Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust

Merve Gülmez
Ericsson Security Research
Kista, Sweden
imec-DistriNet, KU Leuven
Leuven, Belgium
merve.gulmez@kuleuven.be

Thomas Nyman
Ericsson Product Security
Kista, Sweden
thomas.nyman
@ericsson.com

Christoph Baumann
Ericsson Security Research
Kista, Sweden
christoph.baumann
@ericsson.com

Jan Tobias Mühlberg
imec-DistriNet, KU Leuven
Leuven, Belgium
Université Libre de Bruxelles
Brussels, Belgium
jan.tobias.muehlberg@ulb.be

***Abstract*—Rust is a popular memory-safe systems programming language. In order to interact with hardware or call into non-Rust libraries, Rust provides *unsafe* language features that shift responsibility for ensuring memory safety to the developer. Failing to do so, may lead to memory-safety violations in unsafe code which can violate safety of the entire application. In this work we explore in-process isolation with Memory Protection Keys as a mechanism to shield safe program sections from safety violations that may happen in unsafe sections. Our approach is easy to use and comprehensive as it prevents heap and stack-based violations. We further compare process-based and in-process isolation mechanisms and the necessary requirements for data serialization, communication, and context switching. Our results show that in-process isolation can be effective and efficient, permits for a high degree of automation, and also enables a notion of application rewinding where the safe program section may detect and safely handle violations in unsafe code.**

I. Introduction

Rust is an emerging system programming language with memory safety guarantees and performance characteristics close to traditional system programming languages such as C and C++ [1]. Rust is designed to make programs difficult to exploit by attackers by providing compile-time type-checking and memory management based on ownership rules. These static analyses in Rust are conservative: While the compiler will never accept an unsafe program, it may reject safe programs. To work around this incompleteness of compile-time analyses but also to accommodate the need to interact with inherently unsafe low-level operating system interfaces or hardware, Rust provides *unsafe Rust* [2]. In unsafe Rust it is possible to, e.g., dereference a raw pointer or to modify a mutable static variable. Rust then relies on the developer for the correctness of unsafe parts of a program, and failure to ensure correctness will lead to memory errors that may compromise safe parts of the program.

A common use of unsafe Rust is to call C library functions. While the Rust ecosystem is growing, it is not feasible to reimplement common library functionality in Rust right away and the Rust community embraced a Foreign Function Interface (FFI, [2]) to conveniently call non-Rust code. Foreign functions are always assumed to be unsafe by the compiler and it obliges the developer to integrate C libraries safely. Common causes of unsafety regarding the FFI are library interfaces that are not thread-safe, pointer arguments that are not completely sanity checked, and the use of raw pointers. The use of FFI is pervasive: According to Li et al. [3] more than 72% of packages on the official Rust package registry (crates.io) depend on at least one unsafe FFI-bindings package. With many high-profile open-source projects such as the Mozilla Gecko browser engine (≈ 10% Rust[1]) adopting Rust, ongoing efforts to add support for the language to the Linux kernel[2], and Microsoft announcing the uptake of Rust in the Windows OS,[3] the language is gaining a strong foothold across industry sectors and developing easy-to-use and efficient ways to safely integrate legacy libraries through FFI is highly relevant [4].

Earlier approaches exist to make invoking unsafe code safer, e.g., by executing unsafe code in a separate process [5] or by utilizing Memory Protection Keys (MPK) to provide in-process heap isolation [6], [7], based either on developer knowledge or on automated inference in a compilation framework [8], [9]. Separating safe and unsafe program sections into multiple processes has the advantage of combining comprehensive heap and stack isolation with the potential of recovering execution of the safe application part after a crash of the unsafe process, albeit incurring substantial runtime overheads. In-process isolation, in comparison, has mostly been used to isolate safe from unsafe heaps only and with exceptions leading to program termination, yet with much smaller overheads.

This Paper and Contributions. In this work we study Secure Rewind and Discard of Isolated Domains [10] to protect Rust applications that make use of unsafe language features. As a mechanism that enables compartmentalized in-process isolation of safe and unsafe program sections, our approach relies on MPK to provide stack and heap protection, and allows the safe compartment to recover from violations caused in unsafe code, albeit without relying on a custom compiler and with much smaller overheads than earlier related work that achieves similar properties by means of process-based isolation. Our study also engages with the question of how to fairly compare

[1] Mozilla Gecko repository: https://github.com/mozilla/gecko-dev

[2] Rust for Linux project: https://rust-for-linux.com/

[3] “You will actually have Windows booting with Rust in the kernel in probably the next several weeks or months, which is really cool.” – David Weston, director of OS security for Windows, at BlueHat IL 2023.

different approaches to compartmentalization and isolation as they require different approaches to memory management, data serialization, and communication between compartments, which incur a majority of the overheads. Specifically, we make the following contributions:

- We present Secure Rewind and Discard of Isolated Domains for FFI to protect the integrity of a Rust application from memory-safety violations in unsafe program parts or C libraries, utilizing in-process isolation. This approach further increases the Rust application's availability through a secure rewinding mechanism.
- We implement stack and heap protection for Rust-FFI as a Rust crate by leveraging in-process isolation and the Secure Domain Rewind and Discard (SDRaD) C library. We provide an easy-to-use API with a high degree of automation for sandboxing programmer-selected functions.
- We compare the costs of context switching and serialization for process-based and in-process isolation using libpng and snappy libraries as case studies. Specifically, we evaluate different serialization approaches in Rust.

We open-source our prototype and experimental evaluation data under a BSD license on GitHub: https://secure-rewind-and-discard.github.io/ .

We discuss the Rust FFI and approaches to isolating unsafe code in Sect. II and explain the objectives of our study in Sect. III. In Sect. IV and V we detail SDRaD-FFI, and experimentally compare our prototype with related approaches. We report on a security evaluation, lessons learned, and limitations of our approach in Sect. VI. Finally, in Sect. VII and VIII, we discuss our approach in the context of recent related work and draw conclusions.

II. Background

A. Rust-FFI

Rust is a modern systems programming language that is designed to prevent memory-safety defects through a strong type system combined with built-in compile- and run-time checks. Memory-safe Rust will restrict arbitrary casting, prevent temporal memory-safety bugs through a set of ownership rules on data objects, and perform bounds-checks on static and dynamic data allocations. Together, these properties ensure *safe* Rust code is *sound* and will not exhibit undefined behavior [11].

However, Rust programmers also have access to an *unsafe* superset of Rust which may exhibit undefined behavior. With unsafe Rust it is the programmer (not the compiler) who is responsible for ensuring the soundness of the code. Unsafe Rust is generally required when interfacing directly with hardware, operating systems, or *other languages*.

Foreign function interface. The Rust Foreign Function Interface (FFI) enables the sharing of code and functions between Rust and other programming languages. Since the target language may not conform to the memory-safety properties enforced in Rust, FFI calls are considered inherently unsafe. As Rust and other memory-safe system programming languages are being deployed gradually, it is frequently necessary for Rust programs to interface with legacy libraries written in unsafe languages, such as C and C++. Recent work on the security of such multi-language programs [4] demonstrate that the interplay between safe and unsafe languages can undermine existing mitigations against memory attacks.

Different approaches for sandboxing legacy code from the safe Rust code have been proposed [5], [6], [8], [12]. In what follows, we describe the most relevant approaches in detail.

B. Process-based Isolation

Process-based isolation is an essential concept in most operating systems. It protects the system's integrity and resilience by providing the following features: 1) *integrity*: each process runs in its own virtual memory space that prevents a malicious process from accessing the memory of another process, and 2) *resilience:* each process has its own failure boundary so one process' failure does not affect others.

Multi-process software architectures. Compartmentalizing large applications into distinct, isolated processes is a well-known design pattern used both for security and reliability. Multi-process software architectures, such as *site-isolating* browsers [13] come with two significant drawbacks: 1) they are complex to engineer and maintain, and 2) come with associated memory and process-to-process communication overhead that is exacerbated as the number of processes increases.

