# Fidelius Charm: Isolating Unsafe Rust Code

Hussain M. J. Almohri
Department of Computer Science
Kuwait University
almohri@ieee.org

David Evans
Department of Computer Science
University of Virginia
evans@virginia.edu

## Abstract

The Rust programming language has a safe memory model that promises to eliminate critical memory bugs. While the language is strong in doing so, its memory guarantees are lost when any unsafe blocks are used. Unsafe code is often needed to call library functions written in an unsafe language inside a Rust program. We present Fidelius Charm (FC)[1], a system that protects a programmer-specified subset of data in memory from unauthorized access through vulnerable unsafe libraries. FC does this by limiting access to the program's memory while executing unsafe libraries. FC uses standard features of Rust and utilizes the Linux kernel as a trusted base for splitting the address space into a trusted privileged region under the control of functions written in Rust and a region available to unsafe external libraries. This paper presents our design and implementation of FC, presents two case studies for using FC in Rust TLS libraries, and reports on experiments showing its performance overhead is low for typical uses.

## 1 Introduction

Rust is designed to provide strong memory safety, but provides a way to escape its strict checking rules using an explicit `unsafe` keyword. This enables systems-level Rust programming, and supports easy integration with libraries written in unsafe languages such as C. Code within an unsafe region can use all memory in the Rust program's process in arbitrary ways, jeopardizing all the safety guarantees made by the Rust compiler. Unsafe regions enable calling unsafe and untrustworthy external libraries through Rust's foreign function interface (FFI). When using FFI, the Rust compiler cannot reason about memory vulnerabilities, repudiating all the safety guarantees Rust programmers work so hard to obtain.

The goal of this work is to enable FFI calls while isolating some of the already allocated memory, limiting the potentially-vulnerable external code (running within the same address space) from reading sensitive data. Rust has no mechanism to isolate an unsafe external function or limit its impact. A practical isolation mechanism must be able to limit the data available to an unsafe external function while allowing it to execute normally without the need to modify or even inspect the unsafe code (which may only be available as a binary). Traditional ways to address this problem use computationally-intensive runtime systems for compartmentalization of untrustworthy code, or complex (and seldom usable) modifications of the language's compiler for monitoring and sandboxing vulnerable or unsafe regions of the code. These solutions would require major changes to either the Rust compiler or the unsafe code itself, both of which we want to avoid. Instead, we focus on a solution that leverages existing language and operating system mechanisms.

While no prior work addressed memory isolation for unsafe Rust regions in particular, several previous works have sought to confine code executing within a single address space. Recent work such as Shreds [7], lwC [14], and SpaceJMP [12] provide thread-like abstractions for isolating memory. Codejail [29], a memory sandbox system, has fewer dependencies and provides a sandbox of the memory for unsafe libraries. Our approach reverses the sandbox model by isolating a subset of the trusted region of the program and providing the rest of the memory to the unsafe libraries.

All the previous works, except Codejail, require some static analysis or special abstractions. We aim to have a practical and lightweight solution that allows programmers to make choices about which parts of the memory to protect, that requires only mem-

[1] A first version of this work was accepted for the proceedings of CODASPY'18.

ory page permissions that are supported by modern hardware, and that avoids hard modifications to the operating system, only involving simple kernel extensions. Our approach is to (i) move sensitive program data to protected pages before entering unsafe code, (ii) allow unsafe code to run normally without modifications, (iii) restore visibility of the protected state when unsafe code execution completes, and (iv) incorporate a precise and efficient kernel-level monitor to ensure unsafe code cannot circumvent protections.

**Threat Model** Our solution assumes a trusted starting state in which the operating system, and underlying hardware, are not malicious and are implemented correctly. FC is designed to protect the memory of processes executing *FC-ified* Rust programs, in which sensitive memory regions are protected by isolated secure compartments when interacting with foreign function interfaces. Thus, we assume the code written in Rust is trusted except when using an `unsafe` block to call an external library function.

We assume attackers are remote and do not have root privileges on the target machine (that is, the machine on which FC operates and is subject to attacks) or any way to interfere with program execution other than through the foreign function called by the Rust program. We assume the attacker can use memory vulnerabilities in the unsafe code to access data allocated by the trusted program region, for example, using poorly checked memory boundaries. Thus, the attacker has access to all memory pages available to the process in which the untrusted code executes, except memory pages that are under FC's protection. The attacker aims to exploit vulnerabilities in untrusted code to gain control over the program state in a Rust program.

**Contributions** We present Fidelius Charm[2], a Rust language library and kernel extension support for creating in-memory *secure compartments* by protecting sensitive data in memory from an unsafe function execution. We discuss the design decisions made for developing FC, which intends to hide a subset of the trusted memory regions allocated while executing Rust code. In particular, our work has four primary contributions:

- **Extending Rust's memory ownership** by allowing functions to own their allocated data in *secure compartments* and limit access from unauthorized functions (Section 2).
- **Designing strong kernel-level protection** to maintain the integrity of FC's secure compartments by monitoring and protecting the underlying APIs, such as `mprotect`, as well as performing stack inspection to ensure that access can be re-enabled only by the intended safe Rust code (Section 2.4).
- **Controlling data sharing** between a Rust function and a unsafe library function while providing strong isolation of the safe and unsafe code using a narrow and controlled data interface (Section 2).
- **Testing FC in case studies** to explore its effectiveness and implementation efforts (Section 3).

