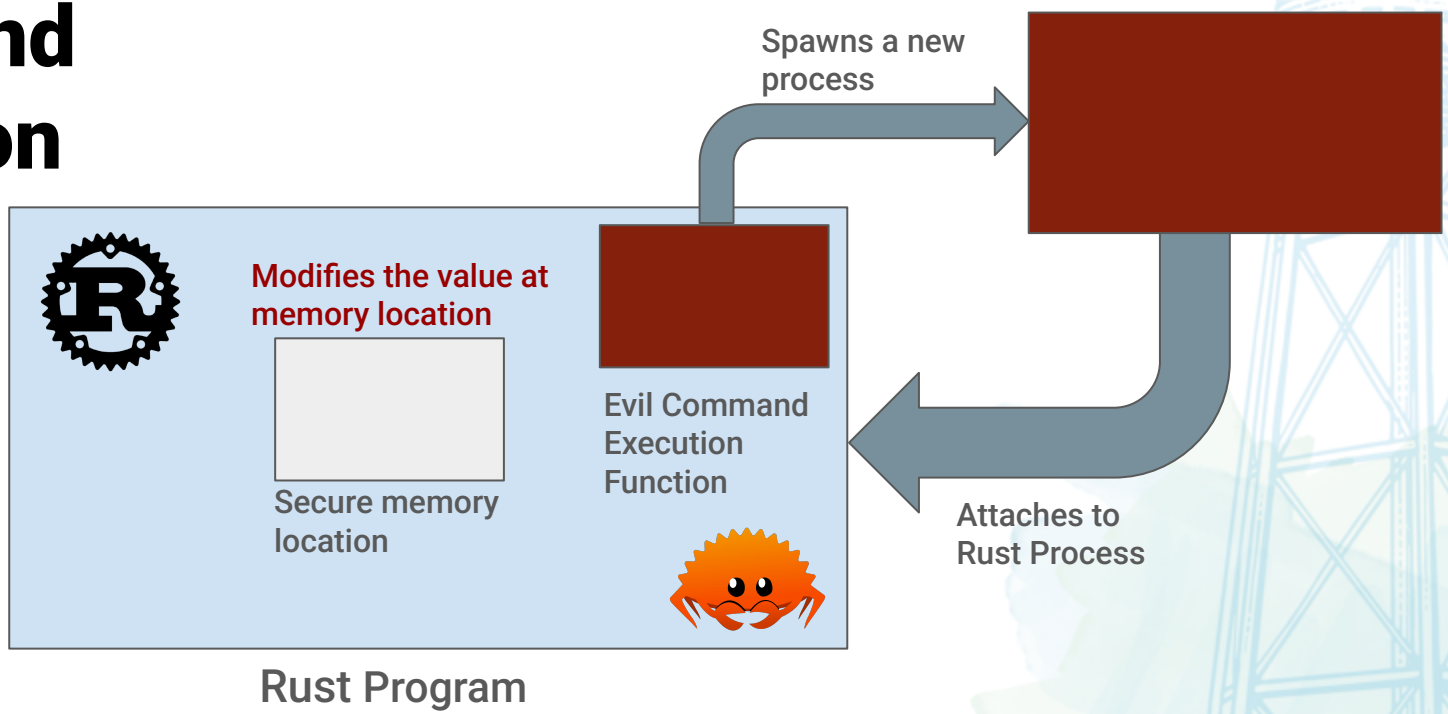
Command Execution



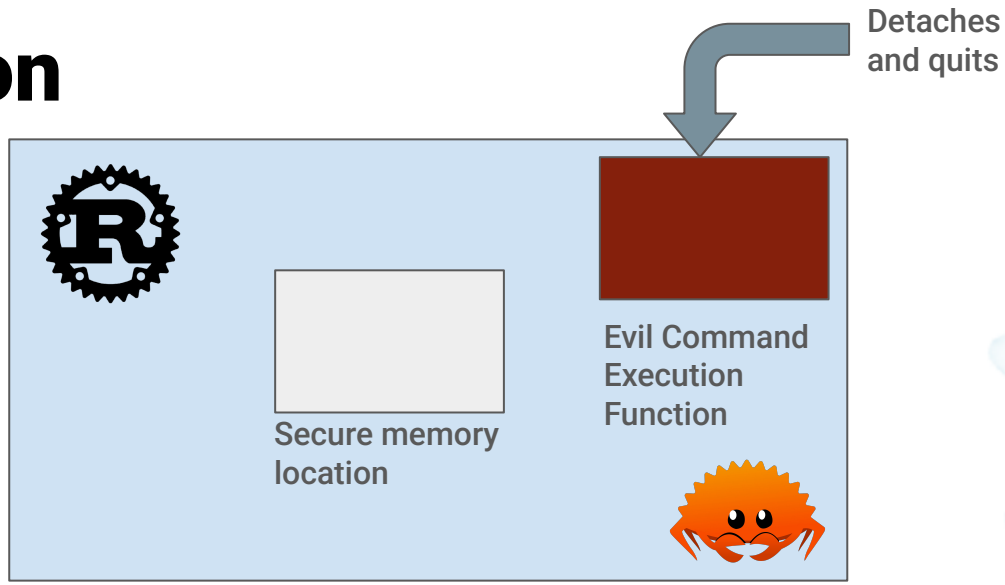
Command Execution



Command Execution



Command Execution



Command Execution

Requires sudo privileges.

```rust
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(())
}
```

Memory update part

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