The engineering challenges with process-based isolation stem from moving from a monolithic program with a single thread of control to a concurrent programming model that necessitate multiple isolated processes to co-ordinate their execution and to communicate results to each other. For example, the incorporation of site isolation to the Chrome browser is the result of a multi-year engineering effort [14].

Automatic software compartmentalization. The problem of *automatically* splitting an existing monolithic program into multiple, compartmentalized components requires program transformation of local function calls into *remote procedure calls* (RPCs) that occur across multiple processes. Sensible boundaries for such RPCs are highly application-specific, e.g., for Chrome, they are defined by the interfaces between the browser Chrome and the isolated component, such as the renderer. In most prior work automatic software compartmentalization the boundary is based on existing library APIs that, using source-to-source translation, can be turned into RPC calls. Source-to-source translation is generally invasive and requires changes to the compiler which rarely make their way into upstream toolchains.

Sandcrust. In this work, we primarily focus on the use case of automatically compartmentalizing unsafe library interfaces in Rust programs. Sandcrust [5] is an easy-to-use sandboxing solution for compartmentalizing Rust applications to execute unsafe code in a separate, isolated process. Unlike solutions that require compiler-based source-to-source translation Sandcrust builds on the metaprogramming capabilities provided by the Rust macro system. This is the key selling point for Sandcrust's ease-of-use: to compartmentalize functions belonging to an

```
sandbox!{
    fn F (...)   {
        unsafe {
            ...  //  unsafe  code
        }
    }
}
```

Listing 1: Transform unsafe F with Sandcrust's `sandbox!` macro to a remote procedure executed in an isolated process.

unsafe library interface, they are simply annotated with a `sandbox!` macro provided by the Sandcrust crate (Listing 1).

Under process-based isolation, for different compartments (i.e., processes) to communicate and exchange data with each other, they must do so via *inter-process communication* (IPC) primitives. IPC typically comes with high overheads due to the context switching associated with scheduling different process and data crossing security and failure boundaries. In Sandcrust, argument passing and communicating return values between the main process and sandboxed code is handled over IPC channels. To allow the passing of Rust (and C) objects across the process boundary, one must be able to serialize, and deserialize, any data type that is to be transferred. As we will show in Section IV-D the overhead associated with serialization and deserialization is the principal source of run-time overhead for compartmentalization in Sandcrust. Consequently, minimizing the overhead of object passing to and from the sandboxed code is an important consideration for making automatic program compartmentalization in Rust more tractable.

C. In-Process Isolation

In contrast to process-based isolation, *in-process isolation* is based on the notion of creating compartmentalized security domains within the memory space of a single application process. The main perceived benefit of in-process isolation is that since transitions from one domain to another stays within the same process, in-process isolation can significantly reduce the run-time cost of context switching compared to traditional process isolation. This is especially beneficial when domain transitions are frequent.

Software fault isolation. In-process isolation is enabled through *software fault isolation* (SFI) [15], a technique for using program transformations to establish *logical* protection domains. The enforcement of such protection domains can either leverage software-based checks inserted through compiler-based program transformation [16]–[19], binary rewriting [20], or hardware-assisted enforcement [6], [21]–[32].

Memory protection keys. The inclusion of *memory protection keys* (MPK) [33] to commodity processors has prompted advances in SFI solutions that leverage hardware assistance. The *protection keys for userspace* (PKU) mechanism in 64-bit x86 processors by both Intel [34] and AMD [35] has received, by far, the most attention from academia.

Protection keys for userspace. PKU associates each 4KB memory page with a 4-bit *protection key* which is stored by the operating system (OS) in the page table entry. An additional 32-bit, CPU-specific, user accessible, *protection key rights register* (PKRU) stores a 2-bit value for each of the 16 possible protection keys, controlling whether associated memory pages are writable or accessible. The validity of the memory accesses is enforced in hardware based on the PKRU configuration.

Because the PKRU is accessible from user space, the access control policy enforced by PKU can be updated without the need to call into the OS kernel. This characteristic makes PKU significantly more efficient compared to OS-enforced memory access control; OS-involvement is only needed when a memory page's protection key is updated. However, as the PKRU policy is entirely controlled from user space, it can also be subverted by adversaries that can control writes to the PKRU. This risks violating the isolation guarantees of PKU-enforced in-process isolation unless PKU is augmented with protection for unauthorized writes to the PKRU. Previous work has explored compiler-based code rewriting [22], binary inspection [24], system call filtering [30], and hardware extensions [28] to harden PKU/PKRU security.

PKRU-Safe and XRust. Automatic compartmentalization of multi-language Rust applications has been explored by Liu et al. [8] and Kirth et al. [6]. These prior works focus on portioning the heap into distinct protection domains for safe and unsafe Rust code, including calls occurring via FFIs. While these compartmentalization approaches share similar goals to process-based isolation, they are not directly comparable as process-based isolation encompasses not only the application heap, but all data, including the stack and static allocations. As discussed in Section II-B, process-based isolation can further provide a degree of resilience against memory corruption whereas conventional SFI approaches terminate the application as soon as a violation of a domain boundary is detected.

In this work, we are concerned with comparing process- and in-process isolation for Rust-FFI under settings which provide similar isolation guarantees. Consequently, PKRU-Safe and XRust do not meet our requirements.

Secure Rewind and Discard. Secure Rewind and Discard of Isolated Domains [10] allows compartmentalizing an application into distinct domains and restoring the execution state of the application in case of memory corruption in the compartmentalized domain. SDRaD is a C library that realizes this scheme using in-process isolation based on PKU. Developers can enhance their applications with rewinding capability by leveraging SDRaD APIs. For example, a compartmentalized domain can be created by assigning a custom user domain index with `sdrad_init()` call. Memory management for separate domain heaps is provided via `sdrad_malloc()` and `sdrad_free()` calls, leveraging the TLSF allocator [36]. Developers can use `sdrad_enter()` to enter and run code in a previously initialized domain and `sdrad_exit()` to exit from the domain. In case a memory corrupting event is detected inside the domain, the execution flows is rewound to the point where the domain was initialized. The application can then take an alternate action to avoid the offending event.

Because restoring the process to a prior point of execution under an adversary model where an attacker has access to the application memory requires strong isolation guarantees that encompass both the application, stack, heap, and static data, SDRaD provides a stronger isolation model compared to XRust [8] and PKRU-Safe [6]. Depending on the configuration, the SDRaD APIs can provide both confidentially and integrity guarantees for the isolated domains.

The main of the drawbacks of SDRaD is that manual effort is required for integrating SDRaD API calls into an application and the current implementation only supports C code. Therefore, while SDRaD provides a better match for the isolation properties we require, it requires augmentation in order to be usable with Rust code.

III. Problem Statement

The goal of this study is to apply in-process isolation as implemented by the SDRaD library to the Rust FFI. Doing so, will allow developers to isolate foreign functionality in an in-process sandbox with resilience to potential memory safety violations. If such a fault occurs, the sandbox is discarded and an error is signaled to developer-provided handler code that may either take steps to recover from the fault (and avoid it being triggered, if possible) or fail gracefully.

An important objective here is to use Rust's metaprogramming capabilities to allow for automatic software compartmentalization in the same way as Sandcrust does for process isolation. As discussed above, such ease-of-use is crucial for the adoption of a hardening mechanism, as high complexity and cost have proven to be one of the major obstacles to the improvement of software security.

The validation phase then compares the resulting prototype with Sandcrust as a representative of automated process isolation for the Rust FFI. In particular, we look at the performance overhead of the two solutions, micro-benchmarking the context switch cost as well as evaluating real-world applications for different data transfer volumes. Moreover we compare the security posture of the two solutions. All study results are discussed in Section VI along with lessons learned.

IV. Prototype Implementation

In order to protect the Rust FFI using in-process isolation, we provide *SDRaD for Foreign Function Interfaces (SDRaD-FFI)*, a Rust crate containing a Linux library for the 64-bit x86 architecture. It allows developers to leverage metaprogramming in Rust to conveniently wrap functions that should be executed in an isolated domain. Under the hood, it uses the SDRaD C library API with PKU as the underlying isolation primitive and it supports compilation with different serialization crates.

