UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Fine-grained Library Sandboxing for Rust Ecosystem

Permalink
https://escholarship.org/uc/item/5kq7s1jj

Author
Zhou, Tianyang

Publication Date
2023

Peer reviewed|Thesis/dissertation

eScholarship.org
Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Fine-grained Library Sandboxing for Rust Ecosystem

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Electrical Engineering (Computer Engineering)

by

Tianyang Zhou

Committee in charge:

Professor Deian Stefan, Chair
Professor Xinyu Zhang, Co-Chair
Professor Truong Nguyen

2023

Copyright

Tianyang Zhou, 2023

All rights reserved.

The Thesis of Tianyang Zhou is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego
2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to all who have provided their support and assistance throughout this research journey.

First and foremost, I owe a great deal of gratitude to my advisor, Prof. Deian Stefan. Throughout my master's study, his academic guidance has been pivotal. His readiness to allow me space for research exploration, coupled with his essential assistance during my Ph.D. application process, is something I deeply appreciate.

Moreover, I would like to extend my deepest thanks to Prof. Shravan Narayan and Prof. Caleb Stanford, who have been my research mentors, consistently offering their patience and detailed guidance. Their capacity to provide specific suggestions and a clear roadmap has been invaluable in developing my research capabilities.

Also, I want to express my gratitude to Dr. Tal Garfinkel. His unique insights have often illuminated my path, helping me navigate through the complex journey of my research work. Thanks are also due to Prof. Ranjit Jhala and other members of the PLSysSec lab, who have been tremendously helpful in providing insightful feedback and suggestions on my work.

In addition, I extend my heartfelt thanks to my defense committee members, Prof. Xinyu Zhang and Prof. Truong Nguyen, for their for their support and valuable suggestions on my thesis defense.

Lastly, I would like to express my sincere appreciation to my parents and friends. Their financial and emotional support is always my source of strength and inspiration. Their support has made this journey less daunting and more rewarding.

The entire thesis, in full, is currently being prepared for submission for publication of the material. Zhou, Tianyang; Narayan, Shravan; Stanford, Caleb; Jhala, Ranjit; Garfinkel, Tal; Stefan, Deian. The thesis author was the primary investigator and author of this material.

ABSTRACT OF THE THESIS

Fine-grained Library Sandboxing for Rust Ecosystem

by

Tianyang Zhou

Master of Science in Electrical Engineering (Computer Engineering)

University of California San Diego, 2023

Professor Deian Stefan, Chair
Professor Xinyu Zhang, Co-Chair

Rust, a modern programming language prioritizing memory safety, has an ecosystem that is still under active development. Taking advantage of the Foreign Function Interface (FFI), Rust can directly leverage existing C libraries like libjpeg, libzstd, and libsqlite, eliminating the effort to re-implement these in Rust. However, while Rust's robust compiler ensures Rust code's memory safety, it cannot do the same for linked C libraries, potentially endangering the memory safety of the entire program if vulnerabilities exist.

To address this issue, we present RLBox-Rust, a fine-grained library sandboxing framework for Rust. RLBox-Rust employs sandboxing technology to isolate C libraries used in Rust, ensuring Software Fault Isolation. This guarantees memory safety even in the presence of vulnerabilities in C libraries. Building upon sandboxing, RLBox-Rust designs and implements a novel sandbox binding mechanism, enabling Rust developers to use sandboxed C libraries intuitively and securely. With Rust's robust type and macro systems, we've established a set of memory safety-checking mechanisms that employ static (compile-time) and dynamic (run-time) checks to maximize the assurance of type and data integrity. We utilized WebAssembly (Wasm) technology for sandboxing, ensuring high performance and cross-platform compatibility of sandboxed C libraries as well as strict runtime environment isolation and control-flow integrity. We evaluate the RLBox-Rust framework through different use cases, and our experiments demonstrate that RLBox-Rust can ensure memory safety with an acceptable performance overhead, enabling Rust developers to safely use existing C libraries with minimized migration effort.

Introduction

Rust represents an innovative programming language endowed with robust mechanisms that ensure memory safety. It incorporates a comprehensive type system alongside an efficient ownership structure that enforces borrow checking, thereby guaranteeing memory safety at both the compile-time and run-time phases. Furthermore, Rust integrates a lifetime check mechanism. This mechanism not only guarantees memory safety that prevents use-after-free vulnerabilities but also enables zero-overhead memory management and garbage collection, creating a balance between safety and efficiency. Moreover, Rust is a versatile language with a low-level aspect, which enables compilation into native code. Concurrently, it has a high-level optimization capability that allows the Rust code to operate at speeds comparable to those of C/C++. Owing to these characteristics, Rust stands as a promising language for the creation of safe and efficient software. These features make it a potentially suitable substitute for C/C++ in the development of critical systems. Under this circumstance, many developers have begun to rewrite a large number of legacy C/C++ codes in Rust, like the `sudo-rs`, `Servo` and `GNU-coreutils` projects. All of these projects are aimed at replacing the original C/C++ code with Rust code, thereby providing a more secure and efficient alternative to the original programs.

However, the practical implementation of this strategy may not always be feasible due to the significant investment of time and effort required from many developers to rewrite the original C/C++ code in Rust. This is especially challenging for well-established projects with a large codebase, where converting every line of code to Rust is far from a trivial task and may require an extended period before the program is mature enough to be accepted by the public. Fortunately, Rust's capacity to compile programs into native code, give it the ability to link to existing C/C++ code via the Foreign Function Interface (FFI), which provides a practical solution. This capability allows code written in Rust to directly interface with legacy C/C++ code, thus enabling developers to reuse existing C/C++ code within Rust and gradually transition the codebase from C/C++ to Rust. This methodology has been widely embraced by numerous projects. For instance, the Firefox web browser has commenced using Rust to rewrite some of its components, linking them to the original C/C++ code through FFI. Similarly, the Linux operating system kernel has employed Rust for a portion of its kernel and has established an interface enabling Rust developers to write kernel modules in Rust. Moreover, the Windows operating system has announced that they have begun utilizing Rust to write segments of their code.

Benefits from the FFI feature, Rust is capable of directly utilizing existing C/C++ libraries, including widely used C libraries such as `libjpeg-turbo`, `libzstd`, and `libsqlite` . These C-based libraries have had extensive usage in numerous projects, not just those written in C/C++, but also those in other languages like Python, Rust and other languages that have an interface to allow code to interact with libraries written in compiled languages like C. This phenomenon is primarily attributed to the maturity and stability of these libraries, which also have been extensively optimized for efficiency. At the same time, These libraries have large code bases with a wide range of testing and debugging processes over a long period of time. As a consequence, transcribing these libraries into Rust would require plenty of time and effort, and the performance of rewritten Rust code may still not match the efficiency of the original C code. Thus, employing these C libraries in Rust through the FFI has presented as the most practical approach currently available.

While Rust's robust compiler guarantees the memory safety of its own code, it is unable to extend the same safeguard to linked C libraries. This shortcoming potentially breaks the value of Rust's memory safety assurances. If vulnerabilities are present in the linked C libraries, the memory safety of the entire program may be exploited due to the shared memory space between the compiled Rust code and the C libraries' code and data. This problem will become particularly severe in critical systems because vulnerabilities in widely used C libraries are not uncommon. An example is CVE-2019-11922[20], which is a buffer overflow vulnerability in the `libzstd` library. If such a vulnerability is exploited, it may lead to memory corruption and the attacker may be able to access the memory space of the entire program, including the data space in Rust code. This vulnerability is not a singular instance; numerous similar vulnerabilities exist in other C libraries. The most considerable drawback of C/C++ is the ease with which developers can always unconsciously introduce memory safety issues in C/C++-based programs, which is largely due to the absence of built-in mechanisms in C to ensure memory safety. Consequently, these vulnerabilities in C libraries introduce a high risk to the memory safety of the entire program, no matter whether the rest of the program is composed of safe Rust code or not. In this scenario that C libraries are exploited, the assurances offered by Rust's unique mechanisms to ensure memory safety are effectively nullified. Therefore, it is essential to provide a new approach to ensure the memory safety of the Rust program when linked to C libraries.

In this thesis, we introduce RLBox-Rust, a novel fine-grained library sandboxing framework for Rust. Inspired by the concept of Software Fault Isolation (SFI), RLBox-Rust does not aim to eliminate vulnerabilities in C libraries. Instead, it seeks to isolate these libraries from Rust code, thus preventing vulnerabilities in C libraries from being exploited to compromise the memory safety of the rest of the program, including other sandboxes or the host application based on Rust.

In our implementation, we utilize the WebAssembly virtual machine as the sandboxing environment for C libraries. The WebAssembly virtual machine has been widely employed in web browsers for environment isolation. It has several characteristics that make it a suitable choice for our sandboxing environment, including its compatibility on different platforms and different CPU architectures, its high performance, and its ability to provide environment isolation as well as the memory safety and control-flow integrity of the sandboxed environment. In the implementation of RLBox-Rust, we choose to use the wasm2c tool to convert WebAssembly bytecode into equivalent C code, which can be compiled into native code and linked to the Rust code through FFI. Through this approach, we can leverage the WebAssembly virtual machine by simply linking the converted C code to the Rust code, thereby enabling the Rust code to interact with the sandboxed environment.

Leveraging Rust's robust type system, RLBox-Rust offers Rust Bindgen, based on the `bindgen` crate. With the bindings generated from C header files via Rust Bindgen, RLBox-Rust has the ability to acquire type information for all APIs and data structures in C libraries. Utilizing this type information, RLBox-Rust subsequently generates a sandboxing wrapper for each struct, enum, and function present in C libraries, as well as the information for static checking. We also utilize our RLBox Bindgen to generate the bindings for the wasm2c-generated C code, which enables RLBox-Rust to interact with the sandboxed environment, the WebAssembly module.

This mechanism enables RLBox-Rust with the capacity to provide an intuitive and user-friendly Value-accessing API for developers to interact with the virtual environment through RLBox-Rust. The RLBox-Rust provides the `Tainted<T>` struct, which indicates that the value `T` is in the sandboxed environment. Around this struct, RLBox-Rust provides a set of APIs to access the value in the sandboxed environment, including exchanging data between the sandboxed environment and the host Rust application or invoking functions in the sandboxed environment.

RLBox-Rust ensures the memory safety of the host Rust application by deploying a set of rules that enforce both static and dynamic checking during the interaction between the host and RLBox-Rust framework via the Value-accessing API.

The static checking is processed by RLBox-Rust's code generator, which generates type-checking code based on the type information from the C libraries, as derived from Rust Bindgen. This type-checking code is only executed at compile time, allowing the compiler to verify the consistency between the type of the value passed to the APIs and the type of the value defined in the C libraries. Following this verification, the type-checking code will be optimized by the compiler, resulting in zero runtime overhead for all static checking.

Conversely, dynamic checking is performed during program execution. It is responsible for ensuring that access to the sandbox environment stays confined within the memory space of the sandboxed library. Additionally, dynamic checking verifies when the value exchanged from the sandboxed environment to make sure it satisfies domain safety requirements.

In summary, static checking maintains type integrity, while dynamic checking ensures memory integrity. Both of them are essential to ensure the memory safety of the host Rust application.

The ”fine-grained” sandboxing mechanism in RLBox-Rust implies that each sandboxed instance is isolated from the rest of the program, including both the host Rust application and other sandboxed instances. From a developer’s perspective, this feature allows them to choose to create a corresponding sandboxed instance for each C library, or even multiple multiple sandboxed instances could be created for a single C library to isolated different stages of library usage, such as isolating the compression and decompression stages of the `libzstd` library. So that the potential vulnerabilities in the compression stage will not affect the decompression stage. This flexibility provides developers with the ability to adjust the granularity of sandboxing based on their specific requirements, as well as the trade-off between security and performance.

We evaluate RLBox-Rust by applying it to two distinct usage scenarios for Rust applications to interact with C libraries. In the first scenario, the Rust application will directly invoke the raw APIs in C libraries through C bindings from Rust Bindgen. We pick the benchmark suite Image Compression Benchmark (ICB)[25] as the testbed for this scenario and write a Rust application to invoke the `libjpeg-turbo` library through direct C bindings to encode the test images into different quality, then decode them. Then we rewrite the application to invoke the `libjpeg-turbo` library through RLBox-Rust. We compare the performance overhead of the two versions of the application. The result shows that the RLBox-Rust will introduce a performance overhead of 29.75% on average for encoding and 32.34% for decoding.

In the second scenario, we utilize a real-world Rust wrapper of a C library as the testbed. The Rust wrapper we evaluated is the `zstd-rs` crate, a wrapper of the `libzstd` library. We rewrite the `zstd-rs` crate to interact with the `libzstd` library via RLBox-Rust. Subsequently, we evaluate the performance overhead between the original `zstd-rs` crate and the RLBox-

Rust version, using the benchmark suite provided by the `zstd-rs` crate. This suite includes compression and decompression performed on the entire Silesia compression corpus[1], which is a collection of files used to evaluate the performance of data compression. Given that the original `libzstd` is compiled with the SIMD-enabled, we also compile the RLBox-Rust version of `libzstd` with the SIMD-enabled via the SIMD everywhere (SIMDe) feature, in an attempt to minimize performance penalties. The results demonstrate that the RLBox-Rust version of `zstd-rs` introduces an average performance overhead of 41.25% for compression and 36.91% for decompression.

In practical applications, the performance overhead introduced by RLBox-Rust is likely to be less significant than the results obtained in our benchmarks. This is due to the design of the benchmarks used in our evaluation, which are specifically constructed to stress the performance of the library APIs, as the whole benchmarks are only interacting with the library APIs without any other operations. In contrast, invocations of libraries within real-world applications typically comprise only a fraction of the entire application. Thus, the performance overhead introduced by RLBox-Rust in a practical setting is likely to be less pronounced than what our evaluation results might suggest.

The contributions of this thesis are as follows:

- **Propose a new approach to isolate the C libraries in Rust applications:** We propose a new approach that mitigates the memory safety vulnerabilities in the Rust applications that interact with unsafe C libraries. Instead of rewriting the C libraries in a memory-safe language, we propose to isolate the C libraries in a sandboxed environment, and then safely interact with them through our sandboxing wrapper. In our implementation, we use WebAssembly as the sandboxing environment because of its portability and security features. This approach allows developers to reuse the existing C libraries without the rewriting effort, which is more practical in real-world applications.

- **Design and Implement RLBox-Rust:** We design and implement RLBox-Rust, a fine-grained sandboxing framework for Rust applications to safely interact with C libraries. RLBox-Rust provides a user-friendly Value-accessing APIs, which allows developers to reuse the existing Rust binding wrappers of C libraries with minimal effort while migrating the original wrappers to RLBox-Rust. RLBox-Rust also provides a set of static and dynamic checking rules to enforce memory safety during the interaction between the host Rust application and the sandboxed environment.

- **Evaluate the RLBox-Rust on two real-world usage scenarios**: We port two kinds of Rust applications to RLBox-Rust, one is the Rust application that directly invokes the raw APIs and the other is the Rust wrapper of a C library. We evaluate the performance overhead of the RLBox-Rust comparing to the original version of the applications. The results show that RLBox-Rust introduces an overhead less than 50% for both scenarios, which is acceptable for most real-world applications.