Compiler Unsoundness

Compiler can wrongly assume certain code is safe!

Leads to bugs, crashes and even vulnerabilities!

Compiler Unsoundness

Compiler can wrongly assume certain code is safe!

Leads to bugs, crashes and even vulnerabilities!

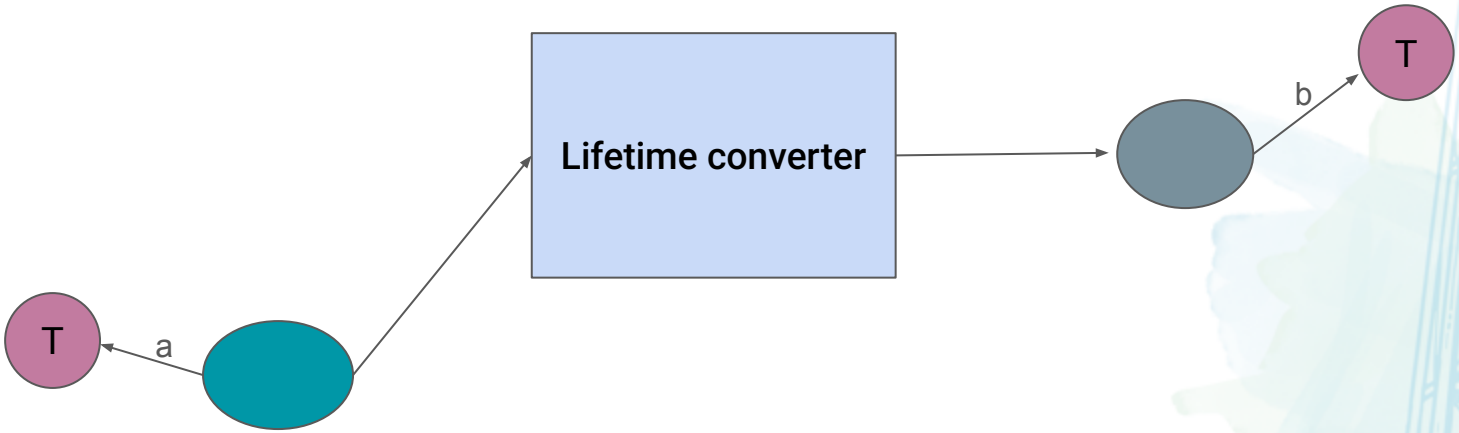
Lifetime : scope during which a reference is valid.

