Securing Mixed Rust with Hardware Capabilities

Jason Zhijingcheng Yu*
National University of Singapore
Singapore
yu.zhi@comp.nus.edu.sg

Fangqi Han*
National University of Singapore
Singapore
fangqi_han@u.nus.edu

Kaustab Choudhury
National University of Singapore
Singapore
kaustab.c@gmail.com

Trevor E. Carlson
National University of Singapore
Singapore
tcarlson@comp.nus.edu.sg

Prateek Saxena
National University of Singapore
Singapore
prateeks@comp.nus.edu.sg

Abstract

The Rust programming language enforces three basic *Rust principles*, namely ownership, borrowing, and AXM (Aliasing Xor Mutability) to prevent security bugs such as memory safety violations and data races. However, Rust projects often have *mixed code*, i.e., code that also uses unsafe Rust, FFI (Foreign Function Interfaces), and inline assembly for low-level control. The Rust compiler is unable to statically enforce Rust principles in mixed Rust code which can lead to many security vulnerabilities. In this paper, we propose CapsLock, a security enforcement mechanism that can run at the level of machine code and detect Rust principle violations at run-time in mixed code. CapsLock is kept simple enough to be implemented into recent capability-based hardware abstractions that provide low-cost spatial memory safety. CapsLock introduces a novel *revoke-on-use* abstraction for capability-based designs, wherein accessing a memory object via a capability *implicitly* invalidates certain other capabilities pointing to it, thereby also providing temporal memory safety automatically, without requiring software to explicitly specify such invalidation. Thus, CapsLock is the first mechanism capable of providing cross-language enforcement of Rust principles. We implemented a prototype of CapsLock on QEMU. Evaluation results show that CapsLock is highly compatible with existing Rust code (passing 99.7% of the built-in test cases of the 100 most popular crates) and flags Rust principle violations in real-world Rust projects that use FFI or inline assembly. We discovered 8 previously unknown bugs in such crates in our experiments.

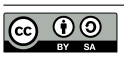
CCS Concepts

• **Security and privacy → Software and application security**;
• **Computer systems organization →** *Architectures.*

Keywords

Rust; Memory safety; Capability-based security

*Both authors contributed equally to this research.



CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744861

ACM Reference Format:
Jason Zhijingcheng Yu, Fangqi Han, Kaustab Choudhury, Trevor E. Carlson, and Prateek Saxena. 2025. Securing Mixed Rust with Hardware Capabilities. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744861

1 Introduction

The last decade has seen a rapid growth in popularity of the Rust programming language, driven by its promise of memory safety. High-profile projects such as Linux [17], Chromium [3], and Windows [10] have started to adopt Rust in their development. The security of Rust programs, however, comes with caveats.

Rust statically enforces a set of strict rules, which we call the *Rust principles*, that are designed to provide memory safety as well as thread safety in Rust code. However, these static checks restrict program expressiveness, so Rust programs often resort to implementing some functionality in *unsafe* Rust. Unlike safe Rust, in explicitly marked unsafe Rust code blocks, the compiler only *assumes*, instead of enforcing, the Rust principles. This reopens the floodgates to unsafety as unintentionally violating Rust principles in unsafe Rust is easy. As of this writing, for example, RustSec [1], a Rust security advisory database, has catalogued over 180 memory corruption vulnerabilities.

Moreover, Rust is a systems programming language where efficiency and compatibility are crucial. Rust programs often have to interact with software written in other languages. For example, unsafe Rust code often invokes external libraries written in C/C++ or contains inline assembly for low-level or performance-critical operations. Prior works report that 44.6% of unsafe functions are linked to foreign items [6]. Therefore, enforcing Rust principles in Rust code alone is insufficient.

In this work, our goal is to create a system that can detect run-time violations of Rust principles in *mixed Rust* code, which consists of both safe and unsafe code blocks, and potentially also includes use of FFI and inline assembly. Specifically, we want to extend enforcement of Rust principles *across language boundaries*. Such a system can be used to run tests on Rust programs to catch violations where they occur, much in the spirit of memory safety checkers commonly used in fuzzing or pre-deployment testing.

Dealing with cross-language enforcement is the key challenge. It is tempting to consider implementing run-time detectors for Rust principle violations in all language compilers of interest as a conceptual solution. However, this is a cumbersome approach: it needs

continuous update as language compilers evolve. Miri, a tool for run-time detection of Rust undefined behaviours [4], is capable of enforcing dynamic versions of the Rust principles [18, 35] and detecting their violations accurately. However, Miri relies on MIR (Mid-level IR), a Rust-specific high-level intermediate representation, which contains sufficient Rust semantics such as information about reference borrowing and ownership transfer. It is thus unable to enforce these principles in any language other than pure Rust. Consequently, it also does not work for mixed Rust programs.

Another option is to consider a weaker goal, namely that of employing a memory safety checker as a proxy for catching violations of Rust principles. For example, one can use a full memory safety checker like SoftBound+CETS [27, 28] or a partial safety checker like AddressSanitizer [30], to detect run-time memory errors. However, a detector for memory safety violation is *not* the same as a checker for Rust principles. We want to detect the violation of Rust principles when and where it occurs, not its after-effect that might manifest in a memory error somewhere later in the program execution. Violating Rust principles causes undefined behaviour in normal testing, which may or may not lead to memory errors. Such violations can also result in security-related logic errors, e.g., reading the wrong state in security-sensitive data in privileged checks, that are not necessarily a memory safety violation. In our experiments, Rust crate maintainers consider several of our reported Rust violations as serious enough to fix. We discuss the importance of detecting Rust principle violations as opposed to only memory errors in Section 2.2.

Our work. We answer the above research question by presenting CapsLock, a capability-based hardware abstraction that captures Rust principles at the *machine code* level in a language-agnostic manner, thus achieving cross-language Rust principle checking. Our key observation is that we can abstract out the core principle from the aliasing models underlying the Rust principles [18, 35] into a mechanism that we call *revoke-on-use*. In the revoke-on-use mechanism, an existing pointer can derive new ones through borrowing, similar to reference borrowing in Rust. Upon each use of a pointer for accessing memory, the revoke-on-use mechanism requires that the architecture implicitly revoke (i.e., invalidate) the pointers that conflict with the access, which is defined based on both the borrowing relationship and the permissions of the capabilities.

We show that the revoke-on-use mechanism is well-suited to be implemented as a hardware capability for memory access. Specifically, we show how to extend a prior capability-based design that supports flexible memory delegation and revocation [41].

We have implemented a prototype of the CapsLock architecture on QEMU [8] based on the 64-bit RISC-V architecture [36]. To enable bug detection in Rust programs, CapsLock requires instrumentation of the Rust code to perform borrowing based on the pointers or references involved in each operation. We modified the default heap memory allocator of Rust as well as the `rustc` compiler to inject such CapsLock-specific borrow instructions. All external code requires no special instrumentation.

Artefacts. We have publicly released the code and the data used in this work on Zenodo [42].

Experimental evaluation. To evaluate the practicality of CapsLock, we focus on its effectiveness in bug finding as compared to Miri [4], the official tool for detecting undefined behaviours in Rust programs. We perform tests with the 100 most popular crates on crates.io [2]. They include both safe Rust and mixed Rust code. First, CapsLock is fairly compatible with real-world Rust code. It is able to pass 99.7% of the default test cases that come with the crates. Second, by applying CapsLock to additional crates that Miri cannot handle due to unsupported FFI or inline assembly, we were able to discover 8 previously unknown bugs that violate Rust principles with their *normal built-in tests*. We reported them to the corresponding maintainers, and several of them have since been patched. The results show that despite being an abstraction of the Rust principles suitable for hardware-level implementations, the revoke-on-use mechanism is effective in finding bugs in mixed Rust code already with our QEMU implementation. Lastly, we compare CapsLock with AddressSanitizer [30], a popular and well-maintained partial memory safety checker, and ThreadSanitizer [31], a thread-safety sanitizer. In our experiments, CapsLock catches strictly more bugs than those alternative detectors, highlighting the advantage of checking for Rust principles over the other respective notions.

Our QEMU-based CapsLock implementation also achieves over twofold run-time performance improvement over Miri on average. We expect better performance in a hardware implementation, though we leave that as future work.

Contributions. We propose the revoke-on-use mechanism, a novel way to enforce Rust principles across languages at the level of machine code. It is implementable on recent hardware capability designs which already provide spatial safety, to further provide temporal safety and maintain Rust principles. Our QEMU-based CapsLock prototype yields a readily usable tool for detecting Rust principle violations in mixed Rust code.

2 Overview

Rust statically enforces three principles on pointer use that prevent security bugs including memory safety violations, data races, and unsafe aliasing. We review these key Rust principles with examples of security bugs and then describe our new CapsLock abstraction.

Motivating examples. Listing 1 shows three snippets (using the C programming language as an example) each with a different type of security bugs. Case (a) is a temporal memory safety violation, specifically a use-after-free. The code writes to `buf` after freeing the space, corrupting any data that resides in the reclaimed space. Case (b) is a data race. Two threads concurrently update the shared `buf` with one of them freeing it in the process. The outcome depends on how the two threads are scheduled. If freeing is scheduled before the other thread updates `buf`, a use-after-free occurs. Even when we ignore freeing, the resulting value of `buf[10]` is nondeterministic. Case (c) is an example of an unsafe aliasing leading to corruption of security-critical data. The function `memcpy` assumes the source and the destination regions to be non-overlapping, and when this is not the case, as in the example code, can produce corrupted data. This can lead to a wrong result at Line 5, which further causes privilege escalation with the `seteuid(0)` at Line 6.

