Strengthening memory safety in Rust: exploring CHERI capabilities for a safe language

Nicholas Wei Sheng Sim
Wolfson College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: nwss2@cam.ac.uk

August 2020

Declaration

I, Nicholas Wei Sheng Sim of Wolfson College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,620

Signed:

Date:

Acknowledgements

I thank the following people for their help with this work:

My supervisors, Simon W. Moore and Khilan Gudka, for guidance which was essential in focussing the project, and especially for my early familiarisation with CHERI and its LLVM fork;

Alex Richardson, for repeated help with the LLVM compiler and debugging when Rust broke its assumptions, as well as the CheriBSD emulator;

David Chisnall, Andrew Paverd, and colleagues at Microsoft Research, for fruitful discussions on the application of capabilities, sealing, and expounding the utility of mixing capability and non-capability pointers;

and finally the Rust community and Ralf Jung, for fueling my engagement with, and understanding of, Rust semantics.

Abstract

Strengthening memory safety in Rust: exploring CHERI capabilities for a safe language

* * *

The lack of memory safety in C still causes untold numbers of security vulnerabilities up to the present day. Both Rust, a safe programming language, and CHERI, an architecture providing hardware capabilities, claim to provide low-overhead memory safety to prevent exploits.

This work explores the implications and interactions of CHERI capabilities for a safe language, Rust. Previous work typically considered capabilities and type-safe languages as mutually exclusive protections; instead I present evidence that the two mechanisms are complementary. To do this, I implement Rust compilation to a CHERI architecture, demonstrating that capabilities are effective in preventing previous Rust vulnerabilities and detailing how they can be used to maximise memory safety in Rust with minimal overhead.

(14,620 words)

Contents

Chapter 1

Introduction

1.1 Motivation

1.1.1 Background

Many security vulnerabilities are bugs arising from the lack of memory safety in C; an example of a traditional exploit is the buffer overflow. The C language specification includes a number of situations that lead to *undefined behaviour*: akin to a deduction of 'falsity' in logic, after these points the state of the program might be arbitrary. Undefined behaviour allows compilers to assume that certain conditions never hold, enabling powerful (but dangerous) optimisations.

Consequently, crucial checks can be optimised away and invariants violated by compilers, resulting in executables vulnerable to memory attacks [50]. Many systems and compiler programmers frequently underestimate the pervasiveness of undefined behaviour in code that appears to function correctly, and gloss over the subtleties of the C standard [35]. These represent opportunities for vulnerabilities to arise when code is optimised away by compiler transformations [55].

1.1.2 The Rust programming language

Rust is billed as a fast, safe language designed for concurrent systems programming. Safe Rust claims to offer type- and memory-safety, and provides guarantees that no dangling pointers or any undefined behaviour will occur [42], giving both spatial and temporal memory protection.

To achieve this, it uses compile-time and runtime checks, relying on a combination of LLVM optimisations and *Unsafe Rust* code to minimise the performance overhead. However, as with all code that requires fine control over memory, mistakes in writing Unsafe Rust give rise to potential vulnerabilities.

Vulnerabilities in Rust

There have been several unrelated bugs and potential vulnerabilities in Rust, all believed to be unexploited. Some are preventable with CHERI capabilities, and these are examined in Section 5.2:

- Standard library: buffer overflow when pushing to a `VecDeque`, CVE-2018-1000657 [2, 24].
- Standard library: buffer overflow in slice `repeat`, CVE-2018-1000810 [3, 39, 13].
- Standard library: out-of-bounds access indexing into a reversed slice [22, 21].
- Standard library: unsafe Iterator method duplicates exclusive mutable references, violating temporal safety [8].

- Standard library: safe trait implementation allows arbitrary typecasting and thus buffer overflows, CVE-2019-12083 [4, 40].

Others are not mitigated by capabilities:

- Standard library: attempt to read uninitialised memory after appending to a `VecDeque` [26].
- Compiler plugin: documentation plugin allowed arbitrary code execution by a different user while running the compiler, CVE-2018-1000622 [1, 38].

Rust is a language with an explicit priority of memory safety, developed by a community of safety- and security-conscious programmers. Collectively, these flaws demonstrate that even such favourable conditions are not sufficient to eliminate unsoundness and vulnerabilities. On the contrary, more tools are needed to build and operate reliable and secure software.

1.1.3 Capabilities and the CHERI architecture

The CHERI instruction set architecture replaces pointers with *capabilities*, which stop unauthorised access of memory as they are unforgeable in software [58]. Capabilities protect data by associating bounds information and access permissions, giving programs fine-grained mechanisms to enforce the security principles of *least privilege* and *intentional use* [37].

Just as using capabilities in C programs prevents unintended operations on memory, applying capabilities to Rust provides an additional layer of safety, guarding against undiscovered vulnerabilities.

1.2 Contributions

Traditionally, capability mechanisms have been evaluated against C, but the main objective of this project was to evaluate the utility of porting Rust, a safe language, to CHERI. In summary, the contributions of this work are:

- A summary of features and techniques Rust uses to provide memory safety guarantees, and how it optimises these. The differences between Rust and other languages, as they pertain to capability architectures and CHERI. A survey of work relating to safe languages and capability platforms, and efforts to manage undefined behaviour and safety in programming languages.
- Patches to the Rust compiler (1.35) and core libraries which enable compilation to the new `cheri-unknown-freebsd` target utilising 128-bit CHERI capabilities. These compile programs which use the Rust core library without optimisations.
- An analysis of previous errors leading to vulnerabilities in Rust, the memory safety implications, and how they are mitigated using capabilities. Two demonstrative microbenchmarks of vulnerabilities in the Rust standard library, shown to be functional on x86 but prevented by CHERI capabilities.
- An evaluation of the interactions between Rust and CHERI protection mechanisms, and how hardware capabilities enhance Rust in general. An exploration of how features provided by the CHERI architecture can be used to enforce Rust guarantees where its compiler and runtime cannot, even as programs call into untrusted code and the kernel.
- Techniques and approaches that may be used to reduce the overheads posed by memory protection when combining the two approaches. A consideration of how the duplication of functionality can be minimised, and how doing so affects the provided safety guarantees.

- Details of the relevant concerns for future implementers porting Rust to CHERI, or examining CHERI on other safe languages. This includes a discussion on problematic points of the Rust language which may need to be clarified, changed, or implemented differently.

This work is *not* intended to be a complete implementation of CHERI support in the Rust compiler. Rather, it prototypes support for replacing pointers with capabilities in Rust, to illustrate the benefit of CHERI capabilities for a safe language.

1.3 Report structure

This report is divided into six chapters, detailing the above contributions and their context:

Chapter 2 An overview of CHERI capabilities and their mechanism, and how they can be used to improve memory safety. Motivation for the choice of Rust as a safe programming language to study. A survey of work relating to other hardware capability implementations, other mechanisms that provide memory safety, and similar techniques for safe languages including Java and Rust.

Chapter 3 An introduction to the properties of Rust relevant to memory safety and this project in particular. This includes a brief discussion of its semantics, Unsafe Rust, and some common optimisation patterns.

Chapter 4 The main changes made to the Rust compiler and core libraries that enable compilation to CHERI. This chapter outlines the scope of my implementation, hence evaluation, and presents challenges encountered that currently prevent a more complete implementation of Rust compilation to CHERI.

Chapter 5 A long chapter, and main contribution of this work. The evaluation of CHERI capabilities for Rust: why Rust is an ideal language for capability protection, and how its safety is improved by capabilities. This includes demonstration of capabilities against past vulnerabilities, how they can rule out some undefined behaviour, and options for stronger protection and enforcement of Rust guarantees by using features offered by the architecture. Concerns when porting Rust to safe languages.

Chapter 6 The conclusion: recontextualisation of the work, challenges faced, and a detailed recapitulation of contributions outlined above. A discussion of future work in bringing Rust to CHERI.

Chapter 2

Background and related work

This chapter defines and highlights the motivation for studying *capabilities*, and sketches their implementation under CHERI. It defines the property of *monotonicity* and discusses CHERI's key design principles. To contextualise this project, I justify the choice of Rust as a safe language, and conclude the chapter with a survey of related work. An overview of the Rust language is included in Chapter 3.

2.1 Capabilities

Capabilities are traditionally known as unforgeable tokens of authority. CHERI provides *memory capabilities*: unforgeable pointers to continuous regions in memory [60].

Capabilities have a long history: while the concept had been established beforehand, they were implemented and expounded in Multics [7], and there is recurring interest in fat pointers today [15, 30]. They can prevent common classes of attacks, such as the eternal buffer overflow (1,613 published CVEs in 2018 alone [36]), and obviate other mitigations, such as address-space layout randomisation and W^X protection against execution of writeable data.

A recurring theme in this work is the definition and usage of pointers. For instance, the proper usage of a capability renders object bounds checks redundant: can this optimise bounds checks in Rust? We will also see how CHERI's implementation of capability pointers stretches a key language definition in Rust due to its usage in the compiler.

2.2 CHERI: Capability Hardware Enhanced RISC Instructions

2.2.1 Memory capability model

CHERI extends the 64-bit MIPS ISA to support capabilities. Its design emphasises incremental adoption, and the principles of least privilege and intentional use, to mitigate unintended vulnerabilities. Its hybrid approach enables capability code to be used alongside non-capability code, allowing concentration on higher-risk code and libraries [58].

2.2.2 Implementation overview

Capability coprocessor and registers

A capability coprocessor is used to implement the CHERI extensions. The coprocessor holds its own register file, creating a clear distinction between integers and capabilities. Instructions are added for

manipulating these capability registers, such as loads and stores, decreasing the bounds on capabilities, and branching based on capability tags [60].

Memory tagging

To allow pointers to be stored anywhere in memory, an out-of-band tag bit is associated with each (aligned) pointer-sized location in memory. If a location storing a capability is written to by a non-capability instruction, this bit is cleared to preserve the unforgeability of pointers [60].

Capability manipulation

CHERI provides capability instructions for using capabilities, from those defining the bounds of a capability, to loading from and jumping to addresses stored in capability registers. Instructions conform to the property of *monotonicity*: they can only decrease privilege, so to preserve unforgeability [60]. Thus capability bounds (length) cannot be increased, nor can a read-only capability derive a read-write one.

Protection mechanism

If an attempt is made to use a capability to access a different object, or otherwise violate its permissions, a hardware exception (trap) results.

2.2.3 Capability sealing

A capability may be *sealed*, marking it as immutable or non-dereferenceable [58]. Based on the *object type* of a capability, the correct capability may be used to seal it; likewise another specific capability may be used to *unseal* it, again allowing mutation and dereferencing.

Applications of sealing in Rust are seen in Sections 5.5 and 5.6.

2.2.4 Using non-capability code

Code that is not capability-aware is interoperable with capability code, through instructions that translate between capabilities and integer pointers. The latter are protected by the *default data capability*, stored in the *implicit capability register*. This allows protection of non-capability code from capability-aware code and vice versa [60]. The contemporaneous usage of regular pointers and capabilities is known as *hybrid mode*, whereas a program that exclusively uses capabilities is deemed to run in *pure capability mode* [57].

This project evaluates Rust under pure capability mode only. The choice has proven to be a practical one: in keeping consistent with LLVM's memory model, the Rust compiler avoids manipulations that commonly cause incompatibility with capabilities. Section 5.3.3 explains the reasons behind this claim.

2.3 Why Rust?

Most evaluations of capability implementations focus on the C language, as a kind of lowest common denominator in computer systems. Further, its stereotypical memory unsafety precedes the reasonable assumption that C programs would benefit from capabilities, perhaps more so than a safe language might.

Rust is an example of a safe language. The following sections briefly motivate the choice of Rust as a language on which to evaluate a capability platform; see Chapter 3 for an overview of language features and semantics relevant to the use of capabilities.

Finally, it provides a foil to C evaluations of capabilities, in both adoption and utility of capabilities.

2.3.1 Bounds checks in Rust

```
1  idx:
2      lui     $1, %hi(%neg(%gp_rel(idx)))
3      daddu   $3, $1, $25
4      sltu    $1, $4, $6      ; idx < arr.len()
5      beqz    $1, .LBB3_2
6      move    $2, $4          ; delay slot
7      dsll    $1, $2, 3       ; offset = idx * 8 (i64 array)
8      daddu   $1, $5, $1      ; ptr = &arr + offset
9      ld      $2, 0($1)       ; retval = *ptr
10     jr      $ra
11     nop
12 .LBB3_2:                    ; prepare information for panic handler
13     daddiu  $sp, $sp, -16
14     sd      $ra, 8($sp)
15     sd      $gp, 0($sp)
16     daddiu  $gp, $3, %lo(%neg(%gp_rel(idx)))
17     ld      $1, %got_page(.L__unnamed_2)($gp)
18     daddiu  $4, $1, %got_ofst(.L__unnamed_2)
19     ld      $25, %call16(core::panicking::panic_bounds_check)($gp)
20     jalr    $25
21     move    $5, $2
22     break
```

Figure 2.1: Generated MIPS64 assembler for Rust's intrinsic indexing on arrays and vectors, with bounds checks. The relevant *unchecked* access occurs on lines 7–9. Rust instructs LLVM that the out-of-bounds case is unlikely, but this does not appear in the generated code. Comments added for ease of reading; `panic_bounds_check` string substituted in place of mangled version.