A. High Level Idea

As in *Sandcrust*, our SDRaD-FFI crate provides a `sandbox!` macro to annotate functions that are to be isolated (cf. Listing 1). Whenever a sandboxed function is called at run-time it executes in a nested domain, i.e., isolated from the caller, using SDRaD. In case of rewinding after detecting a fault, the behavior depends on the return type of the function. If Rust's *Result*-type idiom [37] is used, the fault is encapsulated as an *Err* variant. Otherwise, SDRaD-FFI raises an unwinding `panic` [38] that can be caught using the standard library `std::panic::catch_unwind` function [39] (cf. Listing 2). In either case the developer decides how to resume: they may try to recover, e.g., by dropping the input that caused the fault, or exit the program gracefully.

```
let result  = std::panic::catch_unwind(||{      ①
    F(...);    ②
    });
match result {
    Ok(v)  => ...  , //  Process result  v ③
    Err(e)  => println!("Fault   in  nested  domain!"),  ④
}
```

Listing 2: Example code snippet demonstrating handling unwinding panics raised by isolated function *F* ②. Function `std::panic::catch_unwind` turns an arbitrary function into a *Result*-type function ①. The result ③ and error ④ handling can be done as if a *Result* was directly returned.

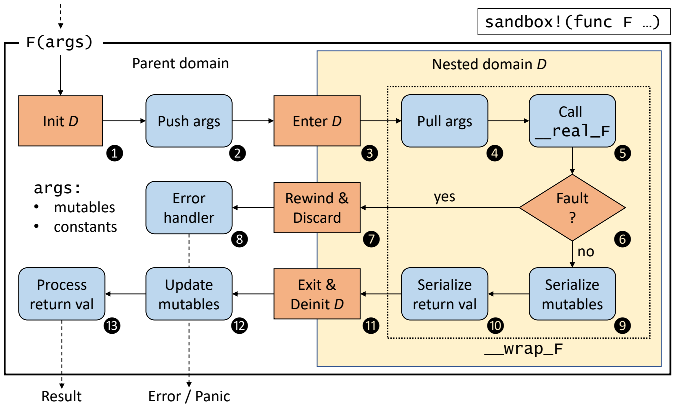


Fig. 1: Transformation of sandbox macro. The angled orange boxed are SDRaD functionality, while rounded blue boxes are new glue logic for parameter passing and error handling.

B. SDRaD-FFI design

The `sandbox!` macro is expanded into calls to the *SDRaD* library and additional glue logic for passing parameters and results between the domains as well as error handling. Syntactically, the definition of a sandboxed function *F* that takes a number of arguments and returns some value is maintained as `__real_F()`, while an additional wrapper `__wrap_F()` is defined to execute the required glue logic in the nested domain. Figure 1 shows the general control flow for executing a call to *F* in a nested domain *D*.

Firstly, domain *D* is initialized as an accessible domain via `sdrad_init()` so that the parent domain can easily push the arguments of *F* into *D*'s memory. To this end, the parent domain serializes the function arguments (cf. Section IV-D) and writes them to a dedicated memory area in the nested domain ❷ (cf. Section IV-C). Next, domain *D* is entered using `sdrad_enter()` ❸ and wrapper `__wrap_F()` is executed in the nested domain area. It first pulls from memory and rebuilds all function arguments using the deserialization method ❹ (cf. Section IV-D). Then, `__real_F()` holding the original function body is executed ❺.

If a memory access violates the nested domain boundaries ❻ the fault is caught by SDRaD, the domain is rewound and discarded automatically ❼, and control-flow is redirected to an error handler which either returns an error value or raises a panic as described above ❽. Similar to as in SDRaD for C code, also Rust applications can be compiled with stack smashing protections (`RUSTFLAGS='-C stack-protector=strong'` [40]) that can trigger a domain rewind in SDRaD-FFI.

On a successful FFI call the results have to be persisted in the safe part of the program. To this end, any mutable local variables used as arguments ❾ and the return value ❿ are serialized and execution returns to the parent domain via `sdrad_exit` ⓫. During this step, the control state of domain *D* is deinitialized to prevent it from being re-entered without a corresponding sandboxed call. The parent domain then deserializes the variables, updates them in the parent domain ⓬, and returns the deserialized return value ⓭.

C. In-Process Communication

SDRaD-FFI requires the parent domain to manage a memory area in the nested domain to transfer data between the domains. SDRaD provides a fixed-size array memory management API for nested domains requiring manual size calculation and error handling to use. To hide these details from the programmer, we leverage the Rust vector type, a dynamically sized and resizable array, for data transfers between domains. Rust vector allocation can be customized by defining specific Allocator traits [41] for different use cases. In our case, we need to create a new vector that should grow and shrink using memory from a specific nested domain. To achieve it, we implemented the Allocator trait for *SdradAllocator* which is passed a specific domain identifier to be used with the SDRaD memory management API. Then a dedicated *SdradAllocator* struct is initialized for each nested domain and if arguments need to be passed from parent domain to a nested domain *D*, a new vector is created for that purpose using *D*'s dedicated *SdradAllocator*.

During a call of a function *F*, all input data is copied to that vector by the parent domain and read out by the nested domain when executing `__wrap_F()`. To this end, the nested domain needs to know vector metadata, i.e., the vector's length, capacity, and backend memory information. The parent domain writes this metadata to a reserved region in the nested domain stack. The nested domain can then create a second instance of the same vector using `Vec::from_raw_parts_in()` function by reading the vector info from its own stack.

However, memory allocated for this vector in Rust is automatically freed when it goes out of scope, i.e., when exiting `__wrap_F()`. As the original instance of the vector references the same memory, that memory would be freed once more when the parent function terminates eventually. To solve this problem, we implemented an SDRaDNestedAllocator that does not free any memory area by itself and we use it to construct the new vector in the nested memory area.

Some of the operations above are not natively supported by the SDRaD API, which we extended (cf. Section IV-E).

D. Serialization and Deserialization

Before entering the newly spawned domain, the function arguments should be passed and redefined in the nested domain. To this end, it is required to track the Rust data types of the arguments and copy all related memory areas into the nested domain area. Sandcrust [5] uses the Bincode serialization crate to pass arguments to another process using IPC. Bincode transforms data into a common binary representation that allows passing data between different platforms. However, as Sandcrust and SDRAD-FFI target only a single platform, this is redundant and Bincode serialization introduces unnecessary overhead. Sandcrust provides a macro-level optimization for *Vec*⟨*u8*⟩ to reduce this overhead. It uses the macro matching mechanism to directly serialize and deserialize data of type *Vec*⟨*u8*⟩ by copying the vector memory area, sending it over the IPC, and later restoring it in another process instead of using Bincode's native serialization scheme. Unfortunately, this solution is not scalable, as it requires providing a dedicated optimized implementation for any other Rust type instance that is not exactly *Vec*⟨*u8*⟩, e.g., *Vec*⟨*u16*⟩ or *Option*(*Vec*⟨*u8*⟩).

Abomonation [*sic*] [42] is a serialization crate that, unlike Bincode and other serialization libraries that provide a platform-independent representations of data types, simply produces a *deep-copy* of all reachable memory that belongs to an in-memory object. However, this raw-memory representation reveals architectural variations and can result in undefined behavior if the in-memory representation of serialized types changes between the time of serialization and deserialization. The latter can occur as a result of changes in the type's definition (which would also affect conventional serialization libraries) or changes in the type's underlying representation, e.g., the use of Rust's `repr(C|packed|align(n))` directives. For this reason, Abomonation is typically not suitable for production use [42] and is primarily of interest as a point of comparison in serialization benchmarks. Nevertheless, in SDRaD-FFI, serialized objects are only persisted across domain boundaries, never leave process memory, and their in-memory representation does not change. Consequently, Abomonation does not introduce undefined behavior in our use case, and is fast compared to other serialization crates [43]. SDRaD-FFI also benefits from Abomonation's ability to deserialize *in-place*, i.e., deserialization results in a shared reference pointing to the serialized data. However, we observed that in-place deserialization prevents Abomonation from respecting memory alignment restrictions in deserialized data [42]. This can lead to suboptimal performance in code tuned to operate with a specific alignment, e.g., to ensure neighboring fields or elements occupy different cache lines, or misbehavior in code that relies on certain alignment for correctness. For our proof-of-concept implementation this limitation did not exhibit adverse effects.