2.1 Rust Principles

Rust enforces three principles to prevent such bugs: ownership, borrowing, and AXM (Aliasing Xor Mutability). The compiler statically

```c
char *buf = malloc(4096);
// ...
free(buf);
// ...
memcpy(buf, inp, 4096);
```

(a) Memory safety violation (use-after-free)

```c
void consumer(void *buf) {
  // ...
  ((int*)buf)[10] = 1;
  // ...
}
int main(void) {
  int *buf = malloc(sizeof(int)
  * 16);
  // ...
  pthread_create(&tc, NULL,
  &consumer, buf);
  // ...
  buf[10] = 0;
  free(buf);
}
```

(b) Data race causing nondeterministic states and potential use-after-free

```c
char buf1[16];
// ...
char *buf2 = buf1 + 2;
memcpy(buf2, buf1, 8);
if (memcmp(buf2, key, 8) == 0)
  seteuid(0);
```

(c) Unsafe aliasing causing security-critical data corruption

Listing 1: Three types of security bugs in C code.

```rust
let mut buf = vec![0u8; 4096];
// ...
drop(buf);
// ...
buf.copy_from_slice(&inp);
```

(a)

```rust
fn consumer(buf : &mut [i32]) {
  // ...
  buf[10] = 1;
  // ...
}
fn main() {
  let mut buf = vec![0i32; 16];
  // ...
  std::thread::spawn(|| {
    consumer(&mut buf[..]);
  });
  // ...
  buf[10] = 0;
}
```

(b)

```rust
let mut buf1 = [0u8; 16];
// ...
let buf2 = &mut buf1[2..10];
buf2.copy_from_slice(
  &buf1[..8]
);
if *buf2 == *key {
  seteuid(0);
}
```

(c)

Listing 2: Rust statically rejecting those programs for violating Rust principles, preventing the security bugs in Listing 1.

checks that the safe Rust code complies with them. The examples in Listing 2, which are Rust versions of the buggy snippets in Listing 1, fail to compile for violating Rust principles.

The *ownership principle* in Rust associates a unique owner reference with each memory object. A memory object is freed and becomes unusable after its owner is dropped (typically, when it goes out of scope), which prevents temporal safety errors. Case (a) violates this principle: `buf` is the owner of the buffer, and is dropped at Line 3. The compiler thus rejects the use of `buf` at Line 5.

Rust relaxes the exclusive ownership principle through a restricted form of aliasing called borrowing. A borrow is a new aliased reference created from an existing reference. In effect, it suspends the reference being borrowed and temporarily allows using the new reference in its place. Rust enforces the *borrowing principle:* a reference cannot outlive the reference it borrows from. In case (a), due to the ownership principle, the owner reference cannot access the object at Line 5. The borrowing principle further ensures that all other aliasing references cannot access it after Line 3. Those two principles, along with compile-time or run-time bounds checking, are sufficient for full memory safety.

The third key principle is *AXM*, short for *Aliasing Xor Mutability*. It enables aggressive performance optimization when combined with the first two principles, in addition to preventing other mistakes due to data races and unsafe aliasing. The AXM principle states that at any given point in the program, each object can have either a singe active (i.e., not borrowed) mutable reference or multiple immutable references, but never both simultaneously. Mutable references are those which allow mutating the object, while immutable references are those that do not. Case (c) shows a violation of the AXM principle. Lines 3–5 create a mutable reference `buf[2..10]` and an immutable reference `buf[..8]`, both borrowing from `buf` and aliased with each other. Rust rejects this behaviour and thus avoids the data corruption resulting from aliasing references in `copy_from_slice`.

The AXM principle enforced across threads can also prevent unintended consequences of data races, as illustrated in Case (b). Lines 9–11 attempt to pass a reference that mutably borrows `buf` to a new thread. Since we cannot statically determine when `consumer` will complete, this borrow needs to be sustained until the end of the whole program. However, this cannot be satisfied, because by the borrowing and AXM principles, the use of `buf` in `main` requires that borrowed reference to die before Line 13. Rust prevents the data race and its potential resulting issues by rejecting this case.

2.2 Exploitability of Rust Principle Violations

Enforcing Rust principles statically is too restrictive for some benign scenarios. As such, Rust allows *unsafe Rust* code blocks for which Rust principle enforcement is disabled. For example, unsafe Rust allows accessing memory through raw pointers (`*const T` and `*mut T`), for which none of the three basic principles are enforced, as opposed to references (`&T` and `&mut T`). Programmers use unsafe Rust extensively [6, 43]. Many data structures and smart pointers in the Rust standard library use unsafe Rust under the hood.

Beyond enabling raw pointers which are not subject to the static checks on Rust principles, unsafe Rust allows interfacing with non-Rust software components through FFI (foreign function interface) calls and inline assembly. Program behaviours behind FFI and inline assembly are beyond the control of the Rust compiler and are also excluded from static checks. Such external code written in other programming languages can also easily violate Rust principles. As a programming language geared towards systems programming with low-level control and low overhead, the use of FFI and inline assembly is widespread in Rust. Prior work has estimated that 44.6% of unsafe functions in Rust projects are linked to foreign items [6].

The Rust compiler is unable to enforce the Rust principles in *mixed Rust code* (i.e., safe Rust code mixed with unsafe Rust code) Instead, those principles become *assumptions* the Rust compiler makes. Violating such assumptions can manifest into memory errors. In particular, the Rust compiler's built-in static analyses and performance optimizations assume—without checking—that unsafe Rust code preserves the Rust principles. When the code fails to uphold these assumptions, the resulting code can have undefined behaviour, causing memory safety or thread safety issues as well

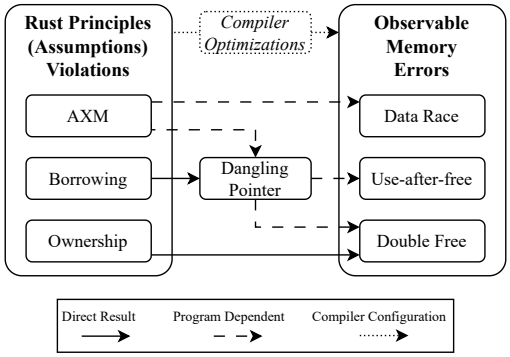


Figure 1: A directed graph illustrating how Rust principle violations manifest into memory errors.

```
1  fn free_xor_alias(raw: *mut u8, obj: Box<u8>) {
2      assert_eq!(*obj, 0);
3      unsafe { *raw = 42; }
4      if *obj != 42 {
5          unsafe { libc::free(raw as *mut libc::c_void); }
6      }
7  }
8
9  fn main() {
10     let mut obj = Box::new(0);
11     free_xor_alias(std::ptr::addr_of_mut!(*obj.as_mut()), obj);
12 }
```

Listing 3: An example Rust program that incurs a logically impossible double free error after compiler optimizations.

as incorrect execution results. As illustrated in Figure 1, under different compiler configurations and program implementations, at least two paths can lead to memory safety violations. One path may be taken even if the program implementation does not appear exploitable, while the other path may be taken even if the compiler disables all optimizations. Therefore, it is always desirable to detect and eradicate Rust principle violations.

Listing 3 shows a Rust program where the AXM violation results in a double free error only if compiler optimizations are enabled. The compiler assumes that, according to AXM, `raw` and `obj` in `free_xor_alias` are not aliased and, therefore, `obj` has been unmodified since the assertion at Line 2. This results in the optimized program evaluating the condition at Line 4 as always `true`, causing a double free error. The example demonstrates the unstable nature of assumption violations. It is thus insufficient to test binaries only for particular compilations. In practice, programmers must ensure that compiler assumptions are always satisfied.

Note that the dashed lines (i.e., program-dependent paths) in Figure 1 are also non-trivial. Section 3.6 further explains how they complicate identifying bugs related to assumption violations, especially with traditional error-detecting tools.

2.3 The CapsLock Abstraction

In this paper, we aim to detect Rust principle violations in mixed Rust code. We propose CapsLock, a new security enforcement mechanism that can be implemented at the level of machine code. CapsLock provides a *hardware capability-based abstraction* [22] which detects Rust principle violations. A capability-based abstraction replaces raw pointers with *capabilities*. Besides the address of a memory location, a capability carries metadata such as memory bounds and permissions that restrict the access that software can make with it. The hardware distinguishes capabilities from raw data and tracks their creation and use.

We choose a capability-based hardware design as the basis of our solution for two reasons. Firstly, such a design is language-agnostic as hardware capabilities work at the machine code level, which all software in a system shares. Inline assembly, in particular, directly exposes this level to the programmer. Capabilities contain extra metadata necessary to enforce policies regarding pointer use. In existing capability-based designs, this metadata includes memory bounds and permissions, which are intended for catching spatial memory safety violations directly. CapsLock extends the capability metadata to capture extra information needed for Rust principles. Secondly, this design allows CapsLock to enable per-program full memory safety while retaining existing benefits offered by the underlying capability-based hardware, such as supporting isolation or secure data sharing which are system-wide properties.

Existing capability-based architectures expose capability-specific instructions and require explicit management of capabilities by the software. This requires extensive changes in software across all programming languages, including assembly code. Our goal requires us to avoid imposing such requirements. We propose a novel *revoke-on-use* mechanism in CapsLock to *implicitly* maintain capabilities and check for potential violations of Rust principles. In revoke-on-use, when software uses a capability for a memory access, the hardware invalidates capabilities whose use would conflict with that memory access later. The notion of *conflicting* capabilities, which we will discuss in greater detail in Sections 3 and 4, is general and simple enough to be followed at the architectural level while providing a useful approximation of Rust principles.

3 CapsLock: High-level Design

CapsLock is based on hardware capabilities. In a capability-based hardware architecture, a capability typically consists of a memory address to access (i.e., the cursor), permitted memory bounds, access permissions, and other metadata such as the capability type. Capabilities can reside in general-purpose registers and memory, but they are distinct from normal data and unforgeable. The hardware maintains whether each register or aligned memory location contains a capability or normal data, and forbids software from converting normal data directly into a capability. Rather, software creates and manipulates capabilities through dedicated instructions. Capability-based architectures require software to use explicitly specified capabilities for memory accesses, and perform bounds and permission checks for each access based on the specified capability. Such checks provide spatial memory safety.