FC's on-demand secure compartments ensure memory protection and isolation within a single process using architectural support for memory page permissions. FC's design is cross-platform and thread-safe, and requires no modification to the Rust compiler. Our implementation uses a Rust library and kernel extension, both of which are available under an open source license. Using FC involves mostly simple modifications to the program's source code, and low run-time overhead (Section 4).

# 2 Design

To control access to memory allocated in Rust while calling an unsafe code, we designed a compartmentalization technique which splits the address space into three regions: (i) a *private region* that is inaccessible from unsafe functions and is fully accessible from safe functions, (ii) an *immutable region* that could be read from any part of the program, and (iii) an *exposed region*, which is accessible from any part of the program. The private and the immutable regions comprise *secure compartments*, which are collections of continuous memory pages with specific permission bits. These compartments isolate sensitive data from unsafe code by arranging memory appropriately on pages and changing their permission bits to read-only or no access. When an unsafe function executes, depending on the program's policies (specified by the programmer), it has limited access to the secure compartments.

## 2.1 Motivating Example

Consider developing a TCP server using worker threads, excerpted in Figure 1. It uses a `Worker` structure to hold data for a client session, and a `Server`

[2]The name Fidelius Charm is inspired by Harry Potter's Fidelius Charm, which is a complex spell to conceal a secret in a person.

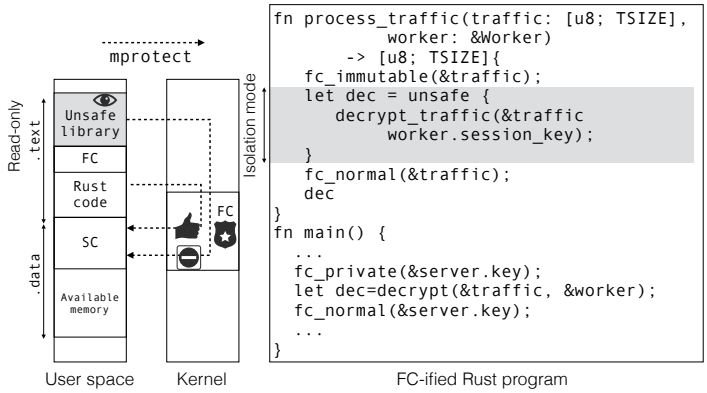


Figure 1: FC's user space and kernel components create and maintain secure compartments in the Rust program's process. Access to `mprotect` calls is restricted to a specific region in the code, which controls the secure compartments. The FC-ified program shows a simplified usage of FC's interfaces to create a private (no permission) secure compartment and an immutable (read-only) secure compartment.Other details of using FC are described in Section 3.

structure to hold data for the main server program. To store client's message for processing, define an array, `let traffic = [0; TSIZE]`. Also, consider a Rust function `process_traffic` that invokes the unsafe external function `decrypt_traffic`, which decrypts `traffic` using `Worker.session_key`.

This simple example includes several aspects that motivate the need for FC. First, when decrypting the traffic using an unsafe library function, the Rust function only needs to share the client's session key in `Worker.session_key`, hiding the server's private key (`Server.key`). Second, the Rust function must protect the original copy of the client's `traffic` and send it as a read-only input to `decrypt_traffic`. Third, when processing a `Worker`'s client, the Rust function must isolate the sensitive data, for example the session keys of another `Worker`'s client. Also, the list of `Worker`s, and other server-related data stored in an instance of `Server` should be secured before executing unsafe code. In Section 3, two case studies show the of FC on actual Rust-based TLS servers.

Since the external library implemented in an unsafe language may have serious security flaws that could allow arbitrary access to memory [25], any call to `decrypt_traffic` using `unsafe` in Rust potentially exposes all of program's memory. FC isolates data objects (i.e., individual variables, referred to as bindings in Rust) to minimize the exposure when calling unsafe code. For example, when calling `decrypt_traffic` to decrypt a client's message using its session key, only the `worker.session_key` is exposed to the unsafe code; `Server.key` and all other sensitive data objects stay in an isolated secure compartment and are temporarily inaccessible throughout the entire program.

## 2.2 Architecture of FC

FC consists of a user space library, a modified Rust program that links to the library, and a kernel module that maintains access control for the program's safe (written in Rust) and unsafe code (arbitrary libraries) regions. Prior to an unsafe call, the programmer adds calls to FC's user space component, which are interfaces linked to the Rust program and facilitate creating secure compartments. As shown in Figure 1, the user memory's data section is divided into pages that form secure compartments and exposed pages with read and write access.

