Preserving Memory Safety in Safe Rust during Interactions with Unsafe Languages

by

Elijah E. Rivera

S.B. Computer Science and Engineering

Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author . . . .

Department of Electrical Engineering and Computer Science

May 20, 2021

Certified by . . . .

Dr. Howard Shrobe

Principal Research Scientist, MIT

Thesis Supervisor

Certified by . . . .

Dr. Hamed Okhravi

Senior Staff Member, MIT Lincoln Laboratory

Thesis Supervisor

Certified by . . . .

Dr. Nathan Burow

Technical Staff, MIT Lincoln Laboratory

Thesis Supervisor

Accepted by . . . .

Katrina LaCurts

Chair, Master of Engineering Thesis Committee

Preserving Memory Safety in Safe Rust during Interactions with Unsafe Languages

by

Elijah E. Rivera

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Rust is a programming language that simultaneously offers high performance and strong security guarantees. However, these guarantees come at the cost of strict compiler checks that sometimes prevent necessary code patterns. The `unsafe` keyword allows developers to bypass these compiler checks, and is used in both pure Rust and mixed-language applications. But the use of `unsafe` undermines the security guarantees of Rust that make it an attractive option in the first place.

We first demonstrate that within a real-world pure Rust application, many uses of `unsafe` can be eliminated,or reduced to formally verifiable standard libraries. We then present Galeed, a system for isolating and protecting the Rust heap from access by other programming languages using Intel's Memory Protection Key (MPK) technology. We demonstrate both the effectiveness and efficiency of Galeed on Firefox, a web browser written in Rust and C++.

Thesis Supervisor: Dr. Howard Shrobe
Title: Principal Research Scientist, MIT

Thesis Supervisor: Dr. Hamed Okhravi
Title: Senior Staff Member, MIT Lincoln Laboratory

Thesis Supervisor: Dr. Nathan Burow
Title: Technical Staff, MIT Lincoln Laboratory

Acknowledgments

I would like to give thanks first and foremost to God, who has made provision for me in every step of my life's journey. "[T]o the only God our Savior, through Jesus Christ our Lord, be glory, majesty, dominion and authority, before all time and now and forever. Amen" (Jude 25 NASB) [1].

Thank you Dr. Howard Shrobe and Dr. Hamed Okhravi for overseeing the work on this thesis project and for providing funding for it. Thank you Dr. Nathan Burow for invaluable supervision, assistance, and feedback throughout the research and writing processes. Thank you Dr. Samuel Jero and Samuel Mergendahl for your additional support through technical assistance and suggestions.

Thank you Prof. Armando Solar-Lezama and James Koppel for your personal and professional support. From my sophomore year to now, you have enabled my research and academic careers to thrive, and I am grateful for the guidance you have provided.

Thank you Marcelo and Deborah Rivera. Thanks Mom and Dad, for being my best advocates throughout my academic career. Thank you for supporting my work, both financially and personally, and for pushing me ever higher into my learning and growth.

Thank you to Sloan Kanaski, Ebenezer Sefah Jr., John Robertson, Paul and Carla Grobler, and Judy Richie. You were there through some dark moments in my MIT career, and have been willing to sit with me in those moments and challenge the lies that I was speaking over myself. Thank you to Lydia Yu and Erica Liu for being willing to listen, to question, and to sit with me and play music to keep our sanity during a global pandemic.

Thank you to those whose names I could not fit in this space. There are not enough words to express how grateful I am to everyone who has had a hand in making me the student, teacher, researcher, and man of God that I am today.

Contents

List of Figures

Chapter 1

Introduction

Many modern-day systems are written in C or C++. These include operating system (OS) kernels such as the Linux kernel [33] or Windows NT [35], mainstream web browsers like Mozilla Firefox [38] and Google Chrome[21, 20], and even other languages' compilers and interpreters (e.g. Python [47]). Unfortunately, C and C++ have little to no enforcement of type or memory safety, and as a result are vulnerable to a host of different types of memory errors.

Memory errors in programs are a major source of errors and exploitable vulnerabilities dating at least as far back as 1996 [44]. At BlueHat Israel 2019, Microsoft disclosed that in the past decade, memory errors have comprised $\sim 70\%$ of discovered vulnerabilities in their products [36]. Google has recently come to the same conclusion after analyzing their own security vulnerabilities since 2015 [22]. We say something is *memory safe* if it successfully prevents these memory errors [60]. We can subdivide these errors into two categories: spatial errors, temporal errors. A *spatial error* occurs when a pointer is dereferenced to a memory address outside of its original bounds, while a *temporal error* is either a use-after-free (i.e. accessing memory after it has been deallocated/freed) or use-before-initialization (i.e. accessing memory before it has been properly initialized).

Many research projects have produced tools which help in detecting/mitigating these errors [9, 29, 56], but nonetheless memory errors remain prevalent in codebases, in everything from operating systems to web browsers. Memory safety bugs are frequently vulnerable to exploitation, leading in some cases even to attacks involving arbitrary code execution [6, 57].

Memory safety is also closely related to *type safety*, the prevention of type errors. A *type error* occurs in memory when a memory location is treated as having a certain type, but is then written to with data that does not represent a valid member of that type. In 1978 Robin Milner famously claimed and then proved that "well-typed programs cannot go wrong" [37] in a sound type system. There have been multiple major vulnerabilities discovered due to a lack of soundness in the type system of C [44, 43].

The programming language Rust guarantees strong memory and type safety for programs in the language [51], guarantees which have recently been formalized and verified by the RustBelt project [26]. Rust's guarantees rely on its ownership system, which implements memory safety as a subset of its type system. By encoding information about the kinds of reference to an object and the lifetime of the object into the type system, Rust is able to utilize existing type checking techniques to statically ensure that programs that compile meet the memory safety guarantees above, and to do so with little to no cost to performance [61, 7]. This combination of safety and performance has proven attractive to the systems community, prompting an increased in the popularity of Rust [27].

Rust's type system is *conservative*, that is, sound but incomplete. The Rust type-checker is *sound*, in that it will never accept a program that is not well-defined within the language model, and thus will not violate the safety guarantees of that model. Rust's type system is *incomplete* in that the Rust type-checker will reject some programs during compilation as false negatives, programs that are considered incorrect by the type-checker but which are actually valid in the underlying language model. Many operations required in low-level systems programming violate the rules of the type-checker but do not necessarily violate the underlying safety model.

In operating systems, these operations include memory-mapped I/O operations, inline assembly code, direct pointer arithmetic and related instructions, and some other data structures with complex ownership requirements. These operations are prevented by different type-checker restrictions and therefore do not compile. For example, memory-mapped I/O operations often necessitate writing directly to a specific constant memory address. However, the Rust type system prevents direct access to any specific memory address, to protect against mutating a piece of data that is potentially pointed to somewhere else in the program.

For user applications, another set of operations which break the rules of the Rust type-checker involve the use of the Rust Foreign Function Interface (FFI) to interact with other languages, especially C/C++. In large pre-established codebases, developers cannot simply rewrite the entire system at once in Rust. Issues of both scale and backwards compatibility are guaranteed to arise. Instead, many of these codebases are being ported over to Rust in small increments (e.g. Firefox [39]). Individual components are rewritten in Rust, and then the FFI is used to connect the Rust component to the rest of the codebase. The FFI makes designated Rust functions externally available to non-Rust components, and enables the use of externally defined functions within the Rust component. However, by default the Rust compiler cannot reason about the safety of functions not written in Rust, and therefore will refuse to compile.

To get around these restrictions, Rust provides a backdoor in the form of the keyword `unsafe`. In Rust, `unsafe` signals to the compiler that the programmer is writing code that he/she knows will not pass the Rust type-checker. The burden of verifying that the code adheres to memory-safety and type-safety falls back onto the programmer. This means that code that includes a dependence on `unsafe` ultimately has the same level of guarantee as pre-Rust solutions: “hopefully the programmer is correct.”

Unfortunately, this backdoor does not just undermine the guarantees of code that contains unsafe sections inserted by the programmer. Since `unsafe` forces the compiler to ignore certain checks, any values modified within or returned by unsafe code could break all type and memory safety guarantees, even once passed to safe code. Once `unsafe` is used, everything it transitively interacts with is also necessarily unsafe.

Worse, many of the Rust standard library data structures also contain some amount of unsafe code. Any code that relies on these data structures also now has a similar lack of the earlier formal guarantees. Now even though we started with a language that is statically safe, using even just the Rust standard libraries can potentially violate this safety. The previously mentioned RustBelt project [26] also includes formal correctness proofs for many of data structures that use `unsafe` within the Rust standard libraries. Rustbelt also provides mechanisms for producing formal correctness proofs for unsafe user-defined structures, which could otherwise undermine memory safety guarantees in the same way as the standard library structures.

The following body of work shows how we can still preserve the memory safety guarantees of safe Rust in the presence of unsafe code. We show that we can often protect safe code in Rust-only applications by simple refactoring, and that we can protect safe Rust code in mixed applications by using newly available hardware technology and our newly developed code transformations.

We investigate extending the formal mechanisms of RustBelt's proof system to data structures in Tock [32], an embedded operating system written solely in Rust but which has multiple uses of `unsafe`. We find that uses of `unsafe` within custom Tock data-structures can be refactored to use structures from the RustBelt-verified standard library, and these refactored structures still compile to the same assembly instructions as their original counterparts.

We then continue trying to preserve Rust memory safety guarantees, turning our attention to Rust's interactions with other languages. We focus on Firefox, a web browser in the process of migrating from C++ to Rust [39]. We propose Galeed, a method for isolating and protecting the Rust heap from independent access by any other programming language using Intel's Memory Protection Key (MPK) technology [25], followed by a method of safely passing structures on the heap from Rust to C++ without compromising memory safety. Together, these approaches help preserve the memory guarantees of the Rust portions of applications that use both Rust and another language (e.g. Firefox), even in the presence of malicious code.

In this thesis, we review the necessary background concepts and related work that this body of work builds off of (chapter 2). We explore first ways of protecting safe Rust from code written in unsafe Rust, and the results of our attempt to apply these techniques to Tock (chapter 3). We then propose and prototype Galeed, a method for protecting the memory integrity of safe Rust from interactions with other programming languages via full memory isolation (chapter 4), followed by methods for automatically transforming external functions to respect Galeed's new heap isolation model of interaction (chapter 5). We review initial microbenchmarks for both of these prototypes (chapter 6), and discuss factors that led to our design decisions and future work to be pursued (chapter 7). Finally, we present related work in the space (chapter 8).