One of Rust's safety features is runtime bounds checks on array accesses: Figure 2.1 shows an example. As arrays have clear bounds, similar hardware checks also occur when dereferencing a CHERI capability. If object bounds are set correctly, the runtime checks are redundant and could be removed as an optimisation.

2.3.2 Broad similarity to C

While Rust espouses very different design principles to C, it targets similar applications, such as systems programming. It emphasises a small runtime, and its compilation and linking process is similar, as opposed to interpreted or JIT compiled languages. An evaluation using Rust is more immediately relevant and readily comprehensible in the context of existing literature, and applicable in re-interpreting it.

FFI: calling into other languages

Rust defines a *foreign function interface* (FFI) that makes it trivial to link against C libraries, so a smaller proportion of the Rust standard libraries need be considered for a meaningful evaluation (Section 4.1).

2.3.3 Object lifetimes and temporal safety

Rust uses *object lifetimes* instead of a garbage collector or placing the burden on users to manage memory. Like manual memory management, this design enforces intentionality, but is enforced by the compiler to prevent dangling pointers and use-after-free unsafety. This is a major feature preventing temporal unsafety, an area for which capabilities have no universal remedy, unlike the spatial integrity they frequently provide. As such, Rust has a built-in mechanism complementary to that provided by capabilities, and the interaction between these is worth examining.

2.4 Survey of related work

Most work examining capability implementations involves C, whereas there has been work examining the object-capability pattern in other languages, which I do not discuss.

The following sections provide an overview of memory safety techniques and their applicability to safe languages, with special focus on FFI in Rust and the Java Native Interface. They give an overview of the main objectives and costs involved in improving memory safety in safe languages, and demonstrate that this is very much an ongoing area of research, with no obvious "best" solution.

2.4.1 Hardbound

Hardbound provides special instructions to set bounds on pointers, preventing out-of-bounds dereferences [15]. Like CHERI, it has no mechanism to handle use-after-free or double-free errors, only identifying invalid dereferences.

It emphasises minimal compiler and memory layout changes, making it theoretically adaptable to many different compilers, including compilers to safe languages such as Rust.

Woodruff et al. detail differences between Hardbound and CHERI [60]; for the purposes of this work, much of the evaluation applies equally or similarly to Hardbound. Notable exceptions would include pointer width compatibility,[1] the comments on enforcing immutability, and protecting against abuse through FFI functions.

2.4.2 AddressSanitizer

AddressSanitizer maintains shadow state to find memory errors, including bounds checks but also use-after-free errors, with a limited possibility of detecting data races [47]. As it does not implement object boundaries on pointers, a correctly chosen offset could result in an out-of-bounds dereference into another object.

It is implemented using LLVM infrastructure, and the primary use target is C code via Clang. There is experimental support for compiling Rust with AddressSanitizer support, as well as the related LeakSanitizer, MemorySanitizer, and ThreadSanitizer [5]. While it is not expected to be useful for Safe Rust code, the main incentives for the Rust project appear to be supporting fuzz testing, finding compiler bugs, and checking unsafe code [29].

As such, it targets a subset of problems in Rust as CHERI capabilities do, albeit in software. While the authors suggest that AddressSanitizer could be run in production code, it suffers a considerable 2× slowdown.

[1] Hardbound stores less information, retaining the pointer width.

2.4.3 Sandcrust: Sandboxing Rust's FFI

Sandcrust is a set of macros and Rust compiler transformations which aim to sandbox and transform C functions called from Rust [31]. Its primary means of achieving isolation is to execute library code in a separate process from Rust caller, communicating via remote procedure calls (RPC) and pipes. This avoids giving libraries access to memory which should not be shared, and frustrates attacks such as control-flow hijacking.

This approach comes with considerable overhead, with Lamowski et al. reporting slowdown factors between 1.3× and 44×, albeit generally at the low end (1.5–8×). Considerable overhead is likely due to data transfer.

CHERI avoids this overhead, and can call foreign functions in the same address space as the Rust program. Interestingly, such a sandboxing scheme could protect data from inspection or modification by callback functions, similarly to capability sealing (Section 5.6). I do not discuss Sandcrust's ancillary functionality of wrapping C functions to fit Rust idioms, itself a valuable contribution.

2.4.4 Robusta: Sandboxing JNI code

The Java Native Interface (JNI) provides full access to the address space of a Java program, and JNI code is completely trusted by Java's security model. Robusta sandboxes native calls by intercepting calls to native functions, building on Google's Native Client. It also executes native code in a separate process, again copying data as necessary. System calls pass through JVM permissions checks, extending the Java security manager to native code [48]: in essence, a reference monitor in the JVM.

Native Client provides separation for the primary purpose of computation, reporting around 5% overhead for computational tasks [62]; Siefers et al. give similar figures for computational tasks on Robusta. Conceivably, such a small overhead could also apply to sandboxing in Rust, on the same tasks and with some optimisation. However, with less computational tasks, copying objects in and out of the sandbox can lead to over 15× slowdown. These broadly correlate with Sandcrust's results.

Note again that Robusta does more than improve memory safety: it also intercepts system calls to ensure they are permitted by the Java security manager.

2.4.5 CHERI compartmentalisation and the JNI

As an alternative to process-based sandboxing, CheriBSD provides compartmentalisation for cross-domain calls (cross-process and system calls). Watson et al. show that compartmentalisation with CHERI capabilities vastly outperforms process-based approaches, as there is no data flow overhead. Compartmentalisation can also be used to restrict access to system calls and other dangerous operations, including tampering with file descriptors [59].

Chisnall et al. use compartmentalisation to sandbox JNI code, eliminating data flow and domain crossing overheads, and enabling sealing (Section 5.6 discusses this in a Rust context), among other benefits [12]. Like Robusta, this includes Java security manager checks on native code. They include a detailed comparison of compartmentalisation to process-based sandboxing.

2.4.6 RustBelt: Verification of Rust's safety properties

One of Rust's goals is to provide a good combination of low-level control and performance with the high-level abstractions and safety. While the language and runtime have been designed with this in mind, the claimed safety properties are not clear in light of the frequent unsafe implementations of functions and data structures in the standard library [16].

Noting that the Rust language does not have formal semantics, Jung et al. define λ_{Rust}, a language modelled on a core subset of Rust. They proceed with a formalisation of this language, placing emphasis on lifetimes and borrowing [28]. These notions are crucial to Rust's temporal safety which underpins its claims to be a safe language. Further, they give a framework for proving the soundness of unsafe code, applying it to Rust primitives, including those that provide shared references and reference-counted memory. They also include a more detailed view of Rust's semantics than this work contains, as I focus on its implementation rather than formalisation.

2.5 Context

This chapter briefly introduced capabilities, which contrast with Rust's ownership and lifetime semantics in Section 3.2. It defined the basic mechanism and properties of the CHERI implementation of capabilities, which are necessary for understanding the rest of this work, especially the evaluation (Chapter 5); it then motivated the study of capabilities applied to safe languages in general, and Rust in particular. The survey of related work again contextualises this project by comparison to other capability mechanisms, and protection techniques used for other safe languages, from software fault isolation to sandboxing.

Chapter 3

The Rust Programming Language

3.1 Overview

This chapter gives an overview of Rust and the major differences from comparable programming languages. It focusses on semantic differences, rather than usability differences, although those also make a significant contribution to code safety.[1] I also discuss some techniques and patterns used by the Rust compiler and programmers for optimisation of Rust code.

For any future effort in porting Rust to CHERI, attention *must* be paid to pointer width and indices (Sections 3.3 and 4.3.1). *Currently, this effectively prevents Rust code from compiling properly for CHERI.*

This work refers to version 1.35 of the Rust compiler.[2]

3.1.1 The Rust programming language

Rust is billed as a fast and safe systems programming language. It uses both compile-time and runtime checks to prevent overflows, widely employing fat pointers for built-in data structures [10]. For example, the built-in Vector (`Vec`) stores data on its allocated capacity and actual length; a CHERI capability pointer to the same vector would also track the allocated capacity as a boundary. This redundancy suggests some room for optimisation.

3.2 Object ownership and borrow semantics

One of the guarantees Rust offers is that Safe Rust should never lead to dangling pointers or memory leaks. Yet it does not have a garbage collector, or expose memory management. Instead, the compiler couples strict ownership analysis with borrow and move semantics to determine the lifetime of an object. As such, some programs which avoid use-after-free or other memory safety bugs may be semantically invalid Rust.

[1] Usability differences include the lack of automatic typecasting, the explicit use of integer types (e.g. `i32`, or `usize` for array indices), and the built-in test harness.

[2] Commit `2210e9a`, nightly build of March 22nd, 2019.
Source: https://github.com/rust-lang/rust/commit/2210e9a6a99c4241d82e85ca71fd291d5ef91c7f.
Patches for CHERI compatibility: https://github.com/CTSRD-CHERI/rust/.

```
// Lifetime ends; 'v' becomes uninitialised in caller.
// Caller is returned a vector, which could be 'v'.
fn take(v: Vec<i32>) -> Vec<i32>;

// Borrowing references from the caller:
// 'w' is defined for the caller afterward and must not change
// 'x' is defined for the caller afterward but could have changed
fn borrow(w: &Vec<i32>, x: &mut Vec<i32>);
```

Figure 3.1: Function signatures for passing a `Vec` to a function. To prevent data races, only one code block can contain a mutable (exclusive) reference to a variable, like `x`, at a time. Multiple immutable (shared) references (e.g. `w`) are allowed.

3.2.1 Ownership semantics

Instead of passing objects by duplicating pointers, a function in Safe Rust will either take ownership of an object, or borrow it mutably or immutably. Figure 3.1 shows some function signatures for moving ownership or borrowing. In contrast to C, the emphasis is not on access, but the nature of the access.[3] As such, pointers cannot be casually duplicated[4] to cause data races or temporal unsoundness generally.

Ownership and borrowing are the foundations of Rust's temporal safety guarantees. They make object provenance clear, preventing bugs including use-after-free at compile time.

3.2.2 Caveat on memory leaks

An exception to the memory leak guarantee is the exposure of `mem::forget` in the core library as a safe function. This allows a programmer to end an object's lifetime without deallocating it. Such a pattern might be used to avoid double-frees in circular data structures. Nevertheless, invoking this function can create a leak, as can a knot-tied reference-counted data structure, though knot-tying is made difficult in light of the single mutable reference rule. Section 5.4.4 gives an example of a memory leak in Safe Rust.

3.3 Definitions of pointers and indices

In Rust, a `usize` is defined to be a pointer-sized integer, conventionally equivalent to C's `uintptr_t`, which *may not be equivalent to* `size_t` [63]. The width of a `usize` derives from the data layout's pointer width, 128 bits for a CHERI capability. Crucially, `usize` is used as the index size in Rust, for indexing into arrays, structs, and all objects, and is hence passed to LLVM instructions such as `getelementptr`.

3.3.1 Implications

Having a 128-bit `usize` means that the Rust compiler instructs LLVM to generate 128-bit indexed versions of intrinsics such as `memcpy`, in addition to indexing. However, not all bit widths for the `memcpy` intrinsic are supported by all targets [34], and CHERI supports a bit width of 64 but not 128.

As it stands, this is an implementation detail which is the consequence of the language semantics. Nevertheless, to target CHERI properly, Rust must either support pointer widths larger than the index size, or use pointer widths differently in code generation. Section 4.3.1 explores the implementation in more detail.

[3] References represent ownership of an object, rather than access to it.
[4] In Safe Rust.

3.4 Unsafe Rust

To provide Rust with more power, *Unsafe* Rust permits several additional actions [44]:

- Dereferencing a raw pointer
- Calling unsafe functions
- Accessing or modifying a mutable static variable
- Implementing an unsafe trait

While these seem fairly innocuous compared to abstractions in C, they impose significant restrictions on Safe Rust. It's important to note that unsafety need not necessarily come from the `unsafe` code block itself, but from the handling of inputs to the unsafe block. Section 5.2.2 shows a bug in a built-in data structure caused by incorrect computation before an `unsafe` block, leading to a buffer overflow.

3.4.1 Optimisation with Unsafe Rust

As seen in Section 3.5, unsafe optimisations are used to avoid redundant checks, whether they are duplicated or simply known to be within bounds. The core library uses unsafe code for unchecked conversions, or other unchecked indexing, like into a Unicode string at a known character boundary.

Another example is to override the default allocation strategy when initialising a vector with data: there is no need to fill the allocated space with zeroes or poison values if it is guaranteed to be overwritten before access. Naturally, there is scope for programmer error here: Section 5.2.1 discusses an off-by-one error in the standard library.

3.5 Array bounds checks

```
1  impl<T, I: SliceIndex<[T]>> Index<I> for Vec<T> {
2      type Output = I::Output;
3
4      #[inline]
5      fn index(&self, index: I) -> &Self::Output {
6          Index::index(&**self, index)
7      }
8  }
```

Figure 3.2: Like built-in arrays, Rust's `Vec` uses intrinsic indexing. The same bounds checks therefore apply to random indexing into a `Vec`, or any structure built on one.

