METASAFE: Compiling for Protecting Smart Pointer Metadata to Ensure Safe Rust Integrity

Martin Kayondo[1,2] Inyoung Bang[1,2] Yeongjun Kwak[3] Hyungon Moon[3,*] Yunheung Paek[1,2,*]

[1]*ECE, Seoul National University,* [2]*ISRC, Seoul National University*
[3]*UNIST*
{kymartin, iybang}@sor.snu.ac.kr,
{kyj05137, hyungon}@unist.ac.kr
ypaek@snu.ac.kr

Abstract

Rust is a programming language designed with a focus on memory safety. It introduces new concepts such as ownership and performs static bounds checks at compile time to ensure spatial and temporal memory safety. For memory operations or data types whose safety the compiler cannot prove at compile time, Rust either explicitly excludes such portions of the program, termed unsafe Rust, from static analysis, or it relies on runtime enforcement using smart pointers. Existing studies have shown that potential memory safety bugs in such unsafe Rust can bring down the entire program, proposing in-process isolation or compartmentalization as a remedy. However, in this study, we show that the safe Rust remains susceptible to memory safety bugs even with the proposed isolation applied. The smart pointers upon which safe Rust's memory safety is built rely on metadata often stored alongside program data, possibly within reach of attackers. Manipulating this metadata, an attacker can nullify safe Rust's memory safety checks dependent on it, causing memory access bugs and exploitation. In response to this issue, we propose METASAFE, a mechanism that safeguards smart pointer metadata from such attacks. METASAFE stores smart pointer metadata in a gated memory region where only a predefined set of metadata management functions can write, ensuring that each smart pointer update does not cause safe Rust's memory safety violation. We have implemented METASAFE by extending the official Rust compiler and evaluated it with a variety of micro- and application benchmarks. The overhead of METASAFE is found to be low; it incurs a 3.5% average overhead on the execution time of a web browser benchmarks.

1 Introduction

Rust is a systems programming language that strongly emphasizes memory safety. It ensures memory safety through its strict ownership model and enforced borrowing rules, unlike C/C++, which is plagued by memory vulnerabilities such as buffer overflows and use-after-free (UAF). The language's security advantages have contributed to its growing popularity, as evidenced by its adoption in real-world projects such as the Linux kernel, Mozilla Firefox browser, and Android operating system [13, 21, 36]. Google attested to Rust's memory safety when they reported a drastic reduction in memory bugs, from 76% to 25%, since the adoption of Rust to the Android OS, and now 32% of Android OS is written in Rust [17].

Rust achieves memory safety by relying on a distinguished memory safety system with rules enforced both statically at compile time and dynamically at runtime. The compiler ensures strict adherence to memory safety rules by performing static analysis during compilation. In cases where the compiler checks cannot be deterministically enforced through static analysis, Rust employs smart pointers to enforce the memory safety rules. Smart pointers carry metadata alongside the data pointer, which is relied on to perform memory safety checks at runtime. The type of metadata varies depending on the intended usage. For example, some smart pointers have metadata representing the length of a dynamically allocated buffer, used to check against buffer address indices and mitigate buffer overflows. Another example is a smart pointer for reference-counted shared pointers, where it has the number of pointer copies as metadata to mitigate use-after-free (UAF) bugs. Therefore, smart pointers and their associated metadata play a crucial role in ensuring Rust's memory safety.

The storage and access of this metadata may vary across different compiler versions, as explained in [6]. In Rust, the metadata is generally stored alongside program data, potentially within reach of malicious actors seeking to exploit vulnerabilities. With knowledge of these implementation details, attackers can maliciously overwrite the metadata, compromising the integrity of the runtime memory safety checks. Additionally, Rust exposes some application programming interface (API) functions that allow the programmer to modify this metadata at will without verification. In such cases, the integrity of the metadata, and consequently the safety checks dependent on it, is left at the mercy of the programmer's proficiency, similar to memory management in C/C++.

*Corresponding authors

Logical bugs resulting from improper utilization of these API functions by the programmer can reintroduce memory bugs that Rust was designed to solve. Consequently, despite Rust's reputation for memory safety, programs written in it can still suffer from memory bugs.

Our analysis of these memory bugs found that a significant proportion of them is caused by smart pointer metadata overwriting and misuse of smart pointer APIs by programmers. For example, many reported vulnerabilities [18,25,26], which received common vulnerabilities and exposures (CVEs) IDs, demonstrate the risks introduced by modified smart pointer metadata. CVEs such as [26] highlight memory bugs resulting from smart pointer API misuse and logical errors committed by programmers. Interestingly, programs with these bugs can still compile successfully because the compiler relies on metadata to enforce memory safety at runtime, only to encounter undefined behavior (UB) at runtime due to tampered metadata.

According to our knowledge, no existing studies comprehensively protect smart pointers' integrity, leaving them within reach of untrusted code and incorrect updates at the run time despite their obvious sacrality. Rust, especially *safe Rust*, is assumed to be memory-safe in principle. Only the remainders in the program, the *unsafe Rust* and external libraries linked through the foreign function interface (FFI) are considered potentially vulnerable. This observation has been the rationale behind most existing works [2, 14, 19, 23] on Rust memory safety, focusing on protecting safe Rust's memory objects from the others. However, this assumption only holds when the integrity of metadata on which the runtime security checks depend is maintained. Even when protected by existing works, a Rust program can still be vulnerable to memory safety violations within safe Rust if the metadata of smart pointers is exposed to unsafe Rust code or external libraries. Current approaches do not specifically address the protection of smart pointer metadata, inadvertently leaving it accessible to code outside of safe Rust.

This paper introduces METASAFE, a compilation framework designed to fortify the runtime protection of smart pointers. Recognizing the critical importance of smart pointers, METASAFE adopts a proactive approach by securely isolating them from program data and storing them in a protected compartment. To safeguard against vulnerabilities arising from API operations that modify metadata, METASAFE inserts sanitization checks that refer to allocator metadata, such as block size, to examine the validity of the attempted metadata updates. By default, METASAFE's implementation caters to smart pointers defined in Rust's standard library, but it empowers developers to define custom sanitization routines for their own smart pointer implementations by defining generic validation routines for developers to extend. As a result, METASAFE ensures the veracity and correctness of metadata, thereby guaranteeing memory safety, even in the presence of logic bugs stemming from API misuse. Note that METASAFE cannot replace existing methods that compartmentalize safe Rust from other components, as its primary aim is to protect the metadata of smart pointers from exploitation. For instance, METASAFE is not designed to prevent attackers from manipulating the memory space of safe Rust by exploiting memory safety bugs in external libraries. Therefore, METASAFE should be used in conjunction with existing compartmentalization approaches to provide comprehensive protection for the Safe Rust components of Rust programs.

Among others, the most significant challenge we overcome in designing METASAFE is that smart pointers could be embedded within a composite data type as a field, complicating its isolation. To address this, we propose two solutions: The first solution treats the entire composite type as a smart pointer and applies METASAFE's protection to it as a whole. The second approach employs a more sophisticated method by casting composite type-embedded smart pointers onto a separate protected shadow memory region. This approach enables finer-grained isolation and protection of smart pointers within ADTs. To further secure the smart pointer region, METASAFE leverages hardware extensions such as Intel Memory Protection Keys (MPK), which prevent malicious attackers from bypassing validation and overwriting isolated metadata.

Another challenge METASAFE faces is relying on allocator metadata for synchronization and assuming the correctness of this metadata. A heap allocator is often a separate component that can be compartmentalized from the rest of the program and maintains the information about each heap object used at runtime to examine the safety of a metadata update. To ensure the correctness of this metadata, METASAFE compartmentalizes allocator metadata in a similar way as it does smart pointer metadata.

Our evaluation of METASAFE on targeted microbenchmarks shows it incurs a 25.5% performance overhead on average. Evaluation of METASAFE on `Servo`, a real world browser shows both solutions incur 3.5% overhead on average showing it is easily adoptable in production. In lightly concurrent environments, METASAFE exhibits a memory overhead of 25.5% on average, while in heavily concurrent environments, it uses up to 8x more memory on average.

This paper contributes to the run time memory safety of Rust programs as follows.

- We present a comprehensive examination of memory safety in Rust from the perspective of smart pointer correctness, which has been understudied despite being a crucial component of Rust's memory safety. Our study reveals the need to protect smart pointer metadata and APIs, filling a gap in the current literature.

- We introduce METASAFE, a framework that improves memory safety by protecting smart pointer metadata and the API uses. We also explore ways to combine METASAFE with existing solutions to provide a complete solution to memory safety issues in Rust.

- Finally, we conduct experiments with real-world CVEs to showcase the effectiveness of METASAFE. We further apply METASAFE to real-world programs and evaluate their performance and memory overhead.

We will open the implementation of METASAFE to the public upon publication of this paper for the follow-up studies on Rust memory safety.

2 Background

Rust delivers statically checked memory safety at the cost of limited expressiveness. To write programs that cannot fully adhere to the statically checked restrictions, developers use either smart pointers (§2.1), unsafe Rust (§2.2), or external libraries FFI.