Chapter 2

Background & Threat Model

In the rest of this thesis, we will assume familiarity with two projects which we based our work off of: the Rust programming language, and Intel's Memory Protection Keys. This chapter explains details of these projects necessary to contextualize the rest of our work, and then closes with a discussion of our threat model for this project.

2.1 Rust

Rust [52] is a programming language that offers low-level control and high performance, while still also being able to offer type safety, memory safety, and automatic memory management. Rust does this by making memory safety a property that is statically checked at compile-time in the same way that type safety is. In fact, memory safety is built into the type system for Rust via the *ownership* system.

In Rust, variables "own" their resources, including allocated memory [53]. When a variable goes out of scope, it is responsible for freeing its owned resources. To prevent memory leaks and double-frees, every resource has exactly one owner. Ownership can be transferred to another variable, which invalidates future accesses to the first owner.

If we want to access a resource without taking ownership of it, we can *borrow* it. Borrowing gives us a reference to a resource. We can either borrow a resource immutably or mutably. There can be any number of immutable references to a resource, but if there is a mutable reference, no other references can exist until the mutable reference is done being borrowed (i.e. goes out of scope). All of these properties are checked at compile-time by the Rust borrow-checker (a subset of the type-checker), and a program which violates any of them will not compile.

To make sure that borrowed references are always valid, Rust also includes the concept of *lifetimes*. In Rust every resource has an associated lifetime, which is the length of time for which it exists. References are not allowed to exist beyond the lifetime of the original resource, a restriction also checked by the borrow-checker. This restriction prevents use-after-free errors.

The combination of these static properties ensures that programs which successfully compile are guaranteed to be memory safe. Having these properties be statically checked also means that Rust does not incur the costs associated with runtime checks, which allows for performance on par with its closest counterpart, C++ [61, 7].

Rust has made claims to memory and type safety from its inception, and these claims have been mostly proven, first with Patina [49] and then more thoroughly with the RustBelt project [26]. RustBelt formalizes a machine-checked safety proof for a "realistic subset" of Rust. The project then extends that proof to semantically verify the safety properties of some Rust core libraries which are forced to use `unsafe` to avoid the compile-time restrictions of the Rust borrow-checker. They also provide an extensible interface to this proof system, which allows developers to check what verification conditions are required of new Rust libraries before they can be considered safe extensions to Rust.

Rust's combination of performance and guaranteed safety has contributed to its increasing popularity of within the programming community, with many projects being written or re-written all or at least partly in Rust [16, 39, 31, 5, 27]. Our work focuses on two such real-world applications. Tock [32] is an open-source embedded operating system whose kernel and drivers are all written entirely in Rust for security purposes. Firefox [38] is a web browser developed by Mozilla Corporation. Firefox was originally written in C++, but has begun the process of migrating to Rust [39].

As mentioned in chapter 1, for many applications the Rust compile-time checks can often be too restrictive when trying to write certain patterns in programs, especially in low-level systems or when interfacing with other languages. To allow developers to bypass compile-time checks, Rust includes the keyword `unsafe`. `unsafe` bypasses compiler checks including the borrow-checker, which means that memory safety is no longer guaranteed in the presence of `unsafe`.

2.2 MPK

Intel Memory Protection Keys (MPK) [25] is a new technology which is currently only available on Intel Skylake or newer server-class CPUs. MPK enables quick switching of read/write permissions on groups of pages from userspace. Each page in the page table is tagged with a protection key. Built-in system calls are available to change which protection key is assigned to a page. Permissions for the protection keys are stored in a new register called the PKRU, and new assembly instructions are available to read from or update the PKRU while in userspace. This means that we can execute a single assembly instruction to toggle read/write permissions on a group of pages all at once. ERIM [65] showed that updating permissions takes between 11-260 cycles, which corresponds to an overhead of $<1\%$. `libmpk` [46] (discussed below) also confirmed a $<1\%$ overhead cost for using MPK, and was able to show that using MPK enables performance improvements of >8x when compared to traditional `mprotect` system calls for process-level permissions.

`libmpk` [46] is an open-source C library meant to serve as a software abstraction around the MPK hardware technology. It claims to provide "protection key virtualization, metadata protection, and inter-thread key synchronization." The library has API calls for initialization, allocating/freeing pages, and setting page group permissions. Additionally, `libmpk` provides an additional set of API calls specifically for setting up a heap within a given page group and then allocating/freeing memory from that heap.

2.3 Threat Model

Our project focuses on mitigating threats explicitly caused by interactions between safe Rust and another unsafe language (either unsafe Rust or C/C++) in mixed-language applications. We assume that the underlying hardware, operating system (OS), and compiler layers are not faulty, compromised, or otherwise malicious. We recognize additional potential threats if these assumptions do not hold, but as these threats exist independently of the cross-language boundary we are investigating, we consider these threats out of scope.

We not assume that the underlying infrastructure is secured, but we also assume that it is providing certain standard protections: data execution prevention (DEP), address space layout randomization (ASLR), and stack canaries.

In our work in chapters 4 and 5, we assume that before we implement our protections, an attacker writing the code in the unsafe portion of a mixed-language application can write code which dereferences any arbitrary memory address. Prior work has shown that in this situation memory safety (and thus application safety) can be violated [45]. The goal of our work is to implement protections such that we prevent the violation of memory safety by limiting the read/write capabilities of the attacker.

Chapter 3

Managing Internal Unsafety

We investigate the use of `unsafe` in Rust-only applications via a case study using the operating system Tock [32] which is written entirely in Rust. In Tock, we see 3 different types of uses of `unsafe`: inline assembly, memory-mapped input-output (MMIO), and interior mutability in data structures.

In Rust, inline assembly is written as an embedded domain-specific language (DSL) using the `asm!` macro. It does not follow the same syntax or semantics as Rust, instead essentially functioning as a separate unsafe language. We treat inline assembly as a special case of our general approach to interactions with other unsafe languages (which we discuss in chapters 4 and 5), and we discuss limitations on this characterization in section 7.1.1.

The memory model assumed by MMIO code is inherently different from the standard Rust memory model, which makes it difficult to make claims about memory safety. There are approaches being separately developed within our group to handle this specific use case [24], so we treat it as out of scope here.

Our project will focus on the third type of unsafety: interior mutability in data structures. We show that we can increase the memory safety of Tock by refactoring data structures to eliminate uses of `unsafe`. We can do so by making use of similar related structures already in the Rust standard libraries and verified by the the RustBelt project [26]. In this chapter, we explain the use cases for these data structures within Tock, and why they were originally written with `unsafe`. We then demonstrate that a refactoring is possible which preserves memory safety guarantees while still compiling to the same bytecode.

3.1 Data Structures and Interior Mutability

The Tock operating system is an event-driven application, where multiple callback functions can be registered. Each callback function requires a separate writable reference to the object on which to issue a callback. By default, this would be disallowed by Rust's borrow-checking rules, which prohibit having multiple mutable references to a single object.

We get around this problem in Rust by introducing *interior mutability*, or the ability to mutate internal values even when given an immutable reference to the structure. Interior mutability is such a common pattern in Rust programs that it was added to the Rust standard library via a struct named `Cell`. `Cell` exposes a safe Rust interface for interior mutability, but it does so internally using many unsafe internal functions which operate on an internal representation struct named `UnsafeCell`. This nested abstraction allows for all unsafe code to be localized within the implementation of the `Cell` interface, instead of requiring the `unsafe` keyword in every usage of the interface. The `Cell` interface in the standard library was proven safe as part of the RustBelt effort.

Tock borrows this nested pattern for many of its core data structures to enable specific variations on `Cell`'s interior mutability, using combinations of `Cell` and `UnsafeCell` to create similar interfaces that are "safe" to use but are implemented using `unsafe` under the hood. The core interface is called `TakeCell`, and the other interior mutability data structures are explained to be specializations of `TakeCell`, optimized for different types of contained data [62]. These data structures are used in many places across the codebase, providing a container around mutable memory.