For spatial safety, Rust implements runtime bounds checks to prevent out-of-bounds accesses. This may appear to harbour large overheads, although with optimisations many checks are elided. In fact, bounds checks mainly apply to random indexing, as seen in Figure 2.1 (page 7). They also apply when Rust's intrinsic indexing is used, through the `Index` and `IndexMut` traits. For example, Figure 3.2 shows that `Vec` uses this intrinsic, and therefore has bounds checks.

3.5.1 Optimisation by eliding checks

Figure 3.3 shows examples of bounds check elision in generated code, based on iterators and detecting when the length is checked. This detection is basic, as we see in Figure 3.4: it is more effective to use built-in iterators in general.

```rust
fn sum_iter(arr: &[i64]) -> i64 {
    let mut sum: i64 = 0;
    for x in r.iter() { sum += *x; }
    sum
}
fn sum_foreach(arr: &[i64]) -> i64 {
    let mut sum: i64 = 0;
    arr.iter().for_each(|x| sum += x);
    sum
}
fn sum_builtin(arr: &[i64]) -> i64 {
    arr.iter().sum()
}
fn sum_loop(arr: &[i64]) -> i64 {
    let mut sum: i64 = 0;
    let mut i: usize = 0;
    loop {
        if i >= arr.len() { break }
        sum += arr[i]; i += 1;
    }
    sum
}
```

```
1  sum_iter:
2      beqz    $5, .LBB0_4
3      nop
4      dsll    $3, $5, 3       ; i = arr.len() * 8
5      daddiu  $2, $zero, 0    ; retval = 0
6  .LBB0_2:
7      ld      $1, 0($4)       ; *arr
8      daddu   $2, $1, $2      ; retval += *arr
9      daddiu  $3, $3, -8      ; i -= 8
10     bnez    $3, .LBB0_2     ; until i == 0
11     daddiu  $4, $4, 8       ; arr += 4 (delay slot)
12     jr      $ra
13     nop
14 .LBB0_4:
15     jr      $ra
16     daddiu  $2, $zero, 0
```

Figure 3.3: Four ways to sum an array in Rust: the first three identically generate the shown MIPS assembler, and could be considered idiomatic Rust. The fourth generates nearly the same code: it uses an index of 1 instead of 8, allowing the omission of lines 3–4! It also omits lines 12–13, which appear to be redundant. Note that no bounds checks are present. Optimised code generated; comments added for ease of reading.

```
1  fn sum_checked(arr: &[i64]) -> i64 {
2      let mut sum: i64 = 0;
3      let mut i: usize = 0;
4      loop {
5          if i > arr.len() - 1 { break }
6          sum += arr[i]; i += 1;
7      }
8      sum
9  }
```

```
1  sum_checked:
2          daddiu  $sp, $sp, -16
3          sd      $ra, 8($sp)
4          sd      $gp, 0($sp)
5          lui     $1, %hi(%neg(%gp_rel(sum_checked)))
6          daddu   $1, $1, $25
7          daddiu  $gp, $1, %lo(%neg(%gp_rel(sum_checked)))
8          move    $6, $5
9          daddiu  $3, $5, -1
10         daddiu  $2, $zero, 0
11         daddiu  $5, $zero, 0
12 .LBB1_1:
13         sltu    $1, $5, $6
14         beqz    $1, .LBB1_4
15         nop
16         ld      $1, 0($4)
17         daddu   $2, $1, $2
18         daddiu  $5, $5, 1
19         sltu    $1, $3, $5
20         beqz    $1, .LBB1_1
21         daddiu  $4, $4, 8
22         ld      $gp, 0($sp)
23         ld      $ra, 8($sp)
24         jr      $ra
25         daddiu  $sp, $sp, 16
26 .LBB1_4:
27         ld      $1, %got_page(.L__unnamed_1)($gp)
28         ld      $25, %call16(core::panicking::panic_bounds_check)($gp)
29         jalr    $25
30         daddiu  $4, $1, %got_ofst(.L__unnamed_1)
31         break
```

Figure 3.4: How not to sum an array in Rust. Observe that `sum_checked` is nearly identical to `sum_loop` in Figure 3.3, save for the comparison on line 5. The compiler fails to detect that our accesses are safe due to the small manipulation of the variable storing the length. Thus an additional branch instruction appears on line 14. Optimised code generated; `panic_bounds_check` string substituted in place of mangled version.

Some common situations in which bounds checks are avoided:

Built-in binary search Checks avoided without unsafe code by providing more information to compiler: uses slices instead of indices [49].

Built-in slice equality Similar to **binary search**, providing more information to the compiler makes a different iteration strategy faster without unsafe dereferencing [51].

Built-in iterators The default iterator applying to arrays (Figure 3.3) uses the unsafe `get_unchecked` method[5] implemented for the `SliceIndex` trait. In theory, this *might* be susceptible to an off-by-one error, but such a mistake is easily discovered in unit tests. Section 3.4 covered how unsafe code is used for optimisation.

Built-in sorting Both built-in sorting implementations, timsort (stable) and quicksort (unstable) use the unsafe `get_unchecked` method. They also use unsafe `ptr` methods to swap and write elements.

Image processing Dröge demonstrates different optimisation strategies without resorting to Unsafe Rust, while still avoiding bounds checks and other unnecessary operations [17]. This is achieved through strategic assertions and optimal use of iterators and other built-ins. Note that similar optimisation through assertions may be fragile or made obsolete through updated code generation, and that optimal use of built-ins may require knowledge of the underlying (unsafe) implementations.

3.5.2 Removing bounds checks with dependent types

Finally, there is an experimental effort [52] to remove array bounds checks using dependent types, using the principles described by Xi and Pfenning [61] and applied to Rust by Beingessner [6]. However, this adds significant compile-time overheads, as it necessarily utilises type-, lifetime, and ownership checkers, which Rust runs before optimisations: this results in heavy computation to remove checks that are usually avoided anyway.

3.6 Summary

This chapter introduced some important semantic characteristics of Rust, and showed how it provides some of its memory safety guarantees. It then examined how Rust minimised the overheads of its guarantees, and the tools that programmers employ to do the same. Both of these inform the evaluation in Chapter 5.

Next, Chapter 4 considers the implementation of the Rust compiler and core libraries, and how they interact with the CHERI ISA. The `usize` (Section 3.3) in particular is a definition that has not translated well to the CHERI architecture, but is consistent with it, if interpreted carefully.

[5] This dereferences an array offset without checking bounds. Other `unchecked` methods include string conversions.

Chapter 4

Compiling Rust for CHERI

This chapter discusses key details in compiling Rust for CHERI, as opposed to other architectures. It details the strategy taken, thereby defining the scope of work in this project. To this end, I observe significant architectural differences that make compilation challenging, and consider design choices for this and future work porting Rust to CHERI.

4.1 Strategy

4.1.1 Host platform

One contribution of this project is an extension of the Rust compiler to target FreeBSD on CHERI128. This is achieved by cross-compilation from an amd64 FreeBSD host; other host platforms should work if they are able to compile the CHERI SDK.

4.1.2 Rust functionality on CHERI

This extension exclusively targets *pure capability mode*, where all pointers are capabilities [57]. This avoids the hurdle of dealing with multiple LLVM address spaces and pointer widths, not supported in the Rust compiler (Section 4.3.2).

In order to evaluate Rust on CHERI, I identified a minimal necessary subset of the standard library, listed below. Not included is the bulk of the Rust standard library, which provides filesystem and network support, general IO, interfaces to operating system synchronisation and threading, and others. I consider these not to have critical significance to capabilities in standalone Rust programs.

`libcore` for key definitions of traits and types, typechecking, compiler intrinsics, and other fundamental operations. It also includes support for calling through foreign function interfaces (FFI), Unicode and number formatting, etc. It could be thought of as a modern version of the standard C headers, apart from the core definitions it provides, which would be built into a compiler.

`liballoc` for heap allocation and built-in data structures and operations on them. `liballoc` also included two (patched) vulnerabilities in the Rust standard library, demonstrated in the evaluation (Sections 5.2.1 and 5.2.2).

Other parts of the standard library will compile for CHERI, but they are not as central to running Rust programs. I have not included them in order to focus on evaluating the Rust compiler and a core set of features.

Sections 4.3 and 4.4 detail modifications to the compiler and these libraries required to compile Rust programs.

4.1.3 Compiler optimisations

Optimisations are *not* used. Optimisations are not supported at this point as they cause the Rust compiler to invoke LLVM APIs with values not supported by the CHERI backend, such as offsets of 128 bits. Section 4.3.1 discusses related issues.

4.2 The Rust compiler and LLVM

Rust uses LLVM as a backend for compilation. The CTSRD project maintains a fork of LLVM with support for the current CHERI implementation, extending the MIPS backend to support CHERI targets, and generic code to handle capabilities [57].

4.2.1 Incompatibilities with the LLVM backend

Due to the pointer width and index size divergence (Section 3.3), two incompatibilities were found with the CHERI LLVM backend. These have been fixed in the current version.

- Rust may attempt to generate an unusual pointer offset when calling or returning from functions.[1]
- Rust sometimes attempts to generate unaligned loads. This is not possible in the pure capability ABI.[2]

Both of these were identified as a result of extensive debugging and code generation failures, requiring analysis across both the Rust compiler and LLVM.

The Rust project also maintains a fork of LLVM, with minor changes, and provide 'known good' combinations of LLVM and Rust compiler commits. This is a good starting point to find a CHERI-supporting LLVM commit which will compile the Rust compiler.

4.2.2 Compiler usage

Enabling the `cheri-unknown-freebsd` target when building the Rust compiler necessitates compiling the complete Rust standard library for CHERI, which is not possible at this stage. The default package manager, Cargo, does not support this situation and cannot be used. I identified and used two workflows for compiling programs:

Integrated compilation with Xargo

Xargo describes itself as a "sysroot manager", meaning that it will (cross-)compile versions of the libraries for target platforms. This is useful where binary releases do not exist for those platforms; it allows the use of modified libraries, and in particular it does not require that the entire standard library is compiled.

By default it only compiles `core`; use of `alloc` is specified through `Xargo.toml`. This approach is recommended, although it can make code generation errors difficult to debug.

[1] Fix accessible at https://github.com/CTSRD-CHERI/llvm-project/commit/39cfdf711a759a4799bd32f50af07f5f6f43c987; thanks to Alexander Richardson.

[2] Fix accessible at https://github.com/CTSRD-CHERI/llvm-project/commit/d8e6acf3d4094270cd55b70341a6f9c4d032db81; thanks to Alexander Richardson.

Note that Xargo *does not expect dependencies (e.g.* `libcore`*) to change!* Therefore, it does not always recompile a "sysroot" after modifications. Invoking it with a build tool might be advisable.

Scripted compilation with `llc` and Clang

This is the traditional compilation-and-linking process.

1. First, libraries are compiled as Rust libraries (`.rlib`) and dynamic libraries (`.so`). This is done by using `rustc`, not Cargo.
2. Then, Rust programs are compiled to LLVM IR (again with `rustc`), manually specifying the Rust libraries for definitions. One can compile to object code directly, but this step can be helpful for debugging and ensuring that all the desired target attributes and flags are passed in the next step.
3. Next, `llc` is used to compile the LLVM IR to object files.
4. Finally, Clang links against the CheriBSD sysroot.

I provide sample Makefiles which use this process for the test programs in this project, including appropriate command-line arguments.

4.3 Changes to the compiler

The `cheri-unknown-freebsd` target has several differences to most existing targets in the Rust compiler. This section documents these differences, their implications, and the modifications required to compile Rust for CHERI.

4.3.1 Pointer width of 128 bits

Differences

Under pure capability mode in CHERI128, all pointers are 128-bit wide capabilities. The Rust compiler has built-in support for 16-, 32-, and 64-bit pointers, through compile-time macros. The pointer width is used to determine the `usize`[3] (Section 3.3), offsets, and sizes used for pointer operations.

Choices and implications

The 128-bit pointer width cannot be changed.

The Rust compiler has two definitions of the target's pointer width: the `target_pointer_width value` in the target specification (Section 4.3.3), and the LLVM data layout string. These values *must* agree to avoid alignment issues.

In any case, the value derived from data layout string is used in more than 50 locations in the compiler, to determine the index size and pointer alignment, and for code generation among others. This includes which index size to use when calling LLVM intrinsics, such as `memset`, and what size integer to provide for an `inttoptr` call; CHERI only supports 64-bit integers for all these cases.

The immediately obvious solution is to support a 64-bit index size, but this is met with a non-functional compiler. As Section 3.3 noted, the Rust language defines the `usize` to be equal to the pointer width. This assumption is sufficiently widespread in the compiler to be impractical to change within the scope of this project. However, if this definition is changed, or the definition of `usize` for typechecking

[3] The maximum index size.

compiled programs can be distinguished from the usage of index sizes for code generation, then the changes in the next section may not be necessary.