2.1 Smart Pointers in Rust

Rust delivers the level of memory safety that C/C++ could never achieve by design. By imposing the programs to adhere to several strict rules, such as ownership, Rust proves the safety of many pointer dereferences at compile time. This design choice of burdening developers to achieve efficient memory safety verified at compile time comes with the limitation in the expressiveness of the language. For example, only the memory objects whose size can be determined at compile time can benefit from the static memory safety check. However, programs often use dynamically sized data structures like linked lists, trees, and hash tables, whose memory safety Rust cannot prove statically.

To overcome this limitation, Rust advises using smart pointers. In support of this, Rust standard library provides several smart pointers for various use cases as summarized in Table 1. For example, a program can use `Rc` to enable multiple ownership of a memory object, which the Rust ownership system does not allow for non-smart pointers. Smart pointers are designed to enable such potentially unsafe behavior without compromising the memory safety of the program by maintaining metadata along with the raw pointer to ensure the safety of pointer dereferences at run time. `Rc` stands for Reference Counted, and it maintains a reference counter along with the raw pointer to the memory object to ensure that the object is not freed while there are still references to it. `Vec` enables a program to use a dynamically allocated buffer similar to `std::vector` in C/C++. As expected, using `Vec` is not supposed to expose the program to the risk of buffer overflow. The `Vec` smart pointer has the length and capacity of the allocated memory chunk as the metadata. On each dereference, the implementation of this smart pointer automatically checks the safety of memory accesses using the metadata, ensuring the absence of out-of-bounds access.

As such, the memory safety guarantee Rust provides through smart pointer relies on the metadata's correctness. An unfortunate observation behind our study is that the metadata is at the risk of corruption by memory bugs due to its storage alongside program data, as we explain later §3.

2.2 Unsafe Rust

Writing performant and expressive code in Rust remains challenging, even with smart pointers. Some features such as *inlined assembly* and raw pointer dereferencing are still strictly prohibited. *Unsafe* Rust is a part of Rust in which some memory system rules are relaxed. It is organized in code compartments wrapped by the *unsafe* keyword. Using *unsafe* Rust, a programmer can: dereference raw pointers, call *unsafe* functions and FFI, use inline assembly, and access or modify a mutable static variable. While it helps write more performant and expressive code, *unsafe* Rust is risky because not only does the compiler forego some safety checks on memory accesses in the *unsafe* region, Rust provides limited safety guarantees, and memory safety relies on programmer's expertise. For example, raw pointers provide no guarantees on pointer validity and do not implement any automatic cleanup. Therefore, writing *unsafe* Rust introduces the same risk of memory bugs in C/C++. This is why *unsafe* Rust has received vast attention in studies on memory safety in Rust.

2.3 Isolated Storage: *In-process memory isolation*

In-process memory isolation is a conventional technique that creates separate compartments within the same process to quarantine untrusted code or to give only specific components access to sensitive data. The isolation can be achieved in many different forms [15]; for example, hardware-based memory protection, virtual memory management, or software-based isolation techniques such as sandboxes. Several works [2, 14, 19] have employed this in-process memory isolation to enhance memory safety in Rust, especially by preventing *unsafe* Rust or FFI functions from accessing the safe Rust's memory objects.

3 Motivation

Existing solutions [2, 14, 19, 23] for mitigating memory bugs in Rust programs focus on isolating unsafe Rust and FFI functions from safe Rust. We find that such a strategy leaves safe Rust vulnerable to memory bugs because they do not safeguard smart pointer metadata (§3.1). Our observation is that smart pointer metadata integrity is at risk of being compromised by memory bugs in unsafe Rust and FFI functions (§3.2) or inappropriately updated by logic bugs in intentional changes (§3.3). We argue that the integrity of smart pointer metadata, which is critical to the memory safety of safe Rust, should be protected.

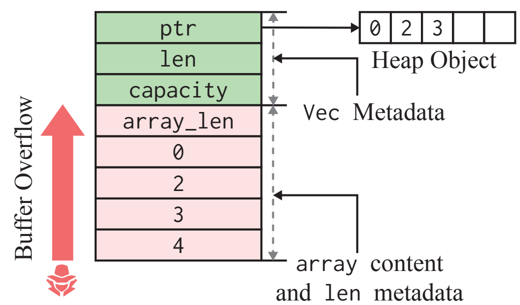
Smart Pointer	Metadata	Purpose	Safety	Vulnerabilities
Box	None	Basic Heap Object	Rust Ownership	UAF
Vec	Len, Capacity	Dynamic Buffer	Spatial	Overflow, UAF
Cell, RefCell	Borrow Counter	Interior Mutability	Temporal	UAF
Rc	Reference Counters	Shared Reference	Temporal	UAF
Arc	Reference Counters	Thread Safe Shared References	Temporal, Thread	UAF, Races
Mutex, MutexGuard	Locks	Thread Safe Interior Mutability	Thread, Temporal	UAF, Races
RwLock	Reference Counters	Similar to Arc+Mutex	Thread, Temporal	UAF, Races

Table 1: General Rust smart pointers, their metadata, intended purpose, and vulnerabilities.

```rust
use std::rc::Rc;
fn main {
    let mut xvec = vec![0,2,3];
    let mut array = [0,2,3,4];

    unsafe {
        let mut ptr = array.as_mut_ptr()
                    .offset(10);
        *ptr = 10;
    }
    //...
    //...
    unsafe {
        let elem = xvec.get_unchecked_mut(1);
        *elem = 5;
    }
    dgb!(xvec);
}
```

(a) A Rust program where a vulnerable buffer is allocated right next to smart pointer metadata



(b) The layout of the stack containing `xvec` and `array` while executing the function in Figure 1a. An overflow on `array` may result in the corruption of `xvec`'s metadata.

Figure 1: An example showcasing the possibility of smart pointer metadata corruption owing to the vulnerability in unsafe Rust

Table 2: Consideration as a source of Rust memory bugs by different works.

Motivation	XRust	TRust	PKRU-Safe	Galeed	METASAFE
Unsafe Rust	✓	✓	✗	✗	✓
FFI Functions	✗	✓	✓	✓	✓
Smart Pointer Integrity	✗	✗	✗	✗	✓

3.1 Existing Solutions for Rust Memory Safety

The memory unsafety of practical Rust programs has recently attracted significant attention.

XRust [19], TRust [2], Galeed [23] and PKRU-Safe [14] are four recent studies exploring the unsafety arising from the use of unsafe Rust and FFI functions. They propose the isolation of the unsafe Rust and FFI functions from the safe Rust by restricting their access to the safe Rust's memory objects. The idea is to confine memory vulnerabilities in unsafe Rust and FFI functions, thereby protecting safe Rust. Each study identifies the set of memory objects to be protected from these untrusted code and applies a protection mechanism that isolates them. XRust [19] requires a programmer to explicitly identify heap memory allocations to be used in unsafe Rust and protect the memory objects allocated by the remaining allocation sites, primarily using software fault isolation (SFI) or guard pages. TRust [2] employs static analysis to track memory objects used in unsafe Rust and FFI functions, and protect the remaining memory objects using SFI and Intel MPK. PKRU-Safe [14] uses dynamic profiling to determine which memory objects the FFI functions access, and protect the remaining memory objects from the FFI functions using Intel MPK. Galeed [23] uses static data-flow analysis to reason about memory allocation and flow between FFI functions and Rust, and protects Rust-allocated memory using Intel MPK. Galeed further wraps Rust-allocated memory pointers used by FFI with *pseudo-pointers* that ensure FFI access to Rust-allocated memory is examined by Rust.

3.2 Memory Bugs Corrupting Smart Pointers

Vulnerable Unsafe Rust Code. The importance of smart pointer metadata remains understudied, and none of these studies give special care to them. They all focus on identifying the memory objects visible from the source code level, i.e., the ones allocated and accessed by the program under the developer's instruction. Unfortunately, a Rust program stores more data in the memory during execution, some of which affect the memory safety of safe Rust. Figure 1a shows an example where a smart pointer is likely to remain unprotected

```rust
1  extern "C" unsafe fn do_array_stuff(ptr: *const c_void);
2  fn main() {
3    let mut buffer = vec![0,2,3];
4    let char_slice = ['a', 'b', 'c'];
5    unsafe {
6      do_array_stuff(char_slice.as_ptr() as *const c_void);
7      buffer.set_len(10);
8    }
9    println!("Element at 100: {}", buffer[99]);
10 }
```

(a) Sending a pointer to FFI from Rust.

```c
1 void do_array_stuff(void* ptr){
2   char string[100];
3   read_from_user(&string);
4   memcpy(ptr,(void*)string, READ_SIZE);
5 }
```

(b) An overflow bug in FFI affecting a pointer received from Rust.

Figure 2: Exploiting FFI memory bugs to overwrite smart pointer metadata.

under the protection of existing solutions. At run time, the program stores `array` alongside the smart pointer for `xvec`, as Figure 1b shows. Should an attacker exploit the buffer overflow vulnerability of the `array` at line 9, they could overwrite the metadata of the `xvec` smart pointer. This manipulation could cause subsequent uses of the smart pointer to fail in guaranteeing memory safety. Applying the existing mechanisms does not protect the smart pointer metadata in this example. PKRU-safe is not designed to mitigate vulnerable, unsafe Rust code, so it does not help. TRust [2] or XRust [19] does not help as well because both the `xvec` smart pointer and `array` will be classified as accessible from the unsafe Rust and thus will be allocated in the same unprotected memory region. `array` is clearly accessible from the unsafe Rust from the example, and the existence of smart pointer implementation, which is composed of unsafe Rust functions, makes the `xvec` smart pointer metadata also accessible from the unsafe Rust.