The second component of FC is the modified Rust program that wraps `unsafe` calls with invocations of FC's functions. For example, in the FC-ified program of Figure 1, `process_traffic` creates a read-only secure compartment. The code following a call to `fc_immutable` changes the program's state to the *isolated mode*, in which some of the original memory page permission bits are modified. A subsequent call to `fc_normal` reverses the program's state to *exposed mode* with all data section page permissions are reversed to read and write. The `main` function creates a private secure compartment that hides the server's sensitive data.

FC's user space library serves as a client for its third component, a kernel module that implements a mandatory access control by separating the code section of the user space into two groups: (i) the code written in Rust that has the right to issue `mprotect` calls and modify page permissions, and (ii) the code

written in arbitrary languages, which cannot make `mprotect` calls on pages that are tagged as secure compartments. As detailed in Section 2.4, the kernel module maintains the secure compartments and mediates access to memory page permissions.

## 2.3 Secure Compartments

FC enables two memory permission modes for its secure compartments: *immutable* (read-only) and *private* (inaccessible). The default memory permissions are set by the process at the time of allocating memory pages, which FC does not change. The design of FC faces a key challenge to maintain the integrity of the secure compartments within a single virtual address space by preventing the unsafe foreign functions from accessing the enclosed memory pages. The problem is that both the trusted Rust code and the untrustworthy foreign function are sharing a process, giving them equal operating system-level privileges for modifying memory page permissions. A seemingly simple solution would be to isolate the unsafe code in a separate process. However, this solution requires nontrivial changes to existing code, and can potentially interfere with concurrency in existing Rust programs, which benefit from Rust's clear and memory-safe concurrency model. Our solution preserves the current concurrency structure of the code while providing an inexpensive mechanism to maintain the integrity of secure compartments, specifically by disabling the unsafe code from subverting the policies set by the Rust code.

**Creating and Reversing Secure Compartments** As shown in the FC-ified code of Figure 1, one private secure compartment holds `server.key` and an immutable secure compartment holds `traffic` and `worker.session_key` (since both are on the same memory page, one call to FC will create a shared secure compartment for `traffic` and `worker.session_key`).

Creating the secure compartments starts with a trusted Rust function call to `fc_immutable(var)` (or `fc_private(var)`) to provide protection for all data objects in the memory page where `var` exists. In the example code of Figure 1, this is repeated twice since the secure compartments must be separated by the level of access provided to the unsafe code (a secure compartment cannot be both immutable and private). First, FC examines the number of allocated pages, determines the page addresses, and applies the appropriate permissions to the pages (i.e., `PROT_READ` for immutable and `PROT_NONE` for private), creating two secure compartments. Next, FC issues a system call, sending the kernel a list of page addresses that will be in the program's secure compartments (regardless of being in immutable or private compartments) to deny using `mprotect` on the specified pages. FC also makes a system call to specify a *designated trusted region* (an address in the code section pointing to a function in FC) in the code, which will be allowed (by the kernel) to make `mprotect` calls for reversing page permissions in the secure compartments. The kernel records the trusted region's address and page addresses in a protected address table (Section 2.4) and monitors requests to modify permissions of the protected pages.

## 2.4 Kernel Module

To designate a trusted region for kernel protection, we implemented a Linux kernel module that traps all `mprotect` calls (from those processes carrying a special signal from FC) and monitors the protected memory pages. The reason to use a kernel module is to secure FC's runtime monitoring within the trusted operating system without source code modifications.

FC's kernel module maintains the secure compartments and ensures that only the trusted Rust code can disable the page protections. The kernel extension determines when a call to change page access permissions is legitimate based on code regions. The idea is to designate a specific address range within the Rust code at the time FC creates a secure compartment. The designated address range is communicated to the kernel extension, which will subsequently only allow modifications of the secure compartments to originate and return to the specified address range. The address range is computed at run time according to the fully linked executable.

For security, FC's kernel extension requires that (i) the loaded code to be immutable across the process, except by the operating system or the program loader, (ii) an address $A$ in the code section (address of a function in FC's code, linked to the Rust program), which the kernel can trust to allow `mprotect` calls to return to, and (iii) the *first* system call from the user space FC, which explicitly asks the kernel to restrict access to `mprotect` except those returning to $A$, is trustworthy (done before any unsafe code is executing).

**Code Section Permissions** The page permissions for the code section (`.text` in the linked ELF) must be set to `PROT_READ`, which is ensured by the Rust compiler.[3] FC's kernel extension monitors the page permissions for the code section and deny all

[3] As tested with the Rust's compiler rustc 1.14.0 (e8a012324 2016-12-16).

mprotect calls to them. This monitoring is only for the parts of the Rust code, which must be given the rights to call mprotect, which is a small subset of the READONLY .text section that fits in a memory page. (there is no theoretical restriction that this code exceeds one page, but our implementation does not need more than one).