3.1 Our Baseline: Capstone

We choose an existing capability-based architecture design called Capstone [41] as our baseline. The reason for this choice is the additional expressiveness of Capstone compared to that of other

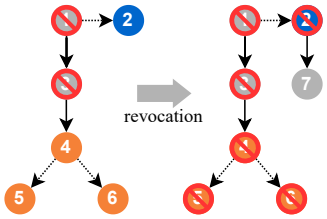


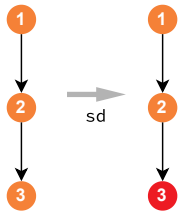
Figure 2: Overview of the revocation mechanism in Capstone (baseline). Each circle represents a capability, with the number indicating the order in which they are created. The colour indicates the capability type: linear (gray), non-linear (orange), and revocation (blue). Solid arrows represent capability derivations that destroy the original capabilities, and dashed arrows represent derivations that retain them.

designs. At the core of such expressiveness are linear capabilities and revocation capabilities, two special capability types in Capstone. Linear capabilities are not duplicable. Capstone guarantees that regions pointed to by linear capabilities do not overlap with other capabilities in the system. In tandem with linear capabilities, revocation capabilities enable a selective revocation mechanism, as shown in Figure 2. Software can pass out a linear capability as well as delinearize it to disable the non-aliasing constraint, but can create a corresponding revocation capability beforehand to allow itself to recover the same linear capability while revoking all capabilities derived from it. This is a default fail-close protection against accidental *capability leaks* that affect classical capability designs [22] and their modern incarnations [37]. It also provides temporal memory safety: the memory allocator creates a revocation capability for each newly allocated object before passing the corresponding capability to the user program and revokes with the revocation capability when the object is freed. Beyond memory safety, the same capability-based mechanism in Capstone also supports isolation use cases, e.g., memory delegation between mutually-distrusting parties.

Capstone falls short. Despite the expressiveness of Capstone, it is not capable of achieving our goal of detecting Rust principle violations. We explain this with the example in Figure 3 (a). Line 13 dereferences a raw pointer `v_raw` to create a mutable reference `v_ref`. Line 14 then passes `v_raw` as an argument of a function call to `use_p()`. After the call returns, Line 15 writes to memory through `v_ref`. The code violates the AXM principle because during the lifetime of `v_ref`, the inline assembly at Lines 4-7 inside `use_p()` mutates its memory through `v_raw`. Detecting this violation requires 1) identifying the derivation relationship between `v_ref` (and `p` in `use_p()`) and `v_raw`, 2) identifying the write operation through `v_raw`, and 3) invalidating `v_ref` thereafter.

In Capstone, to achieve the invalidation of `v_ref` when `v_raw` is used, it is necessary to create a revocation capability for `v_raw` when it is dereferenced at Line 13, and then perform revocation on the revocation capability before Line 5. Both operations require injecting extra instructions, and looking at `use_p()` alone is not sufficient to statically know if `p` corresponds to a revocation capability that needs revocation. Such information may only be determined during run-time. To handle all possibilities, therefore, it is necessary

```
1  use std::arch::asm;
2  fn use_p(p 2 : *mut u64) {
3    unsafe {
4      asm!(
5        "sd x0, 0({addr})",
6        addr = in(reg) p 2
7      );
8    }
9  }
10 fn main() {
11   let mut v 1 = Box::new(0u64);
12   let v_raw 2 = &mut *v as *mut u64;
13   let v_ref 3 = unsafe { &mut *v_raw 2 };
14   use_p(v_raw 2);
15   *v_ref 3 = 42;
16 }
```



(a) (b)

Figure 3: Example of CapsLock detecting a Rust principle violation. Circles represent capabilities. Those in red are invalid. (a) Code with the capability associated with each value indicated. (b) CapsLock maintains capability derivations in trees and invalidates capabilities based on them.

to either check the run-time state (e.g., the capability type) to determine if revocation is necessary before every memory access, or conservatively maintain that a pointer always has a corresponding revocation capability and perform revocation before all memory accesses. Both options are costly and require extensive instrumentation to code in all languages. The distinction between linear and non-linear capabilities in Capstone further complicates the picture, as linear capabilities cannot be duplicated and non-linear capabilities cannot be used to create revocation capabilities.

3.2 Key Idea: Revoke-on-Use

CapsLock simplifies the Capstone model based on a core idea which we call *revoke-on-use*. It eschews extensive software changes while keeping a distilled core which Rust principles and Capstone share.

Observations. Revoke-on-use is based on two observations. Firstly, the act of using a capability to access memory already carries the intention of a program to perform any necessary capability invalidation to allow the access. Instead of requiring explicit handling of revocation capabilities, therefore, we can simply perform revocation implicitly when the software uses a capability to access memory. We thus remove the distinct revocation capability type. Instead, we allow the borrowing operation to build the derivation relationship between capabilities. Secondly, enforcing non-duplicability of linear capabilities is not necessary for guaranteeing exclusive access. A freshly borrowed capability is always unique and the implicit revocation performed at each memory access ensures that while the capability is live, no access to the memory region takes place through other capabilities. Therefore, we also remove the distinction between linear and non-linear capabilities.

Revoke-on-use in CapsLock. With revoke-on-use, CapsLock provides only one capability type. CapsLock maintains the derivation relationships among capabilities as trees. Based on the tree

structures and capability metadata, when software performs memory access using a capability, CapsLock invalidates other capabilities that overlap the accessed memory region and have conflicting permissions. In effect, revoke-on-use ties the intention of capability revocation to concrete memory access. This design choice avoids the need to statically deduce how a pointer will be used. It also avoids extra instructions involving revocation capabilities.

Example revisited. We revisit the example in Figure 3 (a) with revoke-on-use. During run-time, CapsLock associates values in the program with capabilities and maintains their derivation relationship as shown in Figure 3 (b). Capability 1 is created when the software allocates v, and two subsequent capabilities are derived through borrowing when the software converts the reference into a raw pointer at Line 12 and then dereferences it at Line 13. Capability 3 borrows from Capability 2, which in turn borrows from Capability 1. Capability 2 propagates along with the function call argument and into the base address register used in the inline assembly at Line 5. When the software performs the store operation (sd), CapsLock implicitly invalidates Capability 3. This causes Capability 3 to fail the check when it is used at Line 15, and CapsLock reports a Rust principle violation. For CapsLock to work with this example, the only extra information needed is for the capability creation and borrowing operations in the Rust code. Other parts of the code, e.g., for memory access or capability passing, require no change. This applies to both Rust code and code in other languages, including inline assembly (e.g., Line 5).

3.3 Operations on Revoke-on-Use Capabilities

We overview the operations that software can perform on a revoke-on-use capability in CapsLock.

Copy. CapsLock allows all capabilities to be copied. When a capability is copied, e.g., through a mv instruction, CapsLock treats both copies as identical and allows them to be used interchangeably. This avoids the problems with using linear capabilities in Capstone [41], as discussed in Section 3.1.

Borrow. CapsLock allows creating a new capability through *borrowing* from an existing one. We design such a borrowing operation to correspond to borrowing in Rust. Unlike copying, the new capability is not treated as identical to the original even though it points to the same memory location. In effect, borrowing temporarily transfers exclusive access from the original capability to the new one. The borrowing relationship between capabilities forms a *borrow tree* for each allocation of memory object. The root of a borrow tree is the initial capability created through allocation, and each edge connects a capability to another that borrows from it. A capability produced through borrowing can have its memory bounds specified as a subset of the bounds of its parent (i.e., the capability it borrows from). This is to support *partial borrowing*, where non-overlapping subparts of a capability are borrowed by multiple different capabilities and accessed through them independently. Borrowing covers conversions between Rust references and raw pointers. For example, a capability corresponding to a reference can be created through borrowing from a capability corresponding to a raw pointer. This reflects dereferencing a raw pointer to obtain a reference in Rust. Effectively, a leaf capability corresponds to a reference or pointer considered as currently active in Rust.

Creation. Newly allocated objects on heap or stack are associated with capabilities. A full capability-based system resets to an initial state with capabilities that cover all available memory. Memory allocators then derive capabilities for individual objects with tighter bounds or permissions from those initial capabilities. This allows capabilities to confine the set of accesses a software component can make to memory, which is important for isolation. Since we focus on the safety of single programs rather than isolation in this work, for simplicity, we expose a direct Create operation to software which creates a new capability with specified bounds and permissions. The new capability is the root of a new borrow tree.

Revocation. Software can perform *revocation*, i.e., invalidate a specified capability and all capabilities derived from it (in its borrow subtree). This is useful when software frees a memory object.

Load/Store. Every time software performs a memory load/store, CapsLock checks if the capability provided for addressing is valid. CapsLock also invalidates conflicting capabilities based on revoke-on-use for the allowed memory accesses (see Section 3.2).

3.4 Security Invariants

CapsLock guarantees two simple invariants at run-time. Such invariants concern the properties of capabilities throughout their lifetimes (i.e., while they are valid), and can be understood to reflect an approximation of Rust principles but at the machine code level. They define when capabilities are conflicting and require invalidation: upon a load/store operation, two capabilities are conflicting if retaining both leads to violations of such invariants.

Well-nested borrowing. A valid capability either is the root of its borrow tree or has a parent that is also valid. This invariant ensures that a capability cannot outlive the capability that it borrows from, which corresponds to the borrowing principle.

Exclusive access. Consider a capability c. We use $D(c)$ to denote the set of all capabilities derived from c through borrowing (i.e., inside the borrow subtree of c), including c itself. The exclusive access invariant states that during the lifetime of c, before any memory access to a memory location through a capability in $D(c)$, there is no store to an aliasing memory location from outside $D(c)$, and before any store access to a memory location through a capability in $D(c)$, there is no load from an aliasing memory location from outside $D(c)$. This invariant captures the AXM principle: it guarantees the holder of a capability *exclusive access* to memory.

3.5 Difference from Rust Aliasing Models

The Rust compiler statically enforces Rust principles only for safe Rust code. Two *aliasing models* of Rust, Stacked Borrows and Tree Borrows, extend the notion of Rust principles to run-time behaviours of unsafe Rust code. Stacked Borrows [18] tracks references and pointers for each memory location with tags on a stack. Each tag has a type that carries the access permissions and aliasing requirements associated with the pointers. Creating a new reference or pointer through borrowing constitutes creating a new tag and pushing it onto the stack, and each use of a reference or pointer checks the presence of its associated tag on the stack, its permissions, and, in some cases, pops the tags on top of it off the stack. Tree Borrows [35] relaxes Stacked Borrows to remove several undefined

Table 1: Error visibility of collected AXM violations among RustSec entries. Assessments of whether a concrete error is possible are supported by entry descriptions on RustSec.