Changes and limitations

This meant defining a `usize`, and exposing a 128-bit integer type from LLVM as suitable for this value. This is not a functional problem: while 128-bit integer usage is not natively supported on CHERI, it is merely slower and LLVM will generate the correct instructions. However, the overhead is significant, as seen in Figure 4.1.

```
bounds_usize128:
        xor     a5,a2,a0        ; xor upper bits of idx, len
        sltu    a6,a2,a0
        xori    a6,a6,0x1       ; cmp upper bits
        sltu    a7,a3,a1
        xori    a7,a7,0x1       ; cmp lower bits
        movz    a6,a7,a5        ; use lower bits if upper bits eq
        bnez    a6,10510        ; panic
```

```
bounds_usize64:
        sltu    v1,a2,a1        ; idx < len ?
        beqz    v1,1051c        ; panic
```

Figure 4.1: Top: bounds check with 128-bit `usize` under CHERI; bottom: bounds check with 64-bit `usize` under MIPS64. Bounds comparison code when calling into functions which index into a given slice; all code before and after checks omitted.

The other change is to truncate or extend integer types before calls to LLVM intrinsics and pointer operations. An example of the former is `memcpy`: if indexed by a 128-bit length, LLVM silently omits the copy, as it is not defined in the CHERI backend. This change is problematic, as it is possible to miss out the truncation or extension for some intrinsics, then be unaware that the compiler has omitted them. Pointer operations are more limited: we need only concern ourselves with `inttoptr` and `ptrtoint`. Here, we take performance penalties from generating excess instructions and 128-bit integer operations.

4.3.2 Address spaces

Differences

CHERI uses LLVM address space 200 for capabilities, and 0 for non-capability pointers. Supporting pure capability mode only requires use of address space 200.

However, the Rust compiler currently only generates code that uses address space 0, the default address space in LLVM. There is no support for multiple address spaces within or across compilation units either.

Choices and implications

With pure capability mode, only one address space is required, thus the latter problem is avoided. In any case, there is no requirement to support legacy code and pointer manipulation idioms in the scope of this work, thus little reason to support the use of untagged pointers, and hence hybrid mode.

Changes and limitations

The main change was to make Rust aware of LLVM address spaces other than 0; it already specified this as the default. Previous work to support Rust on the AVR platform already (in principle) supported address space 1 for functions, so the changes required were to determine the pointer width correctly from the LLVM data layout string, and to ensure allocations were made to the correct address space, again using the data layout string.

Based on the Rust codebase, it could be difficult but far from impossible to support multiple address spaces in Rust. A far bigger challenge, however, is to support code generation for multiple pointer widths: it will therefore be impractical to consider compiling Rust programs in hybrid mode unless this changes.

4.3.3 Targeting CHERI

Differences

The CHERI backend in LLVM is an extension of the big-endian 64-bit MIPS architecture. To target FreeBSD, it has the triple `cheri-unknown-freebsd` .

Choices and implications

In this case, the question is not so much what to change, but where to change it. For some changes, such as specifying the `cheri128` CPU (the default being CHERI256), the Rust target specification files had relevant fields.

In other cases, such as specifying the `purecap` ABI (as opposed to the MIPS N64 ABI), no such mechanism was present in the compiler. Here, either fields could be added to the target specification, or made locally to their usage.

Changes and limitations

A target specification was added to the Rust compiler, specifying the pure capability ABI, and the CHERI128 CPU.

Instead of adding more fields to the target specification structure and code to parse these fields, I decided to make changes closest to where the data would be used. This minimised code changes. However, this means that when targeting CHERI, pure capability mode would always be used: I do not consider this an issue because the CheriBSD can also run programs compiled with the MIPS N64 ABI, which can also be compiled by the Rust compiler. This choice should be revisited by if CHERI is to be made an official target in the Rust compiler, as it is slightly unergonomic, even if the ABI is not typically configured, and is not specified for other targets.

4.4 Changes to core libraries

The changes to the compiler in Section 4.3 were not sufficient to compile Rust programs. Tests and benchmarks were not run against these libraries.

My attempts to compile the core libraries were cursory, only investigating them as far as their functionality was required to run test programs. Therefore, it is possible that there are problems which I have not noticed. It is also possible that these changes might now be undone without affecting the success of compilation: this is untested.

4.4.1 `libcore`: formatting

Three formatting methods were modified:

- A string writing method, used for writing text to files or the console, was changed not to format strings before writing them. Thus variables cannot be printed.
- A pointer formatting method: this now prints a dummy string instead of an address. This was previously broken by invalid `ptrtoint` calls.
- Number formatting: a method to print numbers in arbitrary radices. Oddly, some of the pattern-matching code refused to compile.

These methods are not relevant to my evaluation as I have used the C `printf` function for printing and formatting.

4.4.2 `libcore`: UTF-8 validation

The UTF-8 validation routine fails to compile into the core library: I did not attempt to debug this, instead removing references to it. Again, this is not relevant to my evaluation as I have not used Unicode.

4.4.3 `libcore`: `memchr`

The `memchr` implementation is similar in functionality to its C version. It performs pointer manipulation and bitwise operations, and did not compile due to an iterator issue.

4.4.4 `liballoc`: macro invocation

Only a minor change was made: when returning an empty vector, to use `Vec::new()` instead of the macro `vec![]`. This was required because the macro was not in scope of the library as I debugged it.

4.5 Summary

This chapter recorded and explained the strategy taken to evaluate Rust on CHERI, which defines the scope of this project. It documented techniques and processes which may be of use to future implementers, as well as specific differences in the CHERI architecture that lead to implementation conflicts with Rust. For example, Section 4.3.1 explains implementation details of the Rust compiler arising from the definition of index sizes, as well as the choices made to resolve the issue for this project.

The strategy taken and compromises made set the stage for Chapter 5, where I evaluate the application of CHERI capabilities to Rust, and Rust guarantees to CHERI. They inform the evaluation and define its limitations.

Chapter 5

Evaluation

5.1 Objectives

Chapter 4 defined the scope of this evaluation, and discussed the compromises that were made to compile Rust's core libraries for CHERI.

This chapter contains the main evaluation of CHERI capabilities in a safe language, Rust. An overview of the focusses of each section:

Section 5.2 Previous vulnerabilities in the Rust standard libraries. It shows that bugs in Unsafe Rust are not substantially different from those in C, and includes microbenchmarks demonstrating that CHERI capabilities provide an effective mitigation.

Section 5.3 The implications of the Rust language and toolchain for CHERI. How Rust's guarantees complement capabilities, and why Rust is an ideal candidate for porting to CHERI due to its memory model and performance.

Sections 5.4 to 5.6 The implications of CHERI capabilities on the Rust language. How Rust's guarantees can be enforced even in Unsafe Rust, and extended to cover cross-domain function calls, without changing language semantics.

Section 5.7 The connection between Unsafe Rust and undefined behaviour, referring to underlying assumptions in the language. How CHERI capabilities rule out forms of undefined behaviour in Rust, by guaranteeing a hardware exception on violation.

Section 5.8 A method to minimise the memory overheads of capability protection by using the hybrid ABI, which allows both capability and non-capability pointers in a single program. A consideration of the conditions that allow this and the guarantees it provides.

Section 5.9 An argument for reinterpreting the previously-discussed `usize` definition, with respect to its usage in the language. Reference is made to community consensus, existing definitions, and types in other languages.

Section 5.10 General concerns and implications when introducing capabilities to safe languages; when they benefit the most and least.

Section 5.11 Chapter summary.

5.2 Errors leading to memory violations in Rust

Despite Rust's attention to safe language design, programmer error is still a rich source of potential vulnerabilities. While the Rust community is generally conscious about memory safety and security, nowhere is this more true than with the language and compiler developers.

This section covers four entirely unrelated memory safety flaws discovered in the Rust standard library, all preventable with CHERI capabilities. Where relevant, demonstrative microbenchmarks were performed on either CheriBSD on Qemu-CHERI with 128-bit capability pointers, or FreeBSD 11.2 (amd64) for the non-capability comparison. In all cases, the Rust compiler was as described in Chapters 3 and 4, specifying the target as appropriate.

5.2.1 Pushing to a `VecDeque`: off-by-one error leads to out-of-bounds write

Cause

Rust's `VecDeque` is a circular data structure stored on a buffer. When this buffer is expanded, elements stored at its (former) end must be moved to its new end. This error was caused by incorrectly using the public capacity, accessed by `self.capacity()`, instead of the private (raw) capacity of the buffer, `self.cap()` (Figure 5.1) [18, 26].

```
1  impl<T> VecDeque<T> {
2      #[inline]
3      pub fn capacity(&self) -> usize {
4          self.cap() - 1
5      }
6
7      pub fn push_back(&mut self, value: T) {
8          self.grow_if_necessary();
9
10         let head = self.head;                          // PRE: 0 <= head < len
11         self.head = self.wrap_add(self.head, 1);
12         unsafe { ptr::write(self.ptr().add(head), value) }  // unchecked
13     }
14 }
```

Figure 5.1: The public-facing definitions for a `VecDeque`'s capacity, and pushing to the end. It will not escape the reader's attention that the `capacity()` might be mistaken for the `cap()`, a likely cause of this error. These definitions are current in Rust. `push_back` annotated with implicit precondition.

Subsequently, the pointer to the deque's `head` would point to the address immediately after the buffer, rather than its start. Pushing to the back of the deque then attempts to write to the address after the buffer, rather than within it.

This appears to have been caused by the similarity of the method names leading to an off-by-one error.

Demonstration microbenchmark

I checked that this spatial violation is caught by CHERI capabilities, but is not detected on x86 FreeBSD. Sample code is shown in Figure 5.2.

Had another data structure been allocated after the deque, its start would have been overwritten by the pushed element. However, I was unable to force this situation without modifying the allocator for the purpose.

```
1  fn main() {
2      use alloc::collections::VecDeque;
3
4      let mut deque = VecDeque::with_capacity(31);
5      deque.push_front(5);
6      for x in &deque { prntf!("%d␣", *x); } prntf!("\n");          // '5'
7      prntf!("%d,␣%d", deque.head as u64, deque.tail as u64);       // '0, 31'
8
9      deque.reserve(30);                              // head should not change
10     prntf!("%d,␣%d", deque.head as u64, deque.tail as u64);       // '32, 31'
11
12     deque.push_back(6);                             // CHERI: length violation!
13     for x in &deque { prntf!("%d␣", *x); }          // '5 0' <- not '5 6'!
14     prntf!("\n%d,␣%d", deque.head as u64, deque.tail as u64);     // '1, 31'
15 }
```

Figure 5.2: A program which writes beyond the end of an allocated `VecDeque` buffer. An off-by-one error in `reserve` results in an unsound call another function, updating `head` to 32. Under x86 without capabilities, the program terminates normally, printing output as shown in the comments.† With CHERI capabilities, execution continues until line 12, which traps due to a length violation.
† The `prntf!` macro is *not* built-in; I define it for convenience. Using the same syntax as the C `printf` function, it avoids considerable unsafe boilerplate while enhancing readability.

Observations

This scenario is a plain example of a bounds violation that is prevented by capabilities. Had the underlying implementation not used an unchecked write (line 12 of Figure 5.1), this would also have been prevented by Rust's bounds checks, as the `VecDeque` is backed by a vector. This is an interesting lesson: bounds checks are useful not only for the end-programmer, but also (presumably meticulous) language developers.

Note that whether this exceeds the bounds is implementation specific! The existing implementation will reserve space for 32 elements here, but if more space had been reserved by `with_capacity` or its underlying code (e.g. 64 elements), the object bounds would have been correspondingly wider. In that case, it would only be detected if the value was written using the built-in bounds-checking indexing, although the pushed value would still disappear.

5.2.2 Slice `repeat`: integer overflow leads to buffer overflow

Cause

This flaw arose from an unchecked write, one pattern of optimisation using Unsafe Rust covered in Section 3.4. It occurs in the `repeat` function on slices (Figure 5.3), which returns a vector containing a slice repeated as specified by the parameter. Here, a buffer overflow occurs when the length of returned vector would overflow the target's `usize`.

This appears to be a simple integer overflow translating to a buffer overflow. Note that this overflow will be caught when compiling Rust code in debug mode, as integer overflow checks will apply to all arithmetic; release mode instead uses two's complement and is *not* undefined.

```
1  impl<T> [T] {
2      pub fn repeat(&self, n: usize) -> Vec<T> where T: Copy {
3          if n == 0 { return Vec::new(); }
4
5          let mut buf = Vec::with_capacity(self.len() * n);
6          buf.extend(self);
7          {
8              let mut m = n >> 1;
9              while m > 0 {
10                 unsafe {
11                     ptr::copy_nonoverlapping(
12                         buf.as_ptr(),
13                         (buf.as_mut_ptr() as *mut T).add(buf.len()),
14                         buf.len(),
15                     );
16                     let buf_len = buf.len();
17                     buf.set_len(buf_len * 2);
18                 }
19                 m >>= 1;
20             }
21         }
22         // omitted: copy into the remainder of the vector
23     }
24 }
```

Figure 5.3: Rust's slice `repeat`, from `alloc::slice`. Parts omitted for brevity. This code attempts to write beyond the end of a buffer. The error is in line 5; it was fixed by checking the multiplication against integer overflow.