Compiler Unsoundness

Task : Given a reference with lifetime a can we return the reference with lifetime b?

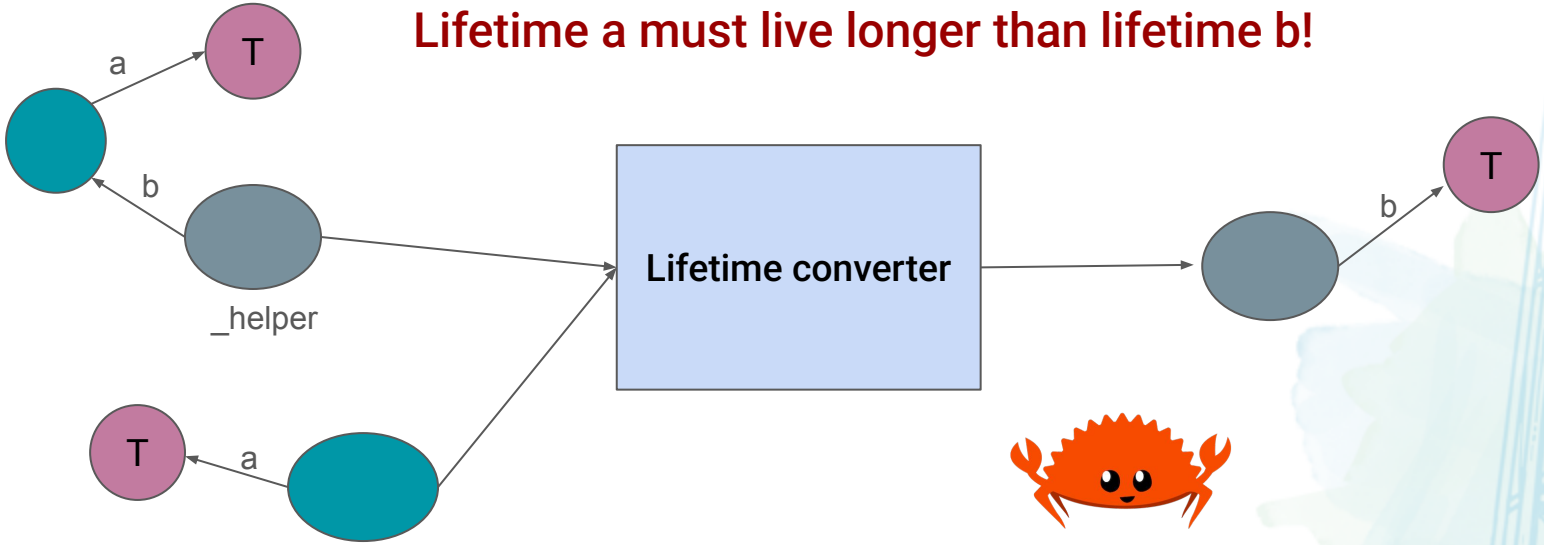
Compiler Unsoundness

Task : Given a reference with lifetime a can we return the reference with lifetime b?