Rust's specialization feature allows traits to have default implementations that can be overridden for more specific types, allowing for optimized trait implementations. [44]. We used this feature to extend the Abomonation trait implementation, optimizing serialization and deserialization of *Vec*⟨*u8*⟩ in the same way as Sandcrust does. As opposed to the Sandcrust implementation, using specialization for traits is more scalable, as it also applies optimizations for sub-type instances of a given type, e.g., for *Vec*⟨*u8*⟩ within *Option*(*Vec*⟨*u8*⟩) in our case.

Our case studies also required passing arguments and results of Rust's *slice* type, which represents sections of vectors, strings, or arrays. Slices are not a *collection type*, i.e., their size is undefined at compile-time and ownership of the underlying memory does not transfer with them, thus deserialization into slices is usually not supported by Rust serialization crates. We employed macro matching to solve this issue by first converting such variables into vectors (for slices of vectors and arrays) or strings (for slices of strings) before transferring them between domains using the above-mentioned mechanism. After the transfer, each variable is cast back to its original slice type.

E. SDRaD Integration and Extension

In the SDRaD-FFI Rust crate, we create a binding to the SDRaD C library using the #[*link*(...)] attribute. The SDRaD APIs are defined in an `extern` block in Rust, and we also define the SDRaD API macros because the C library macro definitions cannot be used directly in the Rust application.

In Rust, the default memory allocator used for dynamic memory management is the system allocator provided by the operating system. However, using alternative allocators such as *jemalloc* or *tcmalloc* is also possible by configuring the Rust compiler. Supporting different allocators may have advantages, such as reducing fragmentation for different applications. However, SDRaD uses the Two-Level Segregated Fit (TLSF) allocator [36], [45] via interposing libc malloc family in the library load order, which is necessary to allow the subheap memory management in the application. That restricts allowing different memory allocators in Rust.

As we discussed in Section IV-C, transferring data between domains requires that the nested domain stack information should be exposed to the parent domain. We extend the SDRaD API with two new APIs: 1) exposing the stack base pointer to the caller, allowing the parent domain to write argument vector metadata in the nested domain stack area. 2) updating the stack base pointer of the nested domain that prevents overwriting information stored on the nested domain's stack by the parent domain. These extensions were sufficient to support the in-process communication mechanism described above.

F. Parallel and Nested Domains

The SDRaD library allows to manage up to 15 in-process domains using PKU. These domains come in different flavors, e.g., accessible or inaccessible by the parent domain as well as persistent or transient. Furthermore, different domains can be used in parallel by different libraries and threads and even be nested to achieve more fine-grained isolation.

We restrict the features of domains in favor of a simplified usage model. By default, the nested domain to isolated foreign functionality from Rust is accessible and persistent to simplify data passing and allow foreign functionality with persistent state on the heap. The developer may manage different domains by providing a custom *Secure Domain Identifier (SDI)* to easily separate persistent internal data, e.g., two different libraries would use two different SDIs in their `sandbox!` definitions.

In principle our implementation also supports multi-threading and nesting of sandboxed calls, however, we consider such use cases out of scope of this study.

V. Evaluation

To evaluate our SDRaD-FFI prototype, we perform microbenchmarks to measure the latency of domain transition via the SDRaD call gate and SDRaD heap allocator function call overheads. We evaluate the performance of SDRaD-FFI applied to function calls from two unsafe C libraries: the compression library *snappy* and the image codec *libpng*.

We evaluated SDRaD-FFI performance using rustc 1.71.0-nightly, Abomonation 0.7.3, bincode 1.3.3, and 2.0.0-rc.3. We compared SDRaD-FFI with Sandcrust using bincode v1.0.0-alpha7. SDRaD-FFI leverages `allocator_api` unstable features. We run all experiments with the "release" profile on Dell PowerEdge R540 machines with 24-core MPK-enabled Intel(R) Xeon(R) Silver 4116 CPU (2.10GHz) having 128 GB RAM and using Ubuntu 18.04, Linux Kernel 4.15.0.

A. Microbenchmark

We perform microbenchmarks to measure the overhead of invoking a function in an isolated domain and the overhead of the SDRaD version of `malloc()` and `free()`. All benchmarks were conducted over 1×10^8 iterations and we calculated the mean execution time and standard deviation across all iterations.

To gauge the overall context switch costs, we sandboxed the `empty()` function and measured its execution time, i.e., the time for entering and exiting the sandbox. We compared SDRaD-FFI to a baseline value without compartmentalization, Sandcrust, and PKRU-Safe. Note that the PKRU-Safe microbenchmark value is an estimate based on the description in the paper [6]. Moreover, PKRU-Safe only compartmentalizes heap memory space, and we report the number here as a reference for another PKU-based isolation mechanism. Figure 2 shows the execution times for all systems under test. While the sandboxed `empty()` function with SDRaD-FFI takes $177.2ns$ (σ=$61ns$), the baseline without sandboxing takes $23ns$ (σ=$21ns$) and Sandcrust takes $8671ns$ (σ=$695.5ns$). SDRaD-FFI is $48.93x$ much faster than Sandcrust and $7.69x$ slower than the baseline function calls. This result is in the same order of magnitude as context switch overheads reported for PKRU-Safe [6].

We profiled the context switch and a main culprit for the overhead seems to be CPU pipeline flushes due to writes of the PKRU register. In SDRaD, each API call involves two such writes for entering and exiting the security monitor. Due to

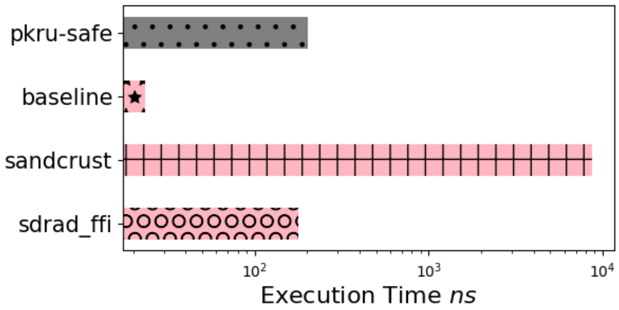


Fig. 2: Microbenchmark of context switch latency for different sandboxing mechanisms. Numbers for PKRU-Safe have been extrapolated from literature.

```
pub fn compress(src: &[u8]) -> Vec<u8>;
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>>;
```

Listing 3: Snappy compress and uncompress function signature

initialization and deinitialization calls, entering and exiting the sandbox accrues four PKRU writes each.

In the second case, we measured the execution time of calling `malloc()` and `free()` functions by allocating and later freeing memory with sizes uniformly distributed over a range from 0 to 4096 bytes. We compare SDRaD with the Rust default allocator. SDRaD takes 66*ns* (σ=40.5*ns*) for `malloc()` and 48.1*ns* (σ=34.2*ns*) for `free()` on average. In the Rust default allocator, the average run time is 44.4*ns* (σ=31.1*ns*), and 33.5*ns* (σ=25.0*ns*) respectively. Thus, in the SDRaD allocator, `malloc()` is on average 1.49*x* slower and `free()` is 1.44*x* slower compared to the Rust default allocator.

B. Snappy

Snappy [46] is a fast compression library that is presented as the FFI example in the Rust Book [2], where the raw snappy C APIs are wrapped by Rust interface functions: `compress()` and `uncompress()` as seen in Listing 3. We sandboxed these functions with SDRaD-FFI, similar to Sandcrust [5].

Compressing and uncompressing randomly generated data of different sizes, we measured the execution time of each operation for 5×10^5 iterations. We evaluate SDRaD-FFI solutions with different serialization crates: bincode and Abomonation. Comparing SDRaD-FFI to Sandcrust, we used the latter's "custom vector" feature for optimized $Vec\langle u8\rangle$ serialization.