Demonstration microbenchmark

I did not demonstrate this by calling the `repeat` method as given: as noted below, attempting to copy more than 2^{64} elements is difficult to do in a controlled way. Instead, Figure 5.4 shows the code used to demonstrate the effects of this bug.

I chose values carefully to demonstrate the overflow on the test machine; other values may work depending on the operating system and allocator.

Observations

This is difficult to exploit, as it would require a long slice or a large number n of repetitions, increasing the chance of a segmentation fault. An exploit would probably have to interrupt the write before too many iterations of the loop, and either spawn a new process or stop the thread executing this loop before the operating system stopped it.

5.2.3 Out-of-bounds indexing into a reversed slice

Cause

An unhandled unsigned integer wrap-around could lead to out-of-bounds slice indexing through its reverse iterator [22]. Figure 5.5 shows the relevant implementation, defined in the core library.

In this situation, the assumptions about the underlying iterator implementation are not obvious, and thus not considered when writing this method.

```
1  fn main() {
2      use core::ptr::copy_nonoverlapping;
3
4      let s: [i64; 3] = [1, 2 ,3];
5      let reps = 6148914691236517207;              // (2 ** 64 + 5) / 3
6
7      // Simulate usize == u64 on CHERI, where usize == u128
8      let mut buf = Vec::with_capacity((s.len() * reps) as u64 as usize);
9      prntf!("capacity: %d\n", buf.capacity() as u64);    // 5
10     buf.extend(&s);
11
12     let mut v: Vec<i64> = Vec::new();
13     v.extend(&s);                                // Manipulate allocation to
14     v[0] = -1;                                   //  ensure the buffers are
15     v.push(-4);                                  //  allocated nearby
16     for x in &v { prntf!("%d ", *x); } prntf!("\n");    // -1 2 3 -4
17
18     {
19         let mut m = 8;      // 8 <= n >> 1; copy enough times to reach 'v'
20         while m > 0 {
21             unsafe {
22                 copy_nonoverlapping(             // generates memcpy
23                     buf.as_ptr(),                // out of bounds on 1st iter
24                     (buf.as_mut_ptr() as *mut i64).add(buf.len()),
25                     buf.len()
26                 );                               // CHERI: length violation
27                 let buf_len = buf.len();
28                 buf.set_len(buf_len * 2);
29             }
30             m >>= 1;
31         }
32     }
33     for x in &v { prntf!("%d ", *x); } prntf!("\n");    // 1 2 3 1
34 }
```

Figure 5.4: Demonstration of how an integer overflow can lead to a buffer overflow when repeating a slice. Three iterations of the copy are performed (lines 20–31) to ensure v is overwritten on the non-capability machine; with capabilities this traps on the first iteration.

```
impl<I> RandomAccessIterator for Rev<I>
where
    I: DoubleEndedIterator + RandomAcc,
{
    #[inline]
    fn idx(&mut self, index: usize) -> Option<<I as Iterator>::Item> {
        let amt = self.indexable();
        self.iter.idx(amt - index - 1)
    }
}
```

Figure 5.5: Previous implementation of indexing into a reversed slice. If `amt - index - 1 < 0`, the index wraps, attempting to access an index larger than the indexable region. This may be unsafe depending on the underlying implementation. This was fixed by checking that `amt > index` before indexing, returning None otherwise [21].

Observations

A very large index (i.e. close to `usize::MAX`) could be passed to the reverse indexing function to get values slightly beyond the end of the slice. As 'slice' suggests, they frequently represent a view into a larger slice, so this could reasonably be expected to be defined.

Whether this is possible is implementation-dependent; if the reverse `idx` is expected to handle bad values, then the underlying `idx` should arguably also handle them. If anything, this example and its resolution is an example of the safety awareness of Rust developers, rather than a fundamental bug.

5.2.4 Iterator method violates Rust's uniqueness of shared references

Cause

An iterator method in the core library should return a mutable slice from a mutable iterator. Instead, it returns a mutable slice for any iterator. Figure 5.6 shows that it does this by calling an unsafe method which operates on raw pointers [8].

```
impl<T> IntoIter<T> {
    pub fn as_mut_slice(&self) -> &mut [T] {
        unsafe {
            slice::from_raw_parts_mut(self.ptr as *mut T, self.len())
        }
    }
}
```

Figure 5.6: Built-in method returning a mutable slice for a (mutable) iterator: except it accepts immutable iterators also. This violates Rust's temporal guarantee that mutable references are never shared.

This bug appears to be caused by copying and pasting the Iterator `as_slice` method when writing `as_mut_slice`. A demonstration of the fallibility of programmers and motivator for safe syntax, neither the language nor the runtime helps in this situation.

Observations

This is a clear violation of one of Rust's temporal safety guarantees, that mutable references must not ordinarily be shared. While this is not strictly undefined behaviour, attempting to mutate the returned object is, defeating the point of getting a mutable slice.

Though this error leads to temporal unsafety, this could be prevented by passing a read-only capability for an immutable borrow. Section 5.5.1 discusses this in more detail.

5.3 Implications of Rust semantics for CHERI targets

Capabilities are typically seen as a mechanism to improve the security of languages and runtimes. It is therefore slightly unusual to consider how a language might improve capabilities. Nevertheless, this section considers how Rust complements CHERI capabilities, presenting a contrast to other safe languages.

5.3.1 Ownership gives complementary temporal safety

One of Rust's main objectives and early selling points was *fearless concurrency*. It uses its ownership model and type system to manage temporal memory safety [44]. These mechanisms prevent dangling

pointers or use-after-free in Safe Rust (Section 3.2), although the use of raw pointers in Unsafe Rust can bypass these checks.

By contrast, CHERI's initial focus was on spatial integrity, only later moving to consider temporal safety by means of tagging. Temporal safety is provided by marking capabilities as *local* or *global*, which restricts the flow of capabilities and thereby preventing their leakage [59].

However, this provision only yields atomic pointer updates and identifiability of pointers; it is not precise enough to guarantee safety across thread or process boundaries, such as foreign function interface (FFI) and system calls [14]. While Rust does not provide temporal protection against other processes misusing its resources, its ownership model guarantees that a thread which has yielded a resource does not attempt to modify it while it has been lent out. Likewise, the ownership model prevents similar conflicts within a concurrent Rust program, complementing CHERI's spatial integrity.

Note: Section 5.6 discusses how CHERI capabilities can be used to make Rust's FFI or cross-process calls safer.

Temporal safety under CHERI

While not inherent to capability systems, CHERI enables a form of revocation, preventing dangling pointers from being dereferenced. See Section 5.4.4 for an example of how revocation can be useful in Rust.

5.3.2 Stronger pointer provenance model in Rust

With a fully-functional Rust compiler for CHERI, porting Rust programs and crates should be straightforward. In particular, common compatibility issues in CHERI due to pointer provenance [14] do not apply in Safe Rust, as the compiler tracks object ownership comprehensively.[1]

Due to Rust's lack of a specification and documented undefined behaviour, programmers are discouraged from performing such capability-incompatible pointer manipulation. It is unclear if problematic pointer manipulation is currently undefined in Rust, but community discussions suggest that it might be in future [27]. As such, unsafe code should not rely on patterns that break provenance analysis in these ways.[2]

5.3.3 Safer code patterns yields easier porting to CHERI

Rust has been designed to guide programmers toward writing safe code, and to highlight potentially dangerous operations. Programmers have less cause to perform unusual or *clever* pointer manipulation, writing code that does not immediately compile for CHERI. Section 4.4 documented the limited incompatibilities in the core library; most changes affect the compiler.

Nevertheless, one should recognise that the difficulty of porting C programs to capability architectures and CHERI in particular is not thought to be too onerous [56]. This must be weighed against the prospect of updating the Rust compiler.

5.3.4 Comparable performance to C

While Rust possesses strong safety features, it also provides good performance. For this reason, it has garnered attention in applications as diverse as astrophysics (favourable comparison to Fortran [9]), GPU programming (comparable to handwritten and domain-specific language generated OpenCL [20]),

[1] An exception is 'pointer shape'; see Section 4.3.1.

[2] For example, the XOR linked list *might* not be implementable in Rust without exhibiting undefined behaviour.

and garbage collection (comparable to C [33]). The Debian project maintains a set of toy benchmarks run against user-submitted programs in diverse languages [19], showing that Rust performs as well as C and C++ under their testing problems and environment. All three come well ahead of the next 'safe' languages, Java and Go.

These suggest that Rust is capable of matching C under some circumstances while being safer, making it a good choice in general. If CHERI implementers are keen on augmenting hardware safety features with a speedy, low-runtime language, then Rust makes a sensible choice, especially for embedded platforms.

5.3.5 Larger pointer size can be offset by removing redundant bounds information

Currently, any memory slice stores a pointer to the relevant object and its length [10]. As bounds information can be derived from a CHERI capability pointer, it is redundant to store the length. Instead, a `CGetLen` instruction could retrieve the bounds information when needed. The overhead of a 128-bit pointer can therefore be offset by the removal of a 64-bit length variable, negating the memory overhead for slices which would apply in other languages.

Impact on data structures

Associating bounds information with pointers is a natural pattern for slices, which have one fixed bound. However, vectors have different bounds for the allocated space and the initialised indices. Currently, these are known as the *capacity* and *length* of the vector respectively. This tension can be resolved in two ways, both requiring capability pointers to the allocation[3]:

Initialised length variable This is similar to the current implementation, where the capacity variable can be dropped. This saves space, as a length variable would be a `usize` (64 bits), where a capability would be 128 bits. However, bounds checks will be required for indexing into vectors, hence could not be completely removed from Rust compilation (to CHERI).

Slice capability pointer A natural solution: this squares with Rust's type system, where indexing is provided by the `SliceIndex` trait implemented for slices. However this requires more space, and it is unclear if deriving a new slice capability for each push/pop operation on the vector will be faster or slower than updating a length variable. This should not affect concurrency when popping as CHERI capabilities support atomic operations: pushing would require the slice capability to be derived from the allocation capability, which could pose a problem.

Memory overhead of CHERI capabilities

The memory traffic overheads of CHERI capabilities for diverse C benchmark programs is within 5–10% [23], with the greatest impact on pointer-heavy workloads. In Rust, some overhead will still apply: capability pointers are always larger, and bounds checks are only occasionally saved. The impact on cache usage is unclear, but it is not unreasonable to suggest that overheads will be at most as large as under equivalent C workloads. Section 5.8 outlines a strategy to reduce the memory impact of CHERI capabilities in Rust.

5.4 Spatial integrity in Rust from CHERI capabilities

This section explores how memory safety can be improved in the Rust compiler and runtime by introducing CHERI capabilities. Section 5.7 instead considers how capabilities can be used to reduce the danger posed by undefined behaviour, enabling changes to the semantics of the language.

[3] Monotonicity implies that the full allocated space must be continuously pointed to; otherwise it is leaked.

5.4.1 Mitigation of traditional vulnerabilities

One class of attacks that Rust aims to mitigate is buffer overflows. In Safe Rust, bounds checks are inserted (Section 2.3.1) to prevent out-of-bounds accesses and writes.

Nevertheless, as Section 3.4 highlights, unchecked accesses are sometimes used for optimisation, by reading or writing using raw pointers. These unsafe operations are just as susceptible to programmer errors as their C analogues, giving rise to the bugs discussed in Sections 5.2.1 and 5.2.2. Capabilities can combat this by preventing out-of-bounds accesses.

5.4.2 Bounds checks removal

With capabilities, bounds checks can be removed altogether: instead of the default panic, a hardware interrupt will occur with each attempted out-of-bounds access. Most Rust programs do not handle panics, instead crashing the program: this is the default behaviour. If desired, CHERI length violations can be caught and the panic handler invoked, rather than terminating the program directly. Therefore the CHERI mechanism is fully compatible with the existing panicking framework.

This is a low-impact improvement, as most bounds checks are elided with compiler optimisations enabled; see Section 3.5 for examples where checks are already optimised. However, they are unavoidable in some situations, such as random array accesses.

5.4.3 Sub-object bound enforcement

In Rust, *slices* can be created from continuous segments of an iterable. Slices can be passed to functions, only allowing them to refer to part of an array, or a *subslice*. Using pointer manipulation, such a function could access indices of the array which are not part of the slice. This is not undefined behaviour as the pointer would point to a valid part of the object, although it must be done in Unsafe Rust.

With CHERI capabilities, a subslice could be passed as a pointer with bounds restricted to the relevant segment of the array. It would then be impossible for a function to access the other elements unexpectedly, again enforcing the principle of least privilege. Additionally, this can enforce Rust's temporal protection when splitting slices with the `split_at_mut` method.

Likewise, this can apply to struct members, generalising the protection to objects with known continuous layouts. The cost of doing this is similar to that of enforcing immutability in Section 5.5.1.

5.4.4 Use-after-free elimination in Safe Rust

As we have discovered, spatial integrity is not an absolute guarantee in Rust. Unsurprisingly, neither is temporal safety, even in *Safe* Rust!

Consider the following example from the documentation, a simplified version of the reference-counted container type, `Rc`, and its simplified `Drop` implementation (Figure 5.7). By overflowing the reference count (using `mem::forget` if memory usage is an issue), one can cause the pointed-to object to be deallocated, even with outstanding references, defeating the reference counter. This creates a use-after-free bug [41].