The rest of this thesis is organized as follows. In Chapter 1, we will discuss the background information of this thesis, including the safety mechanisms in Rust to prevent memory safety vulnerabilities (Section 1.1) , the vulnerabilities that Rust applications may encounter when interacting with C libraries through direct C bindings (Section 1.2), and the existing approaches to mitigate these vulnerabilities (Section 1.5).

In Chapter 2, we will introduce the motivation and overview of our approach, the RLBox-Rust framework (Section 2.1. We will then detail the design and implementation of RLBox-Rust (Section 2.4), as well as the migration process of the existing Rust binding to RLBox-Rust to demonstrate that RLBox-Rust is easy to migrate from the existing Rust applications that interact with C libraries (Section 2.3). Finally, we will discuss the evaluation of RLBox-Rust and demonstrate the performance overhead introduced by RLBox-Rust (Section 2.5).

In Chapter 3, we will conclude this thesis and discuss future work. We will concretely discuss the functionality that RLBox-Rust can provide and future work to improve the usability and performance of RLBox-Rust.

Chapter 1

Backgroud of Rust and unsafe libraries in Rust ecosystem

In this chapter, we explore the background of the thesis, specifically focusing on its memory safety module in Rust and the implications of embedded unsafe libraries in Rust, which may lead to potential vulnerabilities. These vulnerabilities could impact the security and reliability of Rust applications, challenging the assumed safety within the ecosystem. We will detailly discuss several kinds of vulnerabilities in the real world that may be caused by embedded unsafe libraries in the Rust ecosystem, which is the motivation of this thesis. Then we will discuss two kinds of efforts that aim to mitigate these vulnerabilities in C libraries embedded in Rust, as well as discuss the shortcomings of these efforts.

1.1 Rust memory safety module

Rust is a novel programming language that aims to provide memory safety and thread safety in the system programming area. It is designed to provide a safe alternative to C and C++ by enforcing memory safety and thread safety both at compile-time and runtime. To achieve this goal, Rust utilizes a unique memory management model, which is called ownership and borrowing.

The ownership and borrowing model defines that each object in Rust will have a unique owner, and the owner can ”lend” the immutable or mutable references of the object to other objects. The immutable reference is that the object can only be read by the object that borrows the immutable reference, and the mutable reference is that the object can be read and written by the borrowing object. An object can be borrowed by multiple objects immutably at the same time, but the owner can only lend the mutable reference of the object to one object at a time. This model prevents the data race problem caused by multiple objects writing the same object at the same time, or the dirty read problem caused by one object reading the object while another object is writing the object. It restricts the object to be read and written by multiple objects at the same time, which is the key to ensuring thread safety.

To enable the compiler to check whether the object can be borrowed by other objects immutably or mutably at compile-time, Rust introduces the concept of the lifetime, which is the scope of the value and all immutable and mutable references. A lifetime is determined by the scope of the variable that owns the value and references, where the scope can be automatically inferred by the compiler. When the lifetime of the value and references ends, the value and references will be dropped, and the memory occupied by the value will be automatically freed. This process is known as Resource Acquisition is Initialization (RAII).

Through the lifetime mechanism, the Rust compiler can statically infer whether the object can be borrowed by other objects immutably or mutably at compile-time and whether the object has a use-after-free problem. If the compiler detects that the Rust code breaks the ownership and borrowing rules, or the object has a usage that is out of its lifetime, the compiler will report an error and refuse to compile the code.

Rust's ownership and borrowing model, as well as concrete with the lifetime, enables the Rust compiler to statically check the memory safety and thread safety of the Rust code at compile-time. This feature minimizes the possibility of memory safety bugs in Rust programs without requiring extra runtime overhead, making Rust programs as fast as C or C++ programs but with more memory safety guarantees.

1.2 Unsafe libraries embedded in Rust ecosystem

In this section, we will introduce the reasons why unsafe libraries, mainly written in C, are embedded in the Rust ecosystem at high frequency. Then we will introduce different approaches to embedding unsafe libraries in the Rust ecosystem, which is through direct FFI bindings or wrapper libraries. For each way, we will also provide some examples of real-world Rust applications and analyze the advantages and disadvantages of each way.

1.2.1 Reasons why unsafe libraries are embedded in Rust ecosystem

Before Rust was introduced, C and C++ were the most popular programming language in the system programming area. However, these two languages are not designed to be memory safe, which means that programmers have to manually manage the memory so that this process is error-prone and can lead to memory safety bugs, such as buffer overflow, use-after-free, etc. For programs written in C or C++ in core system areas, like operating systems, web browsers, and system utilities, some of these memory safety bugs can be exploited by attackers to take control of the program and, at worst, the whole system. For example, CVE-2021-3156 is a heap buffer overflow vulnerability in the `sudo` program, which is a utility for Unix-like operating systems that allows users to run programs with the root privilege. This vulnerability can be exploited by attackers to overwrite data and pointers in the heap, and finally get the root privilege without correct permission identification[21]. Another example comes from Microsoft. In 2019, Microsoft Security Response Center (MSRC) present that 70% of the security vulnerabilities were fixed and assigned a CVE in Microsoft are caused by memory safety corruptions in C and C++ codebases[15].

Rust provides more compile-time and runtime safety guarantees that make the code written in Rust always considered more secure and reliable than the code written in C or C++. Due to these advantages, Rust has been widely adopted in developing new system programs and rewriting existing programs written in C or C++ such as `sudo-rs`, which is a reimplementation of the `sudo` program in Unix-like operating systems. Or `Servo`, which is a web browser engine written in Rust by Mozilla.

However, C and C++ languages have been used for decades in the system programming area and there have had a large number of software, and tools written in these two languages, including all of the operating systems, web browsers, and system core utilities. What's more, the most widely used libraries in all of the programming languages are written in C or C++, especially in C language. For example, the `libc` library is the standard C library that provides the most basic interfaces for all the C and C++ programs and system-based components to interact with the operating system. The `libjpeg` library is the most widely used library that is written in C for encoding and decoding JPEG images in several famous projects like all the web browsers and OpenCV. Completely rewriting all of these programs and libraries in Rust is neither realistic nor efficient.

Fortunately, Rust programs can be compiled into native code without requiring any runtime, allowing for seamless integration with existing C or C++ programs. This compatibility enables developers to incrementally rewrite components in Rust while maintaining the ability to link with existing C or C++ components during the linking process. The Foreign Function Interface (FFI) facilitates this linkage, enabling Rust programs to call APIs provided by C or C++ programs, or vice versa.

To bridge the API gap between Rust and C or C++, Rust offers the `bindgen` tool, which automatically generates Rust bindings from C header files and has limited support for C++ headers. These generated bindings constitute the Rust code translation of the header files, encompassing function APIs, type definitions, and constants. By utilizing `bindgen`, Rust applications can directly invoke the APIs and employ the types and constants defined in these C or C++ libraries without necessitating additional efforts. This tool significantly streamlines the process of integrating C or C++ libraries into Rust projects and highlights Rust's commitment to facilitating seamless interoperability within the broader programming ecosystem.

Leveraging FFI, several renowned software projects, such as Firefox and Linux kernel modules, have begun to incrementally rewrite portions of their codebase in Rust. By doing so, they retain the functionality of existing components while benefiting from the enhanced security guarantees offered by Rust. This approach has allowed these projects to improve the overall security and reliability of their programs, demonstrating the advantages of integrating Rust into systems programming.

This situation is also encountered by Rust applications that seek to utilize existing C or C++ libraries, especially libraries in C. While several such libraries have been re-implemented in Rust to provide a safer interface for Rust applications, a considerable number of libraries remain challenging to rewrite in Rust due to the complexity of their codebases or limited resources. Under these circumstances, Rust applications need to rely on FFI to leverage the functionality provided by external libraries. This approach is easy to implement and can provide the same functionality and performance as the original libraries written in C or C++ due to the high compatibility between Rust and C and the `bindgen` tool to automatically generate Rust bindings.

1.2.2 Different approaches to embedding external libraries in Rust ecosystem

In the Rust ecosystem, there are two main approaches to embedding external C libraries. The first approach is to directly use FFI bindings generated by `bindgen` to call the APIs provided by external libraries. The second approach, which is more common, is to use wrapper libraries that provide a more Rust-friendly interface for Rust applications to call the APIs provided by external libraries.

Numerous real-world Rust applications employ the first approach, which involves directly calling APIs generated by the `bindgen` tool. For instance, `libproc-rs` is a library designed for obtaining information about running processes on macOS and Linux systems. This library directly utilizes FFI bindings generated by `bindgen` to access APIs provided by the `libproc`library, which is written in C by Apple. Utilizing a C library in Rust applications is common when the library in question is not well-known or lacks a wrapper library, particularly for proprietary or closed-source libraries that are inaccessible for public contributions. In such cases, Rust applications must resort to using FFI bindings that directly invoke the APIs furnished by the C library. This strategy ensures that Rust applications can still benefit from the functionality provided by these specialized or proprietary libraries while maintaining compatibility and interoperability.

However, employing FFI bindings to directly invoke APIs provided by external libraries is not always the optimal approach, as it may give rise to several issues.

Firstly, although `bindgen` generates bindings capable of translating C APIs, types, and constants into Rust code, this translation may not always be Rust-friendly. The generated code might not be Rust's idiomatic style, as it is merely a direct translation of the original C code without any modifications or Rust-specific optimizations. Rust libraries are typically organized into several modules, with each module containing functions, traits, and types that users can employ in an Object-Oriented Programming (OOP) style. In contrast, the direct bindings code is organized into a single module, with all APIs structured in a C-style manner. This non-idiomatic style is inconvenient for Rust developers to use, as it requires them to manually organize the code into modules and make it into a more Rust-friendly style.

Secondly, the generated bindings may not guarantee safety. All APIs generated by the `bindgen` tool are marked as `unsafe` by default, as the tool cannot ensure their safety. On one hand, if the original C APIs interact with raw pointers to access memory, the generated bindings also needs to pass into raw pointers as arguments, which requires Rust developers to manually transform safe Rust objects into raw pointers and vice versa. The process of converting raw pointers into safe objects is always considered unsafe in Rust. On the other hand, if the original C APIs involve unsafe memory operations, the Rust compiler is incapable of checking the memory safety of these operations, potentially introducing various memory safety vulnerabilities through these APIs. A detailed discussion of memory safety vulnerabilities introduced by unsafe APIs will be provided in Section 1.3.

1.3 Vunlerabilities introduced by unsafe libraries in Rust

This section will discuss the memory safety vulnerabilities introduced by unsafe libraries if they are used in Rust applications. We will introduce two common memory safety vulnerabilities including use-after-free, buffer overflow and type confusion.

1.3.1 Use-after-free

Use-after-free vulnerabilities are very common in Rust applications that utilize unsafe C libraries. These vulnerabilities primarily arise because of the Rust compiler's inability to perceive the lifetime of objects within the C libraries. While the Rust compiler manages the lifetime of every object in Rust programs, it cannot do so for objects in C libraries. Consequently, Rust application developers or library wrapper developers must manually manage the lifetime of objects in the C library to ensure that these objects are not freed before they are used. Improper management of object lifetimes within the C library can introduce use-after-free vulnerabilities, leading to memory safety vulnerabilities.

Several CVEs from CVE-2021-45713[22] to CVE-2021-45719[23] are use-after-free vulnerabilities introduced by the `rusqlite` library, which is a wrapper library for the libsqlite3 library written in C. Here is an example trigger code from the GitHub issues[26] for reproducing this vulnerability in `rusqlite<0.25.4` or `rusqlite<0.26.2`, as shown in Figure 1.1.

The code in Figure 1.1 is a use case of `rusqlite` that creates a database connection and updates the hook function in the database connection. The hook function is a callback function that is invoked whenever a row is updated, inserted, or deleted in the database. However, in this case, the `hook` function is freed at line 17, while the enclosure from line 7 to line 17 has ended the variable `hook` has come to the end of its lifetime. However, the address of the `hook` function is still stored in the database connection, which will be invoked at line 20 when the database is updated. This will cause a use-after-free vulnerability, as the `hook` function has already been freed when its lifetime ends. This vulnerability is triggered because although the Rust compiler can manage

```rust
use rusqlite::{hooks::Action, Connection};
use std::sync::{Arc, Mutex};

fn main() {
    let db = Connection::open_in_memory().unwrap();

    {
        let locked: Arc<Mutex<()>> = Arc::new(Mutex::new(()));

        let hook = |_: Action, _: &str, _: &str, _: i64| {
            if let Ok(ref mut mutex) = locked.try_lock() {
                **mutex = ();
            }
        };

        db.update_hook(Some(hook));
    };

    db.execute("CREATE TABLE tbl(i integer)", []).unwrap();
    db.execute("INSERT INTO tbl (i) VALUES (1)", []).unwrap();
}
```

Figure 1.1. Use-after-free Vulnerabilities(CVE-2021-45713 - CVE-2021-45719) in Rusqlite

the lifetime in the Rust code, it cannot manage the lifetime in the external C libraries because all the data and functions in the external C libraries are passed in as the raw pointers. Therefore, the library wrapper developers must manually manage the lifetime of the objects in the external C libraries to ensure that these objects are not freed before they are used, which is a difficult task.

1.3.2 Buffer overflow

Buffer overflow vulnerabilities are likewise prevalent in Rust applications that incorporate unsafe C libraries. In fact, these vulnerabilities often originate from the unsafe C libraries themselves. Within Rust, the compiler enforces static checks during compile time and dynamic checks during runtime to minimize the occurrence of buffer overflow vulnerabilities. However, external C libraries are not subject to these compiler-enforced checks. As a result, buffer overflow vulnerabilities present in external C libraries persist in Rust applications that rely on these libraries, ultimately compromising the memory safety of the entire application.

CVE-2019-11922[20] is a buffer overflow vulnerability that exists in the `libzstd` library prior to version 1.3.8. The `libzstd`, also known as Zstandard, is a compression and decompression library that was developed by Facebook in C and widely used in real-world applications to provide fast and efficient compression and decompression functionalities. In the Rust ecosystem, applications can use the `zstd-rs` library, which is a wrapper library for the `zstd` to invoke the compression and decompression functionalities provided by the `libzstd` library with the same performance and a more Rust-friendly style. However, the `zstd-rs` library is also vulnerable to the vulnerabilities in the `zstd` library, as it is merely a wrapper library for the `zstd` library with several limited safety checks.