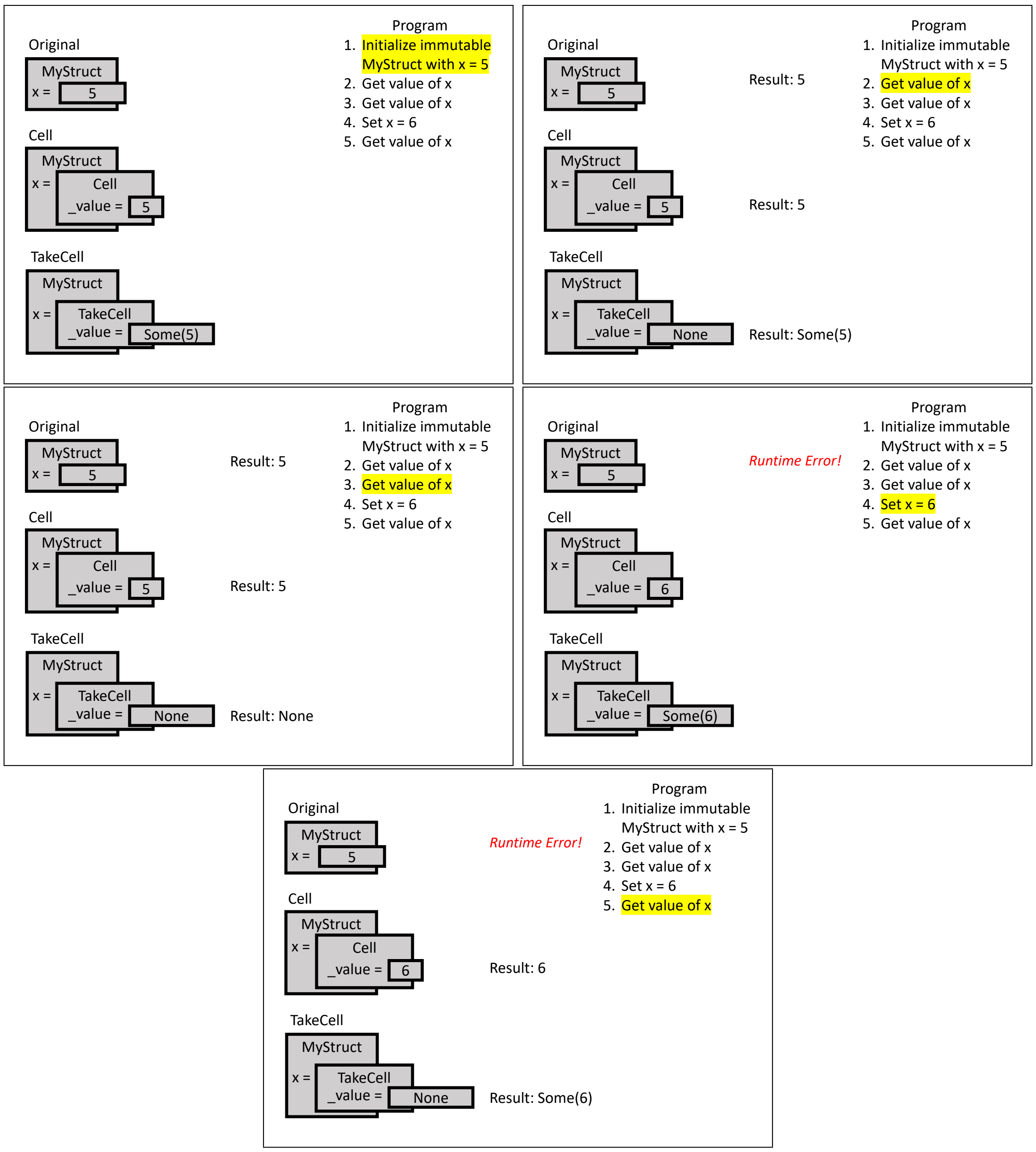
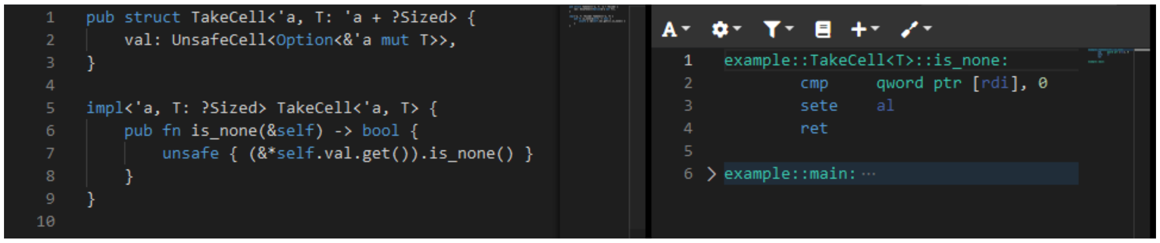


Figure 3-1: *Interior mutability example in* `Cell` *and* `TakeCell`

The key difference between `Cell` and `TakeCell` is that getting data of a `Cell` is done by copying, while getting data out of a `TakeCell` is done by moving the data. After getting data from a `Cell` the data is still there, whereas in a `TakeCell` it is missing, at least until the caller puts it back after operating on it. This is illustrated in fig. 3-1.

3.2 Refactoring Out `unsafe`

Knowing that `Cell` is proven correct enables a simple option for the memory safety of these other data structures: refactor to not use `unsafe` and/or `UnsafeCell`, instead relying only on the already-proven `Cell`.



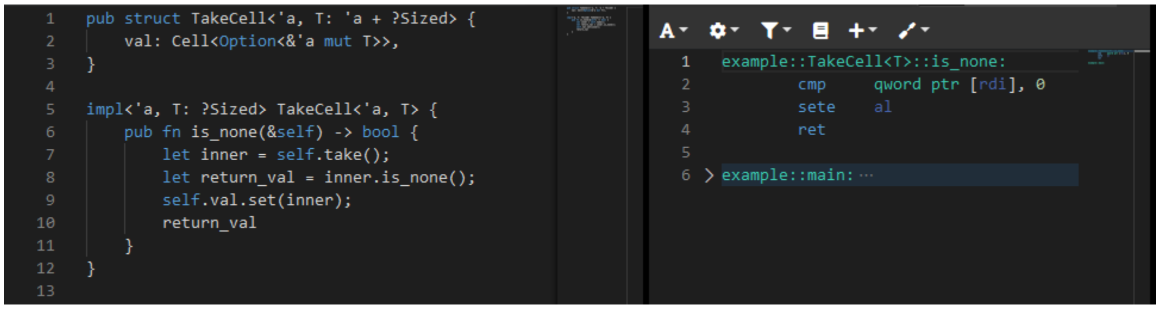
```
pub struct TakeCell<'a, T: 'a + ?Sized> {
    val: UnsafeCell<Option<&'a mut T>>,
}

impl<'a, T: ?Sized> TakeCell<'a, T> {
    pub fn is_none(&self) -> bool {
        unsafe { (&*self.val.get()).is_none() }
    }
}
```

```
example::TakeCell<T>::is_none:
        cmp     qword ptr [rdi], 0
        sete    al
        ret

example::main: ···
```

(a) *Before - unsafe code*



```
pub struct TakeCell<'a, T: 'a + ?Sized> {
    val: Cell<Option<&'a mut T>>,
}

impl<'a, T: ?Sized> TakeCell<'a, T> {
    pub fn is_none(&self) -> bool {
        let inner = self.take();
        let return_val = inner.is_none();
        self.val.set(inner);
        return_val
    }
}
```

```
example::TakeCell<T>::is_none:
        cmp     qword ptr [rdi], 0
        sete    al
        ret

example::main: ···
```

(b) *After - safe code*

Figure 3-2: *The* `is_none()` *method of* `TakeCell`*, both before and after rewritting the method in safe Rust. Screenshots are of Compiler Explorer [19], which compiles the Rust on the left to the assembly instructions on the right.*

We demonstrate that the desired interior mutability for `TakeCell` is made possible directly through `Cell` by taking ownership of the value, changing it, then putting it back. These operations can all be achieved with combinations of methods of `Cell`, instead of having to perform operations directly with `UnsafeCell`. The code for `TakeCell` was written using `unsafe` to ensure performance by operating directly on the underlying pointer, functionality which is exposed by `UnsafeCell`. However, in fig. 3-2 we show that the Rust compiler optimizations are able to recognize the safe pattern and reduce it down to the same set of assembly instructions (see section 6.1).

3.3 Summary

In looking at Tock, an operating system written in Rust, one of the main uses of `unsafe` is to enable interior mutability in data structures. We demonstrate that at least one such data structure can be refactored, eliminating the use of `unsafe` without loss of performance. We leave the application of this technique to the remaining data structures as future work and potentially a target for automation.

Chapter 4

Galeed for Heap Isolation

Much of the current usage of Rust is not limited to Rust-only applications. Instead we see a migration pattern: a longstanding codebase written in a different unsafe language (most often C/C++) is converted piece-by-piece to the equivalent Rust code. The ubiquitous web browser Firefox, our target application for the next few chapters, started their migration from C++ to Rust in 2016 [39]. Mozilla, the maintainers of Firefox, list Rust's memory safety as one of the top considerations for the switch [39].

Mixing Rust with another language (e.g. C++) breaks the Rust memory safety model, even leaving the combined mixed-language application vulnerable to exploit [45]. C++ is not bound by the Rust memory model, nor does it have to obey the restrictions of the Rust compiler. Calling into C++ from Rust breaks any promises of memory safety, and thus such calls must always be marked as `unsafe` in Rust. Unlike our solution for Tock in chapter 3, we cannot refactor away `unsafe` as we do not have a formally verified abstraction of C++ to fall back on as we do for Rust.

In a mixed Rust-C++ application, there are 4 possible patterns of memory access: Rust code accessing Rust-allocated memory, Rust code accessing C++-allocated memory, C++ code accessing C++-allocated memory, and C++-code accessing Rust-allocated memory (fig. 4-1). Rust code accessing Rust memory should never be able to break Rust memory safety (by definition). Additionally, Rust memory safety is independent of accesses to C++ memory.

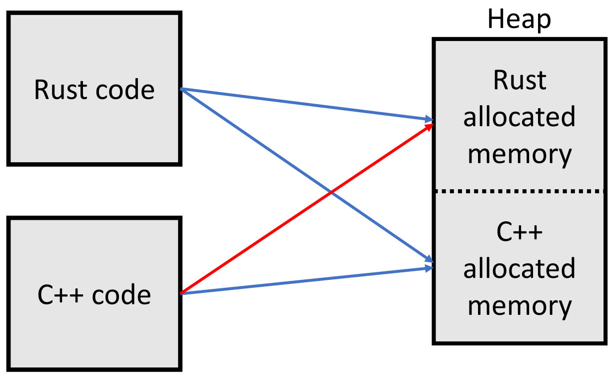


Figure 4-1: *Possible patterns of memory access in Rust-*C++ *mixed-applications*

In contrast, C++ accessing Rust memory (the red arrow in fig. 4-1) could cause any number of violations to Rust memory safety guarantees, up to and including full hijacking [45]. We separate these memory accesses further into two cases: *intended* and *unintended* accesses. An intended access occurs when C++ is explicitly given the location of some part of Rust memory by Rust code and then accesses that Rust memory, while any other access is considered unintended. We assume that if Rust does not explicitly give the other language a memory address, it does not expect the other language to modify its memory, and this assumption holds for our target of Firefox.

In this chapter, we focus on preserving memory safety in the presence of unintended accesses. We assume a lack of intended accesses for the rest of this chapter, and then modify the design to allow for intended accesses in chapter 5.

In order to preserve Rust memory safety in the Rust component of a mixed-language application, we must isolate and restrict Rust memory such that it cannot be accessed by a component written in another language. If only Rust can access Rust memory, Rust memory safety is preserved. In this chapter we propose Galeed as one way to enforce memory isolation using Intel's Memory Protection Key (MPK) technology. We present both a design and prototype implementation of Galeed, with results and benchmarks discussed in chapter 6.

4.1 Design

MPK (see section 2.2 for more details) enables quick switching of read/write permissions on groups of memory pages from userspace. Previous work has shown that using MPK to enforce different levels of isolation is a viable strategy [65, 23, 54], and libmpk [46] even provided a software abstraction for MPK for general-purpose use.

Galeed's approach to Rust memory isolation is to make sure that all of the pages of Rust-allocated memory are in the same page group, and then to use MPK to set permissions on these pages in such a way that external functions are unable to access the Rust memory. If only the given Rust component can access its own memory, and accesses from other non-Rust components to Rust memory are forbidden by MPK, then the program must stay consistent with the Rust memory model despite executing untrusted code in another language.