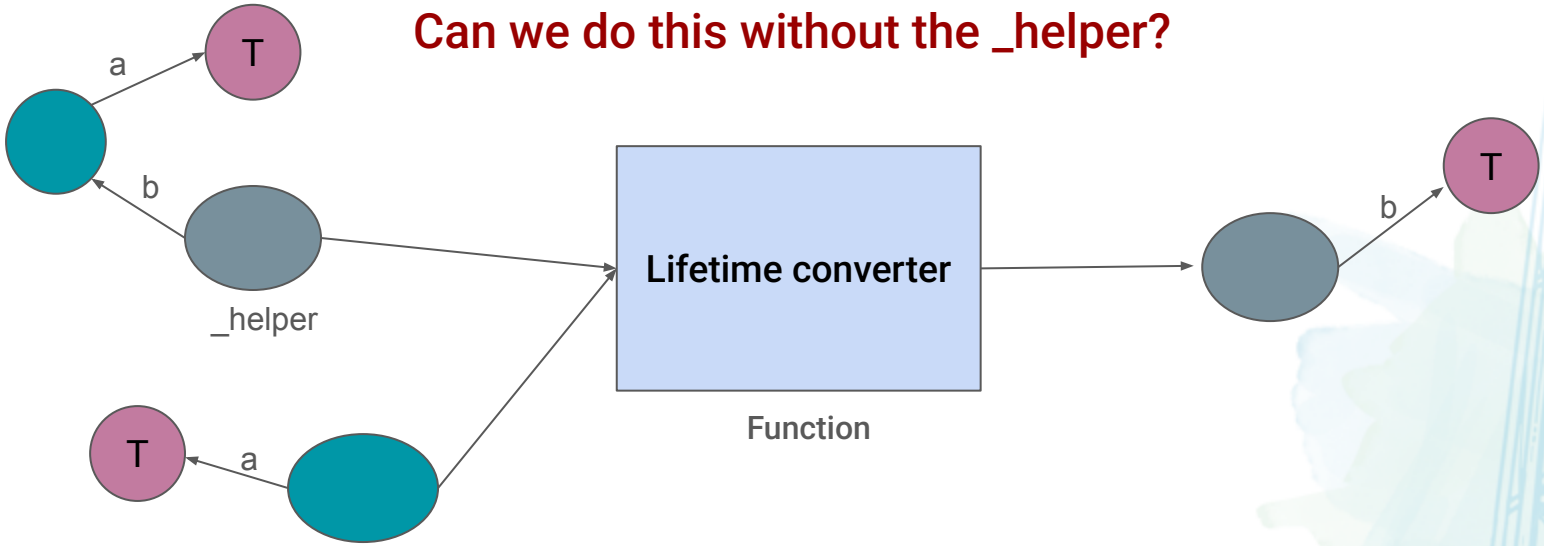
Compiler Unsoundness

Lifetime a must live longer than lifetime b!



Compiler Unsoundness

Can we do this without the _helper?