The vulnerability is triggered by the `ZSTD_buildCTable` function, which is a helper function that is used to build a compression table for the compression stage. The details of this vulnerability are shown in Figure 1.2, which is the source code of the `zstd` before version 1.3.8.

```c
size_t ZSTD_buildCTable(void* dst, size_t dstCapacity,
                  FSE_CTable* nextCTable, U32 FSELog,
    symbolEncodingType_e type,
                  unsigned* count, U32 max,
                  const BYTE* codeTable, ...)
{

    BYTE* op = (BYTE*)dst;
    const BYTE* const oend = op + dstCapacity;
    ...

    switch (type) {
    case set_rle:
        ...
        *op = codeTable[0];
        return 1;
        ...
    }
    ...
}
```

Figure 1.2. Buffer Overflow Vulnerability(CVE-2019-11922) in Zstandard

During the invocation of the `ZSTD_buildCTable` function in the compression processing stage, if the `type` is set to `set_rle`, the function attempts to assign the current value of `dst` as the starting value of the `codeTable`. Subsequently, it returns 1 as the counted size. However, this code

```
RETURN_ERROR_IF(
  dstCapacity==0, dstSize_tooSmall, "not enough space"
);
```

Figure 1.3. Fix of CVE-2019-11922

lacks any checks for the `dst` value, so if the `dst` value is not large enough, the `ZSTD_buildCTable` function will write the `codeTable` to unallocated memory for the dst pointer. This situation can lead to a buffer overflow, creating an opportunity for attackers to overwrite arbitrary memory. This vulnerability is finally mitigated by incorporating an additional check for the capacity at line 14, as shown in Figure 1.3. The check verifies that the `dst` value is sufficiently large to accommodate the `codeTable` prior to the assignment of the `dst`.

Despite the fix before, numerous libraries written in C remain susceptible to buffer overflow vulnerabilities, primarily due to the absence of safety checks and the language's usage patterns in C. Because C developers must directly interact with raw pointers and manually manage memory, which increases the possibility of memory errors, thus buffer overflow vulnerabilities are frequently found in C libraries. Consequently, when these vulnerable libraries are utilized in Rust applications, the buffer overflow vulnerabilities may propagate into the Rust applications, resulting in memory safety issues.

1.4 How Vulnerabilities Break Rust's memory safety

In this section, we will discuss how the vulnerabilities in the external C libraries can be exploited in Rust applications. In particular, we will discuss how these memory safety vulnerabilities can be used to break the data integrity and control-flow of Rust applications.

1.4.1 Data Integrity Corruption

Data Integrity, sometimes referred to as information integrity, is the maintenance of the completeness, accuracy, and validity of data during the entire lifecycle of the data[6]. In other words, data integrity ensures that data remains unmodified in an unexpected manner. In statically typed languages such as Rust, C, and C++, the compiler can infer the size of various data types, including primitive types and some user-defined structs, to ensure the memory area occupied by the data and prevent it from overwriting the memory that belongs to other data. However, in C and C++, pointers are frequently used to access data in memory. Some values pointers pointed to are buffers or arrays of data, whose size cannot be inferred by the compiler and an additional variable is required to store the size of the data. What's more, Even if a pointer points to a single data element with a fixed size, the pointer may still be misused to access other data types with different sizes, leading to type confusion.

Buffer overflow and type confusion are two common vulnerabilities that can be exploited to break the data integrity in applications. For buffer overflow, the attacker can overwrite the data in the memory that does not belong to the buffer, which can lead to the corruption of the data in the overwritten memory area, then the data integrity is broken so that the attacker can control the data in memory to modify the behavior of the application and even execute arbitrary code. One small exploit example that attackers can leverage the buffer overflow to exploit the data integrity and break the application, which is shown in Figure 1.4.

```c
int check_pwd() {
    char input[8];
    int access_granted = 0;

    const char* pwd = "password";
    scanf("%s", input);

    if (strcmp(input, pwd) == 0) {
        access_granted = 1;
    } else {
        printf("Access denied.\n");
    }

    if (access_granted) {
        system("/bin/sh");
    }

    return 0;
}
```

Figure 1.4. Buffer Overflow Exploit Example

In the example shown in Figure 1.4, the programmer use an unsafe function from the standard C library (`libc`), `scanf`, to read user input and store it in the `input` variable. However, the `scanf` function does not verify the size of the input data; thus, if the user enters a string longer than the size of the `input` variable, the `scanf` function writes the excess data to a memory area not allocated to `input`. In this example, the `input` variable has a size of 8 bytes. The compiler may first allocate the `input` variable on the stack, followed by the `access_granted` variable. If an attacker inputs a string exceeding 8 bytes, the `scanf` function writes the excess data to the memory area belonging to the `access_granted` variable, potentially overwriting the data and bypassing the authentication. Consequently, the attacker could execute the code within the subsequent `if` statement and gain control of the computer.

Note that `scanf` is only an example to demonstrate that missing the size check of the data can lead to a buffer overflow. In real-world applications, although the `scanf` function is rarely used, however, the buffer overflow vulnerabilities are still frequently found due to missing explicit size checks of the data, just like CVE-2019-11922[20].

CWE-843[8] enumerates several possible attack scenarios through type confusion, including an example written in C, which is shown in Figure 1.5.

In this example, an explicit pointer type conversion is performed at line 23, which converts the `char*` pointer `defaultMessage` to a `int*` pointer and the value is assigned to `buf.nameID` that is an `union` type. The misuse of the pointer type conversion leads to type confusion, which makes the compiler treat the type of `buf.name` as an integer value and wrongly writes an integer value onto a character array.

1.4.2 Control-Flow Corruption

Control flow refers to the sequence in which a program's instructions are executed. Typically, control flow is determined by the program's code and input data. However, memory safety vulnerabilities can be exploited to disrupt the control flow of the program. In this section, we will concentrate on scenarios in which control flow is compromised by altering function

```c
#define NAME_TYPE 1
#define ID_TYPE 2

struct MessageBuffer {
    int msgType;
    union {
        char *name;
        int nameID;
    };
};

int main(int argc, char **argv) {
    struct MessageBuffer buf;
    char *defaultMessage = "Hello World";

    buf.msgType = NAME_TYPE;
    buf.name = defaultMessage;
    printf("Pointer of buf.name is %p\n", buf.name);
    /* This particular value for nameID is used to make the code
     * architecture-independent. If coming from untrusted input, it
    could be any
     * value. */

    buf.nameID = (int)(defaultMessage + 1);
    printf("Pointer of buf.name is now %p\n", buf.name);
    if (buf.msgType == NAME_TYPE) {
        printf("Message: %s\n", buf.name);
    } else {
        printf("Message: Use ID %d\n", buf.nameID);
    }
}
```

Figure 1.5. Type Confusion Exploit Example 1 (C code)

pointers and return addresses on the stack. These scenarios are frequently exploited in real-world applications and are situations that Control-Flow Integrity (CFI) aims to prevent.

Function pointers are frequently used in programs to implement indirect function calls for callbacks and polymorphism. However, function pointers are also frequently exploited by attackers to alter the control flow of programs. In x86-64, function pointers are implemented as a 64-bit value to store the address of the function. Then when the function pointer is called, the program loads the corresponding function address from the memory to `rax` and jumps to that address to execute the function. However, for an indirect function call, the destination function address is stored in memory, instead of hard-coded in the program like a direct function call.

Thus, if attackers can modify the function pointer's value through a memory safety vulnerability, they can alter the target function address and then hijack the control flow of the program. This type of attack is also known as Call/Jump Oriented Programming (COP/JOP) [5].

Return addresses are used to store the address of the next instruction to be executed once the current function returns. In x86-64 architecture, the return address is stored on the stack, and when the function reaches an `ret` instruction to return, the program pops the return address from the stack and sets the `rip` register to the return address in order to jump to the next instruction. However, without additional protection, return addresses can be modified by attackers who seek to hijack the control flow through memory safety vulnerabilities. This type of attack, which exploits return addresses, is also known as Return-Oriented Programming (ROP)[27].

Through these two exploitation scenarios, the attacker can hijack the control flow of the program. The ”gadget” is a small piece of code that ends with a `ret` instruction for ROP or a `call` instruction for COP/JOP[27]. Attackers can chain multiple gadgets together to form a gadget chain, which allows the program to execute the control flow that even doesn't exist in the program without violating the common memory safety protection mechanisms, such as W$\oplus$X.

1.5 Related work

There have been many attempts to solve the memory safety problem in unsafe C libraries used by Rust applications. In this section, we will discuss several current solutions and their limitations. We will categorize these solutions into two categories: solutions that are based on the improvement of memory safety in the unsafe C/C++ code and solutions that are based on the Software Fault Isolation (SFI) technique.

1.5.1 Memory Safety Improvements

Efforts aimed at enhancing memory safety are primarily focusing on improving the memory safety attributes of C/C++ code or compiled binaries. The fundamental cause of these vulnerabilities is the inherent unsafety of the C/C++ language, which does not provide any guarantees regarding memory safety. Thus, such solutions strive to improve the safety aspects within both the C/C++ language and the compiled binary code.

Data-flow integrity enforcement is a strategy designed to ensure the integrity of the data flow within a program through a data-flow graph[7]. This graph is constructed when compiling the C code to illustrate the data flow of the program, and the integrity of this flow is maintained by checking the data-flow graph at runtime before the data access. This methodology can prevent an attacker from tampering with the data flow and redirecting the data to an area of memory under the attacker's control. Nevertheless, this technique is only capable of preventing corruption of the data flow, and still may cause a false negative issue, which means that an attacker may be able to compromise the program in other ways, such as by hijacking the control flow. Furthermore, this technique lacks efficiency for widespread deployment in real-world applications, as it introduces an overhead ranging from 44% to 103% on SPEC 2000.

CETS[18] and SoftBound[17] are two techniques that aim to enforce the memory safety of the pointers. CETS will check whether the pointer is valid before the pointer is dereferenced, and SoftBound will check the memory bounds of the pointer before the pointer is dereferenced. Both of these two techniques can prevent the attacker from corrupting the memory through the pointer dereference. However, for CETS, it can only prevent the use-after-free vulnerability and for SoftBound, the memory bounds checking is implemented by adding a metadata pointer to each pointer to ensure the validity access of the pointer, but this approach is too strict so it will not be compatible with several existing C/C++ libraries[28]. Moreover, SoftBound will introduce an average overhead of 67% on their benchmarks.

Apart from the data flow and pointer safety, there are also some solutions that aim to improve the control flow safety. CCFIR[30] collects all legal targets of indirect control flow transfer into a dedicated random table, and then checks whether the target of the control flow transfer is legal before the control flow transfer is executed. SafeDispatch[10] statically analyzes the C++ programs and inserts the runtime checks to prevent the vtable hijacking attack. CCFI[14] uses cryptographic hashes to authenticate the return address and function pointers, and vtable pointers. Intel CET[2] uses indirect branch tracking instructions (ENDBR) and shadow stack to prevent both ROP and COP/JOP attacks. However, although these solutions can prevent the control flow hijacking attack, only assuring the control-flow integrity is not enough to prevent the memory safety vulnerabilities, it only prevents the attacker from hijacking the control flow to the attacker's code, but the attacker can still take control of the program's memory space and corrupt the memory.

Cyclone[11] and CCured[24] are two early attempts to implement a safe C dialect. They both introduce a new safe C language with extra type checks or runtime checks to ensure the memory safety of the program. However, these kinds of solutions are not compatible with the existing C/C++ libraries, so the developers have to rewrite the existing code into their safe C dialect, which is not practical for the real-world applications.

1.5.2 Software Fault Isolation

It is difficult to mitigate memory safety vulnerabilities in unsafe C code, all the solutions mentioned above based on improving the memory safety of C code are either not providing enough protection or not practical for the existing C/C++ libraries. Thus, the more practical solutions are targeted at deploying software fault isolation(SFI) to mitigate memory safety vulnerabilities. SFI is a technique based on the fact that memory safety vulnerabilities are not easy to be mitigated in the unsafe C code, so this technique aims to isolate the unsafe execution environment from the safe execution environment through a sandbox. So that even if the unsafe code is exploited and the memory is corrupted, the attacker cannot escape from the sandbox and take effect on the safe execution environment.

XRust[13] modifies the heap allocator to allocate the unsafe code in Rust into a separate memory region and then uses the memory guard pages to achieve the in-process isolation between the unsafe code and the safe code. However, XRust doesn't provide protection to the stack, so the attacker can still corrupt the stack and bypass the safety assurance of XRust. Moreover, XRust doesn't provide any protection for the external libraries in Rust, so the attackers can still exploit the memory safety vulnerabilities in the external libraries. Fidelius Charm[3] uses similar approach to XRust, it provides in-process isolation by separating the memory region into the trusted region and the untrusted region. However, Fidelius Charm also lacks protection for the stack. Attackers can exploit the stack memory and add the pointer to data in the trusted region to implement the unauthorized memory access.

TRust[4] separates both stack memory and heap memory into the trusted region and the untrusted region, and it uses the Intel Memory Protection Keys (MPK) to protect the trusted region from unauthorized memory access. For memory protection on external libraries, it inserts the ”entry gate” and ”exit gate” into the external libraries to prevent the attacker from exploiting the stack pointer. However, TRust takes leverage of the Intel MPK, which is currently only available on the Intel CPUs, so it is not compatible with other architectures like ARM. Sandcrust[12] employs inter-process isolation to separate unsafe code from safe code, placing them into different processes. This approach utilizes Inter-Process Communication (IPC) to facilitate data exchange between unsafe and safe processes. However, the performance overhead associated with IPC is unneglectable, leading to a high performance overhead introduced by Sandcrust. The performance overhead of Sandcrust is at worst 10 times slower than the original program on the snappy benchmark.

All of the solutions mentioned before employ a coarse-grained isolation approach, where the entirety of the unsafe code is placed within a single memory region. This approach of isolation implies that vulnerabilities introduced in one unsafe library can impact other libraries in the same memory region. Consequently, this strategy could still introduce potential security issues, such as data leakage or privilege escalation, by exploiting other libraries within the same unsafe memory region. What’s more, these solutions don’t provide sanity checks for the data in unsafe memory regions when the data is accessed by the safe code, so if the data in the unsafe memory region is corrupted, the safe code is still vulnerable to memory safety vulnerabilities by accessing the corrupted data.

RLBox C++[19] is a fine-grained isolation solution that provides sandboxing for C libraries. It uses software-based fault isolation (Webassembly) or multi-core-based isolation to isolate the C libraries into separate sandbox instances. It also provides an enforcement mechanism to ensure data integrity when data is exchanged from the sandbox to the host area. However, RLBox C++ aims to provide sandboxing for the usage of C libraries in C++ programs, but not Rust programs. Moreover, RLBox C++ lacks the type derivation from the C libraries, so when migrating the existing C++ applications to use RLBox C++, the developers have to manually write all the type definitions for data structures in the C libraries.

Chapter 2

RLBox-Rust: The Fine-grained Library Sandbox For Rust Ecosystem

In this Chapter, we present our motivation, design, and implementation of our solution to the problem of safely using unsafe C libraries in Rust applications, the RLBox-Rust framework. We will first present the motivation and objectives of the RLBox-Rust framework in section 2.1. Then, we will present the overview architecture of the RLBox-Rust framework in section 2.2 to demonstrate how the RLBox-Rust framework can provide a fine-grained C-based library sandbox framework for Rust applications. In section 2.3, we will use a real-world example to demonstrate how easy it is to migrate the existing Rust applications to safely use the unsafe libraries thorough RLBox-Rust. Following that, we will present detailed the design and the implementation of the RLBox-Rust framework in section 2.4. Finally, we will evaluate the performance overhead of the RLBox-Rust framework on two kinds of benchmarks in section 2.5.