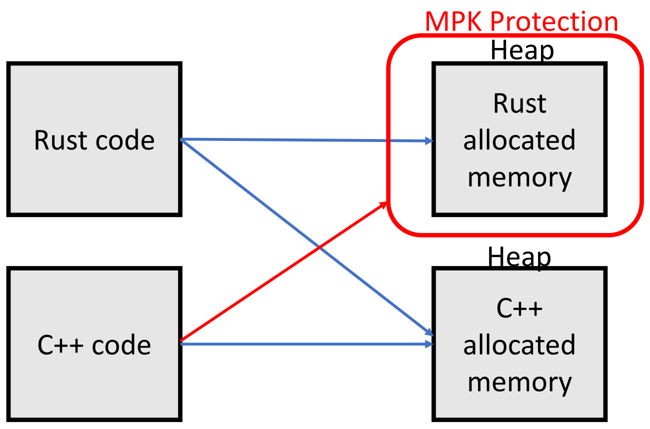


Figure 4-2: *Protection via page-level memory isolation and MPK-enabled permissions switching*

In this work we focus on isolating the Rust component's heap, leaving stack isolation to future work. General memory safety for non-Rust components is out of scope of this work, and well studied in literature [58, 40, 41, 55].

4.1.1 Heap Splitting

All Rust heap allocations must be made in the same group of pages, in order for permissions to be controlled by a single MPK key. All pages within this group must be set to the same MPK key. Permissions on these pages should initially be set to allow all read/write access, since we start in safe Rust with these allocations, and Rust should always be allowed to access its own memory. Allocations that happen in other programming languages should not be able to allocate memory in any of the same pages that Rust is allocating, so that our page-level permissions apply only to Rust memory.

4.1.2 Access Policy

Rust should always be allowed full permissions to access its own memory. The Rust compiler checks provide the memory safety protection that we need in safe Rust, and when using `unsafe` we should refer to the work mentioned in chapter 3.

Any time Rust is about to call an external function, we must first turn off all access permissions on Rust memory. Any time Rust returns from calling an external function, we immediately turn back on the permissions. While execution is controlled by the other language, permissions must remain off, ensuring that only Rust has permission to access its own memory.

Since any user-space application can modify the PKRU (the MPK permissions register), additional care is required to ensure that the external language does not turn its permissions back on. The previous MPK isolation projects all present additional techniques for solving this problem which we could adopt and reimplement, including CFI [2], binary scanning [65, 23], hardware watchpoints [23], and system call filtering via sandbox [54]. We leave this part of the implementation for future work.

4.2 Implementation

We implemented a prototype for Galeed, specialized to interactions between Rust and C++. We also include the code for our implementation at `https://github.com/eerivera/rivera-meng-appendix` [50].

Our implementation currently relies on manually inserted code annotations. In this implementation, we attempt to lay groundwork for future development and automation. We believe that automation and minimal annotation will be key to adoption of these techniques in well-established codebases, and where possible, we attempt to outline a path to full automation.

4.2.1 Initial Allocation

`libmpk` (section 2.2) provides, among other things, a heap API for allocating memory within a page group. We replace the standard Rust allocator with calls to this API (namely `mpk_alloc()` and `mpk_free()`), after updating it to match new typing information in the Linux kernel headers.

Rust provides machinery for writing a custom allocator that can be imported as a crate and used in place of the default. Our prototype does not separate the allocator into its own crate out of convenience, but doing so would allow a developer to switch to this allocator with a handful of lines of code.

In order to ensure that `libmpk` is properly initialized and has a page group assigned to it, we also must include a one-time call to `mpk_create()`. We note additional subtleties and difficulties when using the `libmpk` interface in section 7.4.

4.2.2 Access

`libmpk` restricts all access to newly allocated memory by default. We removed the line of code that did this, so that Rust by default has full access permissions in its own memory.

Code to switch MPK permissions is included on either side of all external function call sites, which have currently been manually identified. The code immediately preceding the call site switches selected permissions off, and the code immediately following the call site switches all permissions back on. We currently switch the Rust memory permissions to read-only at all call sites, but permissions could be selected at each call site by swapping named constants.

```
asm!("rdpkru", in("ecx") ecx, lateout("eax") eax, lateout("edx") _);
eax = (eax & !PKRU_DISABLE_ALL) | PKRU_ALLOW_READ;
asm!("wrpkru", in("eax") eax, in("ecx") ecx, in("edx") edx);
```

Figure 4-3: *Rust inline assembly code for MPK permission switching*

We use the previously mentioned Rust `asm!` macro to directly call the assembly instructions `rdpkru` and `wrpkru` for reading from and writing to the PKRU register which holds the MPK permissions (fig. 4-3). Note that the inline assembly code for switching permissions at any given call site is independent of any local variables or names present at that site. This means that if one could manually identify all external function call sites, one could easily insert the correct code into either end of the call site. Depending on the analysis mechanism chosen, this could be done at either the Rust or LLVM levels.

4.3 Summary

Using MPK we design and implement Galeed, a mechanism for memory isolation between Rust language components and the rest of a mixed-language application. We do this by isolating the Rust heap into a separate page group, and managing permissions on this page group across language components. In doing this, we can be confident that our strategy for preserving memory strategy is as strong as MPK. See section 7.3 for a discussion of MPK strengths and weaknesses.

Chapter 5

Pseudo-pointers

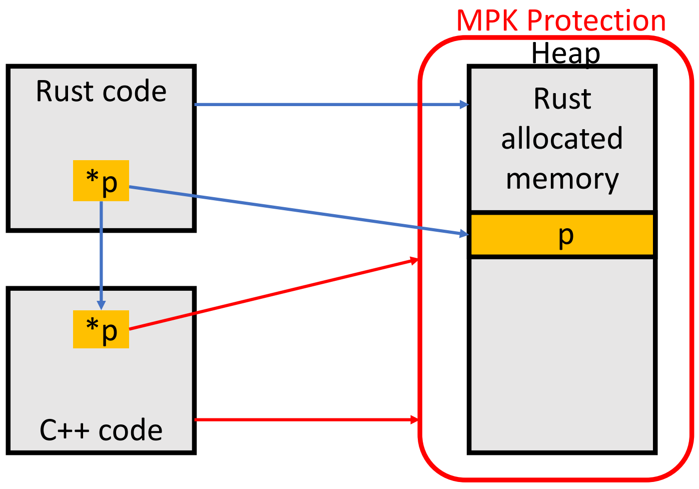


Figure 5-1: *Intended accesses are restricted by default by Galeed*

In the previous chapter Galeed intentionally excluded *intended accesses*, i.e. times when C++ is explicitly given the location of some part of Rust memory by Rust code and then accesses that Rust memory. This most commonly occurs in FFI function calls, by passing as an argument a pointer to a structure in memory instead passing directly by value. In fact, this pattern is employed by many Firefox modules, often due to performance or storage considerations. When we strictly enforce the heap isolation mechanisms, we prevent the external function from accessing the memory pointed to by the passed pointer (fig. 5-1).

In this chapter, we present an option for data flow between that does not require breaking the safety guarantees provided by Galeed's heap isolation in chapter 4. Instead, when external functions need access to Rust memory, we will force the external function to request that Rust make the change in its own memory, a request that Rust can safety-check and reject. We present a design for both the interfaces and underlying machinery required in both the Rust and external functions, followed by a proof-of-concept implementation of this design specialized to Rust and C++. Our benchmark results are discussed in chapter 6.

5.1 Design

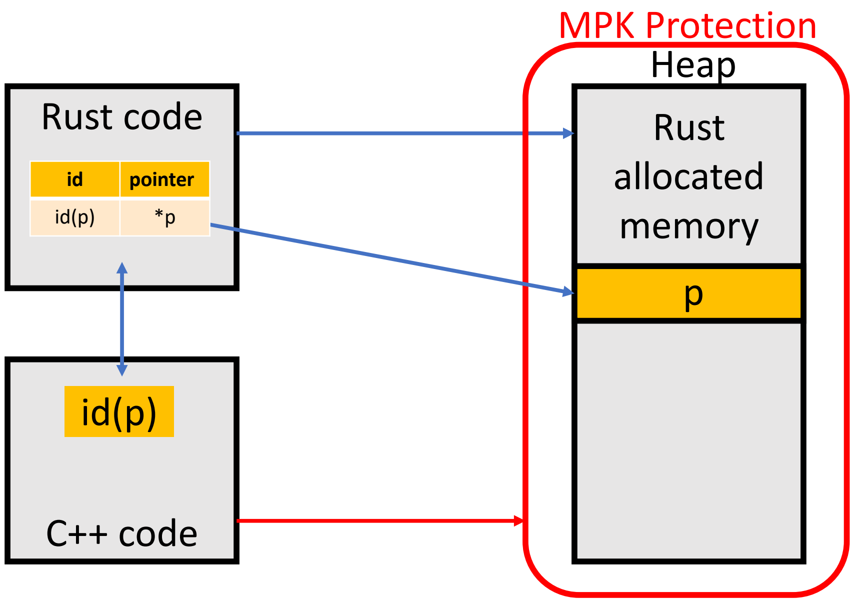


Figure 5-2: *In our design,* C++ *uses pseudo-pointers (e.g.* `id(p)`*) to request that Rust dereference Rust memory*

We introduce *pseudo-pointers*, i.e. identifiers that Rust passes to an external function instead of pointers. Rust keeps an internal mapping of pseudo-pointers to real pointers. Any time a non-Rust component attempts to dereference a Rust pointer, it must present a valid, non-expired pseudo-pointer to Rust via an exposed API, along with the information for the change it wishes to make (if applicable). Rust verifies that the pseudo-pointer is valid and non-expired. In the case of a write request, Rust also verifies that the value to write represents a valid member of the type associated with the memory location. Once verified, Rust executes the request. Since only Rust directly accesses Rust memory, we can keep our heap isolation in place and protect memory safety (fig. 5-2).

We break the design into 3 components to discuss further: necessary properties of these pseudo-pointers, the API which Rust exposes to other external components; and requirements on external functions.

5.1.1 Pseudo-pointer Properties

Pseudo-pointers need to have certain properties in order to function correctly as safe pointer identifiers: uniqueness, automatic expiration, and forgery protection.

Pseudo-pointers must be unique to the memory they represent: each pseudo-pointer must represent exactly one real memory location, and each memory location must be represented by at most one pseudo-pointer. Not only is this necessary for being able to look up the corresponding memory location, but it is also necessary to comply with the Rust borrow-checker.