Vulnerabilities in FFI functions. Figure 2 shows an example where the `Vec` smart pointer becomes vulnerable to a bug in an FFI function. In Figure 2a, the Rust compiler creates a smart pointer for the mutable variable `buffer` on the stack, together with another array, `char_slice`. In line 6, the function passes the pointer to this array as an argument to invoke an FFI function. The callee FFI function, however, has a buffer overflow bug, as shown in Figure 2b. At line 4 of Figure 2b, the function invokes `memcpy` with the inappropriate buffer size, resulting in the overflow into the `buffer`'s metadata that is stored right next to `char_slice`. Any further smart pointer dereferences in safe Rust will use the corrupted one to examine if the memory access adheres to the memory safety, potentially resulting in memory safety violations. We refer the readers to an earlier work [20] for more details about this kind of vulnerability. Attacks of this nature are commonplace in Rust programs that depend on bug-ridden FFI functions, as discussed in a recent study on cross-language attacks [20]. Similarly to the earlier example, neither XRust [19] nor TRust [2] mitigates such attacks. XRust does not consider FFI functions, and TRust stores both `buffer` smart pointer and `char_slice` in the same region which FFI functions can access because `buffer` is used in an unsafe block at line 7. PKRU-Safe [14] and Galeed [23], on the other hand, would successfully mitigate because the FFI function will not access the `buffer` smart pointer during the profiling. Nevertheless, PKRU-safe does not provide the comprehensive protection of the smart pointer in that they do not protect safe Rust memory objects from *unsafe* Rust, which could also have memory safety vulnerabilities.

```rust
fn main() {
    let mut buffer = vec![0,2,3];
    unsafe {
        buffer.set_len(10);
    }
    println!("Element at 9: {}", buffer[9]);
}
```

Figure 3: Misusing smart pointer APIs to update smart pointer metadata.

```rust
use std::ptr;
use std::alloc::{dealloc, Layout};
fn test(input: *mut String){
    unsafe{
        //destructor of the pointed-to value
        ptr::drop_in_place(input);
        dealloc(input);
    }
}
fn main(){
    let x = Box::new(String::from("hello"));
    let p = Box::into_raw(x);
    test(p);
    println!("{:?}",p);
    let tmp1 = unsafe{ Box::from_raw(p)};
    println!("{:?}",tmp1);
}
```

Figure 4: Misusing smart pointer APIs to feed invalid pointers to smart pointers

3.3 Logical Bugs in Smart Pointer Changes

Logical bugs in the legitimate smart pointer changes also threaten its integrity. Smart pointers are supposed to be changed at run time, and their implementation provides the interface for intended updates. However, the Rust compiler cannot statically verify the correctness of such updates regarding memory safety, leaving room for potential logical bugs in intended changes to smart pointers, which could violate memory safety. Figure 3 shows an example of buggy smart pointer updates. In line 4, the length metadata of `buffer` is overwritten to a value larger than the allocated buffer's actual length. Subsequent dereferences using the updated smart pointer, such as the one at line 6, will be examined with inappropriate smart pointer metadata, causing the memory safety violation in safe Rust. Logical bugs like this are repeatedly found in real-world Rust programs [7, 18, 26], suggesting that this is an actual problem demanding a systematic solution. Figure 4 shows another example where `Box` smart pointer is updated with a dangling pointer. In line 12–13, a raw pointer `p` is taken and passed to a function `test`, which frees the memory chunk pointed by `p`. In line 16, the pointer `p` is used to create another smart pointer `Box`, causing the program to commit a UAF violation when the `Box` is dereferenced in line 17. The root cause behind this is that `Box` accepts a pointer without ensuring that the pointer is actually pointing to a live memory chunk.

3.4 Need for Isolation and Sanitization

We argue that simultaneously applying isolation and sanitization techniques is necessary to protect smart pointers. Isolation is essential to protect smart pointers from corruption exploiting memory bugs in unsafe Rust or FFI functions. Such corruption does not happen through the predefined interfaces for smart pointer updates; thus, sanitization on the interfaces alone cannot prevent it. On the other hand, sanitization is also necessary to protect smart pointers from logical bugs in intended smart pointer updates and from the crafted invocation of the updated interface using memory bugs in unsafe Rust or FFI functions. METASAFE fulfills the first requirements by storing smart pointers in a separate, gated memory region and enabling only the intended update interface to write to the region. For the second requirement, METASAFE refers to the memory layout and liveness information available in the memory allocator to validate the correctness of smart pointer updates. These design decisions arise from the observation that smart pointers are rarely updated while frequently used, as we further detail in the following section. Table 2 summarizes the motivation of METASAFE and its consideration as the source of Rust memory bugs from observation compared with other works.

4 Assumption and Threat Model

We assume that a program is primarily written in Rust, but inevitably contains some unsafe Rust blocks or functions, and use external libraries that are written in potentially any language, such as C. To mitigate the risks of using such unsafe code pieces, we assume that the program may use existing compartmentalization schemes such as XRust, Galeed or TRust to protect safe Rust's memory objects from the unsafe code pieces. As presented in Figure 2, Figure 3 and Figure 4, with or without such protection, such a program is left vulnerable to malicious smart pointer manipulation, the threat that METASAFE is designed to fight against.

We consider an attacker targeting a Rust program and knowing the vulnerabilities of the program. This includes memory safety vulnerabilities in unsafe Rust or external libraries and logical bugs in smart pointer metadata manipulation. The programs that compartmentalize such unsafe parts effectively prevent such attackers from corrupting safe Rust's memory objects. However, exploiting such vulnerabilities still enables the attackers to modify certain smart pointer metadata to trick even the *safe* Rust program into making unsafe memory access. Specifically, such an attacker corrupts one or more smart pointer metadata so that safe Rust code is misled to make unsafe memory access. For example, manipulating the bounds in a smart pointer can cause the safe Rust code to access memory out of bounds (e.g., buffer overflow). Finally, in a program hardened by isolating *unsafe* Rust, the attacker may still be interested in corrupting the unprotected objects used by internal *unsafe* Rust by corrupting smart pointer metadata. METASAFE aims to narrow the attack surface by considering all smart pointers regardless of where they are used in the program.

We also consider an attacker aware of Rust's polymorphism and attempting to corrupt the function pointer in a trait object to hijack the program. Using an existing vulnerability, the attacker may overwrite a trait object to execute a desired routine. Such code reuse attacks are highlighted by CLA [20], and we aim to mitigate them by treating trait objects similar to smart pointers.

5 METASAFE Design

Isolation and Validation. Figure 5 provides an overview of how METASAFE isolates smart pointers and validates their updates. METASAFE ensures the correctness of smart pointer metadata by allowing only the implementation of the smart pointer ① to update it and validate new smart pointer metadata whenever updated ②. METASAFE compartmentalizes the smart pointers in a separate memory region, called *gated region* ③, and allows only the smart pointer's implementation to update the metadata, i.e., prevents the others from writing to the gated region ④. On each update through the genuine smart pointer implementation, METASAFE further examines the new metadata value's correctness regarding memory safety ②. That is, METASAFE refers to the ground truth it can find from the memory allocator ⑤ to determine if a new metadata value could cause a memory safety violation.

Compile Time Transformation. To this end, METASAFE performs static analysis and code transformation at compile time as an extension of the Rust compiler and runs with its runtime library, including the augmented heap allocator.

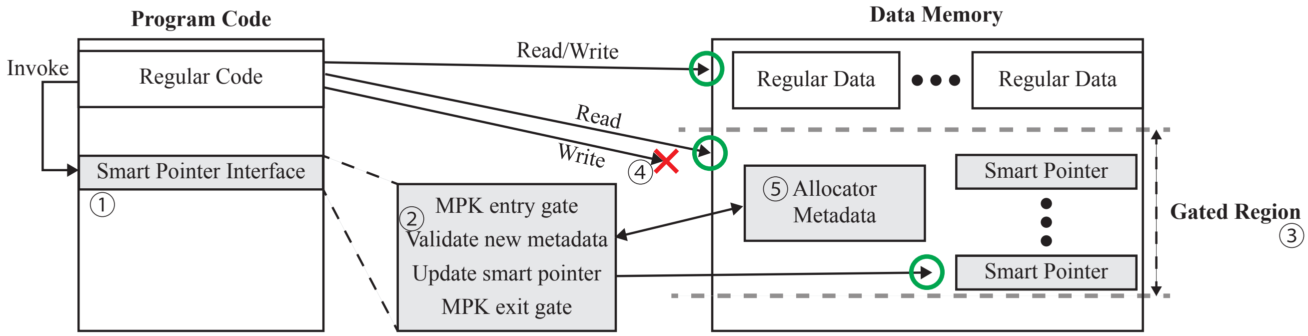
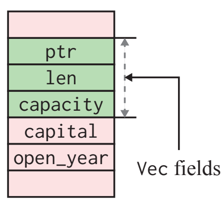


Figure 5: An overview of METASAFE

```rust
struct Firm {
    employees: Vec<String>,
    captital: usize,
    open_year: usize,
}
```

(a) An ADT with a smart pointer field.



