Keeping Safe Rust Safe with Galeed

Elijah Rivera
MIT CSAIL
Cambridge, MA, USA
eerivera@mit.edu

Samuel Mergendahl
MIT Lincoln Laboratory
Lexington, MA, USA
samuel.mergendahl@ll.mit.edu

Howard Shrobe
MIT CSAIL
Cambridge, MA, USA
hes@csail.mit.edu

Hamed Okhravi
MIT Lincoln Laboratory
Lexington, MA, USA
hamed.okhravi@ll.mit.edu

Nathan Burow
MIT Lincoln Laboratory
Lexington, MA, USA
nathan.burow@ll.mit.edu

ABSTRACT

Rust is a programming language that simultaneously offers high performance and strong security guarantees. Safe Rust (*i.e.,* Rust code that does not use the `unsafe` keyword) is memory and type safe. However, these guarantees are violated when safe Rust interacts with unsafe code, most notably code written in other programming languages, including in legacy C/C++ applications that are incrementally deploying Rust. This is a significant problem as major applications such as Firefox, Chrome, AWS, Windows, and Linux have either deployed Rust or are exploring doing so. It is important to emphasize that unsafe code is not only unsafe itself, but also it breaks the safety guarantees of 'safe' Rust; *e.g.,* a dangling pointer in a linked C/C++ library can access and overwrite memory allocated to Rust even when the Rust code is fully safe.

This paper presents Galeed, a technique to keep safe Rust safe from interference from unsafe code. Galeed has two components: a runtime defense to prevent unintended interactions between safe Rust and unsafe code and a sanitizer to secure intended interactions. The runtime component works by isolating Rust's heap from any external access and is enforced using Intel Memory Protection Key (MPK) technology. The sanitizer uses a smart data structure that we call *pseudo-pointer* along with automated code transformation to avoid passing raw pointers across safe/unsafe boundaries during intended interactions (*e.g.,* when Rust and C++ code exchange data). We implement and evaluate the effectiveness and performance of Galeed via micro- and macro-benchmarking, and use it to secure a widely used component of Firefox.

ACM Reference Format:
Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *ACSAC '21: Annual Computer Security Applications Conference, Online.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3485832.3485903

1 INTRODUCTION

Many modern-day systems are written in C or C++. These include operating system (OS) kernels such as Linux [38] or Windows [42], mainstream web browsers like Mozilla Firefox [45] and Google Chrome [26, 27], and even other languages' compilers and interpreters (e.g. Python [56], JVM [37]). Unfortunately, C and C++ have weak 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 [51]. At BlueHat Israel 2019, Microsoft disclosed that in the past decade, memory errors have comprised ~70% of discovered vulnerabilities in their products [43]. Google has recently come to the same conclusion after analyzing their own security vulnerabilities since 2015 [28]. Memory safety remains a significant concern for software security despite the immense research effort expended on addressing it [2, 35, 66, 69], and such vulnerabilities are highly exploitable [6, 19–21, 25, 63, 65, 69, 75].

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" [44] in a sound type system. There have been multiple major vulnerabilities discovered due to a lack of soundness in the type system of C [21, 49, 51].

New programming languages are entering popular use that address the twin threats of memory and type safety violations, most notably Go [15] and Rust [40]. Rust guarantees strong memory and type safety for programs written in it [59], guarantees which have recently been formalized and verified by the RustBelt project [31]. 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 do so with little to no cost to performance [7, 70]. This combination of safety and performance has proven attractive to the systems community, prompting an increased in the popularity of Rust [32, 50].

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 benign programs during compilation, programs that are considered incorrect by the type-checker but which are actually valid in the

ACSAC '21, Dec 06–10, 2021, Online
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8579-4/21/12.
https://doi.org/10.1145/3485832.3485903

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, *e.g.,* doubly linked lists.

Concerningly for incremental deployments of Rust, 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 [46]). 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 they know 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. Unsafe Rust is a security hazard on multiple fronts. Unsafe code transitively removes the safety of code that interacts with memory it modifies. The standard library relies extensively on unsafe to provide safe wrappers around abstractions that the type system cannot verify. Applications written in Rust and C++have been shown to be less secure than hardened C++ [53]. Given the transitive nature of unsafety in Rust, and that Rust will be deployed in mixed-language settings that fundamentally require unsafe code for the foreseeable future, new defenses are needed to preserve Rust's guarantees in mixed-language applications.

Here we consider mixed-language applications consisting of both a memory and type safe component (Rust) and an unsafe component (C++). Such applications pose two threats against their "safe" components. The first is that in practice safe and unsafe languages share a heap, with no abstraction or isolation between them at runtime, such as virtual addresses between processes. This allows efficient communication, but also means that an arbitrary-write vulnerability in C++ can alter memory that notionally belongs to Rust. The second is through the programmer intended interactions, in which Rust gives C++ a pointer to use.

Galeed preserves the memory and type safety guarantees of Rust in mixed-language applications, and consists of two components: 1) a runtime defense that isolates Rust's heap from manipulation by C++, thereby preventing unintended interactions, and 2) a sanitizer for use by developers that secures intended interactions between the safe and unsafe programming language. Galeed thus protects safe code from possible corruptions in the unsafe code with which it interacts. Our runtime defense is built on top of `libmpk` [54], which enables the use of Intel's Memory Protection Keys (MPK) [30] to remove read/write access to the heap. However, the Galeed design is generic and can work with any enforcement mechanism. Our sanitizer replaces pointers passed across the language boundary with identifiers to Rust objects (dubbed *pseudo-pointers*), and turns dereferences of such pointers into function calls back into Rust with the object ID and request operation.