2.1 Motivation and Objectives

Since it's difficult to eliminate memory safety vulnerabilities in unsafe C code, our goal is to provide a fine-grained C-based library sandbox framework for Rust applications to implement software fault isolation(SFI). So that the unsafe C code can be executed in a sandbox, and we also need to provide a safe interface on the Rust side to allow the Rust application to communicate with the unsafe sandboxed code in a safe way. Fine-grained isolation aims to isolate the different instances of unsafe libraries from each other, so that the corruption of one sandbox instance will not affect the other instances and the safe execution environment, making the entire system more secure and reliable.

In detail, the primary goals of RLBox-Rust are to provide a fine-grained C-based library sandbox framework for Rust applications, which can achieve the following goals:

- **Fine-grained Isolation**: The library environment isolation is to isolate the unsafe execution environment of the unsafe libraries from the safe execution environment of the safe Rust applications. The execution environment includes the memory space, the registers and the execution control flow. Under this isolation, unsafe libraries should not be able to access the memory space outside of the sandbox, and the execution control flow should not be able to jump out of the sandbox, except for the ad-hoc APIs registered and safely provided by the application developers. Besides that, the isolation should be fine-grained, which means that different libraries, and even one library in different instances, should be isolated from each other. So that the developer will have the flexibility to trade-off the security and the performance. For example, the developer can choose to isolate the different instances of the same library from each other, to isolation different process parts of the library, or just simply isolate the entire library in one sandbox instance.

- **Type Integrity**: The type integrity is to ensure the type safety when exchanging data between the sandbox fields and the host application fields, which means that the data exchanged between the sandbox fields and the host application fields should be of the same type or the type can be safely converted to the other type such as the shorter bit-width integer type can be safely converted to the longer bit-width integer type.

- **Data Integrity**: The data integrity is to ensure the integrity of the data when exchanging data between the sandbox fields and the host application fields. When the data is going to be copied from the sandbox fields to the host application fields, the RLBox-Rust framework should ensure that the data is not corrupted. When the data is going to be copied from the host application fields to the sandbox fields, the framework should ensure that the data is correctly copied into the sandbox fields and does not overflow the sandbox fields.

- **Easy Migration**: Since the RLBox-Rust framework is a library sandbox framework developed for the Rust ecosystem and applications, the application developers should be able to easily migrate their existing Rust applications to safely and intuitively use unsafe libraries with the help of RLBox-Rust.
- **Performance**: The performance overhead of the Rust applications that use the RLBox-Rust for unsafe library sandboxing should be acceptable compared to the performance of the Rust applications that directly use the unsafe libraries.

2.2 Overview

The structure of operation for the RLBox-Rust framework is described in Figure 2.1. As a fine-grained library sandboxing system designed for Rust applications, RLBox-Rust is capable of simultaneously generating multiple sandbox instances for multiple libraries, or even various instances of the same library.

When an application requires access to the APIs of an unsafe library, the developer can create a distinct sandbox instance for the particular C library through RLBox-Rust. Subsequently, the developer can either transfer data from the host application fields to the sandbox fields, or directly allocate data within the sandbox fields, in preparation for the execution of the unsafe library APIs. The unsafe library APIs are invoked within the confines of the sandbox fields, implemented by safe APIs generated by the RLBox-Rust framework. Upon execution of the unsafe library APIs, the generated data or modified data is still located within the sandbox field in the same instance, and ready to either be transferred back to the host application or utilized further by other unsafe library APIs.

When the host application needs to access the data within the sandbox fields, the developer can transfer the data from the sandbox fields to the host application field. This process involves an

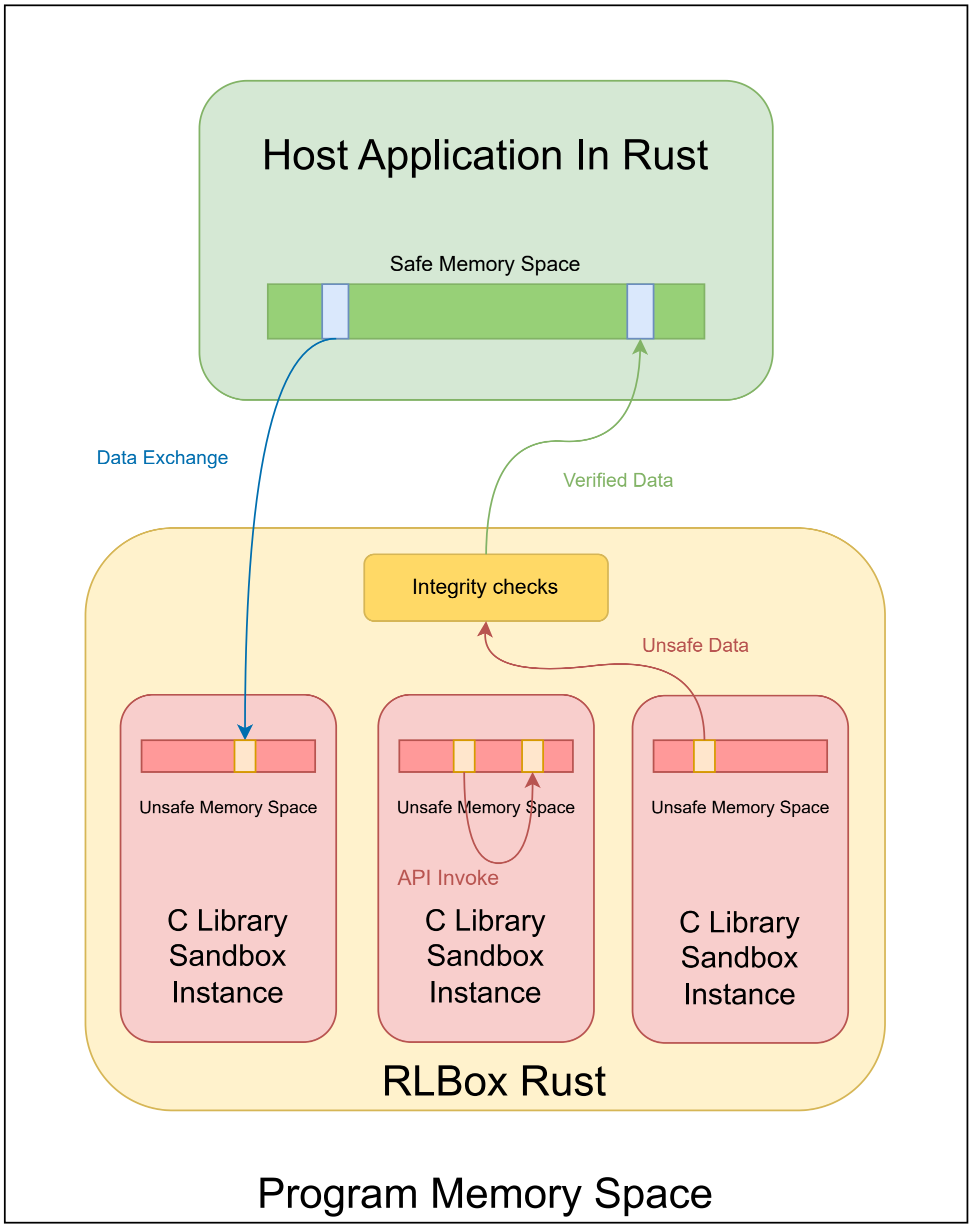

Figure 2.1. Overview Architecture of RLBox-Rust

integrity check to ensure that the data being transferred from the sandbox fields is both type-safe and data-safe - referred as **data sanitization**. Following data sanitization, the data is securely transferred into the host application fields for subsequent usage.

Through this architecture, RLBox-Rust is able to provide fine-grained isolation for unsafe libraries. Upon this isolation, RLBox-Rust provides a unified and transparent interface for application developers to safely use unsafe libraries APIs in the sandbox fields, and finally transfer the data from the sandbox back to the host application fields with enforced type and data sanitization.

2.3 Migration Process

RLBox-Rust is designed to offer an intuitive, user-friendly library sandboxing framework experience. To facilitate a smooth transition for application developers migrating from their current Rust applications with unsafe libraries to the utilization of the RLBox-Rust framework, we exploit the capabilities of the powerful Rust macro system. Through the Rust macro system, the developer can directly use the macro API invocation to invoke the APIs within a sandbox instance by simply specifying the API name and the arguments. This approach simplifies the process of developing efforts to migrate to the RLBox-Rust framework.

Here, we will demonstrate the migration process of a specific Rust application that utilizes the C library, which we will refer to as `increment_buffer` library. The definition of the C library is shown in Figure 2.2.

```c
// definition of lib.h
typedef int*(OnCompletion)(int, int*, unsigned int);

void incrementBufferWithCallback(int* buffer, int length,
    OnCompletion cb);

// definition of lib.c
void incrementBufferWithCallback(int *buffer, int length,
    OnCompletion *cb) {
  int i;
  for (i = 0; i < length; i++) {
    buffer[i]++;
  }
  int* buffer_half = cb(buffer[i - 1], buffer, length);
  for (i = 0; i < (length - length / 2); i++) {
    buffer_half[i]++;
  }
}
```

Figure 2.2. Example C Library

The C library contains an API, `incrementBufferWithCallback`, which takes in a buffer and a callback function pointer with type `OnCompletion` as arguments. The API firstly increments the entire buffer by one, and then invokes the callback function to get the pointer to half of the buffer. Finally, the API increments the second half of the buffer by one again.

In the current use of a C library, the developer of a Rust application is required to directly invoke raw C APIs via bindings. These bindings are generated by Rust Bindgen, which necessitates the developer to create a build script responsible for compiling the C library and generating the corresponding bindings. To generate the bindings for RLBox-Rust, the developer only needs to incorporate two additional processes: building the sandboxed version of the C library and generating bindings and sandbox bindings, based on the original build script. These two bindings are designed to work in corporate for both the original C library and the sandbox primitive.

In our implementation that employs WebAssembly as the sandbox primitive, the developer is required to compile the C library into WebAssembly. Subsequently, wasm2c is used to generate C code that serves as the sandbox primitive. Ultimately, two separate bindings are generated for the original C library and the sandbox primitive C code.

After developers have generated proper bindings for the RLBox-Rust, they can directly import the RLBox-Rust APIs and invoke the APIs of the C library within the sandbox instance just similar to the current way of invoking the C library through bindings. To demonstrate the similarity, we show two examples of invoking the `incrementBufferWithCallback` API. The first example is the current solution that directly invokes the C library through bindings, which is shown in Figure 2.3. The second example is the solution that utilizes the RLBox-Rust framework, which is shown in Figure 2.4 and Figure 2.5.

Compared with the current solution, the RLBox-Rust solution marks all the values in the sandbox as `Tainted` type, which will be detailly discussed in section 2.4. When the developer wants to mark a function as the callback function, the developer only needs to add the `#[tainted_callback]` attribute macro with the type definition of the callback function. The

```rust
mod bindings {
    include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
}

fn on_complete(
    result: i32,
    buffer: *mut i32,
    length: u32
) -> *mut i32 {
    let buffer = unsafe { std::slice::from_raw_parts_mut(buffer,
  length as usize) };
    unsafe { buffer.as_mut_ptr().offset((length / 2) as isize) }
}

fn main() {
    const LEN: usize = 23;
    let mut buffer = [0 as i32; LEN];
    for i in 0..LEN {
        buffer[i] = i as i32;
    }

    unsafe {
        bindings::incrementBufferWithCallback(
            buffer.as_mut_ptr(),
            LEN as u32,
            on_complete
        );
    }

    for (i, item) in buffer.iter().enumerate() {
        if i < LEN / 2 {
            if *item != i as i32 + 1 {
                println!("expected {} but got {}", i as i32 + 1,
  item);
                panic!("Bad value");
            }
        } else if *item != i as i32 + 2 {
            panic!("Bad value");
        }
    }

    println!("Succeeded");
}
```

Figure 2.3. Example of direct invocation of `incrementBufferWithCallback` API

```
#[tainted_callback(rlbox::bindings::OnCompletion)]
fn on_complete(
    result: Tainted<i32>,
    buffer: Tainted<*mut i32>,
    length: Tainted<u32>,
) -> Tainted<*mut i32> {
    let length = length.untainted_clone(|_| true);
    let buffer = buffer.assemble_vector(length as usize);
    let buffer_untainted = buffer.untainted_clone(|buffer| {
        for (i, item) in buffer.iter().enumerate() {
            if *item != i as i32 + 1 {
                println!("expected {} but got {}", i as i32 + 1,
    item);
                return false;
            }
        }
        true
    });
    buffer.index_ptr((length / 2) as usize)
}
```

Figure 2.4. Callback function definition in RLBox-Rust

RLBox-Rust framework will automatically generate the corresponding callback function code that can be registered into the sandbox.

For API invocation, the developer only needs to invoke the API from using unsafe block by using the `sandbox_invoke!` macro with the API name and arguments, as shown in Figure 2.5. When the data is accessed from the host Rust application, the `Tainted` value needs to firstly be sanitized by using the `untainted_clone` API, which passes into a verifier function that verifiers the data of the value.

In this way, the developer can easily migrate the current Rust application with unsafe libraries to the RLBox-Rust framework and take advantage of the sandboxing capabilities of RLBox-Rust.

2.4 Design and Implementation

In this section, we will present the basic structure of the RLBox-Rust framework and how we design and implement the RLBox-Rust framework to achieve the primary functionalities that

```
fn main() {
    let sbx = WasmSandbox::new(/* parameters */);
    const LEN: usize = 23;

    let mut tainted_buffer = tainted_vec!(sbx.clone(), [0 as i32;
LEN]);

    for i in 0..LEN {
        tainted_buffer.index_mut(i).update(i as i32);
    }

    // register the callback
    let cb = register_on_complete(sbx.clone());

    sandbox_invoke!(
        sbx.clone(),
        rlbox::bindings::incrementBufferWithCallback,
        &mut tainted_buffer,
        LEN as i32,
        &cb
    );

    let buffer = tainted_buffer.untainted_clone(|_| true);

    for (i, item) in buffer.iter().enumerate() {
        if i < LEN / 2 {
            if *item != i as i32 + 1 {
                println!("expected {} but got {}", i as i32 + 1,
item);
                panic!("Bad value");
            }
        } else if *item != i as i32 + 2 {
            panic!("Bad value");
        }
    }

    println!("Succeeded");
}
```

Figure 2.5. Example of invocation of `incrementBufferWithCallback` API in RLBox-Rust

help to provide safety for the Rust applications that we described in section 2.1. we will describe the detailed design and implementation of all the components of the RLBox-Rust framework and the reason why we design and implement the RLBox-Rust framework in this way and why this design can be beneficial to the structure and extensibility of the RLBox-Rust framework. We will also introduce some tricks we used to achieve the functionalities of the RLBox-Rust framework. More detailed implementation can be found in our GitHub repository.