**Designating the Trusted Region** As described in Section 2.3, FC's user space component invokes the kernel component with a system call (FC reuses existing system calls to avoid modifying the kernel) before executing an unsafe function. After creating secure compartments, a call to fc_protect(var) will send the page address on which var exists along with the address of the trusted region to the kernel (through a system call). fc_protect(var) computes the address of the trusted region as a fixed offset relative to the linked address of fc_protect(var) itself. That is, depending on the number of instructions in fc_protect(var), the offset is manually computed and hard coded in FC's code and is relative to the address given to fc_protect(var) by the linker. The offset must only be changed if the logic of fc_protect(var) was changed.

Designating the trusted region must come from a trusted part of the code, which is code written in Rust and is assumed to have compile-time memory guarantees. As the programs must always start execution in Rust's main function, assuming the programmer has FC-ified the program around all calls to unsafe foreign functions, the first use of fc_protect(var) is trusted to be from the Rust program. In the example shown in Figure 2, a new line of code is added to the body of process_traffic (from Figure 1), which performs the trusted call. Upon receiving the call, FC's kernel module disables mprotect and records the address of the protected page and the trusted region's address in the protected address table. The trusted code region's address is the address to which subsequent mprotect calls on the protected memory pages must return. The call to fc_normal perfroms a system call asking FC's kernel module to first allow calls to mprotect and then reverse page permissions. Before re-enabling mprotect, the kernel checks the instruction pointer of the requesting task to verify the return address from the call against the recorded address in the protected address table. When a malicious call to fc_normal is made, the instruction pointer has an invalid address, and FC's kernel module's policy policy is to kill the process (although other actions could be used for applications where fail safety is important). We will further analyze the security of FC's kernel module and possible attacks in Section 2.5.

## 2.5 Security Analysis

We examine the security guarantees achieved by FC from an attacker's perspective. According to our threat model, the attacker is only capable of a remote attack, for example by crafting requests to a server. FC's effectiveness depends on both the attackers goals and capabilities, and how much data the program exposes to vulnerable unsafe components.

**Protection Against Data Attacks** The main goal of FC's design is to thwart attacks that depend on reading or modifying sensitive data in the program's memory. The attacker's gateway to the program's memory is through unchecked memory boundaries in the unsafe external library. Once exploited, the attacker can potentially search through all accessible memory to find the target data. The attacker's task is easier when the calling frame from the trusted region can be identified (for accessing stack data), and when a reference to memory allocated on heap is passed to the unsafe function.

First, identifying the calling frame enables the attacker to access data allocated in the trusted region. This data is completely protected by FC, except for any data that is on the calling frame's page. Provided that the programmer does not violate FC's intended usage (by not declaring sensitive data objects in the calling frame), the attacker cannot manipulate or read data on the stack. Second, when the unsafe function has a pointer to a memory allocated in the heap, the attacker can identify the region of the memory that is likely to contain sensitive data. As heap is allocated on consecutive memory addresses, the attacker can attempt to access pages that may belong to the heap. FC protects memory in the heap as requested by the programmer. Thus, all memory pages that were designated to move to secure compartments are not accessible by the attacker, when executing in isolation mode. One limitation of FC is that there may be heap memory allocated for libraries that are not visible to the programmer. It is also important to note that any data in memory that is used as indirect jump location or memory reference is potentially sensitive; if such references are exposed to the adversary, they may be corrupted to allow jumps that bypass FC protections or to copy sensitive data into locations that are not protected in a future unsafe call.

**Bypassing FC's protection** Bypassing FC involves issuing calls to mprotect with a list of addresses to be set to PROT_READ OR PROT_WRITE. First, the attacker is required to identify which memory pages are of interest. Second, a separate call to mprotect is needed for each memory page. An alter-

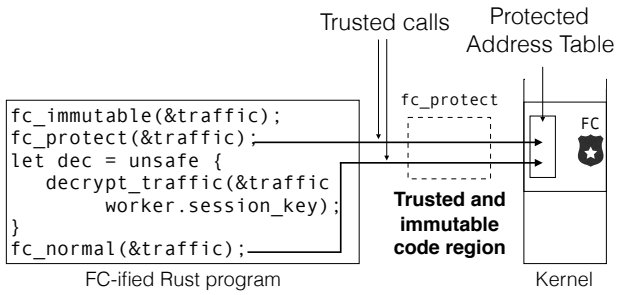


Figure 2: The trusted region is an address (in the code of FC's user space library) to which calls from `mprotect` must return to, which is ensured by FC's kernel module. The kernel module traps and monitor all calls to `mprotect` and maintains a list of page addresses in the protected address table for each thread.

native is that the attacker brute-forces the range of all virtual memory addresses and sets the protection for all pages to `PROT_READ OR PROT_WRITE`.

FC's primary line of defense is the kernel-level discretionary access control based on code regions. As explained in Section 2.4, FC's kernel module only allows `mprotect` to succeed on memory pages that are not in the list of the task's secure compartments. Also, FC will not allow such calls to succeed if the current task's instruction pointer does not indicate the address of FC's library function in the trusted region, which renders the attack unfeasible. There is, however, a possibility of launching a return-oriented programming attack by chaining a set of gadgets within the trusted region to call `mprotect` and trick the kernel to releasing secure compartments. The limitation of this attack is that the only possibility of an execution path is to re-execute the calling trusted function to first release the secure compartments and then make a call to the unsafe library function allowing the attacker to continue execution within the unsafe function. Such an attack is not possible as calls to FC should always create the secure compartments first and then release them, calling the unsafe function in between. Repeating this execution only trigger's FC's kernel module to be cautious of the process and terminate it as this will involve multiple consequent calls to create secure compartments.