To demonstrate the effectiveness of the Galeed runtime defense, we use a case study from Firefox, a web browser in the process of migrating from C++ to Rust [46]. The evaluation results show that our technique incurs a runtime overhead of less than 1%. The security guarantees of the sanitizer are demonstrated on micro-benchmarks, which show reasonable overhead.

Our contributions are as follows:

- We study and systematize threat vectors against Rust in mixed-language applications, *e.g.,* Firefox, into unintended and intended interactions.
- We design and implement a runtime defense for isolating and protecting Rust heap from accesses by unsafe code enforced using Intel's MPK technology, and a sanitizer to verify the security of intended interactions between safe Rust and unsafe code using the idea of pseudo-pointers.
- We perform micro- and macro-benchmarking on our techniques and evaluate their security and performance impact, finding that the runtime defense has less than 1% overhead in our Firefox case study.

2 BACKGROUND & THREAT MODEL

In this section, we provide a quick background on the Rust programming language and Intel's MPK technology at the level that is necessary to contextualize the rest of our work. Our goal is not to be a comprehensive reference on these two relevant technologies. Interested readers can refer to the references cited in this paper for a deeper background on these two technologies.

2.1 Rust

Rust [60] is a programming language that offers low-level control and high performance, while still offering 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 [61]. 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 one needs to access a resource without taking ownership of it, one can *borrow* it. Borrowing gives one a reference to a resource. One 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/C++ [7, 70].

Rust has made claims to memory and type safety from its inception, and these claims have been mostly proven, first with Patina [58] and then more thoroughly with the RustBelt project [31]. 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 within the programming community, with many projects being written or re-written all or at least partly in Rust [4, 22, 32, 36, 46]. Our work focuses on one such real-world, popular applications, Firefox [45]. Firefox is a web browser developed by Mozilla Corporation. Firefox was originally written in C++, but has begun the process of migrating to Rust [46].

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) [30] 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 [73], Hodor [29], and libmpk [54] present intra-process isolation schemes using MPK. Such schemes come with a variety of security issues, detailed by Connor *et al.* [8].

The attraction of MPK is the performance of hardware security schemes. ERIM showed that updating permissions takes between 11-260 cycles, which corresponds to an overhead of <1%. `libmpk` [54] (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` [54] 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 effort focuses on mitigating threats explicitly caused by interactions between safe Rust and an unsafe language (*e.g.,* C/C++) in mixed-language applications. Threats against the underlying hardware [67], operating system (OS), and compiler layers are out-of-scope for this effort. We recognize the importance of threats against these layers, but as these threats exist independently of the cross-language boundary we are investigating, we consider them out-of-scope. For the same reason, we consider security pitfalls of MPK out-of-scope.

We assume standard protections mechanisms such as $W \oplus X$ [52] (a.k.a. data execution prevention (DEP) [41]), address space layout randomization (ASLR) [55], and stack canaries [9] are in-place. We do not attempt to replace these basic protections and our techniques work seamlessly with them. Since unprotected C/C++ code is trivially vulnerable to memory corruption attacks, our work is particularly relevant for cases where C/C++ code is hardened using additional protections such as CFI [5].

We assume a strong attacker, in-line with the related work in this domain [35, 66, 69]. That is, the attacker knows one or more strong memory corruption vulnerabilities in the unsafe portion of a mixed-language application and can use them to achieve an arbitrary write gadget to any writable location in the memory space of the application. Prior work has shown that in this situation memory safety (and thus application safety) can be violated [53]. The goal of our work is to implement protections such that we can isolate the effect of vulnerabilities in the unsafe language (C/C++) portion of the application and preserve the safety of the safe language (Rust) portion of the application.

3 GALEED DESIGN

In this section, we describe the design of Galeed. Galeed has two components: a runtime defense for isolating the Rust heap from unintended interactions, and a sanitizer for securing intended interactions using pseudo-pointers. We describe each component separately for ease-of-understanding, but they are both part of the overall technique for preserving the memory safety guarantees of safe Rust when it interfaces with unsafe code.

Rust is primarily being incrementally deployed: 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 case study, started its migration from C++ to Rust in 2016 [46]. Mozilla, the maintainers of Firefox, list Rust's memory safety as primary reason for the switch [46].

Mixing Rust with another language (*e.g.,* C++) breaks the Rust memory safety model, and leaves the mixed-language application more vulnerable to exploit than a CFI [1, 5] hardened C++ implementation [53]. Our work is general to any unsafe language that interfaces with Rust, but for the sake of simplicity and because of