Abomonation optimization. In the initial development phase of SDRaD-FFI, we noticed that Abomonation is slow for $Vec\langle T\rangle$ type variables. Abomonation crate has a generic trait for $Vec\langle T\rangle$ variables that are serialized and deserialized item by item and the corresponding iterator code takes up around 60% of the run time. However, for $Vec\langle u8\rangle$ such iteration is redundant: the vector underlying memory area can simply copied in bulk.

To optimize Abomonation for $Vec\langle u8\rangle$, we used the unstable specialization feature as explained Section IV-D. Our experiments show that this specialization improved performance significantly as seen in Figure 3a.

Later, we observed that building with optimization level 3 (release mode) yields similar performance for the unmodified Abomonation crate with generic type $Vec\langle T\rangle$. The Rust compiler optimizes the iteration for $Vec\langle u8\rangle$ and inlines Abomonation completely so that each serialization and deserialization step is just a `memcpy()` operation.

Overall, as described in Section IV-D, the memory layout presentation used by Abomonation to serialize and deserialize variables is suitable for optimization, unlike Bincode, which uses binary format representation and cannot be optimized automatically by the compiler in the same way.

```
fn decode_png(png_image: &[u8]) -> Result<Vec<Vec<
    u8>>, String>;
fn is_png(buf: &[u8]) -> bool;
fn png_init() -> Result<(), String>;
```

Listing 4: Sandboxed functions in libpng

png_create_info_struct()	png_create_read_struct()
png_destroy_read_struct()	png_sig_cmp()
png_get_io_ptr()	png_set_longjmp_fn()
png_read_info()	png_get_image_height()
png_get_rowbytes()	png_read_image()

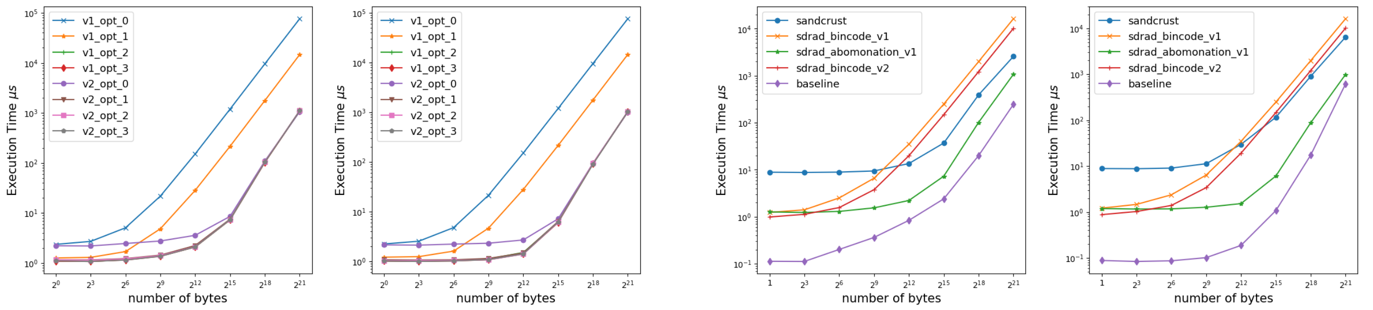
TABLE I: List of *libpng* C functions that are invoked by the sandboxed libpng Rust API in Listing 4.

Snappy benchmark result. We perform the following benchmarking using the original version of Abomonation. Figure 3 show the snappy `compress()` and `uncompress()` function execution times for different byte sizes: 1 byte to 2^{21} bytes. In Figure 3b, Sandcrust performs worst when compressing data smaller than 2^{12} bytes which can be explained by our microbenchmark results from Section V-A.

For data larger than 2^{12} bytes, Sandcrust performs better than SDRaD-FFI with bincode version bincode 1.3.3 and 2.0.0-rc.3. As discussed, the bincode serialization crate is not optimized for serializing and deserializing byte arrays and Sandcrust uses macro-level optimizations for $Vec\langle u8\rangle$ instead of using the bincode serialization. As a result, serialization crates overheads for large bytes are dominant over overheads from isolation methods. SDRaD-FFI with Abomonation crates is faster than the bincode-based versions and Sandcrust.

In general, all compartmentalization methods have an overhead compared to the baseline of at least one order of magnitude if only little data is transferred between the compartments. The overhead is reduced for larger data, but still significant. We profiled the SDRaD-FFI benchmark using the *perf* tool and found that about 60% of the run time is spent in `memcpy()`.

For uncompressing, Sandcrust has slightly better performance than SDRaD-FFI with bincode 2.0.0-rc.3. Even though Sandcrust is optimized to handle variables of type $Vec\langle u8\rangle$, that solution does not apply for other types that contain



(a) snappy `compress()` and `uncompress()` for SDRaD-FFI with Abomonation (v1: original, v2: specialized), compiled with different optimization levels

(b) snappy `compress()` and `uncompress()` for SDRaD-FFI, Sandcrust, and baseline without any isolation

Fig. 3: Execution Time of snappy with different isolation mechanisms for different numbers of bytes.

Vec⟨*u*8⟩. Sandcrust cannot perform the same optimization for the `uncompress()` return value of type *Option*⟨*Vec*⟨*u*8⟩⟩.

SDRaD-FFI with Abomonation does not suffer from this problem by integrating optimizations in the serialization mechanism directly. Its performance comes relatively close to the baseline for large inputs but degrades for small inputs due to the high context switch costs.

C. libpng

The C library *libpng* is used to handle portable network graphic images. We reused the Sandcrust evaluation code published in [47] which sandboxes three Rust functions; their function signatures can be seen in Listing 4. These wrapper functions make calls to a total of ten different libpng C functions listed in Table I to read a PNG image into a vector of row byte vectors. We compared SDRaD-FFI-Abomonation with Sandcrust and measured the execution time of each decoding operation for 1×10^6 iterations and different image sizes.

Table II shows mean and relative-standard deviation for decoding measurement of different images. SDRaD-FFI introduced a worst-case overhead of 11.72% performance degradation compared to a baseline measurement for 5.5*KB* image decoding and it performed significantly better than Sandcrust in all cases.

D. Security Evaluation

We reproduced CVE-2018-1000810 [48] to verify the SDRaD-FFI error handling mechanisms. The `str::repeat` function in the Rust standard library has an integer overflow that causes a buffer overflow. Integer overflows only occur in the release build; Rust checks for these in debug builds. We sandboxed the `str::repeat` function with a specific return value in case a fault occurs and the sandbox is discarded. We found that the buffer overflow causes a domain violation which triggers the rewinding mechanism. Control is transferred to the error handler which returns the specific return value.

VI. Discussion

Below we summarize the insights gained in our study, provide a comparative security evaluation of both process-based and in-process isolation, and outline remaining challenges.

A. Lessons Learned

As we have seen, the in-process isolation approach clearly outperforms process isolation in terms of context switch overhead. This is not surprising, as the development of PKU technology is explicitly designed to reduce context switch cost (be it for switches to the kernel or to other user processes).

Nevertheless, even for modestly sized arguments, the context switch cost starts to get dominated by the cost of data transfer between domains. Here, the data serialization method used can significantly impact performance and thus it is crucial to optimize it for the use case at hand.

One important insight here is that in our scenario, where data is exchanged between isolated code running on the same platform, serialization can essentially be simplified to copying the required data between different memory domains and making sure that data is interpreted using the correct types.

Such an optimization is straight-forward for vector-type variables, however, care must be taken where to implement these optimizations. We found that it is more elegant and scalable to use traits and specializations within the serialization crate instead of implementing optimizations using macro type matching, as is done in Sandcrust.

In this study, we adapted in-process isolation as provided by SDRaD, which involves many options for domains and a rather complicated programming model. We strived to control this complexity and provide a simple interface for isolation of unsafe code, similar to Sandcrust.

The Rust macro system proved to be a powerful tool in this respect as its meta-programming capabilities allow to automate argument and result passing. Moreover, the *Result* type and *panic* concepts could readily be integrated with the sandbox design. If possible, unsafe functions should use the *Result* return type idiom, so that they can easily be sandboxed without the developer having to write a custom panic handler.