2.4.1 Design Overview

To achieve the primary goals of the RLBox-Rust framework, we design and implement the RLBox-Rust framework in the following components:

- **Sandboxing Primitives**: Provide isolation functionalities to isolate the unsafe library code from the host application code, as well as the interface to exchange data and APIs to modify the function table to register callback functions.
- **RLBox Bindgen**: Automatically generate the Rust bindings from the C libraries and automatically infer the information of types to generate helper functions to bridge the gap between the sandbox fields and the host application fields.
- **Value-Accessing APIs**: Provide a set of user-friendly APIs, including traits, structs, and macros, to allow the application developers easily and safely access the data and APIs in unsafe C libraries.
- **Static and Dynamic Integrity Checks**: Provide static and dynamic integrity checks to ensure the type integrity and domain integrity of the data exchanged and API invokes.

To utilize the sandbox primitives, the unsafe C libraries need to be wrapped or recompiled into a sandboxed library. In our implementation, we use the WebAssembly sandbox to provide the sandboxing primitives so that the unsafe C libraries need to be compiled into WebAssembly modules, we will specify the details of the WebAssembly sandboxing primitives in section 2.4.2.

Then, to bridge the gap between sandbox fields and host application fields, it is necessary to address differences between the implementations, such as in our implementation that using WebAssembly as sandbox primitives, the size of pointers in wasm32 is 32 bits, while the size of pointers in the host application for the x86-64 architecture is 64 bits. To handle this, we have developed the RLBox Bingen to automatically infer information in the C header files about pointer types and generate a corresponding ”shadow” struct to deduce the memory layout of structs within the sandbox field. Additionally, we have generated helper functions to ensure that the structs in the sandbox fields are compatible with our RLBox-Rust framework and can be easily utilized with our Value-Accessing APIs. The specifics of the RLBox Bingen will be discussed in Section 2.4.3.

To provide a user-friendly interface for application developers to safely and efficiently migrate existing Rust applications to the RLBox-Rust framework, we have developed a set of Value-Accessing APIs. These APIs enable developers to access data and APIs within sandboxed libraries and the sandbox itself. The Value-Accessing APIs can be broadly divided into three parts: sandboxing operation APIs, the `Tainted<T>` struct, and procedural and declarative macros. The details of the Value-Accessing APIs will be discussed in Section 2.4.4.

Type integrity and domain integrity are essential to ensure the safety of the data and APIs in the RLBox-Rust framework. To ensure the type integrity and domain integrity of the data and APIs, we take advantage of the Rust type system to provide the static type verifiers when the developers invoke the APIs in the sandboxed libraries to make sure that the data and APIs are used correctly. Additionally, when data is exchanged between the host application and the sandbox, we use the size information of the data deduced by the RLBox Bingen to ensure that data size is consistent between the host application and the sandbox. And when the developers need to convert the data from sandbox fields to host application fields, we will enforce the domain integrity of the data to ensure that the data is within the domain of the corresponding type. The specifics of the static and dynamic integrity checks will be discussed in Section 2.4.5.

2.4.2 Sandbox Primitives

The sandboxing primitives are the core sandboxing components of the RLBox-Rust framework. The sandboxing components are responsible for the library environment isolation, including memory space and stack isolation, as well as execution control flow isolation. As well as the interface for the application developers to safely access the data and APIs provided by the unsafe libraries. The sandboxing primitives can both be developed from scratch or reused from the existing sandbox technologies such as WebAssembly. In our implementation, we reuse the existing WebAssembly to provide the sandboxing primitives.

Memory isolation ensures that the code and data within the sandbox cannot directly access the memory space in the host application or other sandbox instances. This isolation applies to both heap memory space and stack, preventing memory corruption and information leakage from the unsafe library to the host application and other sandboxes. Under this isolation, even if the unsafe library in the sandbox is compromised, the affected memory space is limited to the sandbox itself, keeping the host application and other sandboxes safe.

Control Flow Isolation makes sure that the control flow should not escape the sandbox in any case, except for explicitly registered and safe APIs provided by the application developers. This requirement prevents library code from hijacking the control flow of the host application to execute malicious code outside of the sandbox. To strengthen control flow isolation and provide secure access to some host application APIs, the sandbox should offer two features: Address Space Control and Explicit API Registration.

Address Space Control requires all executable code to access memory space within the sandbox. This constraint necessitates compiling unsafe libraries in a way that enforces strict control over the memory address space used by the library code, instead of using general-purpose compilers that do not provide in-process memory isolation. This approach ensures that library code can only access the memory space inside the sandbox, confining the control flow and preventing the execution of malicious code outside of the sandbox.

Although Address Space Control provides control flow isolation, real-world libraries often require developers to supply callback functions for error handlers and other events. Strictly enforcing Address Space Control would prevent developers from registering callback functions from the host application, making the library less usable. To address this issue, the sandboxing primitive should also provide an Explicit API Registration feature. This feature allows developers to securely register callback functions from the host application, enabling the library code to execute host application-provided callback functions, while maintaining safety through strict static and dynamic integrity checks. The RLBox-Rust framework will provide this feature, discussed in detail in section 2.4.5.

Given the strict isolation and control flow restrictions imposed by sandboxing primitives, they must also offer a set of interfaces for developers to safely exchange data between the unsafe library and the host application, as well as access APIs provided by the unsafe library in a secure manner. This set of interfaces will include APIs for accessing isolated memory space, for the purpose of reading and writing data. At the same time, it should also provide APIs for accessing library APIs, for the purpose of invoking library functions. Furthermore, accessing APIs towards the function table is also required to support the Explicit API Registration feature.

WebAssembly: A High Performance and Secure Solution

WebAssembly (Wasm) is a binary instruction format designed for safe execution and seamless integration with multiple programming languages. Wasm is intended to be executed within a sandboxed environment, such as a web browser, to ensure isolation, security, and portability[16]. These properties make Wasm an ideal candidate for implementing sandboxing primitives in the RLBox-Rust framework. In the rest section, we will demonstrate how Wasm fulfills the requirements of sandboxing primitives as previously described and why we use wasm2c, which is a compilation-based implement of Wasm, to fulfill these primitives in the RLBox-Rust framework. Additionally, we will discuss the benefits of using Wasm as sandboxing primitives, including performance, security, and compatibility with a wide range of libraries.

For Memory Isolation, Wasm provides a linear memory model, where each Wasm module has its own linear memory space[9]. This isolated memory space ensures that memory accesses within the Wasm module cannot directly access the host application's memory or other Wasm modules' memory spaces. In the popular Wasm runtime implementations, such as Wasmtime and wasm2c, the linear memory space is implemented as a virtual, isolated memory space, which prevents the Wasm module from accessing the out-of-bound memory space. This isolation feature will effectively mitigate the influence of certain classes of memory safety bugs such as buffer overflows and use-after-free bugs, which the effection will be isolated within the sandbox.

Wasm also makes some guarantees for several common problems related to unsafe pointer usage and undefined behavior[16]. In Wasm, Unlike traditional architectures, referencing into the function and variables in static memory is not implemented by pointers. Instead, Wasm maintains the static function index and static variable index in a separate section, so that the code in the Wasm module can only access the function and static variables through the index in complete pattern.

For Control Flow Isolation, Wasm utilizes a stack-based virtual machine to execute the Wasm module. So codes in the Wasm module will be restricted inside of the sandbox. The call stack of the Wasm module is protected and invulnerable to buffer overflow attacks[16]. In this pattern, the return-oriented programming (ROP) attack will be effectively mitigated because the attacker cannot control the return address of the function call in the Wasm stack.

Coarse-grained Control-flow Integrity (CFI) means that the control flow of the program is restricted to a set of valid targets (e.g., function entry points) instead of redirected to arbitrary locations. At the same time, the return address of the function call is also restricted to the targets that are exactly after a function call instruction. Wasm has these properties by design, in which the stack is protected and the return address of the function call can keep unmodified. At the same time, for the direct function call, the target of the call instruction will be an index in the function table section instead of a function pointer toward the function. If the function call target comes to an unexpected position, the Wasm module will use a memory trap to capture it and stop the program. For the indirect function call, the target functions are also required to be registered in the indirect function table before they can be invoked from an indirect function call instruction. This feature provides both the safety for indirect function invokes and the ability to register the outside function into the indirect function table. Based on these features, Wasm can ensure the coarse-grained CFI for the Wasm module.

Utilizing wasm2c to Implement the Sandbox Primitives

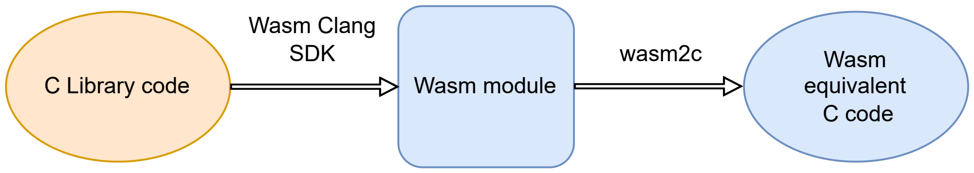

Figure 2.6. Process of Compiling C library code into Wasm and equivalent C code

In our implementation of the RLBox-Rust framework, we have chosen to use wasm2c, a tool within the WebAssembly Binary Toolkit (WABT). This tool enables the compilation of Wasm modules into equivalent C source and header files[29] within the same functionality and value semantics as the original Wasm module. e. The resultant C code encapsulates memory space and stack operations, providing a virtual environment for the execution of Wasm code. Specifically, each WebAssembly instance is represented by a C struct containing a virtual memory space, a virtual function table, and metadata, which includes the size of the virtual memory, the pages and maximum pages of the virtual memory, as well as the size of the virtual function table and other meta information of sandbox instance. The wasm2c runtime further provides several APIs for managing sandbox instances, such as `wasm2c_instantiate` for instantiating a new Wasm module, and `wasm2c_free` for freeing the module respectively. During the execution of the generated code, all data interacting with the Wasm module is stored in the virtual memory, and all indirect function calls are redirected to the virtual function table. The process of compiling C library code into Wasm and equivalent C code is shown in Figure 2.6.

The translated functions present in Wasm modules are implemented in C, maintaining the same semantics as the original Wasm module. To implement the stack-based virtual machine, the generated C code employs a set of local variables to simulate the stack pointer and stack frame. When the `load` and `store` instructions are executed, values are stored exclusively within the virtual memory space through `DEFINE_LOAD` and `DEFINE_STORE` macros. As a result, all code and corresponding data access within the Wasm module remains isolated from the host application. To ensure security, the generated C code also performs boundary checks when accessing the virtual memory space to prevent out-of-bounds memory accesses.

Additionally, the generated code by wasm2c also provides a set of APIs for accessing the virtual function table in the sandbox instance. These APIs include APIs for registering functions into the virtual function table get the index of registered functions. When indirect functions are invoked in the Wasm module, the wasm2c runtime implements the `CALL_INDIRECT` macro and passes the function type, function index, and function arguments to the macro. The macro then performs a lookup in the virtual function table to retrieve the function pointer and invokes the function with the provided arguments with the check of the function type. This approach ensures that all indirect function calls are indexed in the function table, and the function type is checked before the function is invoked.

By compiling the generated C code into a shared library, it can be loaded by the host application to serve as the sandboxing primitive. This approach preserves performance levels comparable to that of native library code while ensuring a high degree of security and isolation, making wasm2c an ideal choice for our Wasm runtime implementation.

2.4.3 RLBox Bindgen

The RLBox Bindgen is responsible for the generation of bindings of unsafe libraries and sandbox-accessing APIs. Our implementation leverages the Rust Bindgen, and extends it to fulfill the specific requirements of the RLBox-Rust framework. RLBox Bindgen generates bindings from multiple sources: the C header files of the wasm2c-generated code that provide sandboxing primitives and APIs, and the original C header files of the unsafe libraries. The latter is crucial,

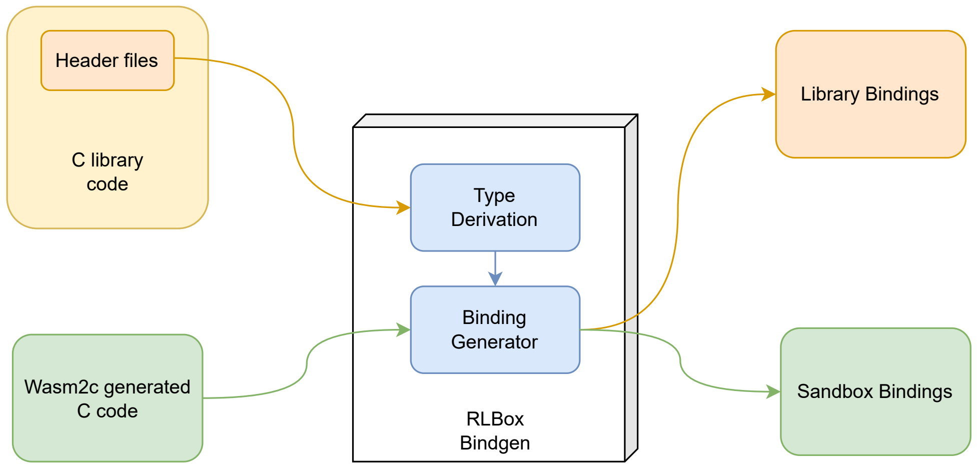

Figure 2.7. Functionality of RLBox Bindgen

as it offers type information and data structure details vital for the RLBox-Rust framework's automatic inference capabilities. These inferences are then used to generate code that is vital for the operation of the RLBox-Rust framework. This approach significantly reduces the developer's manual workload associated with writing bindings for Wasm sandboxes, presenting a noteworthy advancement over the RLBox C++ implementation[19]. Then, after removing the redundant part from two sets of bindings and merging them together, the RLBox Bindgen will generate the final bindings for the unsafe libraries. This process of bindings generation is shown in Figure 2.7.

On the top of the Rust Bindgen, we introduce a new struct derive type named `Shadow`. This derivative is utilized to identify all the structs and enums in the unsafe libraries that will be later processed by procedural macros. Specifically, any struct marked with `Shadow` will result in a new shadow struct. This shadow structure is utilized to automatically infer the memory layout of the corresponding struct within the sandbox environment.

Because of the architectural differences in our implementation, where the pointer size in the Wasm sandbox is 32 bits - potentially differing from the host environment's pointer size - addressing this difference is crucial. Our solution replaces all pointer types in the shadow struct with the `u32` type. Consequently, the memory layout within the sandbox can be directly inferred from the shadow struct in the host environment. This approach ensures that a mapping of address offset can be established between the sandbox and the host for each field within the struct via the corresponding shadow struct.

The `Shadow` derive type works an additional role in facilitating the automated generation of code that implements traits and helper functions as defined in the RLBox-Rust value-accessing APIs. This is enabled by the procedural macros' ability to fully access the type information of each field within the struct, thereby significantly simplifying code generation and ensuring correct field handling.