(b) Layout of the Firm ADT in memory

Figure 6: A general ADT may include a smart pointer as a data field.

METASAFE can employ any existing in-process isolation mechanism to protect the gated region. For example, our implementation of METASAFE uses Intel MPK to protect the gated region by associating the gated region pages with pkey 1 and the rest with pkey 0. The heap allocator is augmented to enable the program to determine with which pkey a heap chunk is allocated, so that the program can store smart pointers in the gated region and the others outside the gated region (Figure 5). During the compilation, METASAFE identifies smart pointers and their implementations to transform the program to utilize the METASAFE runtime. METASAFE first identifies all smart pointer types and the corresponding implementation. Using the identified types, METASAFE associates each heap allocation site with the corresponding type and transform the program to request chunks from the gated region when allocating for the smart pointer. The smart pointer update functions are wrapped with appropriate entry and exit mechanisms proposed in existing mechanisms [2, 14, 35].

5.1 Challenges

The succinct and elegant objective of isolating smart pointers and gating all their updates presents several challenges.

C1. Ground Truth for Metadata Validation. Smart pointer metadata is not static and is often updated legitimately by the smart pointer implementation at run time. METASAFE is tasked not only with validating the correctness of such legitimate updates but also with scrutinizing unintended alterations originating from misbehaving unsafe Rust or FFI functions. Consequently, METASAFE must have access to a reliable reference or ground truth to ensure metadata updates' validity. This ground truth must provide key information about the memory objects, such as the bounds or type.

C2. Identifying Smart Pointers. The Rust official document does not state any criteria for a type to be a smart pointer. Smart pointers are essentially composite types and are treated similarly to other composite types defined in Rust programs, as described in §2.1. Smart pointer types often implement `Deref` trait to appear as if they are primitive pointer types by overriding the * operator. However, whether or not implementing the `Deref` trait cannot be a criterion because it is not a requirement for a type to be a smart pointer type. Having a statically determined list of smart pointer types is not a viable option either because Rust allows users to define their own smart pointer types.

C3. Embedded Smart Pointers. Smart pointer objects can be embedded within another object of a composite type, as Figure 6 shows. Compartmentalizing such smart pointers is not straightforward because they are supposed to be located within the same heap chunk or stack slot with the other fields of the composite type. Storing the entire object that embeds a smart pointer in the gated region is a viable option, but it limits the extent of METASAFE security guarantees, as we further explain in §5.2.4.

C4. Securing References to Smart Pointers. Safe Rust occasionally accesses smart pointers indirectly through their references. This poses another risk when the referenced object is not a smart pointer and can potentially be accessed by unsafe Rust code or FFI functions. Corruption of such pointers can lead to the same consequences as direct corruption of smart pointers, as it can result in Safe Rust using a counterfeit smart pointer.

```
impl<T, A> MetaUpdate for Vec<T, A> {
    fn validate(&self) -> bool {
        metasafe::isLive(self.ptr) &&
        metasafe::getSize(self.ptr) >=
        self.capacity()*sizeof(T) &&
        self.capacity() >= self.len()
    }
}
```

Figure 7: Vec's implementation of the MetaUpdate trait.

5.2 MetaSafe Compiler

MetaSafe extends the Rust compiler to identify smart pointers and transform the program to utilize MetaSafe runtime to protect smart pointers.

5.2.1 `MetaUpdate` Trait

MetaSafe defines a trait called `MetaUpdate` that a smart pointer type can implement to be recognized as a smart pointer. The trait defines a function that MetaSafe invokes to validate updates to the smart pointer implementing the trait, named `validate`.

For each smart pointer, the developer is supposed to provide the implementation of this `validate` as well so that the function can examine the genuineness of the smart pointer metadata, i.e., if the new metadata adheres the property that the smart pointer must satisfy. For example, the `Vec` smart pointer is considered genuine if it meets the following three criteria, as outlined in Figure 7. First, the data pointer must accurately point to the correct memory object. Second, the object should be adequately sized to contain the `capacity` number of elements. Finally, the `capacity` must be at least as large as `len`, the number of elements the vector currently contains. Similarly, the `Rc` smart pointer is genuine only if at least one of its counters is not zero and its pointer is live. For some smart pointers such as `Box`, validating the liveness of the pointer suffices. We also implemented `MetaUpdate` traits for the data structures that the `collections` in Rust's standard library (`std`) implements. These data structures include `LinkedList`, `Iter`, `IterMut`, `IntoIter`, `BtreeMap`, `BinaryHeap`, `VecDeeque`.

As Figure 7 shows, MetaSafe's heap allocator provides two special interfaces that developers can use to obtain ground truth for examining the smart pointer updates. Invoking `isLive` enables the `validate` to determine if the pointer is live, and `getSize` returns the size of the object associated with the pointer.

During the compilation, MetaSafe automatically inserts calls to these `validate` functions at the end of every function in smart pointer implementation that modifies its fields in a fashion similar to Rust's `Drop` glues. If the call to this function returns `false`, MetaSafe interprets it as a memory bug and aborts the program.

5.2.2 Identifying Smart Pointers

The first step of MetaSafe's smart pointer-aware compilation is to identify smart pointers at High-level Intermediate Representation (HIR) level in the Rust compilation flow. MetaSafe considers every and only the type that implements `MetaUpdate` as a smart pointer type. Using the classification result, MetaSafe annotates each heap and stack allocation site whether it allocates for a smart pointer. To be more specific, MetaSafe associates each heap allocation site with the corresponding type ID that the Rust compiler generates for each type during the compilation and classify the type IDs into two categories: smart pointer types and non-smart pointer types. For the stack allocation site, MetaSafe does not need the type IDs, so it only annotates each site with a boolean value indicating whether the allocation is for a smart pointer or not.

5.2.3 Storing Smart Pointers in Gated Region

MetaSafe transforms the program at LLVM IR level to let the program store smart pointers in the gated region using the type IDs and annotations that it created in the previous step.

The stack allocation sites, which are the execution of `alloca` instructions, are transformed to allocate its slot from the gated region if the allocation is for a smart pointer. To this end, MetaSafe creates and maintains one more stack in the gated region similar to existing safe stack or shadow stack techniques often used to defeat return-oriented programming (ROP) attacks [2, 4, 12].

MetaSafe similarly transforms heap allocation sites to allocate smart pointers from gated regions. One difference is that it makes the program deliver a bit more information to the heap allocator so that it can provide more information about a heap chunk for subsequent validation of metadata updates. We call this information as *CIndex* and the information, and the heap allocator uses this to determine how each allocation request will be handled, as we detail in §5.3. Specifically, MetaSafe uses 1 as the CIndex for smart pointers and derived it from the type ID for non-smart pointers.

One challenge in this transformation is in the fact that many types a Rust program uses are generic types whose type ID is determined only at the monomorphization stage of the compilation. For this reason, MetaSafe actually obtains the exact type ID during the LLVM IR generation from MIR. Actual smart pointer identification also happens at the same time.

5.2.4 Handling Embedded Smart Pointers

As elucidated in §5.1, one of the main challenges confronting MetaSafe pertains to safeguarding smart pointers nested within another composite type as a field. Figure 6a shows an example of such composite type containing a smart pointer, and Figure 6b shows the memory layout of the composite type.

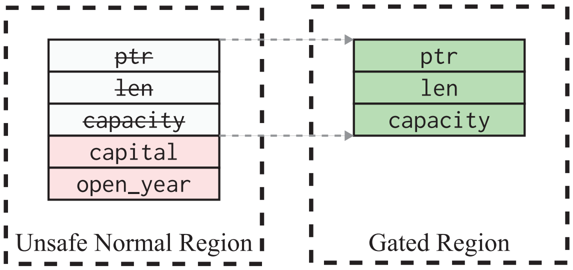


Figure 8: Redirecting smart pointer struct fields to the gated region.

Such a composite type containing a smart pointer cannot be stored entirely in the gated region because this requires all memory instructions that may write to the composite type object, including non smart pointer fields, to be granted access to the gated region. This potentially leads to a large number of memory instructions being granted access to the gated region, which is undesirable.

To avoid this undesirable relaxation of access control, METASAFE stores shadow copies of the smart pointer fields in the gated region and redirects all memory instructions that access the smart pointer fields to the shadow copies, as illustrated in Figure 8. To this end, METASAFE instruments the program to redirect all memory references pertaining to the `employees` field, which is a smart pointer, thereby steering them toward the corresponding objects within the gated region. To be more specific, we make two design choices on top of our observation on creating and working with the shadow smart pointers.

Secure and Efficient Redirection for Heap Objects. An important observation enabling secure and efficient redirection to the shadow copies of smart pointers is that METASAFE maintains a type-pooled heap, and the ratio of composite types containing smart pointers is low. This observation first allows METASAFE to be capable of determining the exact amount of shadow memory required for containing smart pointers. Whenever the heap allocator creates a new pool for a smart pointer-containing composite type, it can deterministically compute the amount of shadow memory required for the pool and prepare it. Moreover, this observation allows METASAFE to redirect memory references to the shadow copies of smart pointers securely and efficiently by using a simple offset computation relying only on heap metadata. At the moment of the redirection, the program is given only the original smart pointer field's address, along with the composite type object's base address, from which the heap allocator determines the pool that the object belongs to with the object's offset. From the base address of the pool and the object's offset, the heap allocator can compute the base address of the shadow copy's pool and its offset and return it to the program. This flow of obtaining the shadow copy's address is secure because the program does not use the content of the composite object that resides outside the gated region.