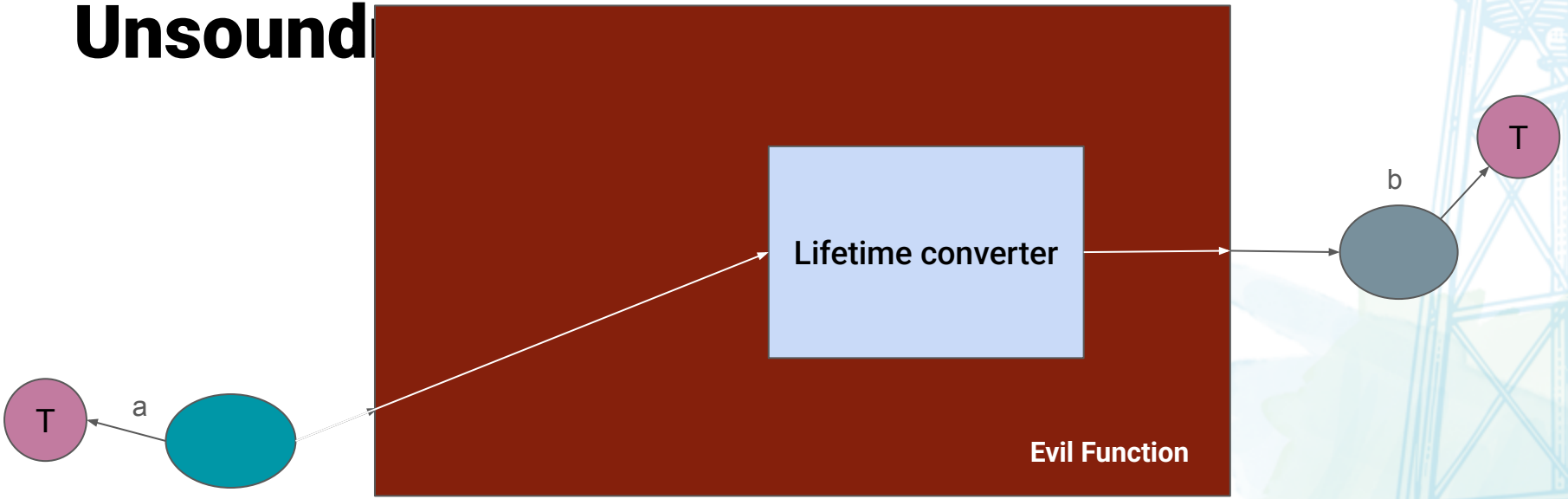


Compiler Unsoundi



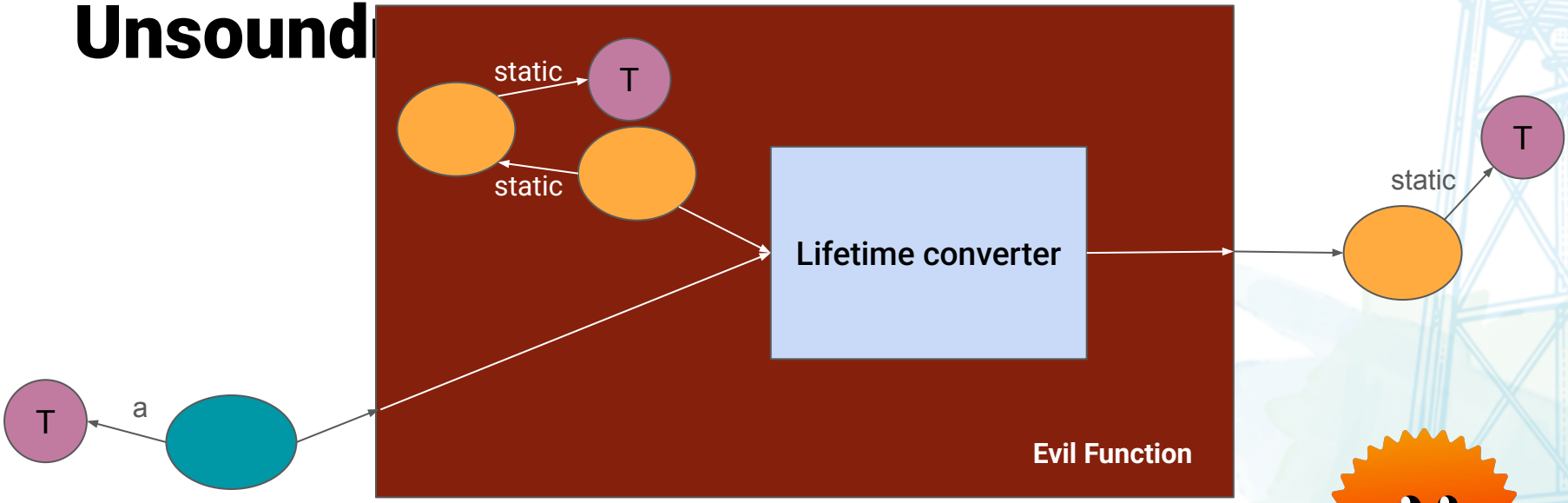
Credits : https://github.com/rust-lang/rust/issues/114728

