# RusTEE: Developing Memory-Safe ARM TrustZone Applications

Shengye Wan
The College of William & Mary
George Mason University
swan@email.wm.edu

Mingshen Sun
Baidu Security
sunmingshen@baidu.com

Kun Sun
George Mason University
ksun3@gmu.edu

Ning Zhang
Washington University in St. Louis
zhang.ning@wustl.edu

Xu He
George Mason University
xhe6@gmu.edu

## ABSTRACT

In the past decade, Trusted Execution Environment (TEE) provided by ARM TrustZone is becoming one of the primary techniques for enhancing the security of mobile devices. The isolation enforced by TrustZone can protect the trusted applications running in the TEE against malicious software in the untrusted rich execution environment (REE). However, TrustZone cannot completely prevent vulnerabilities in trusted applications residing in the TEE, which can then be used to attack other trusted applications or even the trusted OS. Previously, a number of memory corruption vulnerabilities have been reported on different TAs, which are written in memory-unsafe languages like C.

Recently, various memory-safe programming languages have emerged to mitigate the prevalent memory corruption bugs. In this paper, we propose RusTEE, a trusted application mechanism that leverages Rust, a newly emerged memory-safe language, to enhance the security of TAs. Though the high-level idea is quite straight-forwarding, we resolve several challenges on adopting Rust in mobile TEEs. Specifically, since Rust currently does not support any TrustZone-assisted TEE systems, we extend the existing Rust compiler for providing such support. Also, we apply comprehensive security mechanisms to resolve two security issues of trusted applications, namely, securely invoking high-privileged system services and securely communicating with untrusted REE. We implement a prototype of RusTEE as the trusted applications' SDK, which supports both emulator and real hardware devices. The experiment shows that RusTEE can compile applications with close-to-C performance on the evaluated platforms.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Mobile platform security*.

## KEYWORDS

TrustZone, Rust, Trusted Applications, Memory-safety

**ACM Reference Format:**
Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. 2020. RusTEE: Developing Memory-Safe ARM TrustZone Applications. In *ACSAC '20: Annual Computer Security Applications Conference, December 07–11, 2020, Online.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ACSAC '20, December 07–11, 2020, Online*
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

In recent years, TrustZone has been leveraged extensively to provide security protection on the ARM platforms [4, 8, 26, 43, 52]. It enables system-wide isolation by creating a Trusted Execution Environment (TEE) for security-sensitive code and data protection and therefore protects the TEE's software from the untrusted Rich Execution Environment (REE). The isolation is enforced via the hardware features built in the processor as well as the system bus interconnect. Due to the protection of hardware-assisted isolation, it becomes common for TrustZone-based systems [4, 8, 47] to assume the trust of entire TEE, including the trusted applications (TAs) running in the TEE. Also, the functionalities of TEE systems are extended dramatically by installing various TAs in the trusted isolated environment.

Though TrustZone technology can assure isolation between TEE and REE, dozens of software-based vulnerabilities have been reported to compromise the entire TEE system [13, 22, 51]. Among the reported vulnerabilities, most of them are caused by memory corruption of the memory-unsafe TAs [9]. The risk of TEE systems being compromised will increase along with the number of TAs installed. Under the latest ARM TrustZone architecture, the term "Trusted Application" only refers to an application that should be trusted to run in TEE, but it does not mean the application is bug-free. Due to two architectural features of TAs, namely, conducting the cross-world communication with the REE and invoking kernel-privileged system-service APIs, TAs could be manipulated by REE-side attackers to compromise the entire TEE system. Researchers propose to move the execution of TAs from the TEE to the REE and thus prevent one vulnerable TA from corrupting other TAs or the Trusted OS [8, 10, 43]. Though these solutions can effectively mitigate the risk of vulnerable TAs, they will inevitably introduce non-negligible overhead over the system.

Recently, many programming languages focus effort on enhancing their memory-safety, and several new languages are proposed with memory-safety as one of the goals, such as Rust and Go. Meanwhile, researchers have applied the memory-safe languages from upper application layer (e.g., Intel SGX Enclave programs [48]) to lower system layer (e.g., embedded system OSes [30, 31]). One precondition to the engineering effort to rewrite the code base in these memory-safe languages is relatively small, so that developers can afford to convert the existing software into the memory-safe style. Meanwhile, since ARM TrustZone is proposed to protect a limited number of small security tasks, TAs become another ideal target to be rewritten in the memory-safe language.

In this paper, we propose a mechanism called RusTEE to build TrustZone-assisted applications in the memory-safe style, using Rust [35] as the programming language. The basic idea is to leverage newly emerging memory-