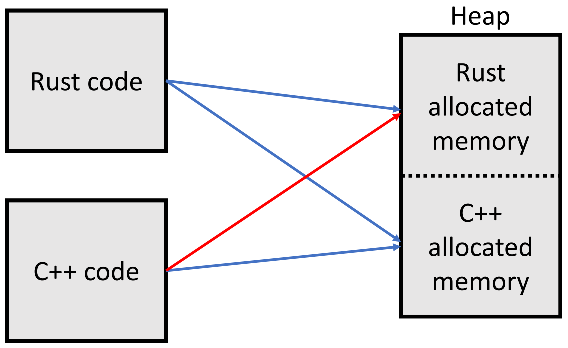


Figure 1: Possible memory accesses in Rust-C++ applications

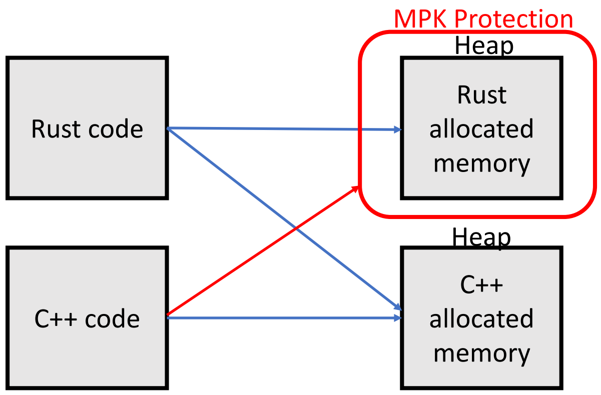


Figure 2: Protection via page-level memory isolation and MPK-enabled permissions switching

its heavy usage in Firefox, in the discussion below we focus on C++. 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.

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. 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.

In contrast, C++ accessing Rust memory (the red arrow in fig. 1) could cause any number of violations to Rust memory safety guarantees, up to and including full control-flow hijacking [53]. 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. An example of an intended interaction is when Rust parses a message and passes a pointer to it to C++ for further processing. An example of an unintended interaction is when an arbitrary write gadget (*e.g.,* a dangling pointer in C++) is used to modify a data structure in Rust memory when such an interaction was not conceived of by the developer.

3.1 Preventing Unintended Interactions via Heap Isolation

First, we focus on preserving memory safety in the presence of unintended accesses, and then we extend Galeed to secure intended accesses in section 3.2.

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.

3.1.1 Heap Isolation. Intel MPK 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 [29, 62, 73], and libmpk [54] provides 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 stays consistent with the Rust memory model despite executing untrusted code in another language.

Galeed focuses on isolating the Rust component's heap, leaving stack isolation to future work. We emphasize that this is in-line with the related work in the memory safety domain. For example, the 'low-fat' scheme was proposed to protect the heap [16], while it was extended to protect the stack in a follow-on effort [17]. General memory safety for non-Rust components is out of scope of this work, and is well-studied in literature [47, 48, 64, 66].

3.1.2 Heap Splitting. In order to isolate the Rust heap, we split the unified program heap into per safe language component heaps that are protected, and a remaining unified unsafe language heap. Each safe language heap comprises a distinct set of pages with its own MPK key. This allows per safe language permissions to be controlled by a single MPK key. Note that if, *e.g.,* a page used for Rust heap contains a C++allocation, then MPK permissions that operate on the page-level can no longer distinguish between the language heaps and thus appropriate permissions regimes. Note also that the pages for each heap can be interleaved, so long as each page in a language heap is dedicated exclusively to that heap.

3.1.3 Access Policy. Whenever a safe language is executing, its heap and the unified unsafe language heap have full read and write permissions. Leaving the unified unsafe heap accessible still prevents unintended interactions while maintaining safety, see fig. 1. On language transitions, which happen on function calls, permissions are removed for the calling safe-language heap. This permission change is inverted upon return. The Galeed policy invariant

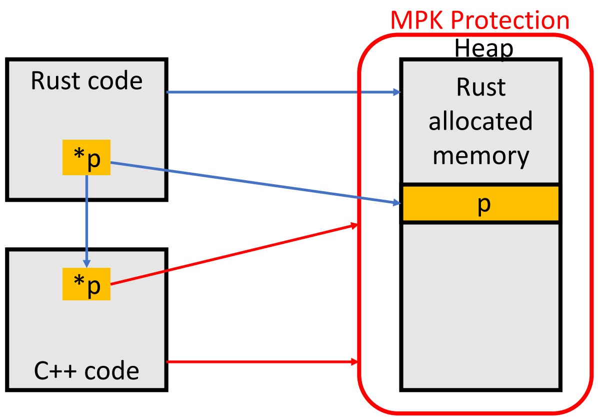


Figure 3: Galeed restricts all accesses by default.

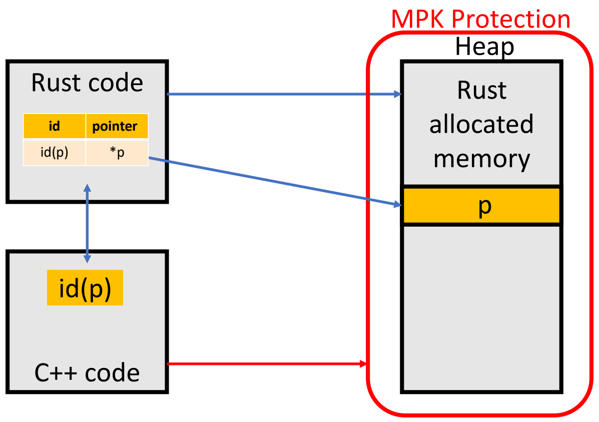


Figure 4: In our design, C++ **uses pseudo-pointers (e.g.** `id(p)`**) to request that Rust dereference Rust memory**

is that a safe language heap is accessible if and only if that safe language is currently execution. This policy results in full isolation of the language heaps, which is overly restrictive in practice. We next discuss how to relax this regime while maintaining safety with pseudo-pointers.