RUSTSEC-	Type	Error Possible/in PoC	CVSS Score
2020-0023	Vulnerability	✓/ ✗	9.8, Critical
2021-0031	Vulnerability	✓/ ✗	9.8, Critical
2021-0114	Vulnerability	✓/ ✗	N/A
2022-0007	Unsound	✓/ UAF	N/A
2022-0040	Vulnerability	✓/ ✗	N/A

behaviours. Unlike Stacked Borrows, it maintains references and pointers in a tree, which means it allows but distinguishes multiple references borrowed from the same source. For example, it allows a mutable borrow to co-exist with an aliasing immutable borrow before the first mutation through it.

Revoke-on-use in CapsLock is generally more relaxed compared to Stacked Borrows or Tree Borrows. One notable difference, for example, is that both Stacked Borrows and Tree Borrows invalidate pointers upon borrowing, while CapsLock does so more lazily, only when capabilities are used for accessing memory. Another difference is that CapsLock allows all mutable references to be reserved and treated as immutable references until the first mutable access through them, whereas Rust disallows this in general and only allows it in specific scenarios to support the two-phase borrow pattern. CapsLock also does not enforce protectors for function arguments, which both Stacked Borrows and Tree Borrows include. While those differences mean that CapsLock is unable to detect all cases of Rust principle violations, we have chosen such a design to keep revoke-on-use capabilities simple and general enough to be useful for other use cases of capabilities rather than specific to Rust that can be implemented generically at machine code level. CapsLock shows that simple adjustments to Capstone allows it to detect some Rust principle violations without losing the general usefulness of hardware capabilities.

3.6 Comparison with Existing Tools

Existing techniques for run-time bug detection fall in two main categories: techniques that focus on other notions of safety rather than Rust principles, and techniques that focus on Rust principles. Techniques in both categories are insufficient for our research goal. We discuss their limitations from the perspective of their designs below. Sections 6.2 and 6.3 show such limitations empirically.

Gap between Rust principles/revoke-on-use and memory safety. Section 2.2 has discussed how violations of Rust principles sometimes, but not always, cause violations of memory safety. The exact outcome depends on various factors such as the specific compiler implementation and optimizations enabled. Furthermore, while memory risks always exist upon a violation, they sometimes require specific program implementations to generate an observable effect. This significantly complicates program testing and undermines the language's safety guarantees.

Recall from Figure 1 that, among the three categories of assumption violations, AXM violations exhibit the most paths to various memory errors that are simultaneously the hardest to detect. Table 1 summarizes the error visibility of our collected RustSec entries

$$
\begin{array}{rl}
c ::= (a, a, p, b) & \text{capabilities} \\
p ::= \text{RW} \mid \text{RO} \mid \text{NA} & \text{permissions} \\
b ::= \perp \mid i & \text{parents} \\
m ::= \emptyset \mid m, i \mapsto c & \text{capability maps} \\
a \quad \text{physical addresses} & i \quad \text{capabaility identifiers}
\end{array}
$$

Figure 4: CapsLock capabilities and related constructs.

that violate AXM (Section 6.2 explains how they were collected). Although all 5 entries can lead to memory errors, AddressSanitizer detected only one PoC program. The 4 entries categorized as vulnerabilities, including 2 with CVSS scores of 9.8 (Critical), are undetected because their PoC implementations have not yet elevated the bug to an observable level. Such vulnerabilities are thus undetectable by memory error detectors even on manually crafted PoC programs. Attempting to construct an automated approach with memory error detection tools that cover such cases requires a more exhaustive search scheme.

Consequently, existing mechanisms designed to enforce memory safety are fundamentally incapable of reliably detecting Rust principle violations. These mechanisms include software-based bug detection methods such as SoftBound+CETS [27, 28] and AddressSanitizer [30]. As Section 6.2 will show, this gap prevents them from detecting many security vulnerabilities in Rust code.

Limitations of existing run-time borrow checking techniques. Dedicated tools model the root cause of the risks (i.e., assumption violation) instead of specific symptoms. Since such methods focus directly on the language specification, they are capable of providing better soundness compared to memory error detectors, covering all legal compiler implementations and program executions.

Miri [4] is a popular undefined behaviour detection tool maintained by the official Rust team. It provides run-time detection of Rust principle violations, not just spatial memory safety violations and data races [1]. Miri achieves this through a run-time borrow checker following either Stacked Borrows [18] or Tree Borrows [35], selectable through a flag. However, to enforce an aliasing model, Miri requires run-time object-level information (i.e., which object a reference or pointer points to), Rust-specific type information such as whether a reference is mutable, as well as pointer provenance (i.e., how each pointer was created). Such information is available in MIR, a high-level IR which Miri interprets, but missing at the machine code level. Therefore, Miri is unable to support FFI or inline assembly. When it encounters an FFI call or an inline assembly block, it reports an unsupported operation and aborts the run. We will show in Section 6.3 that this limitation causes Miri to miss Rust principle violations in real-world code.

4 CapsLock: Design Details

4.1 Capabilities

Figure 4 defines CapsLock capabilities and related constructs. A capability c contains a pair of physical addresses encoding the starting

[1] Beyond Rust principle violations, Miri is also capable of detecting other types of undefined behaviours, e.g., data type confusion. We focus on Rust principles only in this work as they are the most special characteristics of Rust.

```
1  use_p:
2   sd       x0, 0(a0)
3   ret
4  main:
5   # ...
6   li       a1, 8
7   mv       a0, a1
8   call     exchange_malloc
9   create   a0, a0, 8
10  sd       x0, 0(a0)
11  sd       a0, 24(sp)
12  borroww  a0, a0
13  sd       a0, 16(sp)
14  borroww  a0, a0
15  sd       a0, 8(sp)
16  call     use_p
17  ld       a1, 8(sp)
18  li       a0, 42
19  sd       a0, 0(a1)
20  # ...
```

Listing 4: Relevant 64-bit RISC-V binary code corresponding to the example in Figure 3 instrumented with CapsLock instructions. The code is simplified to serve an illustrative purpose and does not exactly match our implementation. Instructions in orange are injected, and instructions in blue are extended with CapsLock load/store operations.

Table 2: CapsLock instruction semantics.

Instruction	Rule
Create (a_l, a_r)	$a_l < a_r \quad i \notin \mathrm{dom}(m)$ $m' = m[i \mapsto (a_l, a_r, \mathrm{RW}, \bot)]$
BorrowR (i, a'_l, a'_r)	$m(i) = (a_l, a_r, p, b) \quad p \neq \mathrm{NA} \quad i' \notin \mathrm{dom}(m)$ $a_l \leq a'_l < a'_r \leq a_r \quad m' = m[i' \mapsto (a'_l, a'_r, \mathrm{RO}, i)]$
BorrowW (i, a'_l, a'_r)	$m(i) = (a_l, a_r, p, b) \quad p = \mathrm{RW} \quad i' \notin \mathrm{dom}(m)$ $a_l \leq a'_l < a'_r \leq a_r \quad m' = m[i' \mapsto (a'_l, a'_r, \mathrm{RW}, i)]$
Drop (i)	$m(i) = (a_l, a_r, p, b) \quad m' = \mathrm{RevokeW}(m, S)$ $p \neq \mathrm{NA} \quad S = D_m(R_m(i)) \cup \{R_m(i)\}$
Load (i, a)	$m(i) = (a_l, a_r, p, b) \quad a_l \leq a < a_r \quad p \neq \mathrm{NA}$ $m' = \mathrm{RevokeR}\,(m, D_m(S)) \quad S = O_m(a) \cap \overline{A_m(i)}$
Store (i, a)	$m(i) = (a_l, a_r, p, b) \quad a_l \leq a < a_r \quad p = \mathrm{RW}$ $m' = \mathrm{RevokeW}\,(m, D_m(S)) \quad S = O_m(a) \cap \overline{A_m(i)}$

and ending addresses of the permitted access region. The permission p encodes the permitted access operations on the capability, which can be read-write (RW), read-only (RO), or no-access/invalid (NA). The parent b is either null ($\bot$) or the identifier of another capability c' which c borrows from. All capabilities that exist in the system are tracked in the capability map m, which maps a capability identifier i to the corresponding capability $m(i)$. The parent components of all capabilities in a capability map define a collection of tree structures (i.e., forest) over the capabilities, which we call the borrow trees. The parent of a capability, if it is not $\bot$ (in which case the capability is the root of a borrow tree), then corresponds to its parent in the borrow trees. We say a capability c_1 is derived from c_2 if c_1 is in the borrow subtree of c_2. CapsLock capabilities do not include the exact cursor address as in Capstone. Instead, such an address is specified by the tagged integer value itself.

The capability map defines the system state relevant to our discussion of CapsLock. We do not model memory and register states as they remain the same as in existing systems, except that each address-wide location can be associated with a capability identifier.

4.2 Instructions

We describe each instruction on capabilities with a rule that derives a new capability map m' of the system from the previous one m. When the preconditions on the rules are not satisfied, the system thus gets stuck, which indicates that an invalid operation is encountered, e.g., due to violating Rust principles. Table 2 lists the instructions and their corresponding rules, which reference auxiliary definitions in Appendix A. We introduce them informally with the running example in Figure 3.

Listing 4 shows the binary code (in 64-bit RISC-V) corresponding to the example instrumented with CapsLock instructions. The `main()` function stores `v`, `v_raw`, and `v_ref` at offsets 24, 16, and 8 to the stack pointer `sp`, respectively. A Create instruction (Line 9) injected immediately after the heap memory allocation (Line 8) creates a new capability with bounds that match the address range of the allocation. The new capability has a permission of read-write and is the root of its borrow tree (i.e., it has $\bot$ as its parent). Subsequent injected BorrowW instructions at Lines 12 and 14 produce capabilities that correspond to `v_raw` and `v_ref` respectively. The two capabilities cover the same range, but each has the earlier one as the parent, signifying that it borrows from the latter. CapsLock extends the behaviours of the load/store instructions (`ld` and `sd`).