For example, every struct within the C library will be automatically equipped with a helper function named `new_tainted`. This function is used for the creation of a new struct instance in the sandbox, initializing all fields to a provided value within the sandbox if the field is a pointer type, or a direct value if the field is a primitive type. Given that the procedural macros can access the number of fields and the type of each field, the helper function can be generically constructed with all fields as arguments, maintaining exact type correspondence with the field type. As a result, this approach ensures that the helper function can statically infer the type of each argument and perform the correct static type checks when the struct is created in the sandbox.

In order to implement the traits defined in the RLBox-Rust value-accessing APIs, the procedural macros leverage the address offset mapping between the sandbox and the host for each field within the struct. This mapping is instrumental in generating the `mem_map` function, which retrieves the corresponding address of a field in the sandbox from the address of the field in the host environment. This mapping is critical during the stage of value copying, whether it is from the host environment to the sandbox, or vice versa. The address offset is needed to calculate the address of each field within the struct, ensuring that all values are accurately copied into their expected locations. Detailed implementation of the value-accessing APIs is presented in Section 2.4.4.

In C header files, type aliases are often employed to simplify complex types, such as multi-layered pointers, into a more understandable alias. Rust's Bindgen does not unravel these type aliases and retains them in the generated bindings. This behavior could lead our procedural macros to incorrectly interpret the field type of the struct To deal with this issue, we have modified the code of our RLBox Bindgen to include support for unraveling type aliases in the generated bindings. This adjustment enables our procedural macros to accurately interpret the field type of the struct.

2.4.4 Value-Accessing APIs

Although the Rust Bindgen is able to generate all the APIs both for sandbox and for C libraries, as the sandboxing primitives are implemented separately from the RLBox-Rust framework, they only offer a set of low-level APIs that are not user-friendly and may not be safe for direct access. Furthermore, different sandboxing primitives may present varying APIs for accessing data and functions provided by unsafe libraries. To address these challenges, we developed the value-accessing APIs to offer a unified, easy-to-use, and safe interface for application developers to interact with sandboxing primitives, including moving data between the sandbox and the host application, registering callback functions into the sandbox and invoking functions in the sandbox. In our implementation, the value-accessing APIs are crafted in Rust, utilizing macros, traits, and structs to create a consistent interface for developers to access data and functions in sandboxing primitives and unsafe libraries. With the help of the value-accessing APIs, the developers can easily and intuitively migrate their existing Rust applications to use the RLBox-Rust framework to safely use unsafe libraries. In this section, we will detailly introduce the design and implementation of our value-accessing APIs.

RLBox trait

The `RLBox` trait, implemented by all primitive types, structs, enums, and `std::String` within the RLBox-Rust framework, prescribes a series of functions used to access metadata associated with a value. Including functions that return the size and the shadow size of the value (indicating the size of the value within the sandbox) in bytes, as well as other helper functions to downcast the value into a specific sub-trait.

The `RLBox` trait is inherited into three sub-traits. The `RLBoxBasic` trait is implemented by most of the primitive types. The `RLBoxVector` trait is implemented by some array-like data structures, which includes vectors, slices, fixed-size arrays, and `std::String` (employed to represent string data structures in the unsafe library). Lastly, the `RLBoxShadowStruct` trait is implemented by shadow structures. These sub-traits are used to provide extra functions that are specific to the corresponding types. For example, the `RLBoxVector` trait offers functions to retrieve the length of the vector, and whether the vector is has a fixed size, which is only applicable to fixed-size arrays. Similarly, the `RLBoxShadowStruct` trait offers a `shadow_size` function that returns the size of the struct in the sandbox environment in bytes, which is useful while some library APIs require the size of the struct in the sandbox environment as an argument.

The `RLBox` trait also defines a crucial function, `mem_map`, which is utilized to extract the associated `MemMap` of a value. The `MemMap` plays a pivotal role when a value is transferred between the host environment and the sandbox, by providing the address offset of each field. The ”field” here refers to only the primitive types, such as integers, floats, and pointers. If a field is not a primitive type, like a vector or a struct, the `mem_map` function will recursively call the `mem_map` of the field to retrieve the `MemMap` of the field and then concatenate the mapping of the field to the mapping of the current value.

The `MemMap` type is a tuple, `(Vec<(usize, usize, MemoryMapControl)>, usize)`, that characterizes the memory mapping relationship between the host environment and the sandbox. The first component of the tuple is a vector of triplets, each consisting of the offset in the native environment, the offset in the sandbox, and the type of the value as represented by the `MemoryMapControl` enumeration. These offsets are measured from the beginning of the value, starting at (0,0), and culminate at the end of the value, which can be calculated by the size and the ”shadow” size of the value.

The final `usize` component of the tuple signifies the number of times the last pair is repeated, with ’1’ denoting no repetition. This is particularly beneficial for the efficient memory mapping of arrays and vectors, as the memory mapping for each element within these data structures is identical. This design allows for a streamlined and efficient process of memory mapping, contributing to the overall performance and reliability of the RLBox-Rust framework.

The `MemoryMapControl` enumeration is used to indicate the type of a given field within a structure or an array, which is used to control the behavior when the data is copied between the host environment and the sandbox. It has four possible values:

1. **NoCheck**: The field is a primitive type that can be directly copied between the host and the sandbox without the need for any additional checks.

2. **Check**: The field is a primitive type that requires an additional verification step. This extra check is necessitated by potential discrepancies in size between the host and sandbox environments. For instance, the `usize` and `isize` types in 64-bit systems have a size of 64 bits, while in 32-bit systems they have a size of 32 bits.

3. **Pointer**: The field is a pointer within a structure, array, or vector.

4. **PointerArray**: The field is an array or vector comprised of pointers.

Here are a few examples illustrating the structure of `MemMap` in our implementation which uses the wasm2c sandbox as sandboxing primitive:

- For a primitive type like `i32`, the `mem_map` would be `([(0, 0, MemoryMapControl::NoCheck) ], 1)`.

- For a struct type such as `struct A { a: i32, b: i32 }`, the `mem_map` would be `([(0, 0, MemoryMapControl::NoCheck), (4, 4, MemoryMapControl::NoCheck)], 1)`.

- For a struct type such as `struct A { a: isize, b: usize }`, the `mem_map` would be `([(0, 0, MemoryMapControl::Check), (8, 4, MemoryMapControl::Check)], 1)`.

- For a vector type such as `Vec<i32>` with a length of 8, the `mem_map` would be `([(0, 0, MemoryMapControl::NoCheck)], 8)`.

Memory mapping plays a crucial role in facilitating data exchange between the host and the sandbox. It imparts the essential information that enables fine-grained control during the copying process, thereby ensuring the type safety of the data. The memory mapping information is computed on-demand and cached when a value is initially copied into the sandbox. This approach is adopted as the memory mapping remains constant within the sandbox for the entirety of a given value's lifetime.

Tainted Types

As described in Section 2.4.4, a value with the type implementing the `RLBox` trait can be copied and utilized in the sandbox. This allows it to interact with the unsafe library via the sandboxing primitives. In this section, we present the `Tainted<T>` struct, which serves to annotate a value in the sandbox. Moreover, we detail the corresponding traits, structs, and macros that facilitate the creation and manipulation of such tainted values.

We name the struct `Tainted<T>` as it is used to mark a value is stored in the sandbox. This is because that the memory region of the sandbox is manipulated by the unsafe library, which is not under the control of the developer and host application. Thus, based on the principle of Software Fault Isolation (SFI), we can not guarantee the safety and integrity of the value in the sandbox. Therefore, we mark the value as tainted to indicate that it needs to be sanitized before it can be used in the host application.

There are two types of tainted values in RLBox-Rust: `Tainted<T>` and `TaintedCallback<T>`. The former is used to mark a value that is stored in the sandbox, while the latter is used to mark a callback function that is registered in the sandbox.

The `Tainted<T>` struct incorporates two main components. The first is the `meta` field, a `TaintedMeta` struct which holding all metadata about the tainted value. This includes the value's address in the sandbox, the cached memory mapping, and the 'native' and "shadow" sizes of the value.

The `TaintedMeta` struct also implements the `Drop` trait, responsible for deallocating the value's memory in the sandbox upon the value's deallocating. This design implements a Rust-like memory control approach, eliminating the need for manual memory management.

However, this design could potentially instigate a use-after-free issue, given that the value's memory in the sandbox is released when its `TaintedMeta` is dropped, but the value remains accessible by other tainted values through pointer types.

To solve this, the `TaintedMeta` struct additionally maintains a reference map, registering all the tainted values that point to the value in the sandbox. If a tainted value A contains a pointer type that points to a tainted value B, then A will retain an `Rc<RefCell<_>>` pointer to the `TaintedMeta` of B. The `Rc<RefCell<_>>` (Reference Counting with Reference Cell) is a reference-counting pointer in Rust, which is used to track shared ownership of a value where the value will be deallocated when there are no more references to it. When B is dropped, the reference count of the `Rc` pointer is decremented. If the reference count reaches zero, the memory of B in the sandbox is finally released. This design ensures that a value's memory in the sandbox is not freed until all tainted values pointing to it are dropped.

The second one is the `phantom` field, which is used to store and indicate the type of the value. This is necessary to make the `Tainted<T>` struct generic, which allows the RLBox-Rust framework to support a wide range of data types with different data types through a set of traits to implement the function overloading. So that the developer can use the same API to do the operations on different data types and our framework can automatically select the corresponding implementation based on the type of the data.

There are two corresponding structs for the `Tainted<T>` struct, which are `TaintedRef<T>` and `TaintedMut<T>`. The former is used to mark a reference to a value in the sandbox, while the latter is used to mark a mutable reference. These two structs are useful to implement several helper functions like `index` and `index_mut` for the `Tainted<T>` if the value is a vector type. The `phantom` field of these two structs is also used to store the lifetime of the reference, which can be used to make sure that the reference is valid during the lifetime of the `TaintedRef<T>` and `TaintedMut<T>` structs.

All instances of the `Tainted<T>` struct implement the `as_ptr` method, while `Tainted<*mut T>` instances implement the `deref` method. These two methods serve to facilitate the conversion of tainted values to raw pointer types and tainted pointer types to their corresponding reference types. This is crucial in scenarios where interaction with an unsafe library API necessitates the use of raw pointers as parameters, or when such library APIs return raw pointers as results.

In order to simplify the process of field access within sandboxed structures, we have implemented a pair of complementary macros: `set_field!` and `get_field!`. The `set_field!` macro is designed to bind a given tainted value to a specified field within the structure, while `get_field!` retrieves the value associated with a specific field and then assembles a tainted value for developers to use.

Due to that the usage of `set_field!` and `get_field!` macros may not be entirely intuitive, we introduce an additional procedure macro, `op_field!`. This macro encapsulates both of these two macros, which facilitates more straightforward interactions with struct fields via the familiar dot operation. Thus, instead of using separate set and get macros, developers can directly access or modify struct fields in a more concise manner, such as `let a = op_field!(struct.field)`, or `op_field!(struct.field = a)`.

The implementation of the `op_field!` macro leverages the `syn::parse` function. By defining our custom `OpField` struct and implementing the `Parse` trait for it, the macro is able to parse the input automatically and generate corresponding `set_field!` or `get_field!` code. This design makes possible the use of the `op_field!` macro to simulate the dot operation on struct fields even the Rust programming language does not actually support the overloading of the dot operation.

The `TaintedCallback<T>` struct comprises two components: a phantom field to represent the type of the callback function, and an index field to store the index of the callback function within the sandbox's callback table. The phantom field plays an important role in static type checking when the callback function is passed as a parameter to the unsafe library API. The index field, on the other hand, facilitates retrieval of the callback function from the callback table when the callback function is invoked within the sandbox. So that the index field will be the actual parameter passed to the unsafe library API.

Value Creation and Data Exchange

The `Taint<T>` struct is equipped with a suite of static functions, facilitating the creation of tainted values and enabling data exchange between the host application and the sandbox. It offers methods such as `new_uninit` for the creation of a new uninitialized value within the sandbox, and `new`, which enables the copying of an existing value from the host application into the sandbox.

Upon completion of these initialization procedures, a `Taint<T>` struct will be generated, signifying that there is a value in the sandbox, which is marked as tainted. At the same time, all relevant metadata of this value is generated and encapsulated within the `Taint<T>` struct, in particular, within the `TaintedMeta` struct in the `meta` field. This design provides a structured and efficient approach to managing tainted data within the sandbox environment.

The process of data transfer between the host application and the sandbox is implemented with the help of the memory map, which functions as a bridge between these two distinct environments. The memory map is generated upon the creation of the `Tainted<T>` struct. The following steps outline the data copying procedure from the host application to the sandbox:

1. Firstly, we obtain the memory address of the value in the host application (referred to as the host address) and allocate a memory block of the ”shadow” size within the sandbox, retrieving its virtual address (sandbox address).

2. If the size of the value is consistent between the host application and the sandbox environment, this will indicate that all the fields in the memory map are of the `NoCheck` type. So that in this case, we can proceed by copying the value directly to the memory address

within the sandbox using its size.

3. However, if the size of the value differs between the two environments, we must iterate over the fields in the memory map. The copying process for each field is determined by its corresponding `MemoryMapControl`, and the rules are as follows:

    - For fields controlled by `NoCheck`, we directly copy the field to the sandbox.
    - For fields under `Check` control, we invoke the boundary checker within the `MemoryMapControl`. If the field is found to be out of boundary, an error is raised; otherwise, the field is copied to the sandbox.
    - For fields with a `Pointer` or `PointerArray` control, no immediate action is taken. As these pointer fields cannot be directly copied to the sandbox, they must be subsequently assigned via the `set_field!` macro.

This method ensures that all non-pointer data is safely transferred into the sandbox, maintaining correct size and alignment, with additional checks for non-aligned data. If the data is aligned between the host application and the sandbox, the data can be copied in a single step to ensure efficiency. On the other hand, for pointer data, this approach requires that developers need manually assign pointer fields utilizing the `set_field!` macro, which contains extra checks and more detailed safety guarantees.

The `Shadow` struct, as mentioned in Section 2.4.3, serves as the basis for the implementation of a helper function named `new_tainted` for all structs in the libraries. This function is tasked with generating a tainted struct value within the sandbox environment.

When dealing with fields of a non-pointer type, the function directly accepts a primitive type. Or, if the field is of pointer type, it anticipates a `Tainted<T>` type argument. Within the `new_tainted` function, all incoming `Tainted<T>` type variables are automatically bound to the respective fields of the `Tainted<T>` struct through the execution of the `set_field!` macro.

As a result, this function returns a `Tainted` struct with all fields appropriately initialized. This design provides significant convenience for developers by eliminating the need to manually initialize each pointer field of the tainted struct in the sandbox. Instead, developers can simply call this function and pass different types of parameters to it, which will be automatically bound to the corresponding fields of the tainted struct.

The process of copying data back from the sandbox into the host application is mostly similar to the one previously described. The primary difference, given our implementation choice of wasm2c as the sandboxing primitive, is that the size of the value in the sandbox is always either equal to or smaller than that of the host application. Consequently, for fields with either `Check` or `NoCheck` control, we can always directly transfer the value from the sandbox to the host application.