Pseudo-pointers must automatically expire when the corresponding memory is freed at the latest. If a pseudo-pointer is still treated as valid and used to access memory even after its corresponding memory location has been freed, we have violated Rust memory safety with a use-after-free error.

Pseudo-pointers must be difficult to guess or forge. Ideally this applies even between different runs of the same program, which requires some level of randomization. It should be noted that while forging a valid pseudo-pointer could potentially cause information leaks or even information replacement (both major security risks), neither one has the possibility of breaking memory safety, since the operations are still controlled by safe Rust and are valid operations within the Rust memory model.

Pseudo-pointer management should be automated, and transparent to the developer. This is not a requirement for correct functionality, but is still critical in the push to incorporate these safety changes into existing applications. The more of the process that can be automated, the lower the burden on the developer. A fully automated transparent system for introducing and using pseudo-pointers reduces the surface area for potential mistakes back to at most the surface area of the original program.

5.1.2 Rust API

Pseudo-pointers are functionally useless without the corresponding external-facing Rust API, consisting of functions which can be called by another language in order to read from or write to the memory represented by a pseudo-pointer. For any given structure that will be used in the FFI, the Rust API will have a getter and setter for each field within that struct. The function names for these getters and setters will be automatically generated using a naming strategy that includes both the struct type and the field name. These functions will either be nops or raise errors when asked to perform a memory operation that is inconsistent with its current internal understanding of that memory location, including both type errors and expired pseudo-pointers. These functions must also be entirely in safe Rust, where compile-time and run-time checks automate most of this for us.

5.1.3 External Function Transformation

Pseudo-pointers are passed in place of pointers in every call to an external function, to avoid ever passing a Rust memory location to another language. Before each external function call, we create a pseudo-pointer for the pointer that would normally be passed, and pass that instead. We invalidate the pseudo-pointer once the function returns, for the reasons mentioned in section 5.1.1.

If we rewrite calls to external functions to use pseudo-pointers, we will also need to rewrite the external functions themselves to accept and use these pseudo-pointers everywhere that they would have had a real pointer instead. Pointer dereferences and writes need to be converted into the equivalent Rust API calls from section 5.1.2.

Ideally, these rewrites can be done automatically, which would once again mitigate the burden on the developer. In fact, full automation of these external rewrites would allow us to secure calls to large existing legacy libraries with little to no change, allowing for this technique to be used in cases like migration from a legacy codebase in an unsafe language (e.g. Firefox, originally in C++). Additionally, since developers are often hesitant to make changes (even automated ones) to working legacy code, these rewrites should be able to be performed at compile time instead of modifying the source file.

Aliasing in unsafe languages could stand as a barrier to the full automation above, as it may be impossible to completely determine the full set of pointer dereferences for a Rust object. We note that ours is a conservative approach prioritizing guaranteed safety. In cases where alias analysis fails and a pointer dereference is not transformed, that pointer dereference will be disallowed by MPK permissions and will not violate memory safety. A human-in-the-loop system could be designed for such situations, to enable quick debugging and additional annotations to guide the analyzer. We leave this as future work.

5.2 Implementation

We implement the above extension to Galeed's design specialized to Rust/C++ interactions, noting that our strategy would be effective in securing Rust calls to external functions in many other languages. We keep this extension separate from our previous Galeed prototype, with integration as future work.

For our prototype, we only implement pseudo-pointers for user-defined structs that are intended to be passed across the language boundary. For primitives like booleans, integers, and floating-point numbers, we would normally expect these to be passed by value directly. For other constructs in the language and/or standard library, further work is required to implement the necessary transformations.

As in the last chapter, this implementation will also focus very heavily on automation and developer convenience, for the same adoption reasons previously mentioned.

In places where we currently rely on manual entry, we also lay out a path to more complete automation.

5.2.1 Pseudo-pointers

Pseudo-pointers are implemented as a transparent struct containing a single field, the id of the pseudo-pointer as a signed 32-bit integer. The struct also contains a `PhantomData` field that is the type of the data being pointed to. `PhantomData` is used in Rust for fields that exist at compile time but not at run time. This allows us to make distinctions in code between pseudo-pointers that represent different types, while still having confidence that they will still compile down to 32-bit identifiers once all of the compiler checks are passed.

We also define a specific map struct for pseudo-pointers, including a function that takes a Rust struct, adds it to the map, and returns the corresponding pseudo-pointer, and the reverse function that takes a pseudo-pointer, removes it from the map, and returns the Rust struct. Every time a struct is added to the map, it will be added with a different id, and every time a struct is retrieved, that id becomes invalid. This prevents external functions from attempting to access a struct after Rust has reclaimed it.

It is worth noting that creating pseudo-pointers using this interface requires having ownership of the object. One cannot just have a writable reference to the object. This is how we ensure temporal memory safety, as Rush requires the object's lifetime must extend for at least as long as it is in the map.

Pseudo-pointer support is implemented as an attribute macro that can be added to a struct. This attribute macro creates the global map that will hold all pseudo-pointers of this struct type. Additionally, the macro automatically creates the API that will be exposed to external functions, as described in the next section. In our prototype, we add this attribute manually to structs being passed to external functions, but it is easy to imagine a static analysis tool that could find external function call sites at compile time, determine which structs get passed to those functions, and automatically add the attribute to those structs.

5.2.2 Rust API

The attribute macro is able to generate getter and setter functions for each field of a struct by name. The macro has access to the type information of each field, so these functions are able to carry that type information in their return value and arguments respectively.

These functions use the pseudo-pointer provided as an argument, and go to the appropriate pseudo-pointer map to request access. If the pseudo-pointer is valid, the function proceeds as expected, either reading or writing the appropriate value. If the pseudo-pointer is invalid, the function will panic. We could just as easily have failed silently with a nop, but for prototype purposes we felt that it was better to be immediately alerted of the violation and to halt to prevent further violations.

In addition to generating the getter and setter functions based on the name of a field, we also generate equivalent functions based on that field's position in the struct. This enables some of the low-level automation described in the next section.

5.2.3 External Function Transformation

```
int add5(MyStruct* const p) {
    p->x += 5;
}

```

(a) Before

```
int add5(ID<MyStruct> const p) {
    x = get_x_in_MyStruct(p);
    set_x_in_MyStruct(p, x+5);
}
```

(b) After

Figure 5-3: *Transforming an example* C++ *function to use pseudo-pointers*

In order to use this new pseudo-pointer interface, external C++ functions that once accepted pointers to structs in memory must be modified to instead accept pseudo-pointers, and operations on those pointers must be replaced with the appropriate Rust API getters and setters above. Figure 5-3 shows an example of this transformation.

Instead of placing the burden on developers to manually perform these transformations, we choose to automate this transformation process. We introduce a module-level pass into the LLVM compiler which is enabled by a command-line flag. This pass transforms identified functions by replacing the expected pointer argument with a pseudo-pointer argument. It then traces uses of that argument through the function, replacing load instructions with calls to the correct getter function and store instructions with calls to the setter function. The information needed to determine the correct function can be found in the type information that LLVM preserves.

Currently we rely on manual annotations to determine which functions to transform and which structs within those functions to turn into pseudo-pointers. If we have the ability to automatically identify call sites on the FFI boundary, we can automate this annotation process, subject to the limitations discussed in section 5.1.3.

5.3 Summary

We can enable external functions to have access to Rust memory by requiring them to go through Rust every time they want to access memory, and allowing Rust to accept/reject such a request. Our prototype enables automatic transformations on both sides of the language boundary, which translate unsafe direct memory accesses into safe function calls back to Rust. This allows us to preserve the Rust memory safety previously protected by MPK-enabled heap isolation.

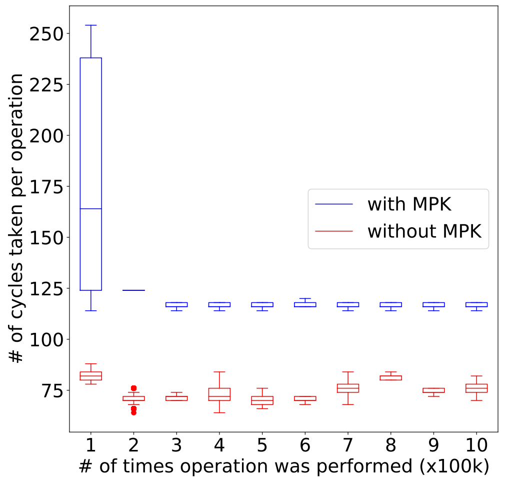
Chapter 6

Benchmarks

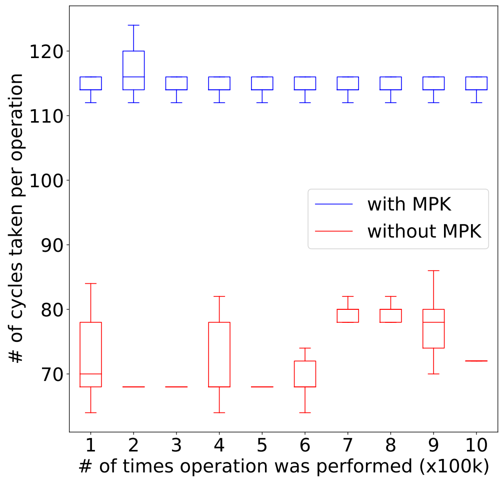
Chapters 3 to 5 have all laid out different code transformations which can strengthen the overall memory safety guarantees of the program. In this chapter we evaluate our safety claims and calculate the performance overhead costs for our prototype implementations of all of these code transformations.

6.1 Refactoring Unsafe Rust

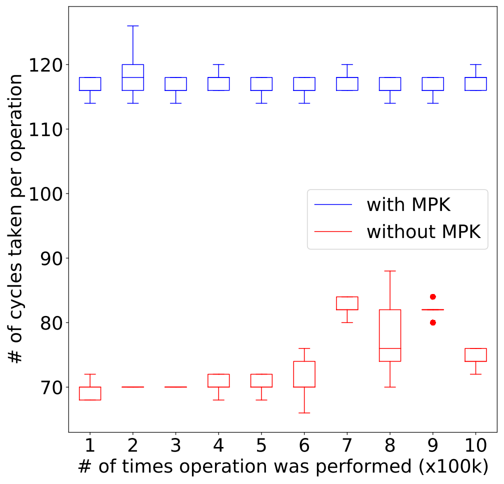
In chapter 3 we demonstrate our claim that data structures in Tock can be rewritten using only safe Rust. We do so by taking a structure called `TakeCell` as a case study and rewriting it. Upon making the code changes to use only safe Rust, we compile the `TakeCell` module down to LLVM and compare it to the compiled LLVM from before the changes. We find that the Rust compiler optimizations are able to correctly recognize the safety patterns and transform the code into the appropriate memory accesses, and so in fact the two LLVM bytecode files were identical (see fig. 3-2 for an example). This means that for `TakeCell`, there is 0% overhead to write in a safe way. We were able to demonstrate this to the maintainers of Tock, and successfully had our pull request accepted.