At Line 2 in `use_p()`, the store instruction (from the inline assembly) invalidates capabilities in the subtree of the capability for `a0` at this point, which is the one created at Line 12. This invalidates the capability for `v_ref` (created at Line 14). Invalidation is represented in Table 2 in the RevokeW$(m, D_m(S))$ auxiliary function, which returns a capability map that is the same as m except that the permissions of all capabilities identified by identifiers in the set $D_m(S)$ are NA in it. The set $D_m(S)$ contains each identifier in S along with identifiers of all capabilities in its subtree. S contains all capabilities overlapping with the accessed address a (denoted as the set $O_m(a)$) which are not ancestors of the capability identified by i (denoted as the set $A_m(i)$. Putting it together, the Store instruction invalidates capabilities in the subtree of any capability that overlaps with the accessed address and is not an ancestor of the capability used (including itself) in the borrow tree. The store instruction at Line 19 accesses memory using `a1` which carries the capability of `v_ref`. The instruction checks this capability, finds it invalid (i.e., with permission NA), and hence reports a Rust principle violation.

4.3 Safety

As with other capability-based abstractions, CapsLock provides spatial memory safety by bounds-checking all memory accesses with bounds encoded in capabilities. Additionally, we present proof sketches which show that the CapsLock design maintains the invariants described in Section 3.4.

Well-nested borrowing. Assume that at some time point, a valid capability with identity i_1 has an invalid parent i_2. From Table 2, the only source of invalid capabilities (i.e., those with the NA permission) is the RevokeW function in Drop and Store instructions. Since the capability identified by i_2 is invalid, there must exist m, m', S, such that $m' = \mathrm{RevokeW}(m, S), i_2 \in S, m(i_1) = (-, -, -, i_2)$, and m, m' are two consecutive capability map states during run-time. Observe that in both Drop and Store, S is of the form $S = D_m(S')$. By induction on the definition of $D_m(S')$, $i_1 \in D_m(S') = S$, hence $m'(i_1) = (-, -, \mathrm{NA}, -)$, which contradicts the assumption that the capability identified by i_1 is valid. Therefore, lifetimes of capabilities in borrow trees are always well-nested.

Exclusive access. Assume that at a time point the capability map state is m_0 and $m_0(i_1) = (a_l, a_r, -, -)$, and $i_2 \notin D_{m_0}(i_1)$ is the identifier of a capability in the subtree of i_1. Without loss of generality, suppose $m_0 \longrightarrow^{\text{Store}(i_2,a)} m_1 \rightsquigarrow m_2, a_l \leq a < a_r$, and $m_2 \longrightarrow^{\text{Load}(i_1,a')} m_3$ or $m_2 \longrightarrow^{\text{Store}(i_1,a')} m_3$. First, it is obvious from definitions that $i_2 \in D_{m_0}(i_1)$ if and only if $i_1 \in A_{m_0}(i_2)$. Therefore, $i_1 \notin A_{m_0}(i_2)$. Since $a_l \leq a < a_r$, $i_1 \in O_{m_0}(a)$. By semantics of $\text{Store}(i_2, a)$, $m_1(i_1) = (a_l, a_r, \text{NA}, -)$. Observe that no instruction changes the permission of a capability from NA to RW or RO. Therefore, $m_2(i_1) = (-, -, \text{NA}, -)$. With m_2, the conditions specified in semantics of neither $\text{Load}(i_1, a')$ nor $\text{Store}(i_1, a')$ are satisfied. This contradicts the assumption that $m_2 \longrightarrow^{\text{Load}(i_1,a')} m_3$ or $m_2 \longrightarrow^{\text{Store}(i_1,a')} m_3$. Thus, before a capability is last used for memory access, it is guaranteed that no mutation to the memory region it covers is made through capabilities other than those in its subtree. Similarly, we can conclude that before the last store access with a capability, no load to its memory region is made through capabilities other than those in its subtree.

5 CapsLock: Implementation

We have implemented a software-based prototype of CapsLock as a readily usable debugging tool for detecting Rust principle violations in mixed Rust code. The implementation consists of two components: 1) a capability-based architecture, which adds support for encoding and maintaining CapsLock revoke-on-use capabilities and related instructions using QEMU [8], and 2) CapsLock-specific instrumentation to Rust code to expose key semantics to the hardware. We expect a hardware implementation to achieve better performance and enable extending the abstraction to a full system and integration with existing capability use cases, e.g., for software compartmentalization. We plan to investigate this in future work.

5.1 Rust-specific Adjustments for Compatibility

In practice, some special cases in Rust programs require adjustments to the core CapsLock design introduced in Sections 3 and 4. Those adjustments remove major false positives, i.e., cases with misidentified Rust principle violations. We include the following Rust-specific adjustments in our implementation.

Raw pointers. Unlike references, software can use multiple aliasing raw pointers created from the same reference in an interleaving way. Therefore, for raw pointers, we enforce a different exclusiveness invariant: while a raw pointer type capability is valid, no store to any memory location that aliases with the capability memory bounds takes place unless through a capability derived from its parent through borrowing (i.e., inside the subtree of its parent in the borrow tree). We add a *type* bit to capability metadata to identify capabilities corresponding to raw pointers.

Interior mutability. Interior mutability in Rust allows mutation even behind immutable references. It requires such mutation to be performed behind an UnsafeCell reference. We extend the type field further to record if a capability corresponds to an UnsafeCell, and further relax the exclusiveness invariant to except aliasing stores from within subtrees of UnsafeCell capabilities: while a capability is valid (i.e., between its creation and last use) no store to any memory location that aliases with the capability memory bounds takes place unless through a capability derived from it through borrowing (i.e., inside its subtree in the borrow tree), *or derived from an UnsafeCell capability it is not derived from.*

Pointer provenance. CapsLock tracks capability propagation to associate correct metadata with pointers. Doing this naïvely at the machine code level easily causes errors. C standard library implementations, e.g., glibc, contain widespread use of pointer arithmetic, including operations which subtract or add pointers together. They are purposefully done, often for performance optimizations. For instructions that take two input register operands, e.g., `add` and `xor`, both registers can be capabilities, in which case it cannot immediately be determined with which capability to associate the result. Our implementation overcomes this problem by allowing multiple capabilities to be associated with each value. Instructions with two input register operands combine the lists of capabilities associated with the inputs to produce the result. When the software later uses the result for load or store for the first time, our implementation resolves its provenance by keeping only the associated capability with memory bounds matching the requested load/store address.

5.2 Implementation on an Instruction Set

We implement an instantiation of CapsLock on RV64 [36] based on user emulation mode of QEMU [8] with the TCG backend.

Capability storage. To maintain maximum binary-level compatibility with existing software, we eschew changing any data in memory and instead store capability metadata in shadow memory inaccessible to the user program, indexed by the virtual address. In particular, compared with designs that store capability metadata inline in memory [37, 41], storing metadata offline has the benefit of avoiding the need to expand the pointer width, which in turn necessitates adjustments to ABIs and changes to all software, including external libraries the Rust program calls into.

Our implementation recycles the metadata storage for capabilities. It maintains for each capability a reference count, corresponding to the number of memory locations containing it. Upon failure to allocate metadata space for a new capability due to lack of free space, our implementation checks the reference counts of all capabilities and scans through all registers. After combining such information, it adds all metadata space associated with capabilities that are invalid and appear in neither memory nor registers to a *free* linked list, from which it allocates metadata space thereafter.

Borrow tree. Revocations in CapsLock consist of invalidating of whole subtrees of borrow trees. We implement this with simple subtree traversals, marking each visited capability as invalid. Since invalidation is irreversible (an already invalid capability never becomes valid), once invalidated, a capability stops playing any role in borrow trees. As a performance optimization, we disconnect a subtree from the borrow tree after invalidating it, thus avoiding paying for invalid capabilities during future invalidations. In this way, capabilities connected to a borrow tree are always valid.

5.3 Rust Compiler Instrumentation

We instrument the Rust code to inject instructions that create, free, and borrow capabilities. Such instrumentations pass the necessary high-level semantics of Rust code to the machine code level. External libraries require no recompilation. The instrumentation on Rust code takes place in three different places.

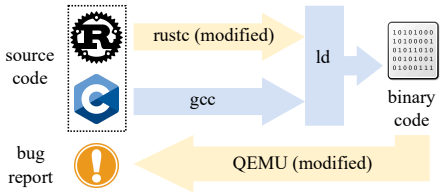


Figure 5: CapsLock in operation with an example program in Rust and C. The shade colour indicates if it is modified for CapsLock: blue for unmodified and orange for modified.

Rust standard library. We modified the default heap allocator `Global` in the Rust standard library to create a capability for every heap allocation and invalidate the associated capabilities every time software frees a heap object. The Rust standard library was also instrumented with the modified MIR pass described below. Since this prototype implementation interposes the standard memory allocator in Rust code but not memory allocators in foreign code, it does not protect memory allocated in foreign code. Note that this is not due to fundamental limitations of the revoke-on-use model. Interposing memory allocators for foreign code will enable such protection, and a hardware implementation of CapsLock can protect all memory even without interposing on memory allocators.

MIR pass. We added an extra MIR pass to the reference Rust compiler `rustc` to inject borrow instructions. This added MIR pass scans through the MIR code for sites of conversions between raw pointers and references, and injects borrow instructions immediately after such conversions. Such instructions are injected as calls to functions that include inline assembly code to perform the borrow. We choose to instrument at the MIR level because it is simpler than Rust source code while keeping the relevant semantics intact.

LLVM frame lowering. We modified the prologue and epilogue generation logic in the LLVM backend to cover stack objects with capabilities. The modified prologue creates a capability for each stack object, while the modified epilogue performs invalidation on the associated capabilities of stack objects one by one.

5.4 Putting It Together

Figure 5 shows an overview of how the components fit together. Take a Rust program that links to a C library as an example. The program is built as normal, except that the Rust code is compiled using the modified `rustc` compiler (including the modified LLVM backend and Rust standard library) that injects CapsLock instructions, with optimizations disabled to preserve as much original semantics in the source code as possible. The resulting binary code is then executed on the CapsLock architecture implemented in QEMU, which then reports any Rust principle violations encountered.

6 Evaluation

We answer three evaluation questions (EQs) in our experiments:

EQ1. Is CapsLock compatible with existing Rust code?
EQ2. Can CapsLock detect new bugs in mixed Rust code?
EQ3. What is the expected performance overhead of CapsLock?

Table 3: Summary of compatibility evaluation results. Legend: P(assed), I(gnored), U(nsupported), O(thers).