This can be resolved by revocation, for which CHERI capabilities provide a foundation by virtue of monotonicity and memory tagging. Revocation prevents programs from reading or modifying memory which has been deallocated, or more importantly reallocated, another common source of vulnerabilities. Note that there are performance overheads associated with revocation [58], and Rust's temporal guarantees mean that its utility is largely constrained to unsafe code. This is discussed further in Section 5.7.3.

```
impl<T> Drop for Rc<T> {
    fn drop(&mut self) {
        unsafe {
            (*self.ptr).ref_count -= 1;
            if (*self.ptr).ref_count == 0 {
                // drop the data and then free it
                ptr::read(self.ptr);
                heap::deallocate(self.ptr);
            }
        }
    }
}
```

Figure 5.7: A potential use-after-free bug, if `ref_count` overflows. This may occur in Safe Rust, as `drop` methods are always safe: the unsafety is encapsulated here, and also in `mem::forget`, which would be used to overflow the count. Example from the Rust documentation [41].

5.5 Capability sealing to protect Rust objects

Capability sealing can prevent a capability from being used to mutate or dereference the memory it refers to, as described in Section 2.2.3. This section describes how the Rust runtime can use this mechanism to enforce program invariants.

5.5.1 Preserving object immutability in Unsafe Rust

Modifying an immutable object is undefined behaviour in Rust, and should be prevented by the compiler in Safe Rust. The only exception to this pattern is the `UnsafeCell`, modifications to which are explicitly excluded from being undefined behaviour. There, the programmer is expected to implement an object safely without causing concurrency bugs.

However, the compiler does not prevent this invariant from being violated in Unsafe Rust, where the programmer can essentially bypass the type system by casting raw pointers. There have been bugs in the standard library [8] (Section 5.2.4) resulting in mutable references to supposedly immutable objects.

Since mutating these is undefined, it is reasonable to protect them from being written to, which can be accomplished with capabilities. This enforces the principle of least privilege on shared references. Sealing can be used to prevent mutation.

Note on differences from C

Enforcing C's `const` in hardware proved problematic for CHERI in the past, due to functions like `strchr` (C), which derive a non-`const` (mutable) pointer from an immutable one [57].

```
char *strchr(const char *s, int c);
```

The proposal of deriving a read-only capability for each immutable reference in Rust does not suffer this problem, as it is not possible to derive a mutable reference from an immutable one. Indeed, this idiom does not occur in Rust as immutability is conveyed by the function signature.

5.5.2 Preserving object immutability across FFI boundaries

No amount of type-checking can prevent a foreign function from modifying a supposedly immutable borrowed object, since they use raw pointers and are not bound to Rust's semantics.

By passing a capability without write permissions to a function, this particular cause of undefined behaviour can be eliminated. This supports the use of *least privilege* even across foreign function interfaces.

5.5.3 Protecting data from callback functions

Another situation that benefits from least privilege are callbacks from FFI code. In this scenario, an FFI function might receive a pointer to data it is expected not to access, but instead merely to return the pointer to the original process at the correct time.

This can be enforced through capability sealing. The foreign function would be unable to dereference the object, but the capability could be unsealed by the original program to regain access to the original data [58]. This allows less trusted code to handle Rust objects while still guaranteeing integrity and confidentiality.

5.5.4 Fine-grained object protection

As an extension of the previous sections, not only can specific objects passed across unsafe or FFI boundaries be protected, but also objects transitively accessed via those objects. CHERI capabilities include an *object type* field, on which sealing and unsealing can be predicated; this can be used to protect a 'confidential object' type (and derivable types) from abuse by Unsafe Rust or FFI functions. This can easily be marked using a Rust trait.

To motivate this protection, consider a library written in Safe Rust. Its authors may wish to restrict the ability of other code to abuse Unsafe Rust to read its data, for instance to protect customer records. Capability sealing presents a viable method to enforce this.

Note that accessing an object which one has a reference to is not undefined. Sealing to prevent pointer dereferences can therefore be used to enforce invariants which other code may not conform to. For instance, a high-performance data structure might attempt a deep copy of its objects, which would be obstructed by sealing in the scenario above.

5.5.5 Efficacy and costs of sealing

However, manipulating capabilities is not free and the overheads of manipulating capabilities should be weighed against enforcement within the existing type system, which is statically enforced but does not protect against unsafe code. Additionally, this does not solve the inherent potential for programmer error in the compiler, such as incorrectly failing to seal a capability.

5.6 Improved safety of FFI calls

A quick survey of Rust crates that use FFI bindings shows that most of them deal with some form of low-level behaviour, including encryption, memory allocation, USB interfaces, and filesystem mounts.[4] With performance or low-level access as the primary goal of most Rust FFI calls, safety may not be a major concern, even if it could allow arbitrary code execution in the Rust program [53].

Most of the discussion below has to do with object capabilities and is thus inherent in the use of CHERI capabilities, but system call protection is a separate feature of CHERI compartmentalisation. Watson et al. describe the mechanism and security implications with regard to C [59]. Many of the points apply

[4] Rust crates are published libraries. A survey of the "Rust package registry" may be of interest: https://crates.io/search?q=-sys; a `-sys` suffix conventionally denotes an FFI crate.

to non-FFI code as well, although protection across domain boundaries is perhaps more significant as FFI calls cannot be type- or borrow-checked.

5.6.1 Prevention of use-after-free from FFI

The use of FFI functions can bypass Rust's object lifetime model, as there is no way to ensure that an FFI function has not stored or leaked references to borrowed Rust objects. This can lead to later temporal unsoundness in the Rust program.

While this cannot be completely prevented with capabilities as-is, Section 5.7.3 discusses the possibility of using revocations to manage this risk.

5.6.2 Enforcement of object boundaries

Objects can only be passed to C as raw pointers. As such, a buffer overflow through an FFI function is no more difficult than in C natively, and no less serious.

A simple example of an overflow is a C string with a missing NUL terminating byte. To overcome this, Rust strings instead store a length value and a byte array, making them non-interchangeable with C strings. However, if a programmer neglects to convert between the formats, passing a Rust string to a C function will quickly lead to an overflow.

Object capabilities easily protect against this by storing and enforcing object bounds. Importantly, they protect the rest of the calling program's address space, preventing a cross-domain call from accessing data which was not explicitly passed to it.

5.6.3 Protection of system calls

In addition to protecting the caller's memory, CHERI can be extended to protect access to system calls in FFI functions. This is achieved by funnelling system calls through classes which only permit safe calls [59]. A caller can thereby restrict the privileges of a callee function, preventing bugs in called libraries from propagating to the host program.

5.7 Strengthening unsafety

This section considers the danger posed by undefined behaviour in Rust, and how it can be mitigated by CHERI capabilities, enhancing the semantics and guarantees of the language while not impeding compiler transformations and optimisation.

I address four of the nine forms of undefined behaviour documented by the Rust project, and show how they are made safer with capabilities.

5.7.1 Rationale for Unsafe Rust

In Safe Rust, the compiler accepts programs which avoid memory safety problems using its type- and borrow-checkers. Unsafe Rust enables programs which would not otherwise pass this static analysis [44].

Programmers bypass these checks by using raw pointers, unsafe traits, or using mutable static variables, all of which could violate temporal memory safety. Reaching undefined behaviour is possible with unsafe code, where it is not supposed to be possible in Safe Rust.

5.7.2 From unsafe code to undefined behaviour

A requirement of the Rust language is that safe code should *never* be able to exhibit undefined behaviour: unsafe code must be handled thoroughly such that safe functions are defined on all inputs. Thus Unsafe Rust and undefined behaviour are inextricably linked.

Undefined behaviour in Rust is fairly simple to comprehend: the core list contains only 9 points [45],[5] none of which are challenging to avoid, unlike signed integer overflow in C. The following sections show how the CHERI architecture can be used to restrict undefined behaviour, and thereby make Unsafe Rust safer.

5.7.3 Restricting undefined behaviour with CHERI capabilities

The undefined behaviour below takes the form of invariants assumed by the Rust compiler or LLVM; they all involve the usage of pointers or references. Transformations based on these assumptions are made safer, by guaranteeing a trap if the invariant is violated.

LLVM pointer aliasing rules

Breaking LLVM's pointer aliasing rules is forbidden. In essence, the rules state that a pointer may not be used to access memory apart from those addresses it is based on.

CHERI capabilities rule this out, due to monotonicity. In fact, they impose stronger conditions, as a capability formed by an `inttoptr` cannot be dereferenced.

Mutation of non-mutable data

CHERI capabilities can prevent the mutation of data reached through a shared reference by enforcing object immutability as detailed in Section 5.5.1. In this case, the references are known to be immutable in Rust semantics, so this is easy to compile and enforce.

Enforcement of the `noalias` model

Likewise, CHERI capabilities can rule out violations of LLVM's `noalias` model in Rust by similar means. To check this, first note that the model will not be violated in Safe Rust, as the compiler enforces uniqueness of mutable references. It is thus only necessary to prevent the mutation of an object through a reference if other references to that object are being used.

This can be done by deriving a read-only capability for immutable references. As the compiler enforces the uniqueness of mutable references, the two measures combined prevent violations of this model in unsafe code.

Caveat A mutable reference can first be converted to a raw pointer, and subsequently duplicated in unsafe code. This would prevent the Rust compiler from enforcing the uniqueness of mutable references or generating non-writeable capabilities, abandoning Rust's temporal safety guarantees.

Dereferencing null or dangling pointers

This invariant is traditionally used to show that a pointer is not null or dangling, if it has been previously dereferenced. That is to say a pointer which has already been dereferenced is safe to dereference again,

[5] *The list is non-exhaustive.* Note: Rust has no formal semantics or specification, making the idea of particular undefined behaviours rather arbitrary.

which ties in with the aliasing rules discussed above. It is then considered safe to optimise away null checks [55, 32], possibly causing an invalid dereference.

While capabilities cannot prevent an attempt to dereference a pointer, they can prevent dereferences from succeeding. For instance, a null pointer or an arbitrarily-constructed pointer from `inttoptr` will fail to dereference on CHERI, as it would not possess a valid capability. This removes much of the danger from dereferencing a null or dangling pointer, by guaranteeing that it will trap rather than compromising the integrity or confidentiality of memory.

Dangling pointers By themselves, capabilities do not offer protection against dereferencing a dangling pointer, as they derive from a previously-valid object. However, revocation allows dangling pointers to be identified and marked, preventing them from being used.

5.8 Hybrid ABI: Minimising the memory footprint of CHERI capabilities

5.8.1 Safe Rust is memory safe

Observe that the bugs covered in Section 5.2 occur in Unsafe Rust: either from the exposure of unsafe methods as safe, thereby failing to handle all the cases, or through errors in the unsafe code itself. This is no coincidence: Safe Rust is not expressive enough to generate memory safety errors, as it lacks the memory manipulation primitives to do so. This is evinced by the additional actions programmers may take in Unsafe Rust [42].

Being satisfied that Safe Rust is indeed memory safe, references that are only manipulated in Safe Rust need not be protected by capabilities, reducing the overall memory overhead.

5.8.2 Pointer provenance for Unsafe Rust

Due to Rust's strong provenance model, all references can be traced to their original object, and so can raw pointers generated from that object. Thus all references used in Unsafe Rust can be tracked to their instantiation at compile time, and marked to indicate that a capability pointer should be used. Unsafe Rust code would therefore only encounter capability pointers.

Likewise, this analysis can determine all objects reached from unsafe code: similar analysis is already performed to prevent modifying immutable objects in Unsafe Rust.[6]

Note that raw pointers can be manipulated to point to a different object, even if this is undefined behaviour in Rust. However, if each raw pointer is a capability pointer, such code will still be unable to violate memory safety.

5.8.3 Reduction in memory overhead of capabilities

Such a scheme would use non-capability pointers in the default LLVM address space for memory only accessed by Safe Rust code. This results in reduction in the memory overhead of capabilities, due to the use of ordinary pointers; the overhead being constrained to objects that are manipulated in unsafe code. This is possible using the hybrid ABI, rather than the pure capability ABI implemented in this project.

[6] For completeness, unsafe code can modify immutable objects by casting to (Rust) raw pointers. The provenance of these raw pointers is still tracked by the compiler; see Stacked Borrows for an effort using provenance to enforce semantics on Unsafe Rust [25].

Note that Rust is considered to be similarly performant to C (Section 5.3.4). However, C lacks a separation of 'safe' and 'unsafe' code, and hence lacks a practical mechanism to distinguish low-risk code, making this optimisation impossible. It is possible to focus on dangerous libraries, traditionally those providing high-performance data parsing, and only use capabilities there. Nevertheless, this ignores the fact that serious vulnerabilities in C code have been unexpectedly found in otherwise safe-looking code, sometimes due to the subtleties of the C standard [35]. Consequently, capability protection could be optimised for a lower footprint on Rust code than C, potentially making it the ideal language for a CHERI platform.