# 3 Case Studies

FC assumes a knowledgeable programmer who uses its core functions to perform the necessary protections. In general, FC's protections could be highly automated, but our current implemention only automates stack protection. As automating heap protection involves many low-level details including implementing a custom memory allocator, we leave it for future work. We present here our experience in FC-ifying a server based on Rust's `openssl` crate (`https://github.com/sfackler/rust-openssl`); we also FC-ified a similar Rust TLS server based on the `hyper` crate (`https://github.com/ctz/hyper-rustls/`) but since the experience and results were similar we only describe `openssl` in detail here. We show performance results for both in Section 4.

**FC-ifying openssl** The `openssl` crate relies heavily on the foreign function interface to fully implement a TLS server based on the functionality provided in the original `openssl` library implemented in C. Similar to `hyper`, the server using `openssl` can be FC-ified for protecting the acceptor object when handling a client.

An acceptor object contains the server's credentials. The acceptor is stored in a `Mutex`, which is in turn managed by a heap memory under an instance of `Arc`. The objective of the FC-ified `openssl` implementation is to secure the heap page that contains the server's private key. For each incoming connection, at the time the client is served in a dedicated thread, the program no longer needs access to the acceptor. Thus, after the thread receives a pointer to the acceptor (`acceptor.clone`), locks the `Mutex` (`acceptor.lock`), and finally accepts the connection (`acceptor.accept(stream)`), a call to `fc_protect(acceptor_addr)` will result in a secure compartment for the heap page on which the acceptor resides. This secures the private key which will be inaccessible to the unsafe code. The call to `fc_auto_stack` and the corresponding call to `fc_auto_stack_reverse` will automatically protect the stack pages, and the call to `kernel_disable!` and `kernel_enable!` (`kernel_disable!` and `kernel_enable!` are macros to facilitate using FC's interface with the kernel `fc_protect` as described in Section 2.4) that re-

strict access to `sys_mprotect`. The call to a helper macro, `stack_padding!` allocates auxiliary memory on stack, ensuring that the data allocated prior to the call remains on a separate virtual memory page.

```
fn openssl_listener() {
   let acceptor = load_ssl_acceptor();
   stack_padding!();
   let acceptor = Arc::new(Mutex::new(acceptor));
   let acceptor_ptr = acceptor.clone();
   let acceptor_addr = memory_page_addr!(*acceptor_ptr);
   let listener = TcpListener::bind(SERVER_IP).unwrap();
   for stream in listener.incoming() {
      match stream {
          Ok(stream) => {
          let acceptor = acceptor.clone();
          let child = thread::spawn(move || {
          let acceptor = acceptor.lock().unwrap();
          let mut stream = acceptor.accept(stream).unwrap();
          fc_private_u(acceptor_addr);
          fc_auto_stack();
          kernel_disable!(acceptor_addr as usize);
          handle_client(&mut stream);
          kernel_enable!();
          fc_auto_stack_reverse();
          fc_normal_u(acceptor_addr);
          });
          child.join().unwrap();
          }
          Err(_) => { /* connection failed */ }
     }
     break;
   }
}
```

Figure 3: Starts a FC-ified server, while protecting the server's private key after a new client connection is established. This function is thread-safe and does not cause segmentation fault for accessing the protected acceptor.

**Attacks Thwarted** FC can be used in various ways depending on the needs. The main goal is to prevent exposing the server's credentials and the program's state (mainly on stack) when interacting with untrustworthy clients and executing unsafe code. For complex servers, the handling function may include references to commodity unsafe libraries that can jeopardize the security guarantees of the program, causing arbitrary data leak. FC guarantees that the explicitly protected data remains unreachable when such an attack occurs. FC releases the secure compartments, when called through `fc_normal`, after completing a client processing. This ensures that the server's credentials are only accessible when the server can make sure a malicious client is no longer connected. A slight limitation of this implementation is when serving clients for long periods, which can cause long delays for concurrently connecting clients (as the acceptor object is locked while serving the client). One can prevent this limitation by generating multiple copies of the acceptor with a pool of worker threads model. Each thread would use FC to protect its own copy.

# 4 Performance

This section reports on our experiments to evaluate the run-time cost of FC, first showing results on a set of microbenchmarks, and then reporting application-level performance measurements on the `openssl` and `hyper` case study applications from Section 3.

## 4.1 Microbenchmarks

For the microbenchmarks, our goal is to understand the cost of each of the operations involved in using FC. We use Rust's benchmarking interface to measure the cost for creating secure compartments, launching and processing a plain `openssl` request, and launching and processing a FC-ified `openssl` request, as implemented in Figure 3.