Secure and Efficient Redirection for Stack Objects. A similar approach also works for the composite objects and embedded smart pointers in the stack. The layout of a function's stack frame is determined at compile time, during which METASAFE creates a shadow stack frame for the smart pointers. At the same time, METASAFE can allocate slots for such smart pointers embedded in the other composite types in the stack and redirect memory references to the shadow copies of the smart pointers statically using the shadow stack pointer.

Handling Contiguous Memory Operations. Nonetheless, this approach necessitates METASAFE's vigilant monitoring of contiguous memory operations conducted on buffers harboring such data structures. For instance, a `memcpy` operation involving a `Firm` type pointer mandates a subsequent `memcpy` operation on the shadow region. This entails further analysis by METASAFE during the code generation from MIR to LLVM, owing to the partial loss of type information at the LLVM level.

5.3 Type-pooled Heap Allocator

METASAFE runtime comes with an augmented heap allocator based on `mimalloc` that manages more than one pool of heap chunks to provide the ground truth for both the chunk bounds and the chunk types. In addition to the size and other existing arguments, METASAFE's allocator takes the index of the desired pool, called CIndex, as an argument. The heap allocator then returns a heap chunk from the pool corresponding to the given CIndex. As mentioned earlier, the heap allocator also creates the pool for shadow copies of the embedded smart pointers when needed. It also ensures that all its metadata are stored within the gated region and serves the allocation requests with CIndex 1 from the gated region.

The heap allocator also implements the two interfaces that `validate` function in `MetaUpdate` needs, namely `getSize` and `isLive` using its metadata.

METASAFE's `is_valid_ptr` is an extension of `mimalloc`'s that checks whether a pointer is part of the heap. Since `mimalloc` pages are classified by object size, a particular page fits a known number of objects.

To implement `isLive`, we extend page metadata of `mimalloc` by adding bits for tracking *liveness* of a given object. The liveness bit is set when the corresponding object is allocated from the page, and unset when the object is freed. With this extension, `isLive` determines if a pointer is still alive in $O(1)$. `isLive` further verifies that the pointer belongs to the correct type pool of chunks. This typed isolation and liveness check reduces the chances of hijacking through UAF bugs because UAF leaves undetected only when a pointer is wrapped again with the same type after the corresponding chunk is also allocated again. METASAFE enables `mimalloc`'s deferred free option to further mitigates UAF exploits.

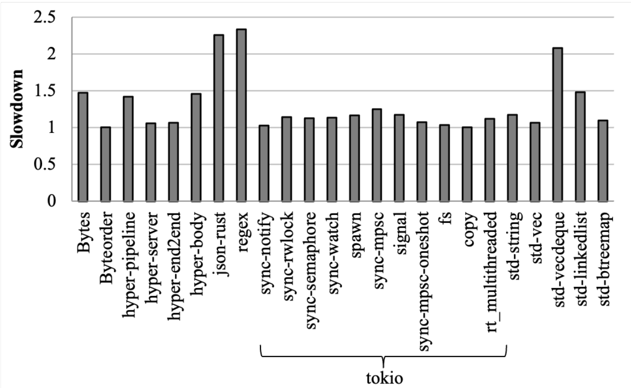


Figure 9: Impact of METASAFE on the execution time of microbenchmarks.

5.4 Protecting the Gated Region

The choice of mechanism for protecting the gated region is orthogonal to the design of METASAFE, and the details of its implementation are not part of our contribution. Nevertheless, we explain two primitives that we use in our evaluation for completeness. For example, TRust [2] and PKRU-Safe [14] have already explored the design space where a Rust program uses software fault isolation (SFI) and Protection Keys for Userspace (PKU), which is also called Memory Protection Key (MPK), to isolate unsafe Rust and FFI functions.

The primary mechanism that METASAFE uses for protecting the gated region is MPK. On a system where MPK is available, we consider it a better choice than SFI in that the transformed Rust program does not frequently enter and exit the functions granted to write to the gated region. To use MPK as a means to protect the gated region, we follow the design choices that many existing studies have already explored [2, 14, 35]. All memory pages belonging to the gated region are associated with pkey 1. The write permission to these pages is granted temporarily only for legal writers to the gated region, such as the smart pointer APIs and heap allocator. METASAFE regards outermost callsites to smart pointer functions as the boundaries for granting and revoking write access. It disallows inlining callee functions at such callsites, clones the callee, and inserts instructions that enable and revoke write access at the beginning and just before every return instruction of the cloned function, respectively. METASAFE then uses the cloned function as the callee at such callsite. Similar to ERIM [35], METASAFE insert one more check just after revoking access to ensure attackers do not redirect the program with write access enabled. It is worth noting that system patching and addressing potential pitfalls, as outlined in [5], are essential responsibilities outside the scope of this work but critical for maintaining the overall security posture.

Benchmark	Full Inlining (ns/iter)	Controlled Inlining (ns/iter)	Slowdown
Bytes	1975	1858	0.9407
Regex	1088948	1432505	1.3155
std-string	1056	1184	1.1212
std-VecDeque	446	531	1.1906
std-vec	311	327	1.0514
std-BTreeMap	20158	21334	1.0583
std-linkedlist	104	108	1.0385
json-rust	681	803	1.1791
Geomean			1.1067

Table 3: Impact on the execution time of controlled inlining of gate callsites that METASAFE uses.

6 Evaluation

We evaluate METASAFE on performance and security. For performance evaluation, we measure the impact of execution time and memory usage when running microbenchmarks §6.1 and a real-world application §6.2. We test if METASAFE stops the exploits real-world CVEs for security evaluation §6.3, and measure the performance impact when METASAFE works together with an existing isolation mechanism §6.4.

Experimental Setup. We build and test METASAFE on a workstation that runs Ubuntu Jammy 22.04.2 LTS with kernel version 5.19.0. The workstation runs on a 12th Gen Intel Core i5-12400 CPU with 6 cores operating at 2.50GHz with 16GB of DDR4 ECC memory. We ran all our experiments under the same system settings. In all our experiments, the benchmarks are compiled with Rust optimization level 3, and we make sure the baselines are executed with the original `mimalloc` allocator without METASAFE components.

Benchmarks. To understand the effects of its design on a lower level, we apply METASAFE to 19 microbenchmarks from widely used Rust crates, similar to the earlier studies on Rust runtime memory safety [2, 14, 19]. We choose these benchmarks with the consideration of smart pointer usage, memory usage intensity, and concurrency. For example, we include `std collections`, many of which heavily update smart pointer metadata, presenting the worst case for METASAFE.

We further investigate the effect of METASAFE on real-world programs by applying it to Servo, an upcoming Rust-written web rendering engine. The web rendering engine is one of the core building blocks of web browsers, which is a widely used application, and has been used for evaluation in several works [14]. In this evaluation, we use three widely used browser benchmarks to evaluate the impact of METASAFE on the performance of Servo.

Note that we do not compare the performance overhead of METASAFE with the others because no existing studies present a mechanism that protects smart pointer metadata.

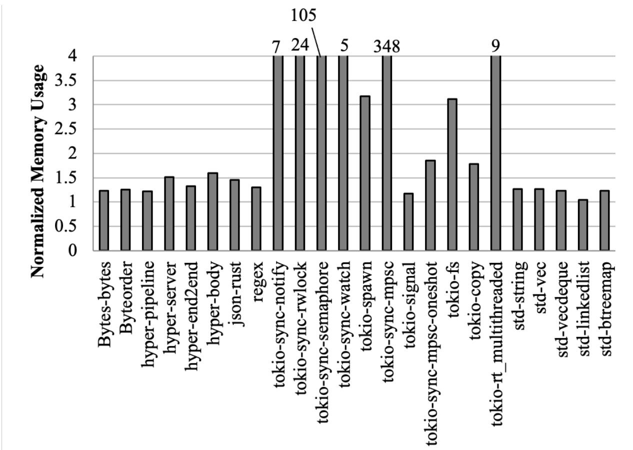


Figure 10: Impact of METASAFE on memory usage.

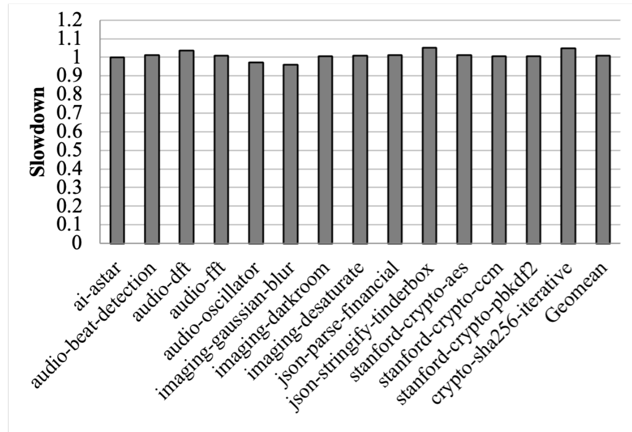


Figure 11: Performance overhead of METASAFE on Kraken.