5.8.4 Limitations

Optimisation opportunity could be limited, however. Many routines in the core library use unsafe code to manipulate objects, such as the built-in sorting algorithms. It is unclear how many routines do this to avoid bounds-checked accesses, as is the sole reason in sorting: recall that runtime bounds checking is redundant in CHERI, but only with the use of capabilities.

A study of the proportion of Rust references not used in unsafe code is therefore advisable. Note that not all references passed to Unsafe Rust need to be marked: only usage for raw pointer operations and FFI calls are relevant in this case, along with specific compiler primitives.

5.9 Distinguishing pointer width and index sizes

Sections 3.3 and 4.3.1 discussed the implications of Rust's implementation of index size or `usize` in the compiler. Here I consider how its definition could be changed or reinterpreted to make it compatible with CHERI.

5.9.1 Definition of `usize`

The `usize` type is classified as a 'machine-dependent integer type'. It is an *unsigned integer type with the same number of bits as the platform's pointer type, which can represent every memory address in the process* [45]. Elsewhere, the Rust documentation states that it is *the pointer-sized integer type*, and that *the size of this primitive is how many bytes it takes to reference any location in memory*, stating that it is 8 bytes on a 64-bit target [46].

Likewise, the `isize` is defined as its signed counterpart. The Rust reference notes that the *theoretical upper bound on object and array size is the maximum* `isize` *value*, even though the `usize` type is used for indexing into slices.

5.9.2 Representing every memory address

There is no pair of morphisms that translate between integers (including the `usize`) and CHERI capabilities. This is due to monotonicity, where integers cannot be transformed into pointers, and enforced by the memory tagging mechanism. Thus no memory address can be recovered solely from an integer.

Notwithstanding the difference between *represent* and *reference*, and the existence of arbitrary pointer values, it seems clear that there should be some sort of correspondence between pointers and `usize` values. Specifically, the text suggests that this correspondence is between *memory addresses* or *locations*, rather than any data associated with a pointer.

Usage in the compiler and core library

This correspondence is echoed in compiler usage and the core library. Usages of `usize` are not concerned with tag information that may be associated with a pointer; they refer to sizes, ranges, and distances.

For instance, the 8 tag bits provided on the AArch64 architecture are of no interest when using a `usize`.

5.9.3 Rust context

This is a complex issue in Rust. Rust RFC #544 [63] is ostensibly a naming issue for this pointer-sized integer type, but it provides an excellent background to the semantics and usage of this type in Rust. Indeed, its complexity is such that the community understands `usize` to be "the size of a pointer by definition.", or "defined…[to be] the same size as *const ()", i.e. a raw pointer. A more cautious opinion gave that "traditionally, `usize == uintptr_t`".[7]

Despite these statements, I claim that the text of the definitions and the usage show that the `usize` need only be large enough to reference memory addresses. Thus 48 bits suffice on CHERI, though 64 might be a more practical value.

This is something of a controversial issue in Rust, and even a seemingly minor clarification may take considerable time to make it into the documentation. From a semantic standpoint, however, to use a 128-bit `usize` on CHERI128 would be not only inconvenient, but incorrect.

5.9.4 Integer types in C

C has several related machine-dependent integer types:

`size_t` Integer type appropriate to represent the size of any object, including indexing into at array. This is the most frequent use of `usize` in Rust.

`ptrdiff_t` Integer type large enough to store the difference between two addresses. In Rust, there is no motive to compare pointers from different allocations, so a `size_t` equivalent suffices.

`intptr_t` Optional signed integer type large enough to hold a pointer. The commonly-understood definition of `isize` in Rust.

`uintptr_t` Optional unsigned integer type large enough to hold a pointer. The commonly-understood definition of `usize` in Rust.

What is important to see is that the specific integer value of a pointer is of no intrinsic interest, and only matters in comparison to other pointers. Further, it is meaningless to compare pointers to different objects. The Rust core library's principal use of the LLVM `inttoptr` primitive is to determine the size of an allocation, which is consistent with the above statements.

Most importantly, the key argument for only having one integer type that corresponds to C's four was that Rust semantics are different, and the require different reasoning and types. This is precisely true: there is no functional use for non-address part of a CHERI capability pointer, and the addressable range should *not* be conflated with the pointer size.

[7] Responses in clarification of the definition of `usize`, from the Rust compiler forum on Zulip. The query included the CHERI example, where 64 bits of a 128-bit pointer are used for addressing.

5.10 Porting safe languages to capability architectures

This chapter has analysed how different mechanisms and protections of the memory-safe Rust language interact with CHERI capabilities. This section generalises many of its observations to other safe languages, giving a preview of the relevant concerns when bringing other languages and runtimes to hardware capability platforms.

5.10.1 Weaknesses in language runtimes

Many safe languages employ type systems to prevent unauthorised reading or writing of data, but their runtimes can be a source of vulnerabilities. For example, a 2017 bug in CPython[8] contained a bug potentially giving arbitrary code execution from a heap overflow [11].

Section 5.2.2 showed a similar exploit prevented by CHERI capabilities, and in the Python example, the heap allocation would not successfully overflow with capabilities. Note that CPython is written in a combination of C and Python, and this routine in particular is written in C. This shows how safe languages suffer vulnerabilities through their runtimes.

5.10.2 Unsafe code

Continuing from the previous section, like Rust's `unsafe`, most 'safe' languages provide escape hatches: an example is Haskell's `performUnsafeIO`. Failing to do so limits the range of behaviour that programs in that language can exhibit efficiently. Similar mechanisms are used to call code written in other languages, such as via the Java Native Interface (JNI).

Chisnall et al. have considered the application of CHERI capabilities to JNI calls [12], and Watson et al. demonstrate compartmentalisation alongside CHERI capabilities [59]. In both cases, protections are applied to what is generally the only significant attack surface of a program, making it an efficient design choice.

Likewise, capability-aware languages could employ capability sealing to guarantee that data is not inspected or modified when passing references through system or cross-process calls, enforcing the principle of least privilege.

5.10.3 Language semantics and implementation

Each language has a different memory model, which may not be fully consistent with CHERI, which may prevent certain idioms from translating properly. C has a notoriously weak memory model, permitting pointer operations which destroy provenance under CHERI, making them non-dereferenceable: in particular, such patterns are used in real-world programs [35].

Nevertheless, when Davis et al. examine the issue for FreeBSD and PostgreSQL, pointer provenance is not as significant an issue as one might expect [14]. It is probable that any pointer provenance issues to do with safe languages come from, as in Rust, the compiler and runtime rather than general programs written in that language.

5.10.4 Non-optimisation: type systems

Languages may use type systems to make significant guarantees about programs: this is especially true in functional languages. Outside of these, Java and Rust have sophisticated type systems; the use of capabilities cannot optimise checks that are made at compile time.

[8] The Python runtime.

A contrasting example is Python, which performs type-checking at runtime: this cannot be optimised by capabilities either, as more information is derived from types in general. Ultimately, it may be the case that capabilities are uniquely useful for weakly-typed languages such as C.

5.11 Summary

This chapter began by demonstrating how effectively CHERI capabilities mitigate Rust vulnerabilities. As further evidence of this point, another vulnerability in the Rust standard library was announced while writing this report [40]: it permitted arbitrary typecasting and hence buffer overflows. Again this is prevented by CHERI capabilities.

It detailed how Rust and CHERI memory protection mechanisms interact, and choices that would maximise the benefit while minimising the cost. This was done in light of Rust's semantics and definition of undefined behaviour, identifying the weakest points in the language and showing how they can be protected with capabilities. Consideration was also given to implementation issues in the compiler, resulting from under-defined semantics, and a strategy to tackle this presented. Finally, it generalised the impact of Rust capabilities to other languages and runtimes, showing the areas which would benefit the most and least from capability protection.

Chapter 6

Conclusion

6.1 Context and review

This project examined memory protection mechanisms at the intersection of hardware and computer architecture, compilers, and programming language semantics. The varied approaches and measures shed light on the subtleties of two orthogonal problems: spatial and temporal integrity. These insights illustrate different ways of enforcing the principles of *least privilege* and *intentional use*, and the compromises and difficulties that arise from their interaction.

Whereas the benefits of hardware capabilities are well-studied for C, this work provides an understanding of their benefits in the context of a safe language, Rust. Each safe language provides different guarantees and enforces them differently; Rust defines semantics and performs most of its enforcement through static checking, rejecting programs with unclear invariants. The strict constraints that this places on programs motivates Unsafe Rust, which provides programmers more control over execution. In the standard library, unsafe code is even used to optimise string processing, a traditional source of C buffer overflows. Sections 3.4 and 3.5 covered situations in which Unsafe Rust is used.

It is in unsafe code that capabilities demonstrate their value, providing dynamic checks where static checking is not possible. Section 5.2 examined how Rust's protections yield to programmer error, giving analyses of previous vulnerabilities in the Rust standard library. Further, it showed that capabilities successfully prevent these vulnerabilities from exploitation. Indeed, in the course of writing this report, another vulnerability in the Rust standard library surfaced [40]: it too is preventable by CHERI capabilities!

The evaluation went on to discuss how capabilities can strengthen and enforce guarantees given by the language, using features like *capability sealing* to protect data given to cross-domain calls. This protection obviates expensive measures like sandboxing, frequently considered too costly to use in many real-world applications. I emphasise that capabilities equip Rust with powerful mechanisms for increasing memory safety at low cost, *without* changing the language semantics or memory model. A similar claim cannot be made for C, due to its weak pointer provenance model [14]. Further, I elaborate on how capabilities interact with, and can rule out several forms of undefined behaviour in Rust. Compiler transformations relying on those forms are thus safe to perform on CHERI, without concern of violating the integrity or confidentiality of data.

I also elucidated how Rust's techniques complement CHERI memory safety, offering a clear pointer provenance model, a safe sub-language, and complementary temporal safety guarantees. These properties make Rust code easier to compile with capabilities, enable the targeted use of capabilities to protect

the most vulnerable data, and avoid the most common vulnerabilities that CHERI architectures do not currently address. Combined, these properties permit optimisations on capability usage not possible with C codebases, showcasing the promise of combining capability architectures with safe languages. With Rust performance comparable to that of C without capabilities, these optimisations could mean that Rust code will run even *faster* than C code under capability protection. This is especially significant as workloads become more bounded by memory throughput, where capability overheads are most noticeable.

Even as CHERI begins to form a foundation for its own temporal memory protection, the study of how the architecture interacts with memory-safe languages forms a basis for realising its potential in the larger ecosystem of modern software development. As a preliminary step, Section 5.10 covers the initial concerns when porting safe languages to capability architectures.

6.2 Challenges

This project has not been without its challenging points. A recurring problem was the conflict caused by changing the pointer width to 128 bits, various aspects of which were recorded in Sections 3.3, 4.3.1 and 5.9. The issue effectively prevents full Rust compatibility with CHERI capabilities, and in this project prevents the use of compiler optimisations. The definition is part of the language specification, and a source of some controversy in the Rust community; I put forth small, plausible changes to remove the conflict with CHERI. A detriment of this issue has been the inability to perform more sophisticated evaluation of CHERI overheads in Rust, as Joannou et al. have done for C [23].

Another challenge was maintaining the relevance of the work. At first glance, the motivation is obvious. Capabilities make C much safer: can they do the same for a safe language like Rust? Could they optimise out dynamic checks like array bounds checks, leading to a measurable performance increase?

The nuances emerge when one considers Unsafe Rust, or the diverse 'escape hatches' in other languages. Perhaps the relevant questions are those of where languages benefit the most, or equally where capabilities are *least* useful. The dearth of literature studying capabilities under safe programming languages does not help; indeed the proposition that capabilities are less obviously useful for safe languages seems increasingly reasonable. These serve to illustrate the value of segmenting safe and unsafe code and enabling fine-grained protection in the language (Sections 5.5.4 and 5.8).

6.3 Scope of contributions

This project was principally concerned with the challenges that arise from bringing a safe language to the CHERI architecture, and how they interact with each other. In providing an account of these concerns, it made a number of contributions:

- Patches to the Rust compiler and core libraries enabling support for CHERI capabilities, without compiler optimisations. Details of the necessary further work to fully support a CHERI target, including the language design choices which pose problems for this task. These modifications formed the basis on which I evaluated Rust on CHERI.
- Verification that the modified compiler produces programs which use capabilities, by demonstrating capability protection against previous vulnerabilities. Analyses of these and other vulnerabilities, which were all present in the Rust core library, showing that capabilities are effective in fully mitigating the errors.
- An evaluation of the interactions between Rust and CHERI protection mechanisms, focussed

on how the two complement each other to provide stronger memory safety guarantees. I elucidated how CHERI provisions such as *capability sealing* bring new memory safety properties to Rust, especially the means to enforce previously-unenforceable invariants on which Rust predicates its memory safety guarantees.

- The evaluation further investigated approaches that reduce the overheads posed by these memory protection mechanisms without reducing memory safety. It proposed configurations that avoid unnecessary duplication of strong protections, but instead work to reinforce the most vulnerable points of Rust programs: the use of unsafe code and cross-domain calls. Further, I highlighted the trade-offs of different combinations of protection techniques and how they affect memory safety, crucial factors for implementers bringing Rust to CHERI.
- I addressed details of the Rust language and implementation that currently prevent full Rust compatibility with CHERI. Of specific interest is the definition of `usize` in the language, changes to which would need to be put to the Rust community. Also discussed is how different aspects of protection and implementation might apply to other safe languages and their runtimes, providing insight into CHERI's implications for the wider software development ecosystem.