(a) Single Read



(b) Single Write



(c) Write then Read

Figure 6-1: *Galeed microbenchmarks*

6.2 Galeed for Heap Isolation

In chapter 4 we present Galeed which isolates Rust heap memory from another language's heap memory using Intel MPK. We implemented a prototype of Galeed to work at the Rust/C++ boundary. MPK is still a new hardware technology. At the time of writing, it is only available on the newest line of Intel's server-class CPUs

(Skylake). Our university was able to grant us remote headless use of one such machine, but we did not have the full access that we would like. While we can still be convinced of our safety claims in this section, our performance evaluations have higher deviations than we would prefer.

6.2.1 Proof-of-concept

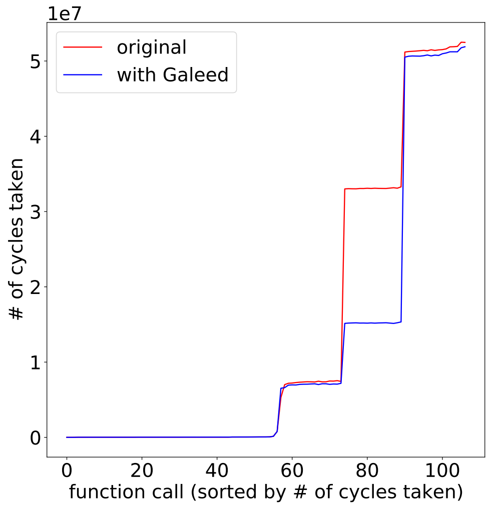
We created a small proof-of-concept application using Rust and C++, where the C++ side has a “library” of functions which took in a pointer from Rust and read from or wrote to that location in memory. We are able to show that disabling read/write permissions at callsites of external functions in Rust using Galeed is enough to force a segmentation fault in the appropriate corresponding C++ function before it has the chance to access Rust memory.

We evaluate the performance overhead of Galeed per function call using microbenchmarks (fig. 6-1). We find that Galeed protections add an overhead of $\sim$50 cycles on average, which is consistent with prior work using MPK [65, 46].

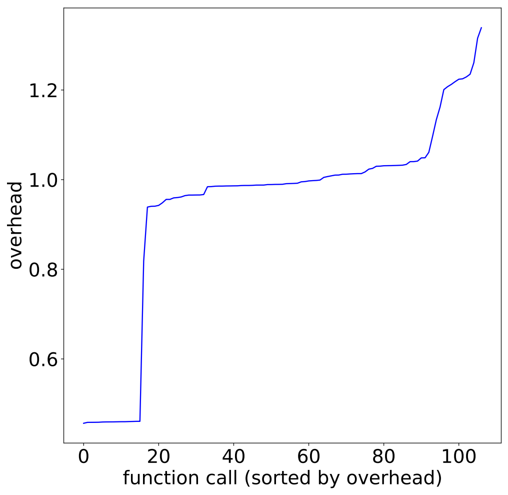
6.2.2 *libpref*

Convinced that our proof-of-concept works, protects the Rust memory safety guarantees, and does so with an acceptable performance overhead, we implement Galeed again within Firefox. We target the *libpref* module within Firefox, which is used to parse a file to collect user preferences. We use Firefox’s own *libpref* module test suite as benchmarks. We discard 5 of the tests which failed on the machine even when running Firefox unmodified. Preliminary troubleshooting indicates that these test failures are due to the headless mode of our server access.

Once again, we were able to show that disabling permissions at external function callsites using Galeed is enough to force a segmentation fault. Additionally, we found that enabling read-only permissions at these callsites is actually sufficient to allow Firefox to run normally and to pass the suite of tests for *libpref*. Read-only access is sufficient to show that Rust memory safety is not broken. Memory errors can still occur with read-only access from the C++ component, but such errors are contained to the C++ component and cannot affect the Rust component's internal memory model.

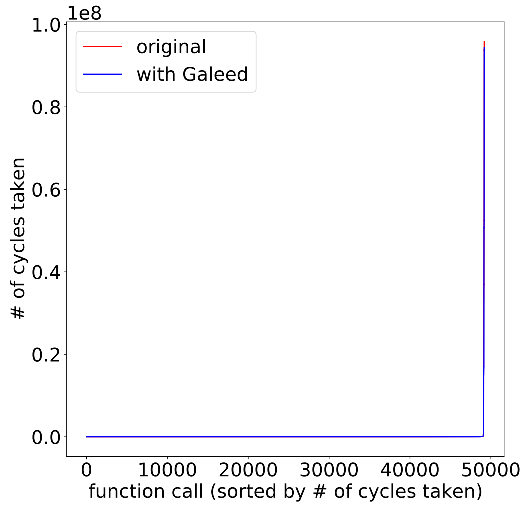


(a) Cycle counts - Rust function calls

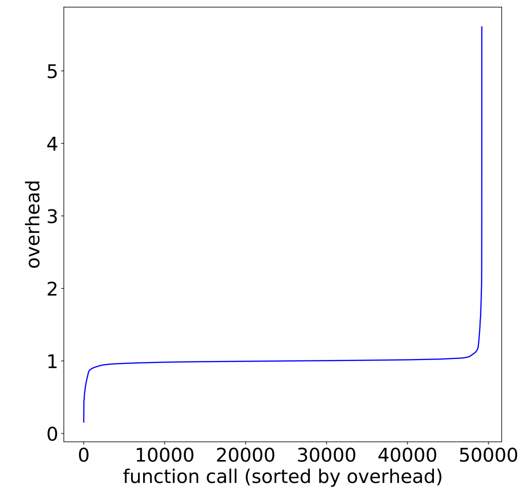


(b) Galeed overhead - Rust function calls

Figure 6-2: *Galeed libpref benchmarks - Rust component*



(a) Cycle counts - all function calls



(b) Galeed overhead - all function calls

Figure 6-3: *Galeed libpref benchmarks - all function calls*

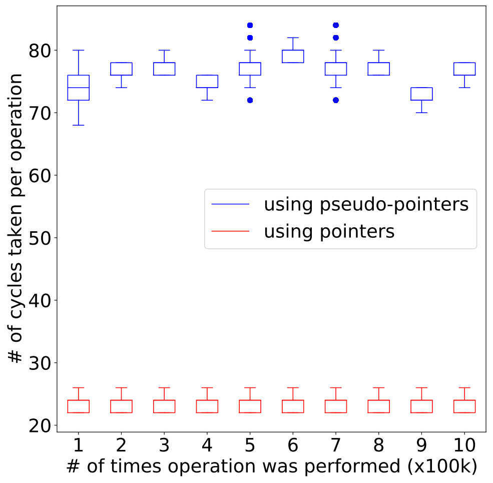
We ran the test suite 1,000 times with the *libpref* module as written, and 1,000 times again after adding Galeed, and we compare the results here. The test suite is written in JavaScript, with Firefox machinery allowing it to hook directly to C++ function calls. Only a subset of these C++ function calls directly call the Rust component (the preference parser). In order to attempt an accurate comparison, we time each function hooked by the test suite using `rdtscp` and report the cycle count for each invocation of the function. We present results both specifically for the parser as well as the overall test suite.

We find an average overhead of <1% using Galeed for the Rust component (fig. 6-2), with an even lower average overhead in the application overall (fig. 6-3). We do see some variance stemming from earlier mentioned constraints on our MPK machine access, and further work is needed to confirm that this is the primary cause. Nevertheless, we are confident in this assessment of the data, as it aligns with results in prior work using MPK [65, 46].

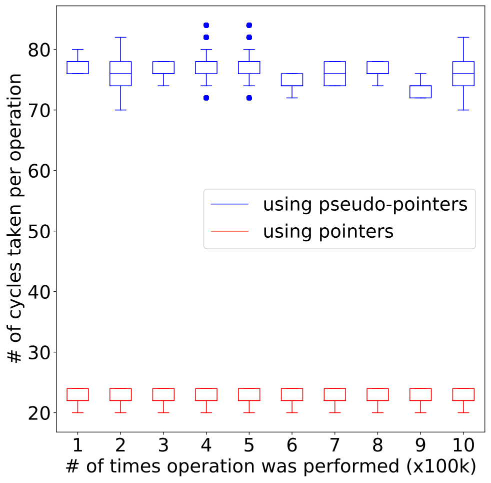
6.3 Pseudo-pointers

In chapter 5 we augment Galeed by replacing pointers being passed across the language boundary with pseudo-pointers, identifiers that could be be used to request information from Rust memory through function calls without needing direct memory access. We implement a prototype that works for Rust/C++. The Rust transformations are inserted manually, but the C++ transformations are performed automatically by a custom pass in the LLVM compiler. Due to the resource limitations on MPK machines mentioned in the previous section, we chose to implement and benchmark this prototype separate from the previous Galeed prototype.

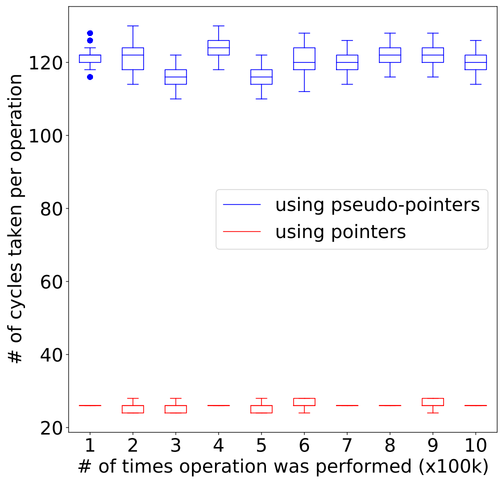
We developed another proof-of-concept application very similar to section 6.2.1, where the C++ side has a "library" of functions which took in a pointer to a Rust struct and read from and/or wrote to that struct. We are able to show that the compiled unit for this application had replaced all pointer dereferences and writes for the Rust struct with the corresponding Rust function calls for that struct. Rust pointers were never accessed from C++, while other pointers not from Rust were left unaffected.