Ours \ Miri	P	I	U	O	Total
P	4948	17	12	1	4978
OOB	1	0	0	0	1
Invalid cap	11	0	0	1	12
O	0	0	1	0	6
Total	4960	17	13	2	4992

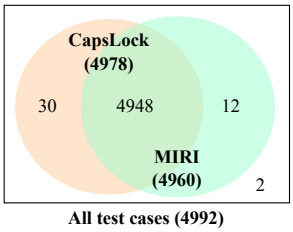


Figure 6: A Venn diagram comparing the numbers of test cases passed on CapsLock and Miri. CapsLock passes 99.7% of the test cases in the 100 most popular crates.

We ran all experiments on a machine with an AMD Ryzen Threadripper 3970X 32-core processor and 96 gigabytes of RAM, inside an Apptainer container with Ubuntu 22.04 LTS.

6.1 Compatibility with Rust Crates

To answer EQ1, we compare the result of running existing Rust code on CapsLock against that of running it on Miri.

Benchmarks. We collected unit tests and integration tests bundled with the 100 most popular (by numbers of downloads) crates on crates.io [2]. We assume that these widely-used Rust projects are representative of idiomatic Rust code and are of relatively high quality. From these crates, we successfully collected and built a total of 5388 test cases[2], from which we further exclude those that fail due to missing dependencies in the environment (e.g., missing software packages) or time out on either Miri or CapsLock. This results in a total of 4992 test cases used in our compatibility evaluation.

Miri configuration. We first tried targeting Miri to RISC-V [36], matching CapsLock. However, for 60 out of the 100 crates covered, Miri reports unsupported use of inline assembly. Further inspection reveals that the inline assembly is in the `core::core_arch::riscv64` and `core::core_arch::riscv_shared` modules, which provide RISC-V-specific intrinsic implementations. In comparison, Rust provides their counterparts in x86-64 through invocations of LLVM intrinsics, which Miri emulates. We decided that Miri did not provide a mature enough support for the RISC-V backend and therefore configured Miri to target x86-64 instead for our main experiments.

Results. We categorize test cases based on whether they pass on CapsLock and Miri, as well as on the cause of failure. Table 3 provides a summary of the evaluation results and Figure 6 shows

[2]Since crate authors may configure each test case to be disabled or ignored for different setups (e.g., target architecture, whether on on Miri), we include only test cases that are (1) present for both setups or (2) disabled or ignored only because of Miri.

a Venn diagram of the test cases passed on CapsLock and Miri, respectively. A majority of the test cases pass with no violations on CapsLock and Miri. CapsLock passes 99.7% of the test cases.

Comparison with Miri. In total, more test cases pass on CapsLock than on Miri. CapsLock passes all test cases ignored by Miri and 12/13 of the tests with operations unsupported by Miri.

Failed cases. A total of 12 test cases (0.2%) pass on Miri but fail on CapsLock. Those test cases are distributed across 5 crates: `crossbeam-utils`, `once_cell`, `parking_lot`, `tempfile`, and `url`. We investigate them and identify two main mismatching corner cases involving interior mutability between the revoke-on-use model of CapsLock and the proposed Rust aliasing models [18, 35]. Unlike revoke-on-use, proposed Rust aliasing models allow bypassing an UnsafeCell to access memory it covers. Since they are defined per memory location, they also allow interleaving uses of multiple references to memory regions covered by the same UnsafeCell, as long as those regions are disjoint. We did not pursue those corner cases further as no officially accepted aliasing model exists yet. [3]

CapsLock is highly compatible with existing code, passing 99.7% of test cases from the 100 most popular crates on crates.io [2].

6.2 Effectiveness in Security Bug Detection

We evaluate the ability of CapsLock to detect known cases of Rust principle violations. We also experimentally determine the gap between Rust principles and two other notions: memory and thread safety. We find that tools designed for detecting issues for the latter are incapable of detecting many bugs caused by Rust principle violations, in line with our discussion in Section 3.6.

Benchmarks. We collect proof-of-concept exploits (PoCs) of 15 known vulnerabilities attributed Rust principle violations from the RustSec Advisory Database [1].[4] We test the effectiveness of CapsLock in bug finding on those 15 PoCs compared to Miri. We also include results of AddressSanitizer [30] and ThreadSanitizer [31], two widely-deployed checkers for memory safety violation and data race detections, to verify the gap between Rust principles and these alternative notions.

Results. CapsLock detects all 15 Rust principle violations across all three types, matching Miri, as shown in Table 4.

Comparison with AddressSanitizer and ThreadSanitizer. Corroborating Section 3.6, tools not designed for detecting Rust principle violations show limited effectiveness. AddressSanitizer and ThreadSanitizer together only detect memory safety violations or data races in 9 of the 15 bugs that violate Rust principles. Among the three principles, they fare most poorly with AXM violations. Only 1 out of 5 AXM violations (RUSTSEC-2022-0007) is detected by AddressSanitizer as it results in a use-after-free, and none is detected by ThreadSanitizer. We inspect the cases they both fail to detect, and find that most (5 out of 6) do not trigger memory safety violations or data races.

Table 4: Evaluation results on collected PoCs attributed to Rust principle violations. A checkmark (✓) indicates that a violation is detected. A double cross (✗✗) for AddressSanitizer (AS) and ThreadSanitizer (TS) indicates missed cases due to the memory or thread safety risks not being visible during test run. A single cross (✗) indicates cases missed despite memory errors or data races visible in the run.

RUSTSEC-	Violation	CapsLock	AS	TS	Miri
2020-0023	AXM	✓	✗✗	✗✗	✓
2021-0031	AXM	✓	✗✗	✗✗	✓
2021-0114	AXM	✓	✗✗	✗✗	✓
2022-0007	AXM	✓	✓	✗	✓
2022-0040	AXM	✓	✗✗	✗✗	✓
2019-0009	Borrowing	✓	✓	✗	✓
2019-0023	Borrowing	✓	✓	✓	✓
2020-0091	Borrowing	✓	✓	✓	✓
2021-0130	Borrowing	✓	✓	✗	✓
2022-0002	Borrowing	✓	✗✗	✗✗	✓
2022-0070	Borrowing	✓	✗	✗	✓
2023-0070	Borrowing	✓	✓	✗	✓
2021-0039	Ownership	✓	✓	✗	✓
2021-0049	Ownership	✓	✓	✗	✓
2021-0053	Ownership	✓	✓	✓	✓

CapsLock is able to detect all known Rust principle violations we tested, matching the results from Miri. Memory safety violation and data race detection tools fail to detect 40% of such violations.

6.3 Improvements for Mixed Rust Code

Miri [4] is the official Rust tool to catch undefined behaviours through runtime testing in mixed Rust code. Note that Miri does not work with code using FFI or inline assembly. Nonetheless, it is widely used. We examine in EQ2 whether CapsLock can detect bugs in programs that Miri does not work with, specifically on programs with FFI or inline assembly.

Benchmarks. We collected built-in test cases of crates for which Miri reported "unsupported operations" from crates.io [2]. Specifically, we include the top 1000 popular crates along with 13000 randomly sampled crates deemed to potentially use FFI.[5] Test cases that do not build automatically through `cargo` are discarded. In total, we obtain 86051 test cases. Miri reports unsupported operations for 10478 of them, which we then run on CapsLock successfully. All crates are the latest versions available at the time of experiments.

Bugs found. CapsLock detects 8 previously unknown cases of Rust principle violations across the FFI, which Miri is unable to handle, as summarized in Table 5. We have manually reviewed the relevant source code to confirm that they are violations of Rust

[3] Stacked Borrows and Tree Borrows capture slightly different requirements, with Tree Borrows being generally more tolerant. It is possible to adapt CapsLock to match either model more accurately, but we avoid this for two reasons. First, we avoid overfitting the ISA to Rust. Instead, we focus on the goal of highlighting the generality of application-level isolation and object-level memory safety. Second, for the Rust use case, we focus only on the core component of the aliasing model, i.e., Rust principles. The other details of the Rust aliasing model are still work in progress and under debate.

[4] At the time of our experiments, RustSec contained a total of 641 entries. We used the following criteria to select the entries to use in our experiments: We first included those in the `memory-corruption`, `memory-safe`, and `thread-safety` categories, and additionally those that mentioned `memory`, `thread`, `race`, or `sound` in the keywords or the advisory main text. For the resulting 333 entries, we then collected 162 PoCs. We successfully built 55 of those PoCs for the 64-bit RISC-V Linux platform. Among them, 15 are attributed to violations of Rust principles and thus relevant to our experiments.

[5] We use a simple heuristic: if the source code contains any of the strings `ffi`, `bindgen`, and `#[link`, we consider the crate as likely to use FFI.

Table 5: Summary of previously-unknown bugs detected with CapsLock. For the maintainer response, ✓ indicates *confirmed and fixed*, and ♦ indicates *confirmed but not fixed*. "(S)" indicates that Stacked Borrows [18], but not Tree Borrows [35], considers the principle as violated, and "(X)" indicates that the bug can cause security issues under the current `rustc` implementation.

Crate	Downloads (Total / 90-day)	Violation	Unsupported Operation Miri Reports	Response
`ring 0.17.8`	171146621 / 28216664	AXM (S)	`ring_core_0_17_8_OPENSSL_cpuid_setup`	✓
`generator 0.8.3`	9520725 / 1367251	AXM	`getrlimit`	
`sxd-document 0.3.2`	877547 / 94741	AXM	`gnu_get_libc_version` through `cargo-wix 0.3.8`	✓
`mozjpeg 0.10.10`	395789 / 43760	AXM	`jpeg_std_error`	✓
`vorbis_rs 0.5.4`	44149 / 16891	AXM	`vorbis_comment_init`	✓
`corgi 0.9.9`	22508 / 6318	AXM	`mi_malloc_aligned`	✓
`cadical 0.1.14`	15567 / 3268	AXM	`ccadical_init`	♦
`libipt 0.1.4`	7854 / 4326	Borrowing (X)	`pt_cpu_errata`	

principles and analyse the root causes. We have reported these issues to the maintainers on GitHub, which hosts the repositories of all eight crates. When communicating the issues, we reproduced simplified versions of the relevant code entirely in Rust to run on Miri and confirmed that Miri also considered them as Rust principle violations. In five cases, the maintainers have confirmed and fixed the bugs. The crates show varying degrees of popularity, ranging from having under ten thousand total downloads to over twenty million monthly downloads. This reflects the fact that violations of Rust principles do occur and the lack of effective tools and methods in detecting such violations in projects that involve FFI.