Table 1 reports the cost of using FC's main interfaces. The *Base* benchmark is an empty closure for measuring the benchmarking interface's cost. The *Padding* benchmark is a closure which isolates two memory pages using $512 \times 64$ byte arrays. *Creating Secure Compartment* is a closure that introduces padding, modifies the isolated page's permissions, communicates the page address to the kernel module, and reverses page permissions and requests the kernel to re-enable access to the specified page address.

The last two rows in Table 1 compare the time require to launch an `openssl`-based HTTP server and processing a single client using a plain Rust implementation and a FC-enabled implementation. The results shown are the average over multiples of multi-iteration tests, in which some of the tests did not distinguish the difference at all. FC's cost is negligible relative to the overall cost of `openssl`. The cost of FC is slightly more noticeable in the system benchmarks presented next. We performed the tests on a vmware Workstation virtual machine on a local disk with Ubuntu 15 as the host system, configured to use two of the available four cores and two GBs of memory.

| Experiment | Time (ms) |
|---|---|
| Base | 0.000024 |
| Padding | 0.0054 |
| Creating Secure Compartment | 0.0105 |
| `openssl` server | 14.342 |
| FC-ified `openssl` server | 14.385 (0.29% increase) |

Table 1: Microbenchmark results.

### 4.2 System Benchmarks

In the second and third settings, we test the performance of unsafe operations in TLS-based HTTP servers, which either use unsafe operations for invoking `openssl` operations or make calls to Rust's `ring` library, which in turn makes unsafe calls to cryptographic functions. We perform a time comparison between a plain HTTP server that doesn't use FC and an FC-ified HTTP server. FC's latency includes the small overhead of the kernel module, which is disabled in experiments without FC.

**HTTP server** Measuring the throughput of a `openssl`-based and a `hyper`-based HTTP server required implementing benchmarking tools for a precise measuring of the contribution of each thread in processing the requests. In the `openssl`-based server, for each request three secure compartments are created prior to handling the process and one secure compartment is created for handling the process. We measure the throughput by running the test for 60 seconds while automatically sending HTTP requests in intervals of 10 milliseconds, collecting the number of successfully processed requests at the end of the experiment. In each iteration, all four secure compartments are created and destroyed. FC maintains modest overhead even when handling 128 simultaneous request threads, reaching a pick decrease of 5% in the number of requests processed in 60 seconds. FC's overhead was noticeable when every request involved 50 calls to `ring` for computing a file's digest, with an average decrease of 13.69% processed requests occurred. Finally, when using a duration of 30–100 seconds with a fixed number of 16 simultaneous requests processed in intervals of 10 milliseconds, the average decrease in the number of requests processed was 8.30% (Figure 4.

In the `hyper`-based server, the throughput was measured similar to the benchmark of Figure 4 in which a growing number of threads send simultaneous requests. All requests were implemented as echo requests. In the FC-enabled version, during the time the client is served, the server's private key is totally isolated and is unusable. To serve multiple clients, for each request, the private key is cloned and once

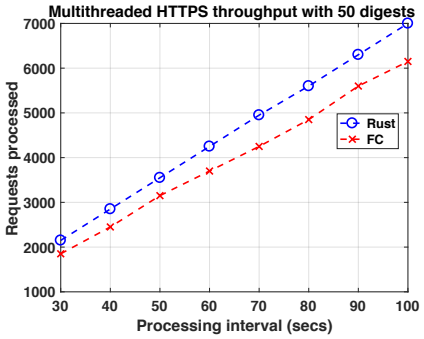


Figure 4: Throughput of an `openssl` HTTP in 30–100 seconds. Each request has 50 calls to `ring`'s digest. In the FC-enabled server, each digest iteration creates and releases a secure compartment.

the shared key is established, the cloned private key will be kept in a separate secure compartment. The result of the experiment is in Figure 5, showing an average decrease of 1.38% in the number of requests processed.

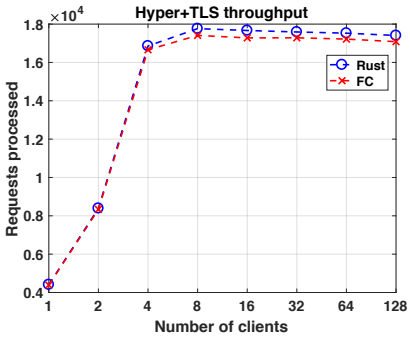


Figure 5: Throughput of a Hyper server using Rust TLS in 60 seconds. Requests are sent locally using $2^k$ threads for $k \in [0,7]$.

## 5 Related Work

The goal of isolating code components (also referred to as application compartmentalization) is a long-standing goal of our community and has been the focus of extensive research in operating systems, programming languages, and architecture. This is the first work to focus on isolating unsafe code within a single address space for Rust programs.

In terms of our memory model and isolation techniques, previous works most similar to FC include Native Client [30], Codejail [29], HideM [9], SeCage [15], Shred [7], lwC [14], and SpaceJMP [12]. We discuss these next, followed by a brief account of classical

work on software fault isolation and the principle of least privilege.