Our heap isolation technique is a *runtime defense* (a.k.a. exploit mitigation), a technique that is meant to run continuously when the application is running in order to provide the protection discussed above. As such, its small performance footprint is crucial for its adoption.

3.2 Securing Intended Interactions via Pseudo-pointers

Galeed's default policy intentionally excludes *intended accesses, i.e.,* times when C++ is explicitly and intentionally given a pointer to Rust memory. This most commonly occurs in FFI function calls, by passing a pointer as an argument to a structure in memory instead of passing directly by value. In fact, this pattern is employed by many Firefox modules, often due to performance or storage considerations. Galeed's default behavior breaks this intended behavior, illustrated in fig. 3.

Here, we present an option for data flow between safe Rust and unsafe code that does not require breaking the safety guarantees provided by Galeed's default heap policy. 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 an implementation of this design specialized to Rust and C++.

We introduce the idea of *pseudo-pointers, i.e.,* identifiers that Galeed passes to an external function instead of pointers. Galeed 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 ensure memory safety (fig. 4).

In contrast to our heap isolation that is a runtime defense, our pseudo-pointer technique is a *sanitizer* [66] meant to be used by the developer to detect and remove vulnerability pre-release. Accordingly its performance budget is much higher [66].

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

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

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 also it is 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 important 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 possibilities for potential mistakes.

3.2.2 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 are automatically generated using a naming strategy that includes both the struct type and the field name. These functions will either be `NOP`s 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.

3.2.3 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 3.2.1.

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 3.2.2.

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. The developer can then debug the code to ensure that all the necessary pointers are transformed. We did not encounter such cases in our analysis, although their possibility remains open.

3.3 Galeed Security Guarantees

Galeed has two security aims: 1) to prevent unintended interactions on the heap in mixed-language applications by providing a runtime defense, and 2) helping developers identify vulnerabilities in intended interactions between languages by providing a sanitizer. Heap isolation in turn has two components – the isolation policy and the hardware mechanism used to enforce it. The isolation policy is simple: a safe-language heap is only accessible when code in that language is executing. Enforcing this requires changing permissions whenever the executing language changes, which is precisely what Galeed does. Such a policy is as sound as its underlying enforcement mechanism.

Galeed uses MPK to enforce heap isolation due to MPK's low overhead. Doing so sacrifices some security. 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. To do so, binary of the unsafe applications are scanned to detect any instance of MPK instruction [29, 73]. If such an instruction is detected, the user is alerted. Such occurrences should be very rare, however, because the unsafe applications are assumed to be buggy and potentially compromised, but not developed to be malicious. Also, note that W $\oplus$ X is deployed, so when unsafe code is compromised, its binary cannot be overwritten by the attacker. The best an attacker can do is to hijack its control by modifying code pointers (*e.g.,* function pointers and return addresses) or use a dangling pointer to write arbitrary data to data (writable) regions.

Alternatively, other techniques can also be deployed alongside Galeed to prevent MPK instructions in unsafe code disabling the access policy, including CFI [1], hardware watchpoints [29], and system call filtering via sandbox [62]. System calls in particular pose a danger to MPK protection regimes [8] that no existing work fully addresses. We do not address the security of MPK isolation schemes here beyond following current best practices of scanning for additional PKRU instructions.

The sanitizer ensures that all pointers given to C++are still used in a memory and type safe manner. It relies on Rust's built in guarantees to do so, by referring all pointer dereferences back to safe Rust for verification before they are completed, and the result returned to C++. By removing all pointers to the Rust heap from C++, Rust maintains the integrity of its heap against intended interactions.

4 GALEED IMPLEMENTATION

We implemented a prototype for Galeed, specialized to interactions between Rust and C++. The full source code for our implementation is available at https://github.com/mit-ll/galeed.

4.1 Heap Isolation

4.1.1 Heap Creation. `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 implementation 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 6.

4.1.2 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. 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 5: Rust inline assembly code for MPK permission switching

We use the 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. 5). 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 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. Rust calls to external functions specify the C ABI and unmangled names, so identifying such external calls is possible in principle.

4.2 Pseudo-pointers

Pseudo-pointers extend our heap isolation mechanism to secure intended interactions between Rust and C++. We implement pseudo-pointers for user-defined structs that are intended to be passed across the language boundary as those are the primary vehicle Rust and C++ use to exchange data. 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.