In general, all values within the sandbox are encapsulated in a `Tainted<T>` struct and, if required in the host application, are copied back with additional sanity checks. For values of an enum type, an initial validity check is required when they are assembled as tainted values. As a result, all `Tainted<T>` values, where T is an enum type, are wrapped as a `TaintedEnum<T>` when they are assembled, which is an alias for `Result<Tainted<T>, u32>`.

Before assembling an enum value into a tainted value, our sandbox framework initially fetches the corresponding `u32` value from the sandbox. The enum type then invokes the static `try_from` function, with the `u32` value as an argument. The `try_from` function is the implementation of `TryFrom<u32>` trait, which is automatically generated by the `Shadow` derive in RLBox-Bindgen.

If the `u32` value can be legally converted to the enum type, indicating the `u32` value is valid for this enum type, the `Result<Tainted<T>, u32>` will be converted to `Ok(Tainted<T>)`. Otherwise, if the conversion is illegal, it will be converted to `Err(u32)`, returning the illegal `u32` value. This approach ensures that enum values used in the host application are always valid for their respective enum types.

Sandbox-Accessing APIs

The sandbox-accessing APIs are implemented to provide a set of user-friendly interfaces that enable developers to engage with sandboxing primitives. These include the creation of sandboxes, the registration of callback functions within the sandbox, and the execution of functions in the sandbox. In RLBox-Rust, all specific sandboxing primitives are obligated to implement the `RLBoxSandbox` trait. This trait defines the basic operations of sandboxing primitives, which encompass memory allocation, memory free, and data exchange between the sandbox and the host application. In addition, it offers a helper function to downcast the sandboxing primitive to a specific type. This provides substantial convenience for developers when accessing the specific APIs of a given sandboxing primitive implementation.

In our design where sandbox primitives are instantiated through WebAssembly (Wasm) sandboxing and wasm2c, the `RLBoxSandbox` trait is implemented by the `RLBoxWasm2cSandbox` struct. This struct contains a `wasm2c` instance, which serves as the actual sandbox instance utilized by wasm2c to execute the unsafe libraries. It also includes a function reference map, which is responsible for storing the map between the references of callback functions and the index of this function in the sandbox function table, given that the callback function is registered within the sandbox.

When an `RLBoxWasm2cSandbox` is instantiated through the `new` method, the sandbox instance is initialized using APIs provided by wasm2c. Subsequently, the function reference map is initialized as an empty map. When registering a callback function into the sandbox, the function reference map will first check if the callback function has already been registered in the function table. If so, the register operation is bypassed, and the index of the callback function in the function table is returned for future use. If not, the callback function is registered into the function table, and a new index is added to the function reference map. This approach guarantees that the callback function is only registered once in the sandbox, thus preventing unnecessary memory usage and performance overhead.

Once all fields are initialized, the `RLBoxWasm2cSandbox` instance is returned, wrapped in a `Rc<RefCell<_>>`. The `Rc` wrapper allows for multiple references to the same sandbox instance, which is beneficial when the sandbox instance is passed into multiple `Tainted` values. The `Tainted` values are used to represent data in the sandbox field, which will be discussed in Section 2.4.4. This method ensures that the sandbox instance will not be dropped if there are still `Tainted` values referencing it, avoiding potential use-after-free scenarios that may arise when the sandbox instance is dropped before the `Tainted` values in the sandbox are still in use. The `RefCell` wrapper facilitates dynamic borrow checking of the sandbox instance during runtime, which is essential for the sandbox instances to prevent race conditions when multiple `Tainted` values access the sandbox instance simultaneously. In our evaluation, we will show that the overhead of the `Rc` and `RefCell` wrappers is negligible and does not affect the overall performance of the RLBox-Rust framework.

In addition to the functionality we mentioned above, we have also implemented several macros for the sandbox-accessing APIs to facilitate the invocation of functions provided by unsafe libraries and the registration of callback functions into the sandbox. One such macro, `sandbox_invoke`, is used specifically to invoke functions provided by these unsafe libraries. The parameters it requires include the sandbox instance, the path of the function to be invoked, and the function's arguments.

As discussed in Section 2.4.3, bindings for all functions present in unsafe library C headers have been generated, as have their corresponding functions within the sandbox primitive. It is crucial to note that all bindings generated directly from unsafe library C headers are unlinked, as we do not intend to utilize the APIs provided by the unsafe library directly. Rather, our design is such that we need to call the corresponding functions from within the sandbox primitive.

To this end, the `sandbox_invoke` macro we have designed and implemented can automatically find the corresponding function from the bindings within the unsafe library, subsequently invoking the matching function within the sandbox primitive. The advantage of this approach is that users need not concern themselves with the specific implementation details of the sandbox primitive. All they need to know is the path of the function they wish to call within the unsafe library. This design greatly simplifies usage for the user while also ensuring their safety.

When the function return, the `sandbox_invoke` macro automatically transitions the return value into a `Tainted<T>` value, leveraging the `IntoTainted` trait. This trait is implemented for various return value types, with the type of the return value being inferred via the original library bindings. The return value is then assembled into a `Tainted<T>` value using different approaches based on the return value's type.

For instance, if the return value is of a primitive type, it will be directly encapsulated within a `Tainted<T>` value. If the return value is a pointer type, a new pointer directed towards the return value will first be allocated within the sandbox, which is subsequently encapsulated within a `Tainted<*mut T>` value. As such, users can directly utilize the return `Tainted<T>` value within the sandbox field as indicated by the generic type, without the need for additional operations to handle various return value types.

To facilitate the registration of callback functions within the sandbox, we have implemented a procedural macro with an attribute, named `register_callback`, which accepts a function type as its parameter. This function type is subsequently employed to enforce static type checking on the callback function slated for registration. Based on the type information furnished by the function type argument, the `register_callback` macro generates a register function and a wrapper function for the callback function.

The register function plays a crucial role in incorporating the callback function into the sandbox's indirect function table. Within the implementation of our wasm2c sandboxing primitive, the register function initially generates a `func_type` linked to the registered callback function, which includes the counts and types of the callback function's parameters and return values. Subsequently, the register function checks the function reference map in the sandbox to determine whether the function has already been registered, returning directly if it has. If not, the register function registers the callback function's wrapper into the sandbox indirect table, adding a new entry to the function reference map. In the end, the index of the callback function along with the function type is encapsulated into a `TaintedCallback<T>` value and returned to the user. As the `TaintedCallback<T>` value is passed as an argument to the `sandbox_invoke!` macro, or incorporated into a tainted struct field via the `set_field!` macro, the RLBox-Rust framework will automatically carry out a type integrity check on the callback function, converting it to its real index within the indirect function table. So that the developers can intuitively use the callback function in the sandbox just as they would in the host application.

The wrapper function serves as an intermediary between the sandbox and the actual callback function. The key responsibility of the wrapper function is to encapsulate all raw parameters, passed from the sandbox, into `Tainted<T>` values before transferring them to the real callback function. Leveraging the callback function's type information, the wrapper function is capable of distinguishing whether an incoming parameter is a primitive or a pointer type. If it is a primitive type, the parameter is directly transformed into a `Tainted<T>` value. In the case of a pointer type, the parameter is wrapped into a `Tainted<*mut T>` value, signifying that the pointer is pointed at a `T` type value within the sandbox. Upon the return from the callback function, the wrapper function then unwraps the returned `Tainted<T>` value and passes it back to the sandbox, conforming to the appropriate raw in-sandbox parameter type. This approach ensures that the callback function can be invoked within the sandbox without any additional modifications.

2.4.5 Static and Dynamic Integrity Checks

When exchanging data between the sandbox or invoking APIs provided by unsafe libraries and the sandboxing primitive, the RLBox-Rust framework is designed to ensure type integrity and data integrity. To accomplish this, RLBox-Rust first derives type information for structs and functions from the C header files of unsafe libraries. It then maintains information of this type when exchanging data between the sandbox and the host application. Leveraging Rust's powerful type system, RLBox-Rust performs static checks for type consistency when exchanging data and invoking APIs. In situations where static checks are insufficient, such as boundary and data integrity checks, RLBox-Rust enforces fast and efficient dynamic checks during runtime. Our evaluations show that the overhead of these checks is negligible compared to the performance of Rust applications that directly use unsafe libraries.

Static-Type Integrity Checks

Static-type integrity checks are performed when the developer invokes APIs provided by Value-Accessing APIs. In particular, the static-type integrity checks are performed in 3 scenarios:

- When the developer invokes APIs provided by unsafe libraries via the `sandbox_invoke!` macro.
- When the developer sets or gets a field of a tainted struct via the `set_field!` or `get_field!`, as well as the `op_field!` macro.
- When the developer registers a callback function via the `tainted_callback!` macro.

To incorporate static-type integrity checks without requiring additional code, we have implemented a set of static assertion macros. These macros are encapsulated within a `if false` block, ensuring they are not executed and subsequently optimized away by the compiler in the release build. This approach thus eliminates any overhead in the release build. We employed three strategies to implement static type checking:

1. **Type Consistency Checking via Value Assignment:** For this strategy, we constructed a collection of static assertion macros that merely execute an assignment operation with the supplied value. The purpose of this method is to check whether the type of the left-hand side is consistent with the type of the right-hand side.
2. **Trait Checking via Generic Function Call:** In this strategy, we created a placeholder function with a generic type. This macro then invokes this function with the provided value. This method is employed to verify whether the argument type has implemented the provided trait.

3. **Type Consistency Checking via Function Call and Values:** In this approach, the macro invokes the provided function with the given values. This is done within an `if false` block, ensuring the function call is not actually executed. This approach is designed to confirm whether the argument types align with the parameter types of the supplied function.

In the RLBox-Rust framework, we've extended the functionality of static assertion macros to support a broad range of type conversions, facilitated by the application of traits. This extension is very useful as it enables the specification of permissible type conversions.

For instance, we've defined all types in the form of `Tainted<T>` to implement the `FromTainted<T>` trait. This approach allows the implicit conversion of the `Tainted<T>` type into the `T` type during static checks, thereby if the developer invokes an API that requires a `T` type argument, they can directly pass in a `Tainted<T>` type value and can still pass the static checks without any additional modifications.

Moreover, we have established that the `&Tainted<T>` type implements the `FromTainted<* const T>` trait, and the `&mut Tainted<T>` type implements the `FromTainted<*mut T>` trait. These two implementations allow mutable and immutable references to the `Tainted<T>` type to be passed into the sandbox. By employing this methodology, we leverage Rust's inherent borrowing mechanism and lifetime verification features to guarantee an enhanced level of memory safety for `Tainted<T>` values. This strategy not only improves the robustness of our system but also keeps the characteristic code style and principles of Rust, thereby preserving Rust's inherent advantages in memory safety.

As mentioned in section 2.4.4, to mark a function accessible for callback function registration, developers are required to utilize the `tainted_callback!` macro on that function. This macro necessitates the function type as an argument in the original library, thereby enforcing consistency in the function signature. Specifically, the macro incorporates a code block at the beginning of the wrapper function, generated by this macro. This particular block executes a mock function call with the designated function type and the corresponding arguments, which are derived from the arguments that will pass into the callback function. If the function signature of the callback function is inconsistent with the argument types, this mock function call will fail to compile, thereby preventing the developer from registering a callback function with an inconsistent function signature. This mechanism is similar to the static type checking `sandbox_invoke!` macro, ensuring consistency in the function signature. This approach enforce developers to maintain the consistency between the callback function registration and its associated argument types, thereby ensuring the type safety of the callback function.

Dynamic-Domain Integrity Checks

While static-type integrity checks sufficiently ensure type consistency within RLBox-Rust, they are inadequate when it comes to boundary and data integrity verifications. For instance, a value with the appropriate type is still possibly falling outside the expected range. Similarly, during operations on sandbox memory, the accessed memory may go beyond the restriction of the sandbox memory range. To mitigate these issues, RLBox-Rust employs a set of dynamic-domain integrity checks at runtime on all data interchanged between the sandbox and the host application. These checks can be categorized into three distinct types:

1. **Sandbox memory boundary checking:** Any operations upon the sandbox memory, including reading, writing, and the pointer arithmetic operations like dereferencing, are needed to be checked to ensure the accessed memory region is strictly restricted to the sandbox memory range.

   A qualified sandbox, like the WebAssembly sandbox implemented in our framework, guarantees that all APIs provided by the sandbox or unsafe library through sandbox will only access the memory region within the sandbox boundaries. Despite this assurance, it does not automatically imply that all access from the host application will be similarly confined to the sandbox memory region. Consequently, this necessitates the execution of boundary checks on all the memory access originating from the host application to ensure the accessed memory regions are always within the sandbox memory range.

2. **User-defined value domain checking:** When developers retrieve a value from the sandbox, although the type of value is assured, it may potentially exceed the expected range due to possible value corruption within the sandbox. For example, a percentage value anticipated to range from 0 to 100 could, if the sandbox is exploited, be corrupted and consequently fall outside this anticipated range, leading to unforeseen behaviors within the host application. To address this issue, RLBox-Rust implements a user-defined value domain check on all values retrieved from the sandbox. Developers are mandated to explicitly provide an anonymous checker function in the `untainted_clone` method, which will then operate on the retrieved value to confirm its compliance with the expected range. If the checker failed the verify the domain integrity of the value, the program will trigger a panic and terminate immediately, preventing further execution. This approach offers developers flexibility by enabling them to define the anticipated value range and corresponding checker function, which may differ across various library APIs.

3. **Overflow checking:** As outlined in Section 2.4.4, when a value utilizes `Check` as its memory control, an overflow check is conducted before passing the value into the sandbox. This is to mitigate the potential type size inconsistency between the host application and the sandbox. If the check is successful, the value is then transmitted to the sandbox. Otherwise, it will raise an error to prevent the overflow in the sandbox.

2.5 Evaluation

To evaluate the performance of RLBox-Rust, we conducted two sets of experiments to measure the overhead introduced by our framework and the sandboxing overhead of the WebAssembly sandbox. In the first set of experiments, we evaluated the `libjpeg` library, which is a widely used JPEG image processing library but doesn't have a rust wrapper. Thus, to use this library in Rust, the developers need to directly invoke raw C APIs. In the second set of experiments, we did the evaluation benchmarks on the `libzstd` library, which is a compression library with a rust wrapper, `zstd-rs`. We rewrite portions of the `zstd-rs` library to invoke the `libzstd` library through our RLBox-Rust framework. Then we compared the performance of the original `zstd-rs` library and the modified version.

2.5.1 Evaluation on direct C API invocation

We evaluate the performance of RLBox-Rust on the `libjpeg` library, which is a widely used JPEG image processing library but doesn't have a rust wrapper. There is an existing rust implementation of the JPEG image processing library, `jpeg-decoder`, and `jpeg-encoder`, but these two libraries don't have as much functionality as the `libjpeg` library, and the `jpeg-encoder` library is not fully compatible with the `libjpeg`.