Compiler Unsoundi



Credits : https://github.com/rust-lang/rust/issues/114728

Compiler Unsoundi



Credits : https://github.com/rust-lang/rust/issues/114728

Compiler Unsoundness

```rust
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.

Build Time Effects

Build time effects refer to actions or optimizations that occur during the process of compiling source code into an executable or library

Build Time Effects



User

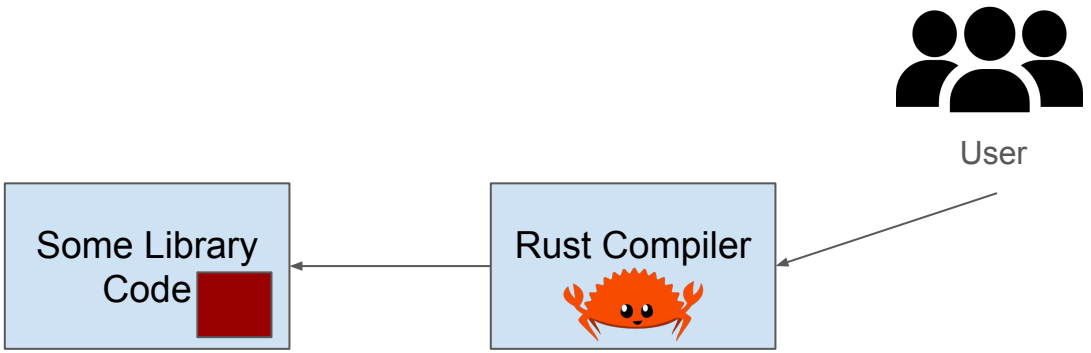
Build Time Effects



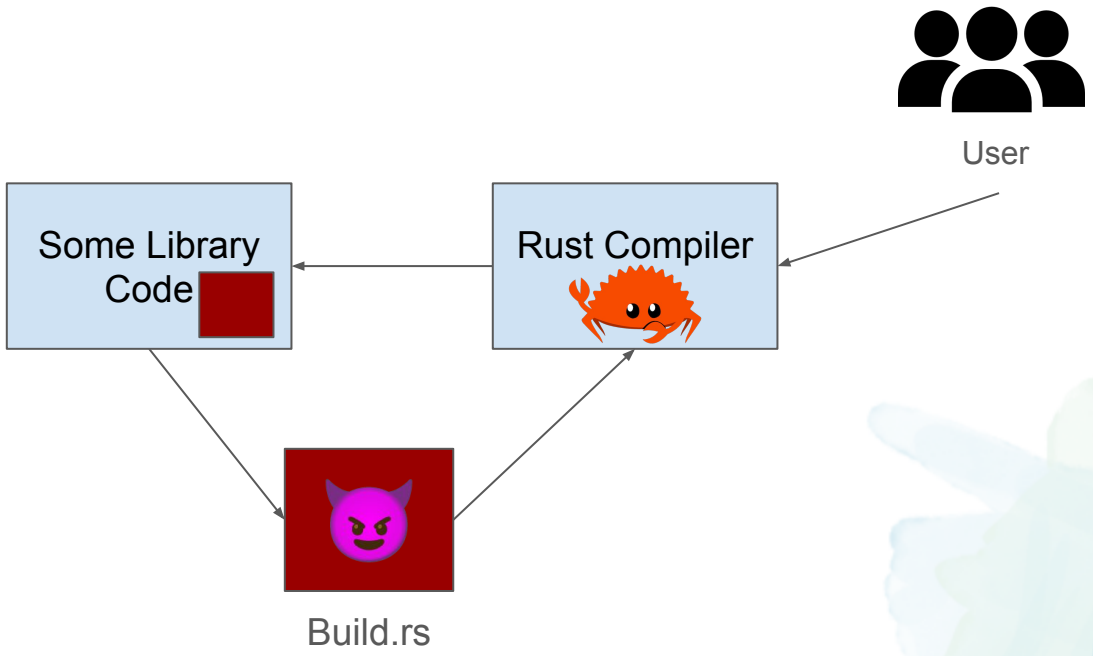
User



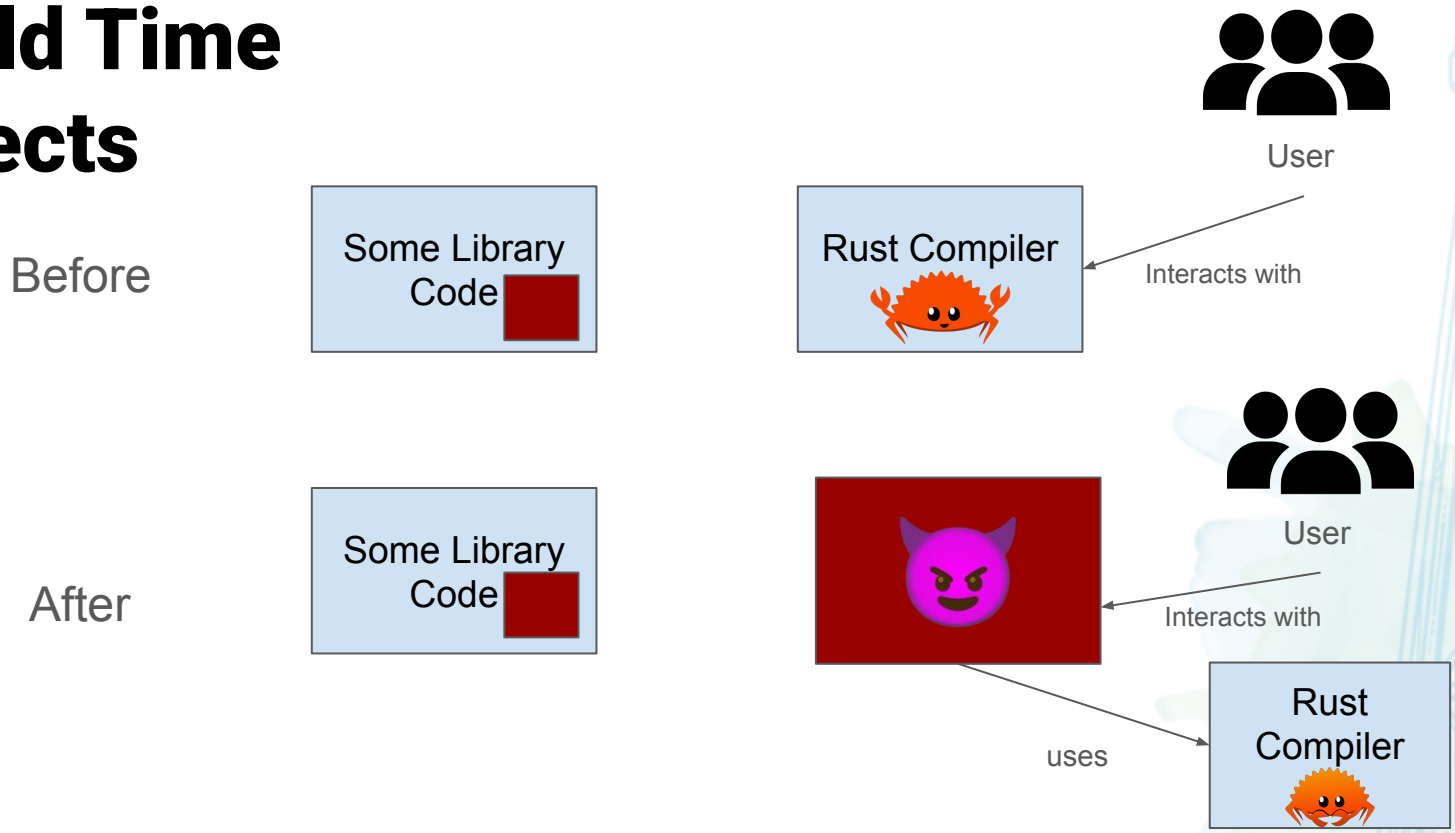
Build Time Effects



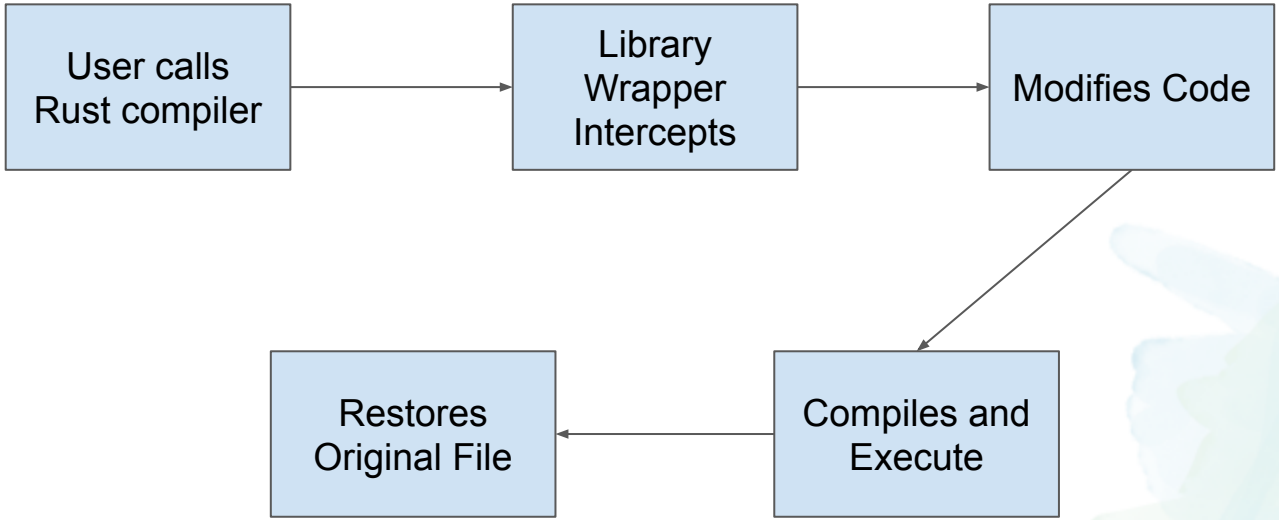
Build Time Effects



Build Time Effects



Build Time Effects



Environment Variables

Configurations can be modified to perform various malicious tasks.

Environment Variables

Configurations can be modified to perform various malicious tasks.