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 pointed data. PhantomData is used in Rust for fields that exist at compile time but not at runtime. 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.

```
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 6: Transforming an example C++ function to use pseudo-pointers.

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.

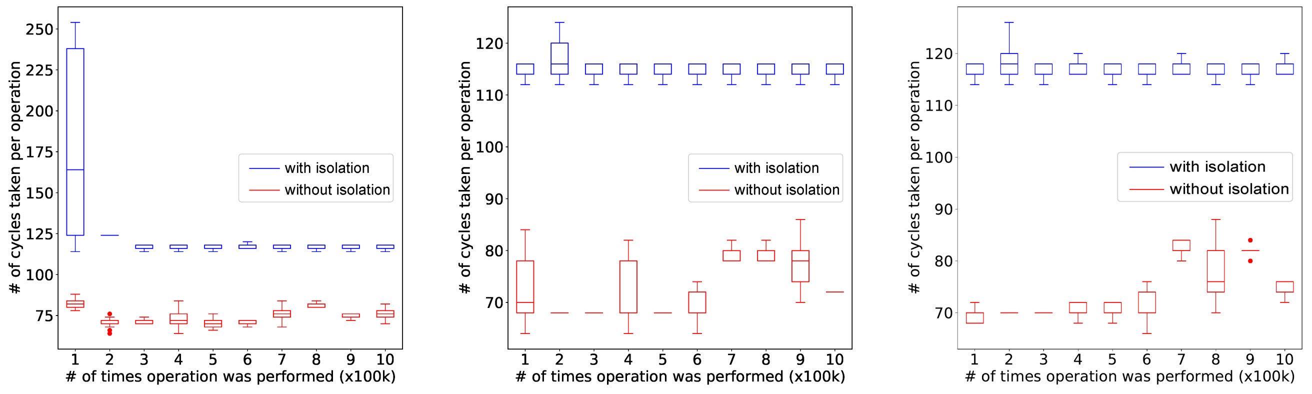
4.2.1 Rust API. The attribute macro is able to generate getter and setter functions for each field of a struct by name. The macro has access to the type information of each field, so these functions are able to carry that type information in their return value and arguments respectively.

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. Other reactions to an invalid pointer (*e.g.,* a NOP instruction instead of a panic) can also be easily used as appropriate depending on the application.

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.

4.2.2 External Function Transformation. 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 6 shows an example of this transformation.

Instead of placing the burden on developers to manually perform these transformations, we automate this transformation process. We introduce a module-level pass into the LLVM compiler which



(a) Single Read **(b) Single Write** **(c) Write then Read**

Figure 7: Heap isolation micro-benchmarks

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 usages 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.

5 EVALUATION

In this section, we evaluate our safety claims and calculate the performance overhead costs for our prototype. Since our heap isolation technique is a runtime defense, in addition to micro-benchmarking its checks, we also evaluate its macro performance overhead in Firefox. In contrast, our pseudo-pointer technique is a sanitizer, so we only perform micro-benchmarking on it because it is not meant to be deployed at runtime.

5.1 Heap Isolation

Recall that Galeed implements its heap isolation using Intel MPK. 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).

We first present a micro-benchmark of our heap separation before a case study of our heap separation in a Firefox library. We use the micro-benchmarks to evaluate the overhead of switching heap permissions and to validate the security properties of Galeed. Figure 7 shows the performance results from three scenarios: a) a single read to an MPK protected page, b) a single write to an MPK protected page, and c) a write and then a read to an MPK protected page. We find that our heap isolation protections add an overhead of ~50 cycles on average, which is tiny and consistent with prior work using MPK [54, 73]. Note that the overhead for the first 100K samples is large due to cold caches, but even then, the overhead is acceptable (~250 cycles). To evaluate security, we verify that C++ dereferencing a Rust pointer causes a MPK segmentation fault, and verify that the expected permissions changes are present in the binary.

5.2 Firefox's *libpref*

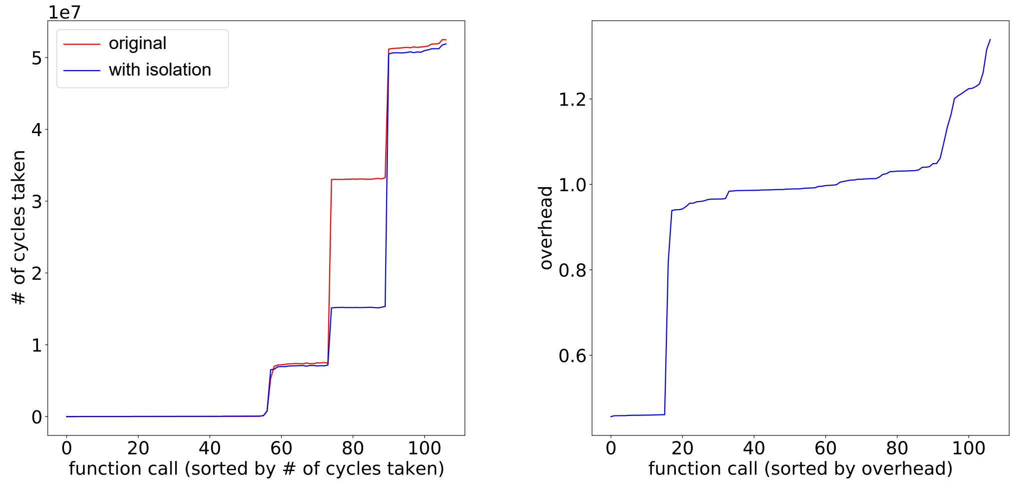
Moving beyond micro-benchmarks, we present an evaluation of Galeed's heap isoaltion on Firefox. We target the *libpref* module within Firefox, which is used to parse a file to collect user preferences. We use Firefox's own *libpref* module test suite as benchmarks. We discard 5 of the tests which test front-end behavior and fail in our headless evaluation environment. As expected, applying heap isolation without modification caused all tests to fail due to heap permissions errors. However, the *libpref* module uses Rust to parse a preference file, after which the results are only *read* by C++. Consequently, pseudo-pointers were not required by this module. Instead, we modify our access policy to allow C++ to read the Rust heap. This reduces security by allowing information leaks that would otherwise be prevented by Rust's memory safety, but allows us to evaluate heap isolation separately.

We ran the test suite 1,000 times with both the unmodified *libpref* module as our baseline, and 1,000 times with heap isolation, 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 heap isolation for the Rust component (fig. 8), with an even lower average overhead in the application overall (fig. 9).

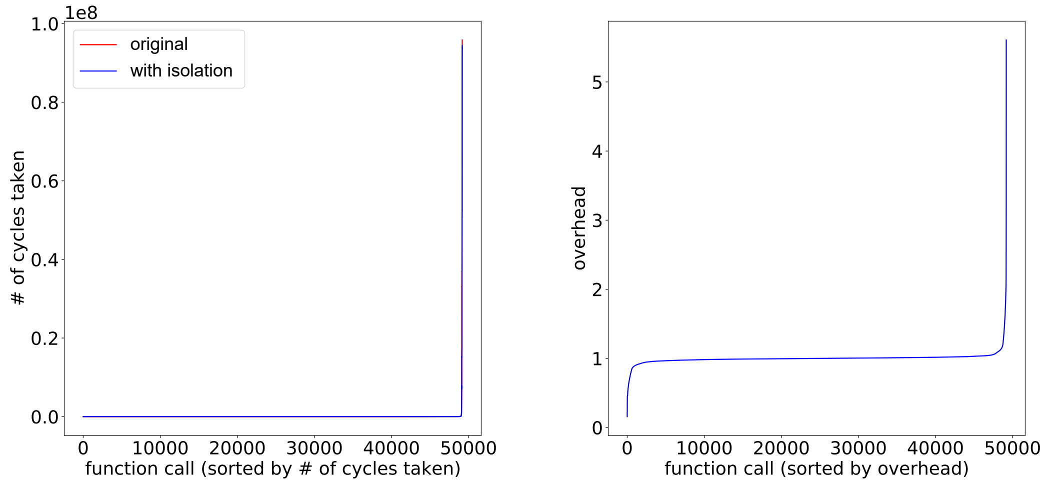
5.3 Pseudo-pointers

To evaluate the functionality and performance of Galeed's pseudo-pointers, we developed a proof-of-concept application in which 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



(a) Cycle counts - Rust function calls **(b) Heap isolation overhead - Rust function calls**

Figure 8: Heap isolation on *libpref* benchmarks - Rust component



(a) Cycle counts - all function calls **(b) Heap isolation overhead - all function calls**

Figure 9: Heap isolation on *libpref* benchmarks - all function calls

were never accessed from C++, while other pointers not from Rust were left unaffected.

We evaluate the performance overhead of adding these additional function calls using micro-benchmarks (fig. 10). 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.