Our evaluation targets the `libjpeg-turbo` library, an extended, high-performance fork of the original `libjpeg` library. We constructed two versions of Rust programs designed to process JPEG images using the `libjpeg-turbo` library. One version utilizes Rust FFI to directly engage the raw C APIs of the `libjpeg-turbo` library, while the other accesses the `libjpeg-turbo` library via our RLBox-Rust framework.

For our image processing tasks, we chose the 8-color bit version of the Image Compression Benchmark (ICB)[25] as our testing dataset. The ICB includes a collection of 14 images, varying in size and content, encoded in the ppm format. Utilizing this dataset, we encoded the images into JPEG format using the `libjpeg-turbo` library, adjusting the quality factors from 0 to 100 in increments of 10. Following this, we decoded back the encoded images to the ppm format using the same library. This encoding-decoding process was repeated 10 times to calculate an average time for both processes to reduce the impact of random noise.

We conducted these experiments on both the original version of the `libjpeg-turbo` library and the RLBox-Rust wrapped version. The resulting data is presented in Figure 2.8.

In Figure 2.8, the x-axis represents the quality factor of the JPEG image in encoding, and the y-axis represents the overhead percentage of the encoding and decoding process. The percentage means the ratio of the execution time of the RLBox-Rust wrapped version to the

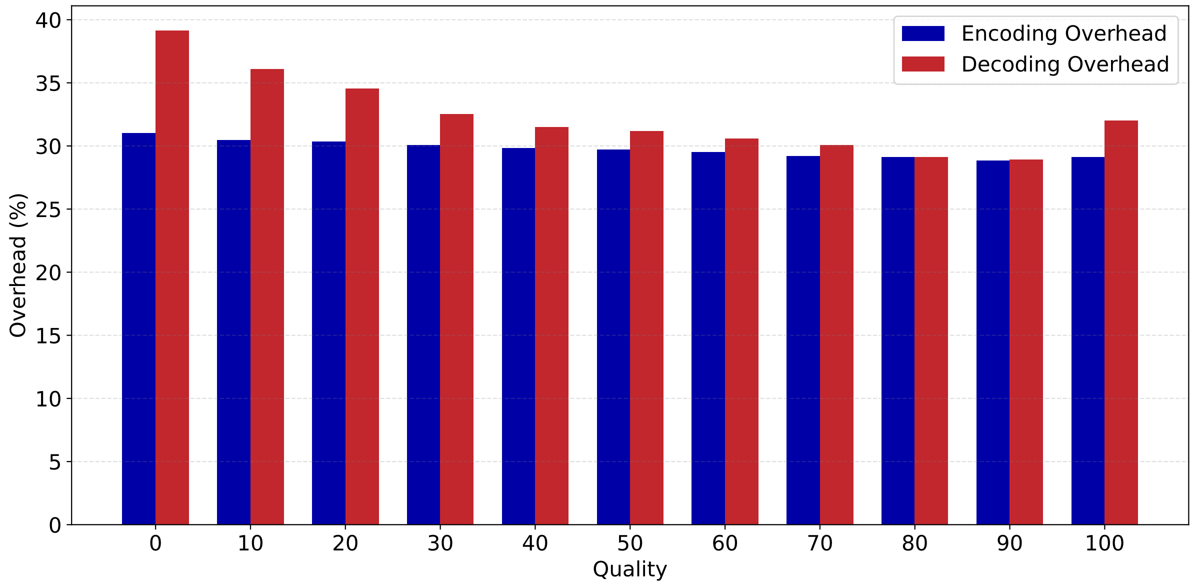

Figure 2.8. Overhead of `libjpeg-turbo` Encoding and Decoding in different qualities

original version.

To analyze the results, we calculated several metrics, including the average overhead, the maximum and minimum overhead, and the standard deviation of the overhead, as shown in Table 2.1.

Table 2.1. Metrics of the overhead of `libjpeg-turbo` Encoding and Decoding

	Average	Maximum	Minimum	Standard Deviation
Encoding	29.75%	31.03%	28.83%	0.65%
Decoding	32.34%	39.14%	28.92%	2.99%

From the results, we can see that the average overhead of the encoding process and decoding process are both around 30%, and the maximum overhead of these two processes is less than 40%. Furthermore, the standard deviation of these two processes is relatively small, which means the overhead is consistent across different quality factors. This result also indicates that the overhead introduced by our framework is relatively stable and predictable.

2.5.2 Evaluation on Rust wrapper library

Another set of experiments is to evaluate the performance of RLBox-Rust on the existing rust wrapper, `zstd-rs`, the wrapper of the `libzstd` library. `libzstd` is a compression library that provides fast compression and decompression speed. In the Rust ecosystem, developers will prefer to use the `zstd-rs` wrapper to access the `libzstd` library as it provides a more idiomatic Rust interface.

To evaluate the performance of RLBox-Rust on the `zstd-rs` library, we need to rewrite portions of the implementation of the `zstd-rs` library to invoke the `libzstd` library through our RLBox-Rust framework. In detail, we need to recompile the `libzstd` library into the WebAssembly module and then replace all the functions invocation of `libzstd` in the `zstd-rs` with the APIs provided by our RLBox-Rust framework.

In the original `zstd-rs` library, `libzstd` is compiled with support for Single Instruction Multiple Data (SIMD). SIMD technology leverages the parallel processing capabilities of modern CPUs to accelerate program execution. However, SIMD instructions vary across different CPU architectures. For instance, the SIMD instructions for the x86 architecture include AVX, SSE, and MMX, among others. These instructions are not compatible with WebAssembly, resulting in the loss of SIMD support when `libzstd` is directly compiled into WebAssembly, since `libzstd` does not natively provide WebAssembly-compatible SIMD instructions like simd128.

To address this compatibility issue, we utilize SIMD Everywhere (SIMDe), a header-only library that facilitates the translation of SIMD instructions across differing CPU architectures. We introduced some minor modifications to the `libzstd` library, replacing the SIMD headers with SIMDe headers. Through these modifications, we recompiled the adjusted `libzstd` library into the WebAssembly module, using SIMDe to translate x86 SIMD instructions into WebAssembly-compatible SIMD instructions. This approach keeps SIMD support within the WebAssembly environment, so that it can reduce the overhead introduced by the WebAssembly sandboxing.

Our evaluation employed the `zstd-rs` library and utilized its provided benchmark. This benchmark involves the compression and decompression of the Silesia compression corpus, a comprehensive collection of 12 files of different types and sizes, including English text, executable files, PDFs, and more[1]. The benchmark process compresses all files using the `zstd-rs` library with various compression levels from 1 to 20 , and subsequently decompresses the compressed files back to their original form. This benchmark provides a thorough and representative test of the `zstd-rs` library's performance and the performance of our rewritten RLBox-Rust version across a range of file types and compression levels, thereby offering a comprehensive evaluation of our framework.

We deploy our benchmark both on the original `zstd-rs` library and our rewritten RLBox-Rust version. The resulting data is presented in Figure 2.9.

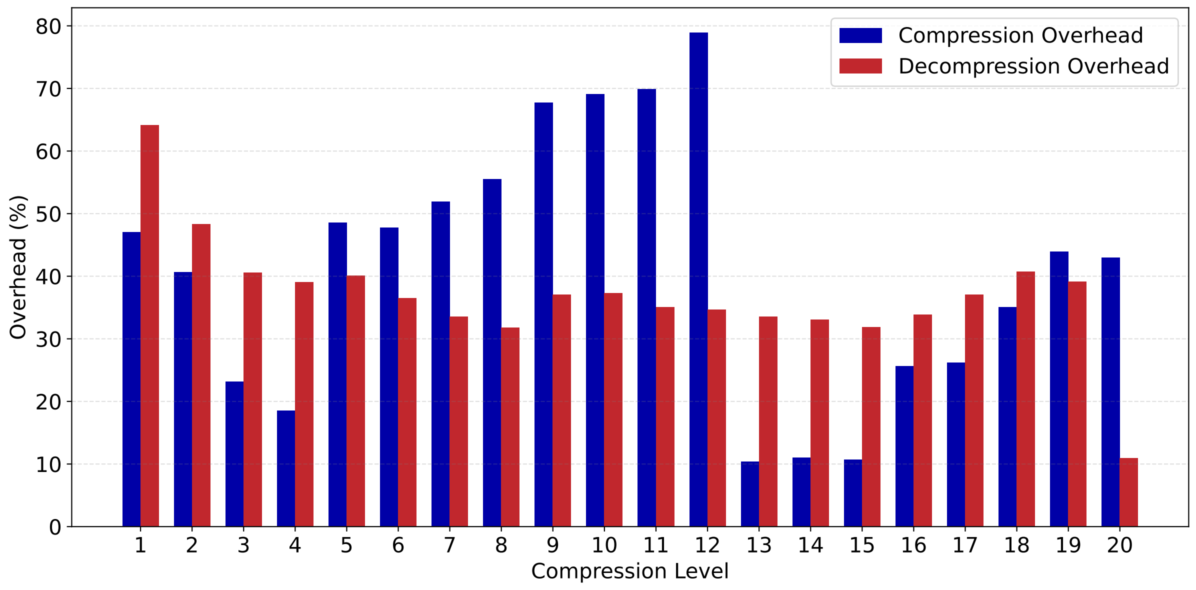

Figure 2.9. Overhead in Compression and Decompression Speeds in different Compression levels

In Figure 2.9, the x-axis represents the compression level of the `zstd-rs` library, and the y-axis represents the overhead percentage of the compression and decompression process.

Samely, we calculated several metrics, including the average overhead, the maximum and minimum overhead, and the standard deviation of the overhead to analyze the results, as shown in Table 2.2.

Table 2.2. Metrics of the overhead in Compression and Decompression of `zstd-rs`

	Average	Maximum	Minimum	Standard Deviation
Compression	41.25%	78.94%	10.40%	20.40%
Decompression	36.91%	64.12%	10.96%	9.22%

The analysis of the results reveals that both the compression and decompression processes incur an average overhead of approximately 40%. Moreover, the maximum overhead observed in the compression process escalates to roughly 80%, which is considerably higher than the average overhead. Furthermore, the standard deviation for both processes is markedly larger in comparison to the benchmark set by `libjpeg-turbo`. This suggests a degree of inconsistency in overhead across varying compression levels, potentially introducing less predictability in the performance of the processes.

Our preliminary analysis of the results suggests that the invocation paths within the `libzstd` library display a degree of variability corresponding to different compression levels. This variability manifests as differing overheads when comparing the original native x86 version and the WebAssembly version implemented via our RLBox-Rust framework. Additionally, the compatibility layer of SIMDe introduces its own overhead to the overall performance of both the compression and decompression processes. Consequently, this overhead also exhibits variability across different compression levels, further contributing to the complexity of the performance in the evaluation.

Chapter 3

Conclusion and Future Work

3.1 Conclusion

In this thesis, we present the RLBox-Rust framework. We show that the RLBox-Rust is a practical and effective approach to sandboxing the unsafe C libraries used by Rust programs. In particular, we show that RLBox-Rust can provide the following benefits:

- **Strong safety assurance**: RLBox-Rust utilize the sandboxing primitive provided by the WebAssembly to provide strong isolation between the unsafe C libraries and the Rust program. Moreover, RLBox-Rust provides a set of static and dynamic safety checks to add an additional safety guarantee to the sandboxed libraries. These safety assurances effectively prevent the vulnerabilities in unsafe C libraries from propagating to the Rust program and other sandboxed libraries.

- **Easy migration**: RLBox-Rust utilize the powerful macro system in Rust to provide an intuitive and easy-to-use API for invoking the sandboxed libraries. This API is designed to be similar to the API provided by the direct bindings to the unsafe C libraries. Therefore, developers can easily migrate their existing Rust programs to use RLBox-Rust from using the direct bindings.

- **High performance**: RLBox-Rust utilizes the WebAssembly and wasm2c toolchain to provide a high-performance sandboxing solution. Our evaluation shows that the overhead introduced by RLBox-Rust will only be less than 1.8x slower than the direct bindings to the unsafe C libraries.

3.2 Future Work

We plan to broaden the capabilities of RLBox-Rust by integrating support for different kinds of sandboxing primitives. WebAssembly is a purely software-based solution, which offers a promising sandboxing primitive. While such software-based solutions are lightweight and straightforward to deploy across various CPU architectures and environments, they typically can not achieve the same performance as hardware-based solutions.

Thus, our future effort will aim to incorporate support for additional sandboxing primitives that leverage specific hardware features, such as Intel's Memory Protection Keys (MPK). Such adaptations are expected to significantly improve performance on specific CPU architectures. Through this work, we hope to provide multiple sandboxing primitives for RLBox-Rust, allowing developers to choose the most suitable sandboxing primitive for the specific use case they expect to deploy.

Acknowledgments

The entire thesis, in full, is currently being prepared for submission for publication of the material. Zhou, Tianyang; Narayan, Shravan; Stanford, Caleb; Jhala, Ranjit; Garfinkel, Tal; Stefan, Deian. The thesis author was the primary investigator and author of this material.

Bibliography

[1] Silesia compression corpus. https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia.

[2] A technical look at intel's control-flow enforcement technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[3] Hussain MJ Almohri and David Evans. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 248–255, 2018.

[4] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. Trust: A compilation framework for in-process isolation to protect safe rust against untrusted code.

[5] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*, pages 30–40, 2011.

[6] J Efrim Boritz. Is practitioners' views on core concepts of information integrity. *International Journal of Accounting Information Systems*, 6(4):260–279, 2005.

[7] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[8] Common Weakness Enumeration. CWE-843. https://cwe.mitre.org/data/definitions/843.html, 2011.

[9] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[10] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Safedispatch: Securing c++ virtual calls from memory corruption attacks. In *NDSS*, 2014.

[11] Trevor Jim, Dan Grossman, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c.

[12] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pages 51–57, 2017.

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

[14] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 941–951, 2015.

[15] Microsoft Security Response Center. A proactive approach to more secure code. https://www.microsoft.com/security/blog/2019/07/18/a-proactive-approach-to-more-secure-code/, 2019.

[16] Mozilla. WebAssembly. https://webassembly.org/, 2023.

[17] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[18] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, pages 31–40, 2010.

[19] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 699–716, 2020.

[20] National Vulnerability Database. CVE-2019-11922. https://nvd.nist.gov/vuln/detail/cve-2019-11922, 2021.

[21] National Vulnerability Database. CVE-2021-3156. https://nvd.nist.gov/vuln/detail/cve-2021-3156, 2021.

[22] National Vulnerability Database. CVE-2021-45713. https://nvd.nist.gov/vuln/detail/cve-2021-45713, 2021.

[23] National Vulnerability Database. CVE-2021-45719. https://nvd.nist.gov/vuln/detail/cve-2021-45719, 2021.

[24] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, 2002.

[25] Rawzor. Image compression benchmark. http://imagecompression.info/.

[26] Rusqlite. Issue # 1048: screwy update_hook lifetime raises SIGSEGV. https://github.com/rusqlite/rusqlite/issues/1048, 2021.

[27] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[28] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[29] WebAssembly. wasm2c: Convert wasm files to c source and header. https://github.com/WebAssembly/wabt/tree/main/wasm2c, 2023.

[30] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.