6.1 Microbenchmark Results

Benchmarks. We run widely used Rust crates to evaluate the impact of METASAFE. We include the `std collections` in the test suite because METASAFE implements the `MetaUpdate` trait for them, affecting their performance. `Hyper` and `Tokio` are famous crates used for asynchronous programming. Using them, we intended to gain insight into the impact of METASAFE on the performance of concurrent programs. Finally, `Regex`, `Json`, `Bytes`, and `Byteorder` represent common data manipulating crates in Rust. All these crates rely on `std collections`, and most of their data structures contain smart pointers as their data fields, i.e., have embedded smart pointers that we discussed in §5.2.4.

Impact on Execution Time. Figure 9 shows that METASAFE slows down the execution by 25.5% on average (geometric mean). What contributes to this overhead are permission switches in smart pointer implementation and redirection for embedded smart pointers. The high overhead on three `std` components, `vecdequeue` and `linkedlist` are due to their frequent smart pointer updates, and the overhead on `regex` and `json` can be explained by their heavy use of embedded smart pointers. One of the implementation detail, METASAFE's controlled inlining at smart pointer update sites (see §5.4), also contributes to the overhead. Table 3 evaluates the impact of this detail, showing that it incurs 10.67% slowdown on average (geomean). This design choice can be revised to perform inlining with care to eliminate this extra overhead.

Impact on Memory Usage. We measure the maximum resident set size (MRSS), the maximum allocated physical memory during a process's lifetime to evaluate the impact of METASAFE on memory usage. Figure 10 shows that for single-threaded and lightly concurrent environments (i.e., all but `tokio`), METASAFE uses up to 27% more memory on average (geomean). In heavily multithreaded settings such as `tokio`—where thousands of threads are spawned, METASAFE uses 8.3× more memory on average. This high overhead can be explained by three things, that is, METASAFE's extra stacks, shadow memory, and METASAFE's segregated memory allocation. To establish the exact cause, we decided to disable segregated memory allocation per type and maintain only two memory regions - one for smart pointers and the other for the rest of the objects, but this showed negligible change in memory overhead. We therefore decided to store all objects on the same stack, a change that showed METASAFE using only approximately 31% more memory in `tokio` benchmarks, which is similar to the overhead in single-threaded and lightly-concurrent environments. For every thread created, METASAFE creates two stacks—one for pure smart pointers and the other for objects with embedded smart pointers. In an environment that creates thousands of threads, thousands of stacks will be created. This explains METASAFE's high memory overhead.

6.2 Servo Results

We use `Servo` to evaluate the performance impact of METASAFE when applied to real-world programs. In particular, we run on Dromaeo [33], Kraken [34] and Octane2.0 [32] on Servo. We modified `Servo` to use our `mimalloc` allocator and protected by METASAFE. For the baseline execution, we use the unmodified `mimalloc` as the heap allocator to rule out the impact of allocator choices on performance. We do not make any other changes to `Servo` or the benchmarks themselves.

Benchmarks. Kraken and Octane2 predominantly evaluate JavaScript performance in web browsers, covering various tasks, including audio processing, image manipulation, encryption algorithms, mathematical calculations, and memory management, with test cases drawn from real-world applications and synthetic scenarios. Both benchmarks emphasize the execution speed and efficiency of JavaScript code,

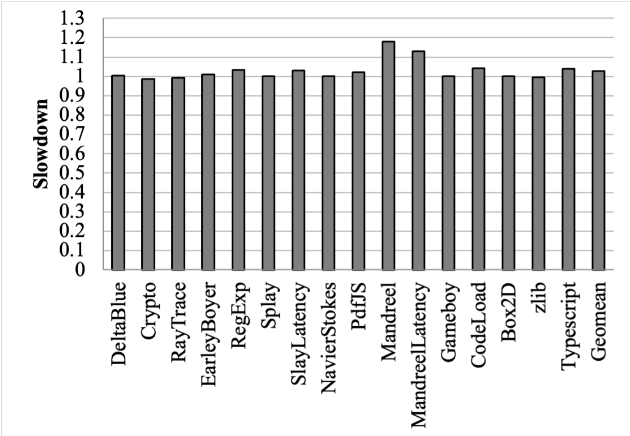


Figure 12: Performance overhead of METASAFE on Octane2.

with Octane2 offering a more robust assessment of various dimensions of JavaScript optimization. On the other hand, Dromaeo, while still covering JavaScript operations like parsing and string manipulation, places a stronger emphasis on DOM manipulation. It measures the time taken by a browser to complete individual test cases in real-world scenarios and generates an overall score based on average completion times. This approach positions Dromaeo as a more accurate indicator of a web browser's rendering engine performance, in contrast to the JavaScript-focused Kraken and Octane2. The tests are run as recommended by the Servo team to ensure reliable results.

Kraken. Figure 11 shows the performance overhead of METASAFE on servo executing Kraken benchmarks. On average, METASAFE slows down the execution the benchmark only by 1.2% (geomean). This result shows that METASAFE is not likely to slow down real-world applications significantly.

Octane2. Figure 12 shows the impact of METASAFE on servo for the Octane2 benchmark. On average, METASAFE incurs an overhead of 3.1% for METASAFE on this benchmark.

Dromaeo. Figure 13 shows the impact of METASAFE on the execution time of METASAFE on Dromaeo. The overhead of METASAFE on Dromaeo is slightly higher than the overhead on Kraken or Octane2. On average, METASAFE incurs a 6.4% geomean overhead. While executing this benchmark, we noticed that it is memory allocation intensive. That is, Servo frequently allocates and frees heap chunks during the execution, where each allocation is likely to be followed by smart pointer updates. Nonetheless, this overhead still remains as low as 6.4%.

6.3 Security

We evaluate the effectiveness of METASAFE in protecting the smart pointer integrity using two experiments. The first experiment focuses on protection against the vulnerable unsafe Rust code, and the other targets the vulnerable FFI functions.

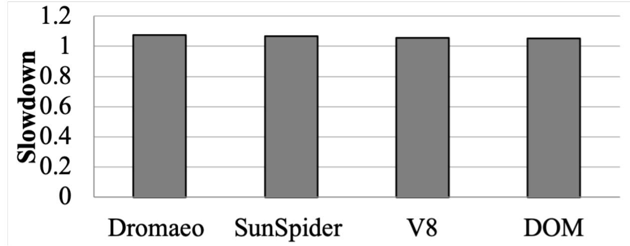


Figure 13: Performance overhead of METASAFE on Dromaeo.

6.3.1 Protection against vulnerabilities in Unsafe Rust

In the first experiment, we test if METASAFE stops the attacks exploiting CVE-2021-25900 [26], which showcases a possible misuse of smart pointer APIs and its catastrophic outcome. CVE-2021-25900 is a vulnerability found from `SmallVec`, a widely used crate providing `Vec`-like buffers, within the stack. In the vulnerable version, the vulnerability is found from the `insert_many` function [1]. The function has an `unsafe` block, where the length of the `Vec` buffer is set to 0. It subsequently calls the `reserve` function to increase the buffer size, but the `reserve` does not increase the size if the `length` is not greater than `capacity`. In this context, the buffer size does not increase, unlike the intention of the caller, because the `length` is set to 0 earlier. Despite this, the `length` is set to a value greater than the `capacity` afterward, potentially causing the safe Rust to misuse this smart pointer to make an out-of-bound memory access.

To evaluate METASAFE's ability to fight against this vulnerability, we ran the vulnerable version of `SmallVec` crate to reproduce CVE-2021-25900 [26]. When we ran this without METASAFE, the program sometimes exits without an error, but it crashed when we used a large buffer. With METASAFE, however, an error is thrown the moment the `line 1069` is executed, halting the program. This bug is caught by `Vec`'s validator call inserted by METASAFE. Note that this misbehavior would elude the existing works including Galeed [23], TRust [2], XRust [19], and PKRU-Safe [14] because each either does not quarantine unsafe Rust from the safe Rust or does not give smart pointer integrity special care and erroneously place it in unsafe Rust-reachable memory region.

6.3.2 Protection against Corruption from FFI Functions

In the second experiment, we evaluate the effectiveness of METASAFE against an attack exploiting vulnerable FFI functions. To this end, we choose to reproduce CVE-2019-15548 [24] from the widely used ncurses library because the library is also used for Rust programs as it is with its Rust wrapper called `ncurses-rs` [39]. Specifically, we wrote a proof-