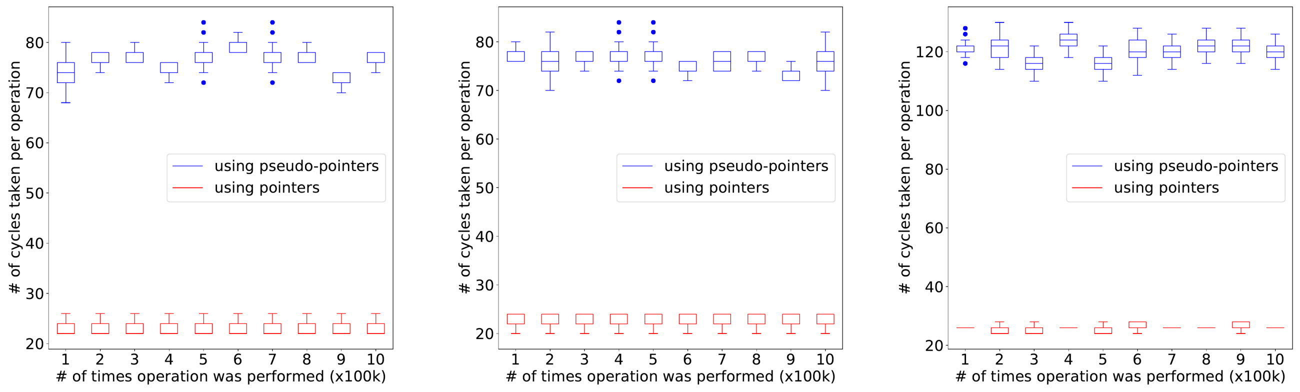
This overhead is considered quite practical for sanitizers, many of which have overheads ranging from 3x to over 10x [66].

6 PRACTICAL LESSONS LEARNED

In this section, we discuss some of the lessons we learned during this effort for practical deployment of a technique like Galeed, how alternative design choices could impact them, and the directions for future work. It is our hope that these lessons not only inform the reader about these practical considerations when using Rust and Galeed, but also they can help researchers be cognizant of some of the practical challenges and pitfalls when developing similar technologies.

6.1 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 started this project, some of the Rust's memory containers that have unsafe code have changed and added new methods (*e.g.,* the Cell library). Developers seeking



(a) Single Read **(b) Single Write** **(c) Write then Read**

Figure 10: Pseudo-pointer micro-benchmarks

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. Such decisions further emphasize the need for technologies such as Galeed that seek to limit the impact of unsafe code, particularly since even a stable and verified Rust language and core libraries will be deployed in mixed language environments for the foreseeable future.

6.2 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 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 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.

6.3 `libmpk`

Galeed also relies on the open-source project `libmpk` [54], a software abstraction developed around MPK. `libmpk` is implemented as a C library, and we currently rely on its 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, preferably in Rust, and optimized for performance. Multiple research projects have shown that memory allocator design has an impact on performance [12, 18, 39]. Future work should include an updated memory allocator that is natively aware of MPK.

6.4 Mixed-Language Application Security

This work is a first attempt to address the security of mixed-language applications, and only considers interactions between compiled languages. Future work should consider interactions between compile and JIT compiled languages such as Python, or with the JVM. Further work is also needed to examine, both statically and dynamically, the full relationship between Rust and C++ applications. It is likely that Papaevripides and Athanasopoulos [53] have only scratched the surface of attacks on Rust/C++applications, motivating this work and future work in this area.

7 LIMITATIONS

Galeed also has a number of limitations that we discuss here. In our prototypes, we intentionally focused on preserving memory safety first, sometimes to the detriment of performance. We rely on the unoptimized `libmpk` library for our memory allocation and deallocation steps. In the pseudo-pointer sanitizer, 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. In addition, in both cases, we made no attempts to allow for LLVM's cross-language link time optimization (LTO).

Galeed can also be further automated, 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.

Moreover, 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 accommodate current Rust/C++ interactions. For example, we do not support strings, which are used in the parsing modules that Firefox has migrated to Rust.

Lastly, our prototype currently depends on having access to the original source code for Rust, and at minimum the LLVM bytecode for C++. Future work can investigate how to retrofit security in cases where only Rust/C++ binaries are available.

8 RELATED WORK

Below we discuss related work in three major areas: formal reasoning about Rust, code/memory isolation (related to our heap isolation), and program transformations for safety (related to our pseudo-pointers).

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 [58] 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 [10, 31] built another formalization of a realistic subset of Rust. RustBelt used the Iris framework for concurrent separation logic [33] to prove memory safety properties. RustBelt went even further and also verified some standard libraries which contained `unsafe`. CRUST [72] 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 [3, 24] presents a language and framework for proving properties of assembly programs and even automating those proofs. TINA [57] automatically lifts inline assembly within C code to semantically equivalent C code, easing the burden of analysis and verification tools.

8.2 Isolation

There have been numerous efforts into efficient and effective isolation at both the software and hardware levels. Many software isolation techniques rely on sandboxing untrusted code [74]. Native Client [77] specifically provides this sandboxing for untrusted browser-based applications, while Vx32 [23] allows native applications to sandbox untrusted plug-ins. Sandcrust [34] 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 address space, and using remote procedure calls (RPC) to communicate between the two languages. Our solution uses the Rust foreign function interface (FFI) instead which is the standard method and allows for lower overhead.