Most (7 out of 8) crates violate the AXM principle. Miri fails to handle their use of FFI, including calls to system APIs (e.g., `getrlimit` and `gnu_get_libc_version`) and application library APIs (e.g., `jpeg_std_error`). CapsLock is able to handle both cases. In one case (`sxd-document`), CapsLock detected the issue when running test cases of another crate that depended on it.

It is worth noting that these bugs were found with *normal built-in* tests and did not require any specialized creation of proof-of-concept exploits. In order to understand whether these bugs would be caught by off-the-shelf memory safety checkers, we tried crafting specific PoCs to check if a memory safety violation can be induced from them. We find that 1 (`libipt`) out of the 8 bugs can lead to a use-after-free under the current `rustc` compiler implementation. Note that despite this, the built-in test cases do not trigger any memory safety issue. As such, tools like AddressSanitizer are unable to detect the bug. To the best of our knowledge, the other bugs currently cause no security violations, but that is subject to changes in `rustc` implementation and the optimization techniques used, as discussed in Section 2.2. Therefore, all 8 bugs can impact security and should be addressed in a memory safe program. We discuss the details of three of those detected bugs in Section 7.

CapsLock detects 8 previously-unknown bugs in cases with FFI and inline assembly which Miri does not support.

6.4 Performance

We answer EQ3 by measuring the time CapsLock and Miri spend running the 4948 test cases from the compatibility experiments (see Section 6.1) which both CapsLock and Miri pass.

Results. Figure 7 shows a histogram that compares the two execution times for each test case. A total of 98.1% of the test cases take less time on CapsLock than on Miri. On average, CapsLock spends 45.9% as much time as Miri on a test case.

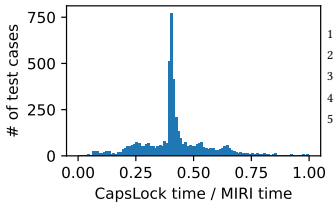


Figure 7: Histogram of the performance of CapsLock compared to Miri (bin size: 0.01).

```rust
let mut r = Box::new(N::new());
let p = &mut *r as *mut _;
r.p = p;
// ...
unsafe { &mut *r.p }
```

Listing 5: AXM violation in `generator` (simplified).

Discussion. It might appear counterintuitive first how CapsLock outperforms Miri despite working at the lower and more detailed binary level. We note two potential factors that contribute to CapsLock's performance advantage. Firstly, Miri maintains high-level metadata that it has access to to cover more types of undefined behaviours than CapsLock does. For example, Miri also handles more general type safety violations, e.g., using an arbitary integer value as a boolean. CapsLock instead focuses on violations of the three principles that are unique to Rust without tracking such information. Secondly, QEMU provides JIT compilation through its TCG engine [8] and is optimized for high performance. In contrast, MIRI operates entirely as an interpreter.

CapsLock achieves more than 2x speed-up over the present implementation of Miri on average.

7 Qualitative Analysis of New Bugs

Our qualitative analysis reveals that CapsLock can detect bugs that require various levels of detection ability.

The easy case: FFI not involved. In the easiest scenario, a test case uses FFI but the bug does not directly involve it. To handle such cases, it suffices to run external code without tracking pointer use therein. An example is the bug found in `generator`, which needs to allocate stack space for its managed threads. On UNIX-like systems, the crate invokes the `getrlimit` system call to decide the stack size limit (`RLIMIT_STACK`). Miri does not support this system call and thus fails to run these test cases. The part of the crate that violates AXM (Listing 5) is entirely in Rust and unrelated to the system

```rust
let mut started = CompressStarted {
  compress: self,
  dest_mgr: Box::new(DestinationMgr::new(writer,
↪  write_buffer_capacity)),
};
unsafe {
  let dest_ptr = addr_of_mut!(started.dest_mgr.as_mut().iface);
  started.compress.cinfo.dest = dest_ptr;
  ffi::jpeg_start_compress(&mut started.compress.cinfo,
↪  boolean::from(true));
}
// ...
unsafe {
  ffi::jpeg_finish_compress(&mut started.compress.cinfo);
}
```

Listing 6: AXM violation in `mozjpeg`.

call. It handles pointers to maintain a tree data structure. The code stores a raw pointer to `r` in `r.p` (Line 3) and then dereferences the raw pointer (Line 5). This results in two mutable references to the same object (`r`), a violation of AXM. Under both Stacked Borrows and Tree Borrows, `p` would become invalid when `r.p` is mutated at Line 3, making the dereferencing illegal at Line 5.

Bugs caused by pointer use behind FFI. More difficult to detect are violations caused by pointer use in external code after Rust code passes the pointer out through FFI. It is no more sufficient to merely run the external code. Rather, it requires tracking how external code uses the pointer. For example, the `mozjpeg` crate provides Rust bindings for MozJPEG, a JPEG encoder library written in C. Miri is unable to run the test cases due to the extensive use of FFI. Listing 6 shows where `mozjpeg` violates AXM. It stores a pointer to `started.dest_mgr` in `started.compress.cinfo.dest` (Line 7), and later invokes the FFI `jpeg_finish_compress` with `started.compress.cinfo` borrowed and passed in as an argument. The C function `jpeg_finish_compress` then uses the raw pointer `started.compress.cinfo.dest` to mutate the content of `started.dest_mgr`. This violates AXM as `started.dest_mgr` is accessed without the owner being borrowed. This violation directly involves FFI. It requires knowing that `jpeg_finish_compress` dereferences `started.compress.cinfo.dest`.

Bugs caused by cross-FFI pointer propagation and use. Even more demanding is the detection of bugs that involve pointers propagated across FFI multiple times. The Rust code may pass pointers to data structures to external code, which then initializes those data structures by storing some pointers in them. Later, Rust code passes in a pointer to one such data structure, and the external code uses pointers in it to access other data structures. Detecting such a bug requires tracking not only how each FFI call individually uses pointers, but also pointer propagation in the global state (e.g., heap memory). For example, the `vorbis_rs` crate provides Rust bindings for a Vorbis codec library in C. Miri fails to run the test cases due to FFI. Listing 7 shows the code related to an AXM violation. The Rust code first allocates memory for several data structures. It then invokes the library to perform initializations (Lines 2 and 7, top), which connect those data structures to one another by storing those pointers (Lines 4 and 9, middle). The Rust code still keeps owner references to those data structures. It then passes them when invoking

```rust
unsafe {
  libvorbis_return_value_to_result!(vorbis_analysis_init(
    vorbis_dsp_state.as_mut_ptr(),
    &mut *vorbis_info.vorbis_info
  ))?;
  let mut vorbis_dsp_state = assume_init_box(vorbis_dsp_state);
  libvorbis_return_value_to_result!(vorbis_block_init(
    &mut *vorbis_dsp_state,
    vorbis_block.as_mut_ptr()
  ))?;
  // ...
  libvorbis_return_value_to_result!(vorbis_analysis(
    vorbis_block,
    ptr::null_mut()
  ))?;
  // ...
}
```

```c
int vorbis_block_init(vorbis_dsp_state *v, vorbis_block *vb){
  int i;
  memset(vb,0,sizeof(*vb));
  vb->vd=v;
  // ...
  vorbis_block_internal *vbi=
  vb->internal=_ogg_calloc(1,sizeof(vorbis_block_internal));
  // ...
  vbi->packetblob[i]=&vb->opb;
  // ...
}
```

```c
int vorbis_analysis(vorbis_block *vb, ogg_packet *op){
  // ...
  oggpack_reset(vbi->packetblob[i]);
  // ...
}
```

Listing 7: AXM violation in `vorbis_rs`. Top: Rust code that invokes the library to initialize Rust-owned data structures, followed by more invocations that pass those data structures as arguments. Middle: Initialization code which stores in those data structures pointers to one another. Bottom: Subsequent dereferencing of those pointers in the library.

more library APIs (Line 12, top). In those library calls, those pointers are dereferenced for mutation (Line 3, bottom). This violates AXM similarly as in `mozjpeg`—mutating an owned object without borrowing the owner—but its detection requires more knowledge about the external code. In particular, `mozjpeg` sets up the data structure pointers in Rust, so knowing if external code dereferences them is sufficient, whereas in `vorbis_rs` both initialization and use of the data structures happen in FFI calls. It also requires knowing where the pointers are stored during initialization.

8 Related Work

Dynamic bug detection in Rust. ERASan [26] and RustSan [9] improve the performance of AddressSanitizer [30] on Rust code by removing unnecessary run-time checks already statically guaranteed by Rust principles. Isolation-based methods for protecting safe Rust code identify potentially unsafe memory accesses and isolate their executions through techniques such as Intel MPK [7, 15, 19, 20, 29], or software fault isolation [5, 21, 24]. Those methods focus on memory safety violations and do not identify Rust principle violations in general. Miri [4] is the official Rust undefined behaviour detection tool based on interpreting Rust programs at the MIR level. It detects

a wide range of undefined behaviours including violations of Rust principles, but does not support FFI or inline assembly due to its reliance on high-level program semantics in MIR.

Cross-language bug detection. FFIChecker [23] uses LLVM-IR-level static taint tracking to find bugs in Rust programs across FFI. It only covers memory leaks and mismatching use of heap allocators, rather than Rust principle violations. Its use of LLVM IR also requires recompilation of all external code with LLVM-based toolchains and prevents it from handling assembly code or dynamic linking, unlike CapsLock. MiriLLI [25] extends the LLVM IR interpreter to emulate the ABIs of FFI calls at the LLVM IR level and call back corresponding Miri functions to perform checks. Due to reusing Miri, MiriLLI covers more varieties of undefined behaviours (e.g., type confusions) than CapsLock. On the other hand, CapsLock detects Rust principle violations, a subset of undefined behaviours unique to Rust, through a simple and general abstraction entirely at the machine code level. MiriLLI shares the same limitations as FFIChecker due to using LLVM IR.