**Sandboxing Libraries** Native Client [30] provided memory sandboxing for libraries, allowing limited interaction with the trusted program using remote procedure calls. Fidelis Charm and Native Client (NaCI) share the main objective of limiting external libraries, however, differ in approach and applicability. In contrast with NaCI's approach for loading libraries in limited containers, FC contains the trusted memory when interacting with unsafe libraries. Codejail [29] proposed an enhancement of the idea by sandboxing a library by disallowing write access to the program's data, making the sensitive memory read-only. The program would selectively allow write access to its data, when a tight interaction with the library is needed. Codejail shares FC's goals in (i) proposing a secure memory sharing model that does not require modifications to the library and (ii) supporting tight and limited interactions with an untrustworthy library. Aside from focusing on Rust, FC is distinguished in that the memory of the trusted program is the sandbox, instead of the library, and unless a data object must be shared with the library, the entire memory allocated either on stack or heap of the trusted Rust code is inaccessible to the library. This key difference in memory sandboxing model overcomes Codejail's limitation of libraries within a specific memory regions.

Various techniques have been developed for confining and limiting processes or groups of processes (e.g., [19, 8, 1, 13, 10, 23, 18]), aiming for isolating vulnerable software from critical system resources. In the design of FC, a sandbox would avoid incuring unnecessary latency as our goal is to isolate code at the fine level of (often frequent) unsafe calls. That said, sandboxing using Intel enclaves [13] can provide improved security for FC's kernel module.

**Memory partitioning** Shared memory among distrustful threads within a single process is addressed by Arbiter [27], which proposes to use memory permission bits to protect a thread's data from another. Arbiter uses to a policy manager that understands programmer-annotated privileges and enforces them. FC uses memory page padding to separate and isolate data objects based on a simple binary permission system (immutable or private). The kernel component in FC only enforces FC's integrity and does not need to enforce application-level policies. HideM [9] takes a radical approach by using split-TCB to show different contents of a page for different CPU operations, hiding data when a specified code region should not have access to it. In contrast, FC temporarily hides the actual memory page and isolates the data when an explicit interaction with an untrustworthy library function occurs. SeCage [15] is similar to HideM in providing different views of the memory according to separated privileges. Using hardware virtualization, SeCage targets a strong adversary model in which the operating system does not need to be trusted.

Shred [7], light-weight context (lwC) [14], and SpaceJMP [12] are new methods for splitting the virtual address space into multiple distinct sections aiming for isolating untrustworthy code. Shreds and lwC introduce abstractions similar to threads. To protect data across a program, Shred provides a set of programming interfaces to request creating and moving data into separate Shreds. At runtime, Shreds are implemented using Intel's memory protection keys. A lwC, in contrast, does not work with memory permissions directly but creates separate address spaces, when the programmer requests that using the programming interfaces. SpaceJMP is similar to lwCs in using multiple virtual address space, differing from lwCs in that SpaceJMP enables memory sharing across multiple processes. While FC is similar to these in that it provides an interface for isolating memory regions, it does not propose a new operating system model and does not require switches between address spaces.

Other related work but farther away have provided interesting contributions for application compartmentalization, mainly with narrow and specific applicability. For example, Mimosa specifically targets cryptographic keys and uses hardware transactional memory to ensure that no process, other than Mimosa, can access the keys. Mimosa uses encryption to hide the keys, when the system is idle. Although FC and Mimosa agree on a high level goal of hiding data objects in memory, FC differs in that it uses memory isolation without the need for hardware transactional memory and works with arbitrary data. DataShield [5] protects data in C++ programs by disallowing pointer dereferencing based on programmer annotations. Song et al. propose a data-flow integrity approach to infer and enforce correct flow of sensitive data in kernel space [24]. Lastly, SOAAP [11] is a reasoning tool for assisting programmers in using application compartmentalization to avoid security and correctness errors. We envision a similar tool for our future work to support programmers with FC and automate the task of locating unsafe regions and the data objects that must be isolated.

**Software fault isolation** has consistently received attention during the past decades. Simple solutions such as placing the faulty code, or the unsafe code, in a separate address space seem viable, al-

though for merely a call to an unsafe function, the unnecessary context switches are too much of a burden. Wahbe et al. pioneered the design of logically separated fault domains within a single address space [26], which was followed by an effort to isolate addressability from accessibility in Opal, a single-address-space 64-bit architecture [6]. The work in [26] described a model in which fault domains are separated based on the code region through restricting the execution of one to jump to another. This model inspired our kernel-centric code region discrimination design, with a fundamental difference; FC would not require an RPC interface to enable cross-domain interactions. In fact, FC imposes no particular paradigm on the program and automates code region separation through a kernel extension.