Many hardware-based isolation techniques rely on additional metadata/tags on pointers and/or memory locations [13]. CHERI [76] and Dover [68] uses PUMP [11, 14] infrastructure are among such efforts.

Multiple compartmentalization projects have been built on top of Intel MPK [30] technology [29, 62, 73]. MPK has been shown to have vulnerabilities that these projects do not prevent [8].

8.3 Compile-time Transformations

The current solution for FFI between Rust and C++ is a project called CXX [71]. 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 the static analysis claims.

Other projects have also used compile-time transformations to strengthen safety for C/C++ code. For a comprehensive list, we refer the reader to systematization of knowledge papers in this area [66, 69].

9 CONCLUSION

The Rust programming language offers a combination of performance and memory safety guarantees which is increasingly drawing developers to use it, but interfacing with unsafe code undermines claims to memory safety. In this paper, we presented Galeed, a technique to preserve memory safety in safe Rust while used in conjunction with unsafe code (*e.g.,* C/C++). Galeed consists of two components: a runtime defense to isolate Rust's heap from external unintended interactions that is enforced using Intel MPK and a sanitizer that automatically replaces raw pointers with pseudo-pointers to secure intended interactions between safe Rust and unsafe code. Our micro-benchmarking of the transformations and macro-benchmarking on Frifox indicate that our runtime defense only incurs minimal overhead and our sanitizer is practical.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. 4Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)* (2009).

[2] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM Computer and Communications Security (CCS'15).*

[3] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium (USENIX Security 17).*

[4] Adam Burch. 2019. Using Rust in Windows. https://msrc-blog.microsoft.com/2019/11/07/using-rust-in-windows.

[5] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1 (April 2017).

[6] Center for Internet Security. 2019. 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.

[7] Catalin Cimpanu. 2019. 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.

[8] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20).*

[9] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. 1999. Protecting systems from stack smashing attacks with StackGuard. In *Linux Expo.*

[10] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. *Proceedings of the ACM on Programming Languages (POPL)* (2019).

[11] Arthur Azevedo De Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In *2015 IEEE Symposium on Security and Privacy.*

[12] David Detlefs, Al Dosser, and Benjamin Zorn. 1994. Memory Allocation Costs in Large C and C++ Programs. *Software: Practice and Experience* (1994).

[13] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. 2008. Hard-Bound: Architectural Support for Spatial Safety of the C Programming Language. *ACM SIGOPS Operating Systems Review* (2008).

[14] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. 2014. PUMP: A

Programmable Unit for Metadata Processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[15] Alan AA Donovan and Brian W Kernighan. 2015. *The Go programming language*. Addison-Wesley Professional.

[16] Gregory J Duck and Roland HC Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*. 132–142.

[17] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers.. In *NDSS*, Vol. 17. 1–15.

[18] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. 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)*.

[19] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland'15)* (San Jose, CA).

[20] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM Computer and Communications Security (CCS'15)*.

[21] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-based Control Flow Integrity. In *Proceedings of IEEE Annual Computer Security Applications Conference (ACSAC'18)*.

[22] Wedson Almeida Filho. 2021. Rust in the Linux kernel - Google Security Blog. https://security.googleblog.com/2021/04/rust-in-linux-kernel.html.

[23] Bryan Ford and Russ Cox. 2008. Vx32: Lightweight, User-level Sandboxing on the x86. In *USENIX Annual Technical Conference*.

[24] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A Verified, Efficient Embedding of a Verifiable Assembly Language. *Proceedings of the ACM on Programming Languages (POPL)* (2019).

[25] Ronald Gil, Hamed Okhravi, and Howard Shrobe. 2018. There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection. In *Proceedings of the IEEE Secure Development Conference (SecDev18)*.

[26] Google. [n.d.]. Chromium. https://www.chromium.org/Home.

[27] Google. [n.d.]. Google Chrome. https://www.google.com/chrome.

[28] Google. [n.d.]. Memory safety - The Chromium Projects. https://www.chromium.org/Home/chromium-security/memory-safety. Accessed on 2021-05-14.