Rust aliasing models. Rust aliasing models formally define valid ways for references and pointers in Rust to alias with one another during run-time. They are formalizations of Rust principles that also cover mixed Rust code. Two common aliasing models are Stacked Borrows [18] and Tree Borrows [35]. Although the latter is generally stricter (i.e., allowing fewer undefined behaviours) than the former, both models have significant overlap. CapsLock is not a proposal of a new Rust aliasing model. It is a design for detecting Rust principle violations in line with existing aliasing models.

Capability-based architectures. Hardware capabilities have a long history [22]. The more recent capability-based architecture designs include Mondrian [39], CHERI [37], and Capstone [41]. Existing work has explored using capability-based architectures for spatial memory safety [11, 14, 37, 39, 41], temporal memory safety [13, 33, 38, 40], and isolation [12, 34, 37, 41]. Most has focused on protecting legacy systems in C/C++. Recent work proposes using capability-based architectures for unsafe Rust to safeguard memory accesses [16, 32]. Those methods only provide protection for memory safety without covering other types of bugs in Rust. Capstone [41] enhances the existing capability-based models by incorporating extra capability types such as linear capabilities and revocation capabilities. Those additions support enclave-like exclusive access guarantees and revocable capability delegations.

9 Limitations and Future Work

Our current CapsLock implementation based on QEMU demonstrates the practicality of using capabilities for detecting Rust principle violations at the architecture level through the revoke-on-use mechanism. We plan to investigate how to enable efficient implementation closer to real hardware in the future. Revoke-on-use is conceptually compatible with existing use cases of hardware capabilities, but the concrete strategy for integrating those use cases together requires more investigation.

10 Conclusions

We present a novel revoke-on-use mechanism for using a hardware capability-based abstraction to enforce Rust principles across languages at the machine code level. CapsLock, our design based on revoke-on-use, yields a readily usable tool for detecting Rust principle violations in real-world Rust projects, including those that use FFI or inline assembly.

Acknowledgements

We thank NUS KISP Lab members for their feedback. This research is supported by a Singapore Ministry of Education (MOE) Tier 2 grant MOE-T2EP20124-0007.

References

[1] [n. d.]. About RustSec › RustSec Advisory Database. https://rustsec.org/.
[2] [n. d.]. Crates.Io: Rust Package Registry. https://crates.io/.
[3] 2023. Supporting the Use of Rust in the Chromium Project.
[4] 2024. Rust-Lang/Miri. The Rust Programming Language.
[5] Hussain M. J. Almohri and David Evans. 2018. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM, Tempe AZ USA, 248–255. doi:10.1145/3176258.3176330
[6] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–27. doi:10.1145/3428204
[7] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. 2023. TRust: A Compilation Framework for in-Process Isolation to Protect Safe Rust against Untrusted Code. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6947–6964.
[8] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA.
[9] Kyuwon Cho, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee. 2024. RustSan: Retrofitting AddressSanitizer for Efficient Sanitization of Rust. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3729–3746.
[10] Thomas Claburn. 2023. Microsoft Is Rewriting Core Windows Libraries in Rust. https://www.theregister.com/2023/04/27/microsoft_windows_rust/.
[11] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. *ACM SIGARCH Computer Architecture News* 36, 1 (March 2008), 103–114. doi:10.1145/1353534.1346295
[12] Lawrence Esswood. 2020. *CheriOS: Designing an Untrusted Single-Address-Space Capability Operating System Utilising Capability Hardware and a Minimal Hypervisor*. Ph. D. Dissertation. Apollo - University of Cambridge Repository. doi:10.17863/CAM.74163
[13] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, La Jolla CA USA, 251–268. doi:10.1145/3620665.3640416
[14] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee. 1995. The M-Machine Multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. IEEE, Ann Arbor, MI, USA, 146–156. doi:10.1109/MICRO.1995.476822
[15] Merve Gülmez, Thomas Nyman, Christoph Baumann, and Jan Tobias Mühlberg. 2023. Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust. arXiv:2306.08127 [cs]
[16] Sarah Harris, Simon Cooksey, Michael Vollmer, and Mark Batty. 2023. Rust for Morello: Always-On Memory Safety, Even in Unsafe Code. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 39:1–39:27. doi:10.4230/LIPIcs.ECOOP.2023.39
[17] Jonathan Corbet. 2022. A First Look at Rust in the 6.1 Kernel [LWN.Net]. https://lwn.net/Articles/910762/.
[18] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked Borrows: An Aliasing Model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–32. doi:10.1145/3371109
[19] Martin Kayondo, Inyoung Bang, Yeongjun Kwak, Hyungon Moon, and Yunheung Paek. 2024. METASAFE: Compiling for Protecting Smart Pointer Metadata to Ensure Safe Rust Integrity. In *USENIX Security '24*.
[20] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In

Proceedings of the Seventeenth European Conference on Computer Systems. ACM, Rennes France, 132–148. doi:10.1145/3492321.3519582

[21] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. ACM, Shanghai China, 51–57. doi:10.1145/3144555.3144562

[22] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Digital Press, Bedford, Mass.

[23] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. 2022. Detecting Cross-language Memory Management Issues in Rust. In *Computer Security – ESORICS 2022*, Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng (Eds.). Vol. 13556. Springer Nature Switzerland, Cham, 680–700. doi:10.1007/978-3-031-17143-7_33

[24] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing Unsafe Rust Programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 234–245. doi:10.1145/3377811.3380325

[25] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. 2024. A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries. arXiv:2404.11671 [cs]

[26] J. Min, D. Yu, S. Jeong, D. Song, and Y. Jeon. 2024. ERASAN : Efficient Rust Address Sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 239–239. doi:10.1109/SP54263.2024.00182

[27] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. *ACM SIGPLAN Notices* 44, 6 (May 2009), 245–258. doi:10.1145/1543135.1542504

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. Association for Computing Machinery, New York, NY, USA, 31–40. doi:10.1145/1806651.1806657

[29] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *Annual Computer Security Applications Conference*. ACM, Virtual Event USA, 824–836. doi:10.1145/3485832.3485903

[30] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (Usenix Atc'12)*. USENIX Association, USA, 28.

[31] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, New York New York USA, 62–71. doi:10.1145/1791194.1791203

[32] Nicholas Wei Sheng Sim. 2020. *Strengthening Memory Safety in Rust: Exploring CHERI Capabilities for a Safe Language*. Ph. D. Dissertation. University of Cambridge, Computer Laboratory.

[33] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–28. doi:10.1145/3290332

[34] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting Software with Code-Centric Memory Domains. *ACM SIGARCH Computer Architecture News* 42, 3 (Oct. 2014), 469–480. doi:10.1145/2678373.2665741

[35] Neven Villani. 2023. Tree Borrows.

[36] Andrew Waterman, Krste Asanovic, John Hauser, and CS Division. [n. d.]. The RISC-V Instruction Set Manual (Volume II: Privileged Architecture).

[37] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, 20–37. doi:10.1109/SP.2015.9

[38] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 608–625. doi:10.1109/SP40000.2020.00098

[39] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, San Jose California, 304–316. doi:10.1145/605397.605429

[40] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Columbus OH USA, 545–557. doi:10.1145/3352460.3358288

[41] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. 2023. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 787–804.

[42] Zhijingcheng Yu, Fangqi Han, Kaustab Choudhury, Trevor E. Carlson, and Prateek Saxena. 2025. Artifacts of "Securing Mixed Rust with Hardware Capabilities". Zenodo. doi:10.5281/ZENODO.14625327

[43] Yuchen Zhang, Ashish Kundu, Georgios Portokalidis, and Jun Xu. 2023. On the Dual Nature of Necessity in Use of Rust Unsafe Code. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco CA USA, 2032–2037. doi:10.1145/3611643.3613878

A Additional Details in Design

The CapsLock instruction semantics in Section 4.2 references the following auxiliary definitions.

Let $c = m(i)$ be a capability. We use $C_m(i)$ to denote the set of identifiers of all children of c in its borrow tree:

$$C_m(i) = \{i' \mid m(i') = (a_l', a_l', p', i)\}. \tag{1}$$

Then we define the capability derived set in m, which contains identifiers of all capabilities in a subtree, written $D_m(i)$:

$$D_m(i) = \{i\} \cup \bigcup_{i' \in C_m(i)} D_m(i'). \tag{2}$$

When S is a set of capability identifiers, $D_m(S) = \bigcup_{i \in S} D_m(i)$.

$O_m(a)$ is the set of identifiers of capabilities that overlap with a:

$$O_m(a) = \{i \mid m(i) = (a_l, a_r, p, b), \quad a_l \le a < a_r\}. \tag{3}$$

RevokeW(m, S) is a new capability map that is the same as m except that all capabilities with identifiers in S are invalid:

$$\text{RevokeW}(m, S) = \{i \mapsto \text{RevPermW}(c, i, S) \mid m(i) = c\}, \tag{4}$$

where

$$\text{RevPermW}((a_l, a_r, p, b), i, S) = \begin{cases} (a_l, a_r, p, b) & \text{if } i \in S \\ (a_l, a_r, \text{NA}, b) & \text{otherwise.} \end{cases} \tag{5}$$

RevokeR(m, S) demotes capabilities not in S to RO:

$$\text{RevokeR}(m, S) = \{i \mapsto \text{RevPermR}(c, i, S) \mid m(i) = c\}, \tag{6}$$

where

$$\text{RevPermR}((a_l, a_r, p, b), i, S) = \begin{cases} (a_l, a_r, p, b) & \text{if } i \in S \\ (a_l, a_r, \text{RO}, b) & \text{if } i \notin S, p = \text{RO} \\ (a_l, a_r, \text{NA}, b) & \text{otherwise.} \end{cases} \tag{7}$$

$R_m(i)$ is the identifier of the root capability of the borrow tree $m(i)$ belongs to:

$$R_m(i) = \begin{cases} i & \text{if } m(i) = (a_l, a_r, p, \bot) \\ R_m(i') & \text{if } m(i) = (a_l, a_r, p, i'). \end{cases} \tag{8}$$

$A_m(i)$ contains all ancestors of i in the borrow tree:

$$A_m(i) = \begin{cases} \{i\} & \text{if } m(i) = (a_l, a_r, p, \bot) \\ \{i\} \cup A_m(i') & \text{if } m(i) = (a_l, a_r, p, i'). \end{cases} \tag{9}$$