**The principle of least privilege** [22] is the theme of a number of previous work that promised least privilege isolation. With resource containers [2] separating access control from execution, isolation progressed further towards decoupling scheduling from security requirements, which were a fundamental design issue with process management in modern monolithic kernels. The work by Provos et. al set a clear goal: privilege separation within an application forbids programming errors in the lower privileged code from abusing higher privileged code [20]. However, privilege separation was a return back to the use of processes as basic blocks for isolation, reusing UNIX per-process protection domains. Privtrans [4] automated privilege separation using programmer annotations, partitioning a program into a monitor and a slave program, continuing the efforts of isolation at the process level. Sthreads in Wedge [3] introduced default-deny compartments within a single monolithic program, spawning new threads, not entire processes, for isolating parts of the program. Sthreads enjoy programmer tagged memory access rights, which are enforced at runtime. Programmer annotated privileges were also introduced for isolating kernel modules [17] from core kernel services to prevent privilege escalation. Similarly, Trellis [16] allows code-annotated privileges (mainly for memory allocations), which are enforced by the kernel at runtime. CHERI [28], a hardware extension relying on a capability co-processor, supports compartmentalization for in-address-space memory isolation targeting the C language.

Finally, RustBelt [21] develops a subet of Rust, namely $\lambda_{Rust}$, and uses to prove the safety of Rust programs. An important result provided by RustBelt is verification of safety while using unsafe interactions with linked libraries. Rustbelt verifies a $\lambda_{Rust}$ program with unsafe code has safely encapsulated the externally linked library using within Rust wrappers.

# 6 Conclusion

Rust provides strong memory guarantees using zero-cost abstractions, but any non-trivial Rust program today includes unsafe code and most fall back on using libraries in unsafe languages. A long-range goal should be to eliminate the need for any unsafe code—developing native Rust libraries when possible, and when arbitrary memory operations are needed using more powerful formal methods to prove the safety of code that cannot be proven safe by Rust's compiler. A practical path to dramatic improvements in program safety and reliability, however, requires combining safe and unsafe code. Incorporating any unsafe code into a Rust program, however, abandons all of the safety guarantees. Fidelius Charm provides a step towards safe incorporation of unsafe code by isolating sensitive data in memory from the unsafe code. We achieved a high level of isolation, without requiring any compiler changes or complex abstractions, and in a way that can be applied to any Rust program when interacting with any unsafe library function.

# References


[1] H. M. J. Almohri, D. Yao, and D. G. Kafura. Process authentication for high system assurance. *IEEE Trans. Dependable Secur. Comput.*, 11(2):168–180, Mar. 2014.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 45–58, 1999.

[3] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, 2008.

[4] D. Brumley and D. X. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, 2004.

[5] S. A. Carr and M. Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, ASIA

CCS '17, pages 193–204, New York, NY, USA, 2017. ACM.

[6] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12:271–307, 1994.

[7] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained execution units with private memory. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, May 2016.

[8] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[9] J. Gionta, W. Enck, and P. Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 325–336, New York, NY, USA, 2015. ACM.

[10] A. Gollamudi and S. Chong. Automatic enforcement of expressive security policies using enclaves. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 494–513, New York, NY, USA, 2016. ACM.

[11] K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with soaap. In *ACM Conference on Computer and Communications Security*, 2015.

[12] I. E. Hajj, A. Merritt, G. Zellweger, D. S. Milojicic, R. Achermann, P. Faraboschi, W. mei W. Hwu, T. Roscoe, and K. Schwan. Spacejmp: Programming with multiple virtual address spaces. In *ASPLOS*, 2016.

[13] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. Sgxbounds: Memory safety for shielded execution. In *EuroSys*, 2017.

[14] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-weight contexts: An os abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 49–64, Berkeley, CA, USA, 2016. USENIX Association.

[15] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *ACM Conference on Computer and Communications Security*, 2015.

[16] A. Mambretti, K. Onarlioglu, C. Mulliner, W. Robertson, E. Kirda, F. Maggi, and S. Zanero. Trellis: Privilege separation for multi-user applications made easy. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 437–456. Springer, 2016.

[17] 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 *SOSP*, 2011.

[18] E. Pattuk, M. Kantarcioglu, Z. Lin, and H. Ulusoy. Preventing cryptographic key leakage in cloud virtual machines. In *USENIX Security Symposium*, 2014.

[19] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *USENIX Security Symposium*, 2002.

[20] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, 2003.

[21] R. K. D. D. Ralf Jung, Jacques-Henri Jourdan. RustBelt: Securing the foundations of the rust programming language. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2018, New York, NY, USA, 2018. ACM.

[22] J. H. Saltier and M. P. Schroeder. The protection of information in computer systems. *IEEE CSIT Newsletter*, 3(12):19–19, Dec 1975.

[23] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. K. Rajamani, S. A. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In *PLDI*, 2016.

[24] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[25] L. Szekeres, M. Payer, T. Wei, and D. X. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[26] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[27] J. Wang, X. Xiong, and P. Liu. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. In *USENIX Annual Technical Conference*, 2015.

[28] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. D. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[29] Y. Wu, S. Sathyanarayan, R. H. C. Yap, and Z. Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *ESORICS*, 2012.

[30] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.