[29] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.

[30] Intel. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual.

[31] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages (POPL)* (2017).

[32] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2020. Safe Systems Programming in Rust: The Promise and the Challenge. *Commun. ACM* (2020).

[33] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *ACM SIGPLAN Notices* (2015).

[34] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS)*.

[35] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 276–291.

[36] Ryan Levick. 2019. Why Rust for safe systems programming. https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming.

[37] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification*. Pearson Education.

[38] Linux Kernel Organization. [n.d.]. The Linux Kernel Archives. https://www.kernel.org.

[39] Rahul Manghwani and Tao He. 2011. Scalable memory allocation. https://locklessinc.com/downloads/Preso05-MemAlloc.pdf.

[40] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[41] Microsoft. 2006. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. Online. http://support.microsoft.com/kb/875352/en-us

[42] Microsoft. 2014. Lesson 2 - Windows NT System Overview. https://docs.microsoft.com/en-us/previous-versions//cc767881(v=technet.10).

[43] Matthew Miller. 2019. 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.

[44] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* (1978).

[45] Mozilla Foundation. [n.d.]. Firefox. https://www.mozilla.org/en-US/firefox.

[46] Mozilla Foundation. [n.d.]. Oxidation. https://wiki.mozilla.org/Oxidation. Accessed on 2021-05-14.

[47] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. 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)*.

[48] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: Compiler-Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*.

[49] Tim Newsham. 2001. Format string attacks. http://hackerproof.org/technotes/format/formatstring.pdf.

[50] Hamed Okhravi. 2021. A Cybersecurity Moonshot. *IEEE Security & Privacy* 19, 3 (2021), 8–16. https://doi.org/10.1109/MSEC.2021.3059438

[51] Aleph One. 1996. Smashing The Stack For Fun And Profit. *Phrack magazine* (1996).

[52] OpenBSD. 2003. OpenBSD 3.3.

[53] Michalis Papaevripides and Elias Athanasopoulos. 2021. Exploiting Mixed Binaries. *ACM Transactions on Privacy and Security (TOPS)* (2021).

[54] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. `libmpk`: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.

[55] PaX. 2003. PaX Address Space Layout Randomization.

[56] Python Software Foundation. [n.d.]. The Python programming language. https://github.com/python/cpython.

[57] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. 2019. Get rid of inline assembly through verification-oriented lifting. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[58] Eric Reed. 2015. 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).

[59] Rust Foundation. [n.d.]. Meet Safe and Unsafe - The Rustonomicon. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html. Accessed on 2021-05-14.

[60] Rust Foundation. [n.d.]. Rust Programming Language. https://www.rust-lang.org.

[61] Rust Foundation. [n.d.]. What is Ownership? - The Rust Programming Language. https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html. Accessed on 2021-05-14.

[62] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*.

[63] Jeff Seibert, Hamed Okhravi, and Eric Soderstrom. 2014. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)* (Scottsdale, AZ).

[64] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*.

[65] Hovav Shacham et al. 2007. 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)*.

[66] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*.

[67] Chad Spensky, Aravind Machiry, Nathan Burow, Hamed Okhravi, Rick Housley, Zhongshu Gu, Hani Jamjoom, Christopher Kruegel, and Giovanni Vigna. 2021. Glitching Demystified: Analyzing Control-flow-based Glitching Attacks and Defenses. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 400–412. https://doi.org/10.1109/DSN48987.2021.00051

[68] Gregory T Sullivan, André DeHon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. 2017. The Dover Inherently Secure Processor. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*.

[69] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*.

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

[71] David Tolnay. [n.d.]. CXX - safe interop between Rust and C++. https://cxx.rs.

[72] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. CRUST: A Bounded Verifier for Rust. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[73] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*.

[74] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*.

[75] Bryan Ward, Richard Skowyra, Chad Spensky, Jason Martin, and Hamed Okhravi. 2019. The Leakage-Resilience Dilemma. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*.

[76] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*.

[77] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*.