```
1  fn main() {
2      env::set_var("PATH", "/tmp/malicious_bin:".to_owned() +
       ↪  &env::var("PATH").unwrap());
3
4      // This will run the malicious `ls` if it exists in /tmp/malicious_bin
5      let output = Command::new("ls")
6      .output()
7      .expect("Failed to execute command");
8  }
```

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

Evaluation

We aim to answer these three questions:

- Q1 : Behaviour of Existing Tools
- Q2 : Prevalence of Attack Patterns in Real-World Code
- Q3 : Prevalence of Attack Patterns in Rust CVEs

Behaviour of Existing Tools

How to classify behaviour:

✓ → the tool successfully identifies 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 examples.

Behaviour of Existing Tools

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.

Prevalence in Real World Code

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.

Real World Code Example

```rust
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.

Prevalence in Rust CVEs

Prevalence in Rust CVEs

1. Grep-cli (RUSTSEC-2021-0071)

Prevalence in Rust CVEs

1. Grep-cli (RUSTSEC-2021-0071)
2. inconceivable (RUSTSEC-2022-0058)

Prevalence in Rust CVEs

1. Grep-cli (RUSTSEC-2021-0071)
2. inconceivable (RUSTSEC-2022-0058)
3. rustdecimal typosquatting (RUSTSEC-2022-0042)

Acknowledgements



Caleb Stanford
Assistant Professor, UC Davis
cdstanford@ucdavis.edu
https://web.cs.ucdavis.edu/~cdstanford/

Takeaway Message

Sometimes breaking things is the best way to keep them safe.