6.4 Further work

6.4.1 Pointer-width sized indices

Changes to Rust semantics

Section 5.9 considered Rust's `usize` definition ("pointer-sized integer"), why this is not desirable, and explored alternatives. As this is both a semantic and performance issue on CHERI (Section 4.3.1), it should be a priority to resolve this definition.

Nevertheless, this is not a straightforward task. A unilateral change would involve large divergences from the upstream code, as the pointer width and index sizes are implicitly used interchangeably in the Rust compiler. Since such a change affects the Rust semantics, and potentially introduces new types, it inevitably will have to be proposed via the Rust RFC system. The issue has been discussed before [63, 54]; I suggest that the least controversial change would be to reinterpret the current rule and leave the introduction of any new types for a later effort. This could take a significant amount of time and human effort to effect, and is entirely beyond the scope of this exploratory work.

Implementation in the compiler

This change could then be implemented in the compiler. As noted above, this would be challenging to do without first achieving consensus on the semantics.

While there are numerous references that do not fully distinguish between pointer size and index size, the CTSRD project has experience resolving similar issues in Clang and LLVM, showing that the problem is not insurmountable.

6.4.2 Runtime and memory overhead analysis

This work demonstrated that Rust protections can work alongside CHERI capabilities and explored the safety and semantic implications. However, the runtime and memory overheads of CHERI have not been explored.

Approaches

To measure runtime overheads, a possibility is to compare against a MIPS FreeBSD target, which I include in the patches provided in this project. These should compile any Rust program with optimisations, although it should be noted that Rust releases are not tested against the MIPS64 Linux and amd64 FreeBSD targets from which this derives [43]. The major reason this comparison is not currently possible is the lack of optimisations, and performance issues caused by 128-bit index sizes (see Section 6.4.1), both beyond the scope of this work.

For memory overheads, similar techniques to those employed by Joannou et al. [23] can be used, by core-dumping Rust programs. Another plausible approach might be to use Rust's Mid-level IR interpreter (*Miri*) to track heap pointers, and compute overheads from there. This would give precise measurement of overheads, as well as present an effective memory analysis tool for Rust code running on any platform.

6.4.3 Fine-grained capability protection

Previous Rust bugs resulting in memory vulnerabilities occur in unsafe code. This is because Safe Rust does not have the expressivity to dereference memory except in specific ways, including arrays and structs. Further, the Rust developers are keen to uphold the tenet that Safe Rust should never exhibit undefined behaviour, and on multiple occasions have removed or marked unsafe code which violated this assumption. They do so on the grounds that this is incorrect behaviour.

It is therefore reasonable to assume that this will continue, and Rust vulnerabilities relating to memory protection will only occur through the use of Unsafe Rust.

Approaches

Section 5.5.4 suggested how certain Rust types requiring particularly sensitive handling could be protected by capabilities, and Section 5.8 examined the implications of protecting objects reachable by unsafe code. In both cases, using regular pointers for the remaining objects could reduce the memory overheads of capabilities on Rust programs.

Both would be challenging to implement, as Rust would first need to be extended to support multiple LLVM address spaces, then additional provenance analysis would be needed to determine which objects should be protected.

However, the benefits are sizeable if non-capability pointers could be used widely: this would result in lower memory overheads than C programs, for instance. This is also safer than focussing capability protection on risky C libraries,[1] as provenance is far clearer in Rust, obviating the possibility of other unchecked code interfering with the capability protection.

[1] Such as image and video codecs

Bibliography

[1] CVE - CVE-2018-1000622. Available from MITRE, CVE-ID CVE-2018-1000622, 2018. Accessed: 2019-04-25.

[2] CVE - CVE-2018-1000657. Available from MITRE, CVE-ID CVE-2018-1000657, 2018. Accessed: 2019-04-12.

[3] CVE - CVE-2018-1000810. Available from MITRE, CVE-ID CVE-2018-1000810, 2018. Accessed: 2019-04-12.

[4] CVE - CVE-2019-12083. Available from MITRE, CVE-ID CVE-2019-12083, 2019. Accessed: 2019-05-31.

[5] Jorge Aparicio. rust-san. [Online] https://github.com/japaric/rust-san, 2017. Accessed: 2019-05-01.

[6] Alexis Beingessner. You can't spell trust without Rust. Master's thesis, Carleton University, Ontario, 2015.

[7] David Elliott Bell and Leonard J LaPadula. Secure computer system: unified exposition and Multics interpretation. Technical report, ESD/AFSC, Hanscom AFB, Bedford MA 01731, 1975. ESD-TR-75-306.

[8] Christophe Biocca. `std::vec::IntoIter::as_mut_slice` borrows `&self`, returns `&mut` of contents. [Online; Rust issue #39465] https://github.com/rust-lang/rust/issues/39465, 2017. Accessed: 2019-04-23.

[9] Sergi Blanco-Cuaresma and Emeline Bolmont. What can the programming language Rust do for astrophysics? *Proceedings of the International Astronomical Union*, 12(S325):341–344, 2016.

[10] Jim Blandy and Jason Orendorff. *Programming Rust.* O'Reilly Media, Inc., 2017.

[11] Jay Bosamiya, Serhiy Storchaka, Leo Kirotawa Silva, Larry Hastings, and Victor Stinner. Issue 30657: [security] CVE-2017-1000158: Unsafe arithmetic in PyString_DecodeEscape. [Online] https://bugs.python.org/issue30657, 2017. Accessed: 2019-05-30.

[12] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. CHERI JNI: Sinking the Java security model into the C. *SIGOPS Oper. Syst. Rev.*, 51(2):569–583, April 2017.

[13] Alex Crichton. std: Check for overflow in `str::repeat`. [Online; Rust pull request #54399] https://github.com/rust-lang/rust/pull/54399, 2018. Accessed: 2019-04-23.

[14] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre

Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 379–393, New York, NY, USA, 2019. ACM.

[15] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. *SIGOPS Oper. Syst. Rev.*, 42(2):103–114, March 2008.

[16] Derek Dreyer. RustBelt. [Online] http://plv.mpi-sws.org/rustbelt/, 2019. Accessed: 2019-05-07.

[17] Sebastian Dröge. Speeding up RGB to grayscale conversion in Rust by a factor of 2.2 – and various other multimedia related processing loops. [Online] https://coaxion.net/blog/2018/01/speeding-up-rgb-to-grayscale-conversion-in-rust-by-a-factor-of-2-2-and-various-other-multimedia-related-processing-loops/, 2018. Accessed: 2019-04-25.

[18] Steven Fackler. Fix capacity comparison in reserve. [Online; Rust pull request #44802] https://github.com/rust-lang/rust/pull/44802, 2017. Accessed: 2019-05-02.

[19] Brent Fulgham, Isaac Guoy, et al. The Computer Language Benchmarks Game. [Online] https://benchmarksgame-team.pages.debian.net/benchmarksgame/, 2019. Accessed: 2019-04-25.

[20] Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D Matsakis. GPU programming in Rust: Implementing high-level abstractions in a systems-level language. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 315–324. IEEE, 2013.

[21] Felix S Klock II. `core::iter`: fix bug uncovered by arith-overflow. [Online; git commit] https://github.com/pnkfelix/rust/commit/f0404c39f272868c1dedc7cda7b0b6dffcb5713d, 2015. Accessed: 2019-04-25.

[22] Felix S Klock II. Implement arithmetic overflow changes. [Online; Rust pull request #22532] https://github.com/rust-lang/rust/pull/22532#issuecomment-75168901, 2015. Accessed: 2019-04-25.

[23] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alexander Richardson, Stacey Son, and A. Theodore Markettos. Efficient tagged memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648, Nov 2017.

[24] Jesse Jones. seg fault pushing on either size of a VecDeque. [Online; Rust issue #44800] https://github.com/rust-lang/rust/issues/44800, 2017. Accessed: 2019-04-12.

[25] Ralf Jung. Stacked Borrows Implemented. [Online] https://www.ralfj.de/blog/2018/11/16/stacked-borrows-implementation.html, 2018. Accessed: 2019-05-30.

[26] Ralf Jung. vec_deque::Iter has unsound Debug implementation. [Online; Rust issue #53566] https://github.com/rust-lang/rust/issues/53566, 2018. Accessed: 2019-04-23.

[27] Ralf Jung and Alan Jeffrey. Provenance: Rust unsafe code guidelines. [Online] https://github.com/rust-lang/unsafe-code-guidelines/issues/52, 2018. Accessed: 2019-05-22.

[28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, December 2017.

[29] Steve Klabnik, Ben Striegel, et al. Implement address sanitizer (ASAN) support. [Online] https://github.com/rust-lang/rfcs/issues/670, 2017. Accessed: 2019-05-01.

[30] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 721–732, New York, NY, USA, 2013. ACM.

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

[32] Chris Lattner. LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #2/3. [Online] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html, 2011. Accessed: 2019-05-20.

[33] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 89–98, New York, NY, USA, 2016. ACM.

[34] LLVM Project. LLVM Language Reference Manual—LLVM 9 documentation. [Online] https://llvm.org/docs/LangRef.html, 2019. Accessed: 2019-04-12.

[35] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In *ACM SIGPLAN Notices*, volume 51, pages 1–15. ACM, 2016.

[36] National Institute of Standards and Technology Information Technology Laboratory. NVD - Statistics. [Online] https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&query=overflow&search_type=all&pub_start_date=01%2F01%2F2018&pub_end_date=12%2F31%2F2018, 2019. Accessed: 2019-04-02.

[37] Peter G Neumann. Fundamental trustworthiness principles. In Howard Shrobe, David L Shrier, and Alex Pentland, editors, *New Solutions for Cybersecurity*. MIT Press, Cambridge, MA, Jan 2018.

[38] The Rust Core Team. Security advisory for rustdoc. [Online] https://blog.rust-lang.org/2018/07/06/security-advisory-for-rustdoc.html, 2018. Accessed: 2019-04-25.

[39] The Rust Core Team. Security advisory for the standard library. [Online] https://blog.rust-lang.org/2018/09/21/Security-advisory-for-std.html, 2018. Accessed: 2019-04-12.

[40] The Rust Core Team. Security advisory for the standard library. [Online] https://blog.rust-lang.org/2019/05/13/Security-advisory.html, 2019. Accessed: 2019-05-27.

[41] The Rust Project Developers. Leaking. [Online] https://doc.rust-lang.org/nomicon/leaking.html, 2019. Accessed: 2019-04-29.

[42] The Rust Project Developers. Meet Safe and Unsafe. [Online] https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html, 2019. Accessed: 2019-04-10.

[43] The Rust Project Developers. Rust Platform Support. [Online] https://forge.rust-lang.org/platform-support.html, 2019. Accessed: 2019-05-27.

[44] The Rust Project Developers. The Rust Programming Language. [Online] https://doc.rust-lang.org/1.33.0/book/, 2019. Accessed: 2019-04-04.

[45] The Rust Project Developers. The Rust Reference. [Online] https://doc.rust-lang.org/reference/, 2019. Accessed: 2019-05-20.

[46] The Rust Project Developers. usize. [Online] https://doc.rust-lang.org/std/primitive.usize.html, 2019. Accessed: 2019-05-28.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 309–318, 2012.

[48] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the JVM. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 201–211, New York, NY, USA, 2010. ACM.

[49] Arthur Silva. Avoid bounds checking at slice::binary_search. [Online; Rust pull request #30917] https://github.com/rust-lang/rust/pull/30917, 2016. Accessed: 2019-04-25.

[50] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you C: Controlling side effects in mainstream C compilers, 2018.

[51] Björn Steinbrink. Improve PartialEq for slices. [Online; Rust pull request #26884] https://github.com/rust-lang/rust/pull/26884, 2015. Accessed: 2019-04-25.

[52] Ulrik Sverdrup. indexing. [Online; Rust crate] https://docs.rs/indexing/0.3.2/indexing/, 2018. Accessed: 2019-04-25.

[53] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, May 2013.

[54] Aaron Turon, Yehuda Katz, et al. Restarting the `int/uint` Discussion. [Online] https://internals.rust-lang.org/t/restarting-the-int-uint-discussion/1131/191, 2015. Accessed: 2019-05-27.

[55] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275. ACM, 2013.

[56] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security '10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[57] Robert N. M. Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, and Jonathan Woodruff. Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide. Technical report, University of Cambridge Computer Laboratory, September 2015. UCAM-CL-TR-877.

[58] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (version 6). Technical report, University of Cambridge Computer Laboratory, April 2017. UCAM-CL-TR-907.

[59] 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. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, May 2015.

[60] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[61] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 249–257, New York, NY, USA, 1998. ACM.

[62] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[63] Richard Zhang, Aaron Turon, and Niko Matsakis. Rename `int/uint` to `isize/usize`. [Online; Rust RFC #544] https://github.com/rust-lang/rfcs/blob/master/text/0544-rename-int-uint.md, 2015. Accessed: 2019-04-12.