(a) *Single Read*



(b) *Single Write*



(c) *Write then Read*

Figure 6-4: *Pseudo-pointer microbenchmarks*

We evaluate the runtime performance overhead of adding these additional function calls using microbenchmarks (fig. 6-4). We found that there is ∼3x overhead for each individual read/write operation; however, when operations are chained the overhead is not ∼6x as expected but instead ∼4.5x, indicating that the compiler toolchain is inserting additional optimizations post-transformation.

Chapter 7

Discussion

There are a number of factors that affect the scope of our current and future work. We discuss some of them here.

7.1 Active Rust Development

Constant changes to the Rust language and standard libraries mean that verification of features will necessarily lag behind language development. In the past year since we first proposed the `TakeCell` changes mentioned in section 3.1, the previously discussed `Cell` library added new methods which RustBelt has not yet been updated to verify. Developers seeking to only use formally verified libraries must be aware of the time delay between language implementation and formal verification, and plan accordingly. Some projects using Rust have pinned themselves to a specific release, to avoid other difficulties with a constantly changing language. This strategy can be adopted to close the gap between implementation and verification.

7.1.1 Inline Assembly

Another ongoing change in the Rust language is its handling of inline assembly via the `asm!` macro. The details of this macro have not been finalized, and inline assembly is still only available on "nightly" builds of Rust. We treat inline assembly like a different language in chapter 3 because it is, with different syntax and semantics, and is necessarily unsafe. However it is also unlike every other language that Rust can interact with, because it does not do so through external function calls (i.e. the Foreign Function Interface, or FFI). The assembly memory model requires knowledge of the underlying architecture in a way that most other modern languages do not. Some of our memory isolation principles might still apply, and we believe it would be interesting future work to see what analysis could be done on Rust's inline assembly once finalized.

7.2 Tock Data Structures

Due to the similarity of `TakeCell` to all of the other interior mutability data structures in Tock [62], we strongly believe that almost all of the data structures could be rewritten using the strategies presented in chapter 3. However, we did not invest the time in doing so. We instead recognize that rewriting interfaces in a way that satisfies the borrow-checker, using only standard library code that has been verified, may mean that the code is potentially more difficult and time-consuming to read or write. We choose to leave this as a potential target for automation in future work, or as a suggestion for expansion of the RustBelt project.

7.3 Memory Protection Keys

We rely on Intel Memory Protection Keys (MPK) to enable/disable permissions on groups of pages, which in turns helps to enforce isolation between Rust and other languages. MPK is a new hardware technology, and is currently only available in the newest Skylake server CPUs. The assembly instructions used to read and adjust MPK permissions trigger segfaults in Intel machines that do not have MPK. This means that solutions such as ours that rely on MPK for memory safety will have to wait at least a few years while the hardware catches up to the consumer market.

MPK by itself has been shown to not be a perfect solution to memory isolation. The PKRU register (used to manage MPK permissions) can be written from userspace, opening the door for a malicious agent to adjust its own permissions. This has prompted projects such as ERIM [65], Hodor [23], and Donky [54], built on top of MPK to strengthen the protections it provides by preventing execution of PKRU instructions. ERIM and Hodor do this by analysing the compiled binary to find these instructions, while Donky provides a runtime sandbox which safely controls the PKRU. Vulnerabilities have been found in the above approaches [8], and finding solutions to these MPK vulnerabilities will be critical in ensuring the memory safety claims of this thesis.

7.4 `libmpk`

Our solution relies not only on MPK, but on the open-source project `libmpk` [46], a software abstraction developed around MPK. `libmpk` is implemented as a C library, and we currently rely on their heap abstractions for allocation and deallocation of memory by calling that library in our allocator. We trust these abstractions by necessity in our prototypes, but in future work these abstractions should be rebuilt in Rust and optimized for performance.

One reason that these abstractions should be rebuilt is that they are currently broken. When first working with `libmpk`, we attempted to use it as-is. Unfortunately, `libmpk` has not been maintained since its paper was published in 2019, and function definitions needed to be updated to reflect changes in the Linux kernel. Additionally, the heap abstractions do not work if following the calling convention mentioned in the paper, and one cannot use both the general `libmpk` library and the heap abstractions at the same time. The general `mpt_init()` function allocates all MPK keys to itself, to then virtualize and distribute across pages as requested. However the heap initialization function `mpk_create()` does not acquire a virtual key created by the above function, but instead tries to request a hardware MPK key directly via system call. This request fails because `mpt_init()` already holds all possible hardware keys, and the program segfaults.

We modified the heap abstractions to successfully initialize required data, and then used solely these abstractions. Multiple research projects have shown that memory allocator design has an impact on performance [12, 34, 15]. We are not experts in memory allocation, and almost certainly introduced a performance overhead when implementing our modifications. Future work should include an updated memory allocator that is natively aware of MPK.

As MPK hardware becomes more prevalent going forward, we hope that more developers will take an interest in using the technology for memory isolation. We hope to see `libmpk` or an equivalent come up-to-date and address some of the current shortcomings, to give developers software-level abstractions for these protections.

7.5 Firefox

For our mixed-language solutions, we targeted Firefox as a case study. Firefox has a large codebase written in C++, that is slowly being migrated to Rust. Memory safety is cited as one of the main reasons for the switch, which makes it an exceptionally apt target for us as we consider how to preserve Rust memory safety. However, working with such a large multi-language codebase brings complications.

One of the largest barriers to progress when working with Firefox has been their build system infrastructure. Large amounts of code are automatically generated from configuration files, which themselves are written in different custom languages. This makes it difficult to manually insert small adjustments along the process, including: linking external libraries (e.g. `libmpk`), enabling LLVM settings on certain files (e.g. enabling our pseudo-pointer transformation), or just manually inserting sanity checks during development. These challenges are common to most large codebases.

7.6 Prototypes

In chapters 4 and 5, we made multiple choices to bound the scope of our prototypes. Here we discuss these decisions, and some of the work needed to push these prototypes to production-readiness.

7.6.1 Performance

In our prototypes, we intentionally focused on preserving memory safety first, sometimes to the detriment of performance. In the Galeed prototype, we rely on the unoptimized `libmpk` library for our memory allocation and deallocation steps. In the pseudo-pointer prototype, we replace C++ pointer access with external function calls, performing this step before either compiler has a chance to potentially optimize some of these accesses away. And in both prototypes, we made no attempts to allow for LLVM's cross-language link time optimization (LTO).

A high performance overhead is a barrier to industrial acceptance. For many applications, the current overhead of our pseudo-pointer implementation is simply unacceptable. We do believe that much of this overhead can be optimized away, and all of these potential optimization targets should be investigated.

7.6.2 Automation

In chapters 4 and 5 we make multiple attempts to call out opportunities for automation, with the end goal being a fully automated compiler process that requires little to no developer input. We have already achieved this on the C++ end with the LLVM pass that automatically replaces Rust struct pointers with pseudo-pointers and inserts the correct function calls, but many opportunities are still available on the Rust side.

Automation reduces the developer burden, which in turn also increases the likelihood of widespread acceptance. It also reduces the opportunities for developer errors like forgetting to insert permissions switching at a call site, further strengthening our safety claims.

7.6.3 Type Support

In our pseudo-pointers prototype, we currently support flat user-defined structs. This covers a large amount of use cases, but must be expanded to accomodate current Rust/C++ interactions. For example, right now we do not support strings, which is a large portion of the use cases in the parsing modules that Firefox has migrated to Rust so far. To promote adoption, we should be able to handle most if not all of the current types of pointers flowing across the language boundary.

7.6.4 Source Code Dependence

Our prototypes all currently depend on having access to the original source code for Rust, and at minimum the LLVM bytecode for C++. We believe this is a valid model to consider, as most projects have all of the source code for both Rust and C++ available at compile time.

Chapter 8

Related Work

Below we discuss works related to our project in 3 major areas: formal reasoning about Rust (chapter 3), code/memory isolation (chapter 4), and program transformations for safety (chapter 5).

8.1 Formal Reasoning about Rust

Our work relies heavily on the inherent memory safety guarantees of the Rust language. Attempts to formalize and prove these guarantees began with Patina [49] in 2015, though the work built upon decades of prior PL theory. Patina formalized a small model of Rust which did not account for `unsafe`, and so the RustBelt project [26, 10] built another formalization of a realistic subset of Rust. RustBelt used the Iris framework for concurrent separation logic [28] to prove memory safety properties. RustBelt went even further and also verified some standard libraries which contained `unsafe`. CRUST [64] also verified memory safety properties of unsafe library code by translating Rust into C code then performing bounded model checking. While limited, this approach did prove to be able to find memory errors in Rust standard libraries.

There is also a body of work around verification of assembly code, which is one of the uses of `unsafe` that we leave for future work. The Vale line of work [4, 18] presents a language and framework for proving properties of assembly programs and even automating those proofs. TINA [48] automatically lifts inline assembly within C code to semantically equivalent C code, easing the burden of analysis and verification tools.

Static analysis tools have also been built on top of Rust to preserve properties beyond memory safety, including the information control flow work by Balasubramanian et al. in 2017 [3].

8.2 Isolation

There have been many research efforts into efficient and effective isolation at both the software and hardware levels.

Many software isolation techniques rely on sandboxing untrusted code [66]. Native Client [69] specifically provides this sandboxing for untrusted browser-based applications, while Vx32 [17] allows native applications to sandbox untrusted plug-ins. Sandcrust [30] targets the same domain as our work: applications that mix Rust and C. Sandcrust offers protection by moving unsafe C code to execute in a new sandboxed process in a different addres space, and using remote procedure calls (RPC) to communicate between the two languages. Our solution uses the Rust foreign function interface (FFI) instead which allows for lower overhead.

Many hardware-based isolation techniques rely on additional metadata/tags on pointers and/or memory locations [13]. CHERI [68] is an extension to the RISC architecture that allows for fine-grained capability checking against a pre-defined memory model, which was demonstrated to be able to provide compartmentalization [67]. The Dover processor [59] uses PUMP [14, 11] infrastructure to provide programmable metadata that is kept completely separate and inaccessible from application data at a hardware level. Policies can be implemented to ensure a variety of properties, including safe MMIO interactions [24].