Prior work on automated software compartmentalization conventionally relies on compiler-based source-to-source translation which, as discussed in Section II-B, faces a high barrier for adoption due to requiring invasive changes to compilers and toolchains. Compartmentalization APIs such as SDRaD

Data size (bytes)	Execution Time [μs]			Execution Time Overhead	
	Baseline	**SDRaD-FFI**	**Sandcrust**	**SDRaD-FFI** vs. **Baseline**	**Sandcrust** vs. **Baseline**
5.5K	264.14 (σ±3.30%)	295.10 (σ±5.33%)	461.86 (σ±3.30%)	+11.72%	+74.85%
64K	4430.93 (σ±4.00%)	4749.53 (σ±3.19%)	7694.12 (σ±2.91%)	+7.19%	+73.64%
380K	9085.24 (σ±3.16%)	9295.79 (σ±2.14%)	11375.59 (σ±2.50%)	+2.32%	+25.21%
895K	13415.76 (σ±3.39%)	14008.36 (σ±2.00%)	20513.90 (σ±2.81%)	+4.42%	+52.91%

TABLE II: *libpng* Decoding Measurements

place the burden of integrating compartmentalization capability on the developers. For languages such as C, such changes can become invasive as the amount of data to be passed across the domain boundaries increases. Languages with strong metaprogramming capabilities, such as the Rust macro system can greatly reduce the complexity for repetitive integration effort, such as argument passing. There can also be benefits in converging on interoperable interfaces for language- and library-level compartmentalizing; by adopting a similar interface as Sandcrust, SDRaD-FFI can be used as a drop-in replacement for Sandcrust that provides improved run-time performance.

B. Security Evaluation of SDRaD-FFI

The security of Sandcrust is based on the mutual isolation of regular user processes. This is a fundamental and well-studied concept of computer systems [49]–[51] and a multitude of language, compiler, and OS-based hardening techniques exists to raise the bar for adversaries to breach the confines of a user process. [14], [52], [53]

In comparison, hardware-assisted in-process isolation has only been widely supported since rather recently and is not without caveats. In particular, the PKU mechanism as implemented by Intel and AMD has known security issues that necessitate additional compile-time effort and runtime hardening to ensure that the in-process isolation is watertight.

The required measures have been discussed before in detail [10], [30], [54], [55], in a nutshell: 1) instrumented programs need to be scanned for illicit writes to the PKRU register which controls access to domain memory [22], [24], [54], 2) a control flow integrity mechanism needs to be in place to ensure that legitimate PKRU write instructions of the security monitor cannot be abused as gadgets by an adversary [23], [54], 3) run-time modification of code must either be prohibited or subject to binary analysis to verify that no new PKRU write instructions are introduced [24], 4) additional system call filtering and hardening of signal handlers is needed to prevent abuse by an adversary who tampers with the PKU mechanism [30], [54], [55]. While these are not unsurmountable issues, especially requirement 3) makes it tenuous to support functionality that requires just-in-time compilation.

However, improvements of the PKU hardware design have been suggested which would make most of the hardening measures listed above redundant [28]. It would be desirable for processor designers such as Intel and AMD to adopt similar solutions to make in-process isolation more secure and usable.

C. Security Impact on Sandboxed Application

In our case studies of *snappy* and *libpng*, we focus on Rust API functions that handle potentially unsanitized inputs that are passed to the underlying C library for processing. In Table I we list the decoder functions that are immediately called by the sandboxed API of *libpng*. If these inputs are attacker-controlled, compromising the security and availability of the application as a whole becomes feasible. A very recent example of such a vulnerability is illustrated in the recent CVE-2022-3857 [56], where a crafted PNG image can lead to a segmentation fault and denial of service. While our case study does not include the `png_write_png()` call involved in the above CVE (but focuses on `png_read_png()` instead) and we did not try to reproduce this particular vulnerability, our example involves similar risks for the Rust application. Since these vulnerabilities exist in many libraries and even automated exploit generation [57] is possible, we believe that API-based sandboxing techniques, including our SDRaD-FFI, provide a valuable additional line of defense to protect critical application logic – in our case the Rust application – from being attacked through vulnerabilities in third-party libraries. Moreover, isolating such code from the main program creates a failure boundary, improving software robustness. Approaches to assess and validate libraries regarding security requirements such as the absence of certain classes of vulnerabilities is difficult and complementary to our approch.

D. Future Work

As a direct consequence of our work, the performance of Sandcrust could be improved by employing a more streamlined serialization crate such as Abomonation. As we have shown, process isolation is inherently slower than in-process isolation due to the context switch cost, which is especially pronounced for low transfer bandwidth between the safe and unsafe parts of the program. Hence we shift focus to in-process isolation.

As discussed before, SDRaD-FFI can natively support parallel and nested isolation of foreign functionality. Future work should investigate which use cases could benefit from these features as well as their performance impact.

Another avenue for improvement is the security of SDRaD-FFI itself. As it is based on the SDRaD C library, our SDRaD-FFI crate is mostly unsafe code itself which increases the TCB. Formal verification could be employed to obtain correctness and security guarantees on the in-process isolation implementation.

One may ask whether SDRaD-FFI can only be used for the Foreign Function Interface or also to isolate unsafe functionality

TABLE III: Comparison of application compartmentalization schemes discussed in Section VII. The first rows shows how many distinct, isolated protection domains are supported. The *isolation type* is either process (⧉) or in-process (□) isolation. The next seven rows indicate what kind of code modules may be isolated w.r.t. stack and heap memory. *Crash resistance* means that the application can continue execution after protection domain violations are detected and contained. We characterize the *development effort* as follows: the scheme requires invasive code changes, e.g., modifying the arguments of functions (*medium*), superficial code changes suffice (*low*), or the scheme can be incorporated *automatically*. The last rows show the geometric mean of the *performance degradation* for TRust [9], Sandcrust [5] and SDRaD-FFI in the Snappy benchmark (Section V) compress and uncompress operations for input sizes 256B, 1KB, 4KB, 16KB, 64K, 256K, and 1GB as used by Bang et al. [9].

		ERIM [24]	**SDRaD** [10]	**XRust** [8]	**Fidelius Charm** [12]	**Galeed** [7]	**PKRU-Safe** [6]	**TRust** [9]	**Sandcrust** [5]	**SDRaD-FFI**
No. of domains		16[1]	15[1]	2	2	2	2	2	2	15[1]
Isolation type		□	□	□	□	□	□	□	⧉	□
Between safe	Isolated stack	–[2]	–[2]	✗	✓	✗	✗	✓	✓	✓
and unsafe code	Isolated heap			✓	✗	✗	✓	✓	✓	✓
Between unsafe	Isolated stack	✓	✓	✗	✗	✗	✗	✗	✓[3]	✓
and unsafe code	Isolated heap	✓	✓	✗	✗	✗	✗	✗	✓[3]	✓
Mixed-language support		✗	✗	✗	✓	✓	✓	✓	✓	✓
Between safe and	Isolated stack	–	–	–	✓	✗	✗	✓	✓	✓
mixed-language code	Isolated heap				✗	✓	✓	✓	✓	✓
Crash resistance		✗	✓	✗	✗	✗	✗	✗	✓	✓
Development effort		med.	med.	low	med.	med.	auto.	auto.	low	low
Performance degradation	Compress	*Numbers not available*						20.5%[4]	1265.70%	155.7%
in Snappy benchmark	Uncompress							50.0%[4]	6340.2%	370.8%

[1] May be less than 16/15 in cases where Linux's MPK support reserves a few domains for specific purposes. [2] Only supports C and / or C++ code.

[3] Due to manipulating the Rust abstract syntax tree (AST) via Rust's macros Sandcrust's sandbox can also be applied to unsafe functions without FFI.

[4] Result based on numbers reported in Table 8 in pre-print of Bang et al. [9] available at time of writing. We limit the comparison to the 256B – 1GB range due to the custom allocator used by SDRaD limiting the size of continuous allocations to slightly under 4GB in size [10].

in general. This interface is a natural place to split a program into two, as there is likely no interdependency between functionality written in different languages and it is convenient to split the program stack at a function call. Yet, for unsafe code blocks part of Rust functions, working on shared objects in memory is essential. Here it is more challenging to split a Rust program into two domains without breaking the Rust runtime and its control-flow integrity and memory-safety guarantees.