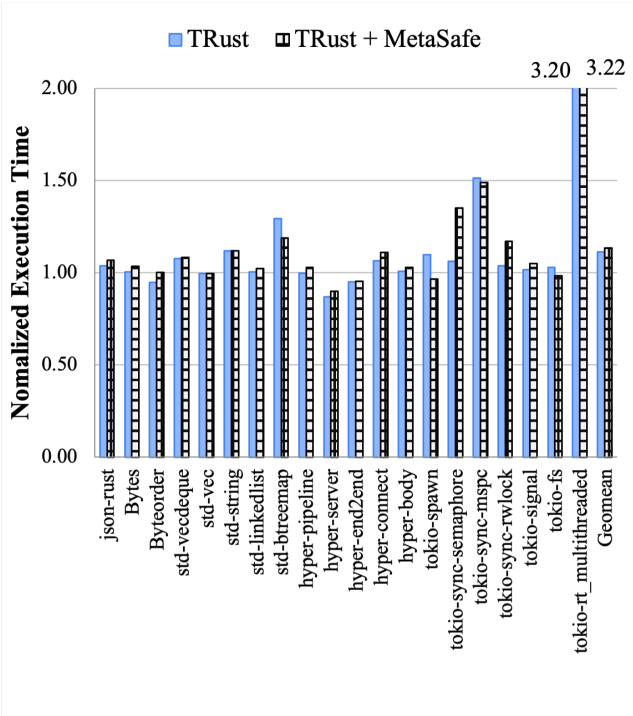


Figure 14: Impact on execution time of METASAFE when it runs with TRust

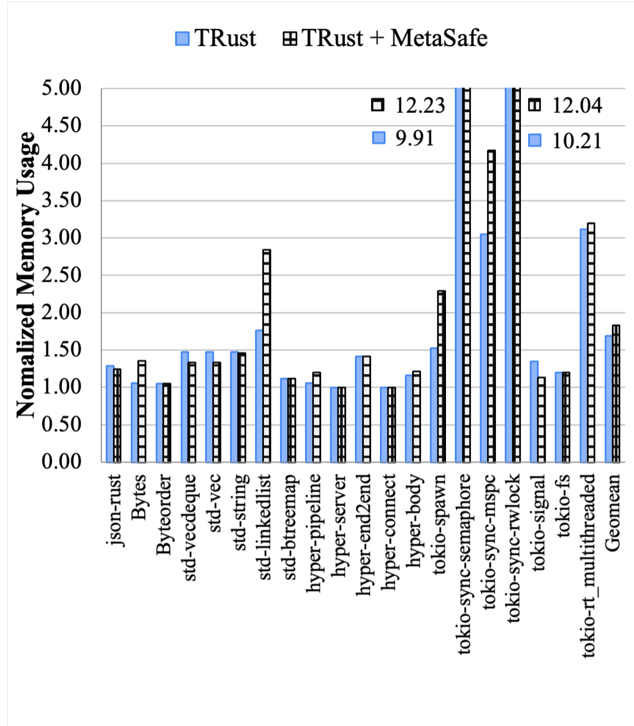


Figure 15: Impact on memory usage of METASAFE when it runs with TRust

of-concept exploit around the buffer overflow vulnerability in the `instr` function in the library. Without METASAFE active, we observed that the buffer overflow could corrupt the smart pointer metadata, leading to memory safety violations in the safe Rust code that uses the corrupted smart pointer. In contrast, METASAFE effectively prevents this type of memory corruption by isolating smart pointer metadata from FFI functions. Other systems, such as TRust or XRust, do not offer this protection, as their smart pointers are likely stored in memory regions accessible to FFI functions. However, systems like Galeed and PKRU-safe can also prevent this corruption because they strictly quarantine FFI functions from any objects not explicitly provided to them.

6.4 METASAFE with TRust

We evaluate the performance impact of METASAFE when it is used with an existing isolation mechanism, TRust [2]. We choose TRust among several possible choices because it is open sourced, and is designed to quarantine not only the external libraries but also the unsafe Rust. Specifically, we adapt METASAFE to leverage TRust's isolation to safely keep smart pointer metadata. METASAFE places the smart pointer metadata in the safe region that TRust protects from unsafe blocks written in unsafe Rust and external code. The validators that METASAFE inserts mediate smart pointer metadata updates by unsafe blocks, enabling proper use of smart pointer APIs. These validators are considered as a part of trusted code blocks, like the safe blocks. With this adaptation, we find that METASAFE does not need to switch the write permission using MPK when running smart pointer APIs because unsafe blocks are already quarantined by TRust using SFI. This positively affects the performance impact of METASAFE when working with TRust.

Figure 14 shows the normalized execution time TRust runs alone and with METASAFE. We use the benchmarks that TRust was evaluated with to measure the performance, and ran the benchmarks on a different workstation running Ubuntu Jammy 22.04.2 LTS, with an Intel 11th Gen CPU with 8 cores and 72GB of RAM because TRust depends on SVF [31] which requires a substantial amount of memory at compile time. As the result shows, METASAFE does not impose significantly more overhead on execution time because most overhead comes from isolation. The overhead of METASAFE running with TRust is 13% on average, while the overhead of TRust without METASAFE is 11%. Figure 15 shows the impact of METASAFE on memory usage. We observe that METASAFE increases the overhead on memory usage because it still has to create the shadow copies of smart pointer fields in the safe region. On average, TRust running with METASAFE uses 83% more memory while TRust as it is uses 69% more memory.

7 Discussion

7.1 Source of Performance Overhead

The METASAFE system appears to adapt proficiently to real-world applications, as evidenced by its performance in the Servo benchmarks. However, a more detailed exploration of METASAFE's functionality and potential overheads can be gleaned from microbenchmarks. Upon evaluating the four microbenchmarks–std-vec_dequeue, std-linkedlist, json-rust, and regex–that exhibited the highest overhead, we discovered a crucial factor: the ratio of useful work accomplished versus gate transitions significantly impacts performance. std-vec_dequeue and std-linkedlist consist of 4 and 6 test functions, respectively. In vec_dequeue, the predominant function frequently calls vec_dequeue::push, a method that alters metadata and is consequently enclosed with call gates by METASAFE. Our analysis revealed that this function was accountable for over 90% of the overhead within this particular microbenchmark. In the case of std-linkedlist, all six benchmarks primarily insert and delete nodes from the list—operations that necessitate metadata modification and transitioning between METASAFE's gates.

We observed that json-rust utilizes composite data structures with smart pointers as fields. All test functions predominantly parse data, repeatedly invoking smart pointer update functions like vec::push that necessitate METASAFE's gates. Thus, the overhead in this microbenchmark originates from both repetitive gate transitions and address masking required to access the shadow memory region. regex presents a similar issue to json-rust, but with 219 tests instead of a handful. From these observations, it becomes clear that when a program frequently updates smart pointer metadata or contains a substantial number of objects with embedded smart pointers as fields, the overhead can be substantial. However, real-world applications involve much more than just transitioning. Memory-intensive programs could also pose problems, as noted in Servo's Dromaeo benchmark. This implies that while METASAFE performs well in some scenarios, particular types of applications may reveal inherent overheads.

7.2 Allocator Metadata Integrity

Although an allocator is primarily for servicing heap memory requests from a program, it usually stores metadata on the chunks of memory allocated. METASAFE relies on allocator metadata as the ground truth to base its validation operations. Should the integrity of this metadata be compromised, METASAFE's protection becomes unreliable. In addition to smart pointer metadata, it becomes natural and essential for METASAFE to protect allocator metadata as well. This makes the allocator functions to be granted write permission to the gated region in addition to the smart pointer functions. Even when an allocator is invoked from an FFI function, METASAFE must enable write access to the protected region. This is done carefully by wrapping exposed allocator function calls with WPKRU routines.

7.3 Hardware Dependence

We realize that METASAFE's choice of Intel MPK alienates other architectures. ARM Domain [8] is a similar protection for ARM CPUs and can be used to replace MPK if available. Software fault isolation (SFI) can be used for a similar purpose for architectures with no similar mechanism.

7.4 Dependence on Source Code

METASAFE can protect a Rust program only when the source code is available, and the program can be recompiled with the protection. We make this assumption because we consider the developers who write Rust programs using unsafe Rust and external libraries for expressiveness and productivity. In such use cases, it is reasonable to assume that we can compile the program with the protection enabled because the tool is used at the time of development. It would be helpful if the same technique could be applied to binaries without recompilation, possibly with the help of a specialized runtime library, but none of the existing works have taken this direction yet.

7.5 Dependence on `validate` Functions

The effectiveness of METASAFE in preventing smart pointer corruption is limited by the correctness and expressive power of validate function. As mentioned in §5.2.1, METASAFE defers the task of implementing validate functions to be called on each metadata update to the developers The only role and guarantee METASAFE delivers is the protected execution of a developer-provided environment, and any incorrect implementation of validate may result in safe Rust using corrupted smart pointers. What may affect this correctness is the expressive power of validate. Developers can write validate only with the trustworthy information that is available at the time of each update, and only the METASAFE-provided interface, isLive and getSize provide such information. We made this design choice because METASAFE primarily aims to prevent spatial safety violations owing to the smart pointer corruption, and validate only needs these two ground truths to ensure spatial safety. Certainly, developers cannot provide powerful validate function if they need unavailable information, especially when they want to ensure a property other than spatial safety violation owing to smart pointer corruption. For example, the developer may want to know where the heap object has been allocated in the code. We believe that exploring the kinds of ground truth that METASAFE can efficiently provide helps improve the expressive power of `validate` functions, and we consider this as a future work.

7.6 Using METASAFE for Rust Libraries

In designing METASAFE, we implicitly assume that the program is primarily written in safe Rust and uses unsafe Rust or FFI functions for productivity and expressive power. We designed METASAFE with this assumption because many examples of software follow this model, but this does not mean that it is the only use case. As noted in an earlier work [23], some legacy programs are incrementally adopting Rust for security by replacing their modules with a Rust version. We believe that the smart pointer integrity must also be considered in such scenarios and they may need a mechanism like METASAFE, but we leave evaluating the efficiency when applied to such Rust libraries for legacy programs as a future work.