Multiple compartmentalization projects have been built on top of Intel's Memory Protection Keys (MPK) [25] technology [65, 23, 54], and `libmpk` [46] is a software abstraction around MPK, meant to provide easier developer access. MPK has been shown to have vulnerabilities that these projects do not prevent [8].

8.3 Compile-time Transformations

The current widely-used solution for FFI between Rust and C++ is a project called CXX [63]. CXX claims to be able to statically analyze both sides of a Rust/C++ boundary, where the Rust code is written entirely in safe Rust using references and obeying borrow-checking rules, and then emit equivalent unsafe Rust code working directly with pointers. This code is what is ultimately compiled into the final application. Our project takes many cues from CXX, but ultimately felt the need to rebuild much of our machinery from scratch. We were not comfortable emitting unsafe Rust code and still claiming memory safety, and could not find a way to validate their static analysis claims.

Other projects have also used compile-time transformations to strengthen safety. AddressSanitizer [55] automatically inserts code instrumentation that, when coupled with their runtime library, is able to detect memory safety violations in C programs. SoftBound [40] automatically transforms C code to also record bounds metadata on pointers, and to check/update this metadata when loading or storing pointers. CCured [42] extends C's type system to include pointer usage information. It uses this information to statically verify that memory errors cannot occur, and automatically inserts runtime checks in places where static analysis fails.

Chapter 9

Conclusion

The Rust programming language offers a combination of performance and memory safety guarantees which is increasingly drawing developers to use it, but `unsafe` in Rust can undermine claims to memory safety. In this thesis, we have shown that we can still preserve memory safety guarantees in the presence of `unsafe`, in both Rust-only and mixed-language applications.

We advocate for the refactoring of unsafe Rust in Rust-only applications into safe equivalents, showing that Rust compiler optimizations are intelligent enough to avoid introducing a performance cost. In situations where `unsafe` represents an otherwise unavoidable compiler restriction, we rely on the increasing power of formal methods tools such as RustBelt to help developers verify memory safety.

We also present Galeed, designed to preserve Rust memory safety in mixed-language applications. We demonstrate that MPK can be used to isolate and protect Rust memory from other languages within the same application, and then present an interface for these other languages to still request access to memory, but in a way that is ultimately still guarded by Rust's memory model. We present initial prototypes for Galeed, automation work in support of both of these prototypes, and potential directions for future improvements on the path to widespread adoption of these safety tools.

Bibliography

[1] *New American Standard Bible.* The Lockman Foundation, 1995.

[2] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. 4control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* (2009).

[3] Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z., and Ryzhyk, L. System programming in Rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)* (2017).

[4] Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K. R. M., Lorch, J. R., Parno, B., Rane, A., Setty, S., and Thompson, L. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium (USENIX Security 17)* (2017).

[5] Burch, A. Using Rust in Windows. `https://msrc-blog.microsoft.com/2019/11/07/using-rust-in-windows`, 2019.

[6] Center for Internet Security. Multiple vulnerabilities in Google Android OS could allow for arbitrary code execution. `https://www.cisecurity.org/advisory/multiple-vulnerabilities-in-google-android-os-could-allow-for-arbitrary-code-execution_2019-088`, 2019.

[7] Cimpanu, C. A Rust-based TLS library outperformed OpenSSL in almost every category. `https://www.zdnet.com/article/a-rust-based-tls-library-outperformed-openssl-in-almost-every-category`, 2019.

[8] Connor, R. J., McDaniel, T., Smith, J. M., and Schuchard, M. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)* (2020).

[9] Criswell, J., Geoffray, N., and Adve, V. S. Memory safety for low-level software/hardware interactions. In *USENIX Security Symposium* (2009).

[10] Dang, H.-H., Jourdan, J.-H., Kaiser, J.-O., and Dreyer, D. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages (POPL)* (2019).

[11] De Amorim, A. A., Dénès, M., Giannarakis, N., Hritcu, C., Pierce, B. C., Spector-Zabusky, A., and Tolmach, A. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy* (2015).

[12] Detlefs, D., Dosser, A., and Zorn, B. Memory allocation costs in large C and C++ programs. *Software: Practice and Experience* (1994).

[13] Devietti, J., Blundell, C., Martin, M. M., and Zdancewic, S. Hard-Bound: Architectural support for spatial safety of the C programming language. *ACM SIGOPS Operating Systems Review* (2008).

[14] Dhawan, U., Vasilakis, N., Rubin, R., Chiricescu, S., Smith, J. M., Knight Jr, T. F., Pierce, B. C., and DeHon, A. PUMP: A programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2014).

[15] Durner, D., Leis, V., and Neumann, T. On the impact of memory allocation on high-performance query processing. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN)* (2019).

[16] Filho, W. A. Rust in the Linux kernel - Google security blog. `https://security.googleblog.com/2021/04/rust-in-linux-kernel.html`, 2021.

[17] Ford, B., and Cox, R. Vx32: Lightweight, user-level sandboxing on the x86. In *USENIX Annual Technical Conference* (2008).

[18] Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., and Swamy, N. A verified, efficient embedding of a verifiable assembly language. *Proceedings of the ACM on Programming Languages (POPL)* (2019).

[19] Godbolt, M., et al. Compiler Explorer. `https://godbolt.org/`.

[20] Google. Chromium. `https://www.chromium.org/Home`.

[21] Google. Google Chrome. `https://www.google.com/chrome`.

[22] Google. Memory safety - the Chromium projects. `https://www.chromium.org/Home/chromium-security/memory-safety`. Accessed on 2021-05-14.

[23] Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M. L., Shen, K., and Marty, M. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019).

[24] Huang, A. Software defined memory ownership system. Master's thesis, Massachusetts Institute of Technology, 2020.

[25] Intel. Intel®64 and IA-32 architectures software developer's manual, 2021.

[26] Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages (POPL)* (2017).

[27] Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. Safe systems programming in Rust: The promise and the challenge. *Communications of the ACM* (2020).

[28] Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., and Dreyer, D. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices* (2015).

[29] Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., and Fetzer, C. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)* (2017).

[30] Lamowski, B., Weinhold, C., Lackorzynski, A., and Härtig, H. Sandcrust: Automatic sandboxing of unsafe components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS)* (2017).

[31] Levick, R. Why Rust for safe systems programming. `https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming`, 2019.

[32] Levy, A., Campbell, B., Ghena, B., Giffin, D. B., Pannuto, P., Dutta, P., and Levis, P. Multiprogramming a 64kB computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017).

[33] Linux Kernel Organization. The Linux kernel archives. `https://www.kernel.org`.

[34] Manghwani, R., and He, T. Scalable memory allocation. `https://locklessinc.com/downloads/Preso05-MemAlloc.pdf`, 2011.

[35] Microsoft. Lesson 2 - Windows NT system overview. `https://docs.microsoft.com/en-us/previous-versions//cc767881(v=technet.10)`, 2014.

[36] Miller, M. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. `https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01-BlueHatIL-Trends,challenge,andshiftsinsoftwarevulnerabilitymitigation.pdf`, 2019.

[37] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* (1978).

[38] Mozilla Foundation. Firefox. `https://www.mozilla.org/en-US/firefox`.

[39] Mozilla Foundation. Oxidation. `https://wiki.mozilla.org/Oxidation`. Accessed on 2021-05-14.

[40] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009).

[41] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. CETS: Compiler-enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)* (2010).

[42] Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2005).

[43] Newsham, T. Format string attacks. `http://hackerproof.org/technotes/format/formatstring.pdf`, 2001.

[44] One, A. Smashing the stack for fun and profit. *Phrack magazine* (1996).

[45] Papaevripides, M., and Athanasopoulos, E. Exploiting mixed binaries. *ACM Transactions on Privacy and Security (TOPS)* (2021).

[46] Park, S., Lee, S., Xu, W., Moon, H., and Kim, T. `libmpk`: Software abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019).

[47] Python Software Foundation. The Python programming language. `https://github.com/python/cpython`.

[48] Recoules, F., Bardin, S., Bonichon, R., Mounier, L., and Potet, M.-L. Get rid of inline assembly through verification-oriented lifting. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019).

[49] Reed, E. Patina: A formalization of the Rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015).

[50] Rivera, E. Preserving memory safety in safe Rust during interactions with unsafe languages–source code appendix. `https://github.com/eerivera/rivera-meng-appendix`.

[51] Rust Foundation. Meet safe and unsafe - the Rustonomicon. `https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html`. Accessed on 2021-05-14.

[52] Rust Foundation. Rust programming language. `https://www.rust-lang.org`.

[53] Rust Foundation. What is ownership? - the Rust programming language. `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`. Accessed on 2021-05-14.

[54] Schrammel, D., Weiser, S., Steinegger, S., Schwarzl, M., Schwarz, M., Mangard, S., and Gruss, D. Donky: Domain keys–efficient in-process isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)* (2020).

[55] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012).

[56] Seyster, J., Radhakrishnan, P., Katoch, S., Duggal, A., Stoller, S. D., and Zadok, E. Redflag: A framework for analysis of kernel-level concurrency. In *International Conference on Algorithms and Architectures for Parallel Processing* (2011).

[57] Shacham, H., et al. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security (CCS)* (2007).

[58] Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., and Franz, M. SoK: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019).

[59] Sullivan, G. T., DeHon, A., Milburn, S., Boling, E., Ciaffi, M., Rosenberg, J., and Sutherland, A. The Dover inherently secure processor. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)* (2017).

[60] Szekeres, L., Payer, M., Wei, T., and Song, D. SoK: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy* (2013).

[61] The Computer Language Benchmarks Game. Rust vs C gcc fastest programs. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html`. Accessed on 2021-05-14.

[62] Tock Project Developers. Mutable references in Tock - memory containers (`Cells`). `https://github.com/tock/tock/blob/master/doc/Mutable_References.md`. Accessed on 2021-05-14.

[63] Tolnay, D. CXX - safe interop between Rust and C++. `https://cxx.rs`.

[64] Toman, J., Pernsteiner, S., and Torlak, E. CRUST: A bounded verifier for Rust. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015).

[65] Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N. O., Sammler, M., Druschel, P., and Garg, D. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)* (2019).

[66] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles* (1993).

[67] Watson, R. N., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., et al. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy* (2015).

[68] Woodruff, J., Watson, R. N., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (2014).

[69] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native Client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy* (2009).