Moreover, unsafe code in Rust is often used to optimize operations for low latency in ways that would not be legal in plain Rust [2]. Given the significant performance overhead of in-process isolation, it might actually be cheaper to use a safe Rust implementation than to isolate an unsafe code snippet. However, certain performance optimization might not be easy to achieve without the use of unsafe Rust [11]. Further research is needed to identify classes of "native" unsafe Rust code for which in-process isolation is practical. In some cases, the Rust standard library may need to be augmented to support execution confined by an isolated domain [8].

VII. Related Work

Several studies have explored the compartmentalizing of applications based on different underlying primitives: hardware-assisted in-process isolation [6], [10], [21]–[32], process-based isolation [5] and software-based isolation [8], [15]. The majority of such approaches address the problem of compartmentalizing applications written in C or C++ [10], [22]–[32]. In this work, we focus on multi-language applications, specifically Rust code calling C or C++ code [5], [6], [8], [9], [12]. As noted in Section II, the majority of prior work on using in-process isolation operate under weaker isolation guarantees than those of comparable solutions based on process-isolation [5]; XRust [8], Galeed [7] and PKRU-Safe [6] only protect the Rust heap from unsafe code while Fidelius Charm [12] only protects the Rust stack.

Independently and concurrently with our work, Bang et al. [9] propose TRust, a scheme with similar goals as Sandcrust [5] TRust protects both the Rust heap and stack from unsafe and mixed-language code. However, unlike SDRaD-FFI, which we designed to act as a drop-in replacement for Sandcrust, TRust relies on compiler-driven compartmentalization of Rust code based on the language-level boundary between safe and unsafe Rust. The principal benefit of compiler-based approaches, such as PKRU-Safe and TRust is relieving the developer completely of the burden of defining protection domains. In practice, automated approaches are restricted to boundaries that can be derived from language-level constructs.

In the case of Rust, approaches such as TRust are not suitable for isolating unsafe code from other unsafe code. Combined with conservative compartmentalization policies that classify any objects that the untrusted code might use as unsafe, this can result in problematic edge cases where nearly all allocations are delegated to the unsafe domain (cf. Table 10 in [9]), negating the benefit of compartmentalization in the first place. Approaches, such as Sandcrust and SDRaD-FFI, which *minimize* developer effort but leave the developer in control can deal efficiently with such cases. Fully automating compartmentalization using compiler-based rewriting can also, at least in the short term, be counter-productive for the adoption of these solutions as invasive changes to mature toolchains are required, which are unlikely to make their way upstream. Point-solutions that can be deployed as discrete libraries or crates offer a more realistic path to industry adoption. Finally, as discussed in Section II, one of the principal motivations for process-based isolation are improved resilience by means of a failure boundary between processes. Apart from SDRaD and SDRaD-FFI, no prior work in in-process isolation provides such a failure boundary between compartmentalized domains.

Table III summarizes our comparison between the relevant compartmentalization schemes for commodity 64-bit x86 processors excluding approaches that require kernel modification [12], [23], [24] for deployment. SDRaD-FFI is the only in-process isolation solution that provides similar capabilities as process-based isolation while significantly improving the run-time overhead compared to the similar Sandcrust in the snappy use case. While SDRaD-FFI remains less efficient than in-process isolation that partitions the program stack and heap in a manner which avoids unsafe code from interacting with objects in the safe area altogether, e.g., TRust, SDRaD-FFI allows programmer-selected unsafe functions to operate safely on data originating from the safe stack or heap. Our benchmarks indicate the measured performance overhead to be highly benchmark-specific, which means performance characteristics are influenced by choice of functions to sandbox.

We posit that schemes such as SDRaD-FFI remain a valid alternative to compiler-based rewriting for use cases that require more flexibility than static analysis can achieve and allow taking developer intent and constraints into account. Static and dynamic approaches to program analysis and vulnerability detection, in particular compiled-program analysis, can inform developers about potentially dangerous functions in libraries, which will allow them to use sandboxing effectively. Furthermore, approaches such as DynPTA [58] or CryptoMPK [31] can inform and complement SDRaD-FFI. Potential future work could further investigate combining compiler-based software compartmentalization, that can provide complete coverage of unsafe code, with developer-guided *further* sandboxing, providing the best of both worlds and defense in depth.

VIII. Conclusions

By allowing to include unsafe code, the Foreign Function Interface (FFI) represents a chink in the armor of Rust's memory safety guarantees [4]–[6], [8], [12]. A line of recent research tries to protect safe Rust code from the fallout of potential memory safety violations in cross-language functionality, e.g., by running such code as a separate process that may be compromised without affecting the safety of the Rust code. In this paper we have studied the use of in-process stack and heap memory isolation to secure the Rust FFI.

Building on the SDRaD C library [10] for secure domain rewind and discard, we developed a prototype, SDRaD-FFI, which allows to conveniently sandbox foreign functionality in Rust. The implementation hides domain operations as well as passing arguments and results between the domains using Rust's macro metaprogramming capabilities. Faults within the isolated code are exposed to the programmer using standard error handling idioms such as *panics* and *Result* types.

We compare our prototype to Sandcrust's implementation of process isolation for FFI [5] and find that in-process isolation outperforms the latter due to its lower context-switching costs. This is further improved by optimized serialization methods for data transfer between isolated domains. We also compare the security of in-process isolation to the traditional process-based approach and find that in-process isolation requires quite extensive hardening to protect the PKU isolation mechanism. However, these shortcomings can be alleviated by updates to the hardware design that have been proposed in related work.

Overall, we conclude that in-process isolation is a usable and efficient security solution for Rust FFI, especially when a lot of data is exchanged between the safe and unsafe portions of the program. Our approach allows to run unsafe foreign code alongside safe Rust code while isolating the latter from possible memory safety violations in the former. Our implementation artifacts is available under a BSD license on GitHub: https://secure-rewind-and-discard.github.io/

Acknowledgments

We thank our shepherd, Jie Zhou, anonymous reviewers, and Niklas Lindskog for their helpful feedback. This work has received funding under EU H2020 MSCA-ITN action 5GhOSTS, grant agreement no. 814035, by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and by the CyberExcellence programme of the Walloon Region, Belgium.

References

[1] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, “Energy efficiency across programming languages: How do energy, time, and memory relate?” in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

[2] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018.

[3] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, “Detecting cross-language memory management issues in Rust,” in *Computer Security – ESORICS 2022*, V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, Eds. Cham: Springer Nature Switzerland, 2022, pp. 680–700. [Online]. Available: https://doi.org/10.1007/978-3-031-17143-7_33

[4] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," in *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, vol. 22, 2022, pp. 1–17. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2022-78-paper.pdf

[5] B. Lamowski, C. Weinhold, A. Lackorzynski, and H. Härtig, "Sandcrust: Automatic sandboxing of unsafe components in Rust," in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, ser. PLOS'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 51–57. [Online]. Available: https://doi.org/10.1145/3144555.3144562

[6] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "PKRU-Safe: Automatically locking down the heap between safe and unsafe languages," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 132–148. [Online]. Available: https://doi.org/10.1145/3492321.3519582

[7] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping safe rust safe with galeed," in *Annual Computer Security Applications Conference*, ser. ACSAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 824–836. [Online]. Available: https://doi.org/10.1145/3485832.3485903

[8] P. Liu, G. Zhao, and J. Huang, "Securing unsafe Rust programs with XRust," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 234–245. [Online]. Available: https://doi.org/10.1145/3377811.3380325

[9] I. Bang, M. Mayondo, H. Moon, and Y. Paek, "TRust: A compilation framework for in-process isolation to protect safe Rust against untrusted code," in *32nd USENIX Security Symposium (USENIX Security 23)*. Baltimore, MD: USENIX Association, Aug. 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/bang

[10] M. Gülmez, T. Nyman, C. Baumann, and J. T. Mühlberg, "Rewind & Discard: Improving software resilience using isolated domains," in *Proceedings of 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '23. Washington, DC, USA: IEEE Computer Society, Jun. 2023, pp. 402–416. [Online]. Available: http://doi.org/10.1109/DSN58367.2023.00046

[11] Rust Team, "The Rustonomicon —- The Dark Arts of Advanced and Unsafe Rust Programming," Retrieved May 22, 2023 from https://doc.rust-lang.org/nomicon/.