7.7 Performance Trade-Offs

By presenting METASAFE, we aim to provide developers with a means to avoid vulnerabilities easily when working with unsafe Rust or external libraries. The primary use cases that we consider are those used for productivity or expressive power. The constraint Rust enforces on the safe Rust program prohibits the program from using particular data structures, motivating the developers to introduce unsafe Rust code. FFI functions are often used to avoid reimplementing something that already exists in other languages. In these cases, developers can simply enable METASAFE to defeat the exploits targeting smart pointers. However, doing this may result in suboptimal performance, especially if a developer introduces unsafe Rust code primarily for performance improvement and the benefit is not significant enough, being amortized by the overhead of METASAFE. For example, if using unsafe Rust brings 5% performance benefit while METASAFE introduces 10% overhead (the geomean from our evaluation), rewriting it with safe Rust and not using METASAFE should bring better performance. This potential performance degradation does not nullify the potential use cases of METASAFE because the overhead of METASAFE is around 10% while unsafe Rust could bring more performance benefit as a recent work comparing the performance of C- and Rust-implementation of same algorithms [41].

8 Related Work

8.1 Memory Safety in Rust

As mentioned in §3, recent studies viewed unsafe Rust and FFI functions as the remaining sources of vulnerabilities and proposed to isolate them from safe Rust. Accordingly, more existing mechanisms designed to enhance the memory safety of Rust programs share the primary goal of preventing corruption from unsafe components propagating to the rest by restricting access to the safe Rust's memory objects.

XRust [19] allocates heap objects that are used in unsafe Rust and FFI functions with separate heap allocator, relying on the manual indication of a programmer. To enforce in-process isolation, XRust uses SFI, instrumenting runtime checks to ensure that objects allocated by the separate allocator would never be able to dereference outside the specified region.

TRust [2] automatically identifies stack and heap memory objects used in unsafe Rust and FFI functions, as well as their allocation sites. It uses SFI to isolate unsafe Rust and Intel MPK to isolate FFI functions.

PKRU-Safe [14] uses dynamic profiling to distinguish heap memory objects the FFI functions access and protect the other from the FFI functions using Intel MPK. Unlike XRust and TRust, PKRU-Safe's focus is FFI and does not handle bugs arising from unsafe Rust. These techniques assume that the vulnerability lies only in unsafe Rust and FFI functions and strive only to isolate them.

Galeed [23] is another work on compartmentalizing Rust programs. It is distinguished from the others by the target use case. Unlike most other compartmentalization works, Galeed aims to protect a Rust program that runs as a part of a larger program written in unsafe language. From the perspective of the threat model, Galeed is close to PKRU-safe. It also considers unsafe Rust as trusted and only prevents non-Rust code from accessing the Rust program's compartment. Specifically, Galeed mediates the access to Rust's compartment from outside using the pointer shared by Rust program, called *shared pointer*, using the concept called pseudo-pointers. These shared pointers delivered to the external code are merely the memory addresses that the external code understands and can be derived from either smart pointers or others used by Rust program. Galeed aims to prevent the misuse of these shared pointers by the external code, while METASAFE aims to prevent the corruption of smart pointers within Rust program. Note that smart pointers remain protected because only Rust code, which includes the ones in unsafe Rust, has access to the smart pointer metadata under Galeed's threat model. However, vulnerabilities in unsafe Rust code, which is known to be prevalent, may still enable an attacker to manipulate safe Rust's memory object, including smart pointer metadata.

As discussed in §3, these either leave safe Rust objects directly accessible from unsafe Rust or do no give special care to smart pointers, potentially leaving them accessible from unsafe Rust or FFI functions.

8.2 Compartmentalization and In-process Isolation

The idea of partitioning or compartmentalizing a program into multiple compartments and granting only the selected portion of the program access to each compartment is not new and has been explored for decades under many different contexts. One of the most widely used primitive is SFI, owing to its independence to architectural support. Many solutions have been published to utilize, adapt and optimize SFI for different use cases [3, 9, 10, 22, 29, 30, 38, 40, 42]. Koning et al. [16] systematically evaluates and compares many different primitives, including SFI, and reports that MPK is effective for compartmentalizing programs thanks to its low permission switch latency. This work led to the development of more intra-process sandboxes [11, 28, 35] using MPK or similar architectural features. The popularity of this technique also motivated a comprehensive study on the level of security and the best practice of MPK- or PKU-based isolation techniques [5, 27, 37].

9 Conclusion

This study introduces METASAFE, a pioneering approach that extensively explores and safeguards smart pointers and their associated metadata storage in Rust. METASAFE is designed to compartmentalize and protect metadata of smart pointers and allocators from unauthorized access using Intel MPK. Furthermore, it ensures metadata integrity by validating modifications to smart pointers through its implementation, utilizing allocator metadata. We tested the METASAFE prototype across microbenchmarks and real-world applications, including a web rendering engine. Experimental results reveal that METASAFE imposes minimal (<7%) overhead on the execution time of real-world applications, indicating its efficiency. We also demonstrated its effectiveness by presenting real-world CVEs that METASAFE could successfully mitigate, thereby attesting to its security guarantees. In essence, METASAFE presents an effective and efficient solution for safeguarding smart pointer metadata, a crucial component for ensuring memory safety in Rust.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2022R1F1A1076100), the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program(IITP-2024-2021-0-01817) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), IITP grant funded by the Korea government(MSIT) (No.2021-0-00724, RISC-V based Secure CPU Architecture Design for Embedded System Malware Detection and Response), the 2023 Research Fund (1.230030.01) of UNIST (Ulsan National Institute of Science & Technology) and Samsung Electronics Co., Ltd. Additionally, this work was supported by the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2023, the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(RS-2023-00277326), the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2020R1A2B5B03095204) and partly supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2020-0-01840,Analysis on technique of accessing and acquiring user data in smartphone). Finally, this work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) under the artificial intelligence semiconductor support program to nurture the best talents (IITP-2023-RS-2023-00256081) grant funded by the Korea government(MSIT).

Availability

In support of open science, we will release the implementation of METASAFE as open-source upon the publication at the following location.
https://github.com/cssl-unist/metasafe

References

[1] Yechan Bae. Buffer overflow in insert_many.

[2] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. TRust: A compilation framework for in-process isolation to protect safe rust against untrusted code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6947–6964, Anaheim, CA, August 2023. USENIX Association.

[3] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*, volume 16, pages 21–24, 2016.

[4] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 10(6):368–379, 2013.

[5] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. Pku pitfalls: Attacks on pku-based memory isolation systems. In *Proceedings of the 29th*

USENIX Conference on Security Symposium, pages 1409–1426, 2020.

[6] National Vulnerability Database. CVE-2021-32810.

[7] National Vulnerability Database. Cve-2019-16138 detail, 2019.

[8] ARM Developer. Cortex-A8 Technical Reference Manual.

[9] U. Erlingsson and F.B. Schneider. Sasi enforcement of security policies: a retrospective. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 287–295 vol.2, 2000.

[10] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 75–88, USA, 2006. USENIX Association.

[11] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[12] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The taming of the stack: Isolating stack data from memory errors. In *Proceedings of the Network and Distributed Systems Security Symposium*, page 17, 2022.

[13] Dana Jansens. Supporting the use of rust in the chromium project, 2023.

[14] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 132–148, 2022.

[15] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 437–452, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *EuroSys*, April 2017.

[17] Abner LI. Google reports decline in android memory safety vulnerabilities as rust usage grows, 2022.

[18] Gentoo Linux. Rust: Multiple Vulnerabilities - GLSA 202210-09, CVE-2021-28875.

[19] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with xrust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 234–245, 2020.

[20] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[21] Mozilla. Mozilla welcomes the rust foundation, 2021.

[22] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. ewasm: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3492–3505, 2020.

[23] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe rust safe with galeed. In *Annual Computer Security Applications Conference*, pages 824–836, 2021.

[24] RustSec. Rustsec-2019-0006, 2020.

[25] RustSec. Rustsec-2020-0038: Memory safety issues in compact::vec, cve-2020-35890, 2020.

[26] RustSec. Buffer overflow in smallvec::insert_many, 2021.

[27] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for {PKU-based} memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, 2022.

[28] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient In-Process isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[29] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. *arXiv preprint arXiv:2002.09344*, 2020.

[30] Christopher Small and Margo I. Seltzer. Misfit: constructing safe extensible systems. *IEEE Concurr.*, 6:34–41, 1998.

[31] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[32] Google Team. Octane 2.0. https://developers.google.com/octane/, 2021. Accessed: 2023-06-04.

[33] Mozilla Team. Dromaeo: Javascript performance testing. https://dromaeo.com, 2021. Accessed: 2023-06-04.

[34] Mozilla Team. Kraken javascript benchmark. https://mozilla.github.io/krakenbenchmark.mozilla.org/index.html, 2021. Accessed: 2023-06-04.

[35] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. {ERIM}: Secure, efficient in-process isolation with protection keys ({{{{{MPK}}}}}). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, 2019.

[36] Stephen J. Vaughan-Nichols. Rust in the linux kernel, 2022.

[37] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by)pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 266–282, New York, NY, USA, 2022. Association for Computing Machinery.

[38] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 203–216, New York, NY, USA, 1993. Association for Computing Machinery.

[39] Jeaye Wilkerson and Evan Cameron. ncurses-rs.

[40] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[41] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. Armor: fully verified software fault isolation. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 289–298, 2011.