[12] H. M. J. Almohri and D. Evans, "Fidelius Charm: Isolating unsafe Rust code," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 248–255. [Online]. Available: https://doi.org/10.1145/3176258.3176330

[13] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1661–1678. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/reis

[14] M. Kosaka, "Inside look at a modern web browser (part 1)," Sep. 2018, retrieved August 28, 2023 from https://developer.chrome.com/blog/inside-browser-part1/.

[15] G. Tan, *Principles and Implementation Techniques of Software-Based Fault Isolation*. Hanover, MA, USA: Now Publishers Inc., 2017. [Online]. Available: https://www.cse.psu.edu/~gxt29/papers/sfi-final.pdf

[16] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '93. New York, NY, USA: ACM, 1993, pp. 203–216. [Online]. Available: http://doi.org/10.1145/168619.168635

[17] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 45–58. [Online]. Available: http://doi.org/10.1145/1629575.1629581

[18] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with API integrity and multi-principal modules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 115–128. [Online]. Available: http://doi.org/10.1145/2043556.2043568

[19] S. Liu, G. Tan, and T. Jaeger, "PtrSplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (ACM CCS)*, 2017. [Online]. Available: https://doi.org/10.1145/3133956.3134066

[20] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 75–88. [Online]. Available: https://www.usenix.org/legacy/event/osdi06/tech/full_papers/erlingsson/erlingsson.pdf

[21] E. E. Rivera, "Preserving memory safety in safe Rust during interactions with unsafe languages," Master's thesis, Department of Electrical Engineering and Computer Science, 2016. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/139052/Rivera-eerivera-meng-eecs-2021-thesis.pdf?sequence=1&isAllowed=y

[22] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 437–452. [Online]. Available: https://doi.org/10.1145/3064176.3064217

[23] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 489–504. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[24] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[25] M. S. Melara, M. J. Freedman, and M. Bowman, "EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments," `arXiv:1907.13245 [cs.CR]`, 2019. [Online]. Available: http://arxiv.org/abs/1907.13245

[26] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 143–156. [Online]. Available: https://doi.org/10.1145/3381052.3381326

[27] H. Lefeuvre, V.-A. Bădoiu, c. Teodorescu, P. Olivier, T. Mosnoi, R. Deaconescu, F. Huici, and C. Raiciu, "FlexOS: Making OS isolation flexible," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 79–87. [Online]. Available: https://doi.org/10.1145/3458336.3465292

[28] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys – efficient in-process isolation for RISC-V and x86," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1677–1694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[29] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran, "Secure and efficient in-process monitor (and library) protection with Intel MPK," in *Proceedings of the 13th European Workshop on Systems Security*, ser. EuroSec '20.

New York, NY, USA: Association for Computing Machinery, 2020, p. 7–12. [Online]. Available: https://doi.org/10.1145/3380786.3391398

[30] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, “You shall not (by)pass! practical, secure, and fast PKU-based sandboxing,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 266–282. [Online]. Available: https://doi.org/10.1145/3492321.3519560

[31] X. Jin, X. Xiao, S. Jia, W. Gao, H. Zhang, D. Gu, S. Ma, Z. Qian, and J. Li, “Annotating, tracking, and protecting cryptographic secrets with CryptoMPK,” in *2022 2022 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 473–488. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00028

[32] Y. Chen, J. Li, G. Xu, Y. Zhou, Z. Wang, C. Wang, and K. Ren, “SGXLock: Towards efficiently establishing mutual distrust between host application and enclave for SGX,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-yuan

[33] Linux kernel development community, “Memory Protection Keys,” Retrieved June 08, 2023 from https://www.kernel.org/doc/html/next/core-api/protection-keys.html.

[34] *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide*, Intel Corporation, 2007, order Number: 325462-076US https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[35] *AMD64 Architecture Programmer’s Manual Volume 2: System Programming. Revision 3.38*, AMD, 2021, publication No. 24593 https://www.amd.com/system/files/TechDocs/24593.pdf.

[36] M. Masmano, I. Ripoll, A. Crespo, and J. Real, “TLSF: a new dynamic memory allocator for real-time systems,” in *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, 2004, pp. 79–88. [Online]. Available: https://doi.org/10.1109/EMRTS.2004.1311009

[37] The Rust Standard Library, “Module std::result,” Retrieved June 08, 2023 from https://doc.rust-lang.org/std/result/.

[38] ——, “Macro std::panic,” Retrieved June 08, 2023 from https://doc.rust-lang.org/std/macro.panic.html.

[39] ——, “Function std::panic::catch_unwind,” Retrieved June 08, 2023 from https://doc.rust-lang.org/std/panic/fn.catch_unwind.html.

[40] Rust Team, “GitHub rust-lang/rust pull request #84197: add codegen option for using LLVM stack smash protection,” Retrieved June 08, 2023 from https://github.com/rust-lang/rust/pull/84197.

[41] The Rust Standard Library, “Trait std::alloc::Allocator,” Retrieved June 08, 2023 from https://doc.rust-lang.org/std/alloc/trait.Allocator.html.

[42] Timely Dataflow, “Abomonation,” Retrieved June 08, 2023 from https://github.com/TimelyDataflow/abomonation.

[43] D. Koloski, “Rust Serialization Benchmark,” Retrieved June 08, 2023 from https://github.com/djkoloski/rust_serialization_benchmark#rust-serialization-benchmark.

[44] The Rust Standard Library Developers Guide, “Using spcialization,” Retrieved June 08, 2023 from https://std-dev-guide.rust-lang.org/code-considerations/using-unstable-lang/specialization.html.

[45] M. Conte, “Github - mattconte/tlsf: Two-level segregated fit memory allocator implementation,” Apr. 2016, retrieved December 7, 2022 from http://github.com/mattconte/tlsf.

[46] Google, “Snappy,” Retrieved June 08, 2023 from https://github.com/google/snappy.

[47] B. Lamowski, “Automatic sandboxing of unsafe software components in high level languages,” 2017. [Online]. Available: https://lamowski.net/docs/Automatic_Sandboxing_of_Unsafe_Software_Components_in_High_Level_Languages.pdf

[48] CVE-2018-1000810, Aug. 2018, retrieved March 09, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2018-1000810.

[49] R. C. Daley and J. B. Dennis, “Virtual memory, processes, and sharing in MULTICS,” *Commun. ACM*, vol. 11, no. 5, p. 306–312, May 1968. [Online]. Available: https://doi.org/10.1145/363095.363139

[50] J. Saltzer and M. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[51] P. A. Karger and R. R. Schell, “Thirty years later: Lessons from the Multics security evaluation,” in *Proceedings of the 18th Annual Computer Security Applications Conference*, ser. ACSAC ’02. USA: IEEE Computer Society, 2002, p. 119. [Online]. Available: https://doi.org/10.1109/CSAC.2002.1176285

[52] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus, “Deconstructing process isolation,” in *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, ser. MSPC ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 1–10. [Online]. Available: https://doi.org/10.1145/1178597.1178599

[53] M. Schwarzl, P. Borrello, A. Kogler, K. Varda, T. Schuster, D. Gruss, and M. Schwarz, “Dynamic process isolation,” `arXiv: 2110.04751 [cs.CR]`, 2021. [Online]. Available: https://arxiv.org/abs/2110.04751

[54] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, “PKU pitfalls: Attacks on PKU-based memory isolation systems,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1409–1426. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[55] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, “Jenny: Securing syscalls for PKU-based memory isolation systems,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[56] CVE-2022-3857, Mar. 2023, retrieved August 30, 2023 from https://www.cvedetails.com/cve/CVE-2022-3857/.

[57] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, “Automatic exploit generation,” *Commun. ACM*, vol. 57, no. 2, p. 74–84, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2560217.2560219

[58] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis, “Dynpta: Combining static and dynamic analysis for practical selective data protection,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1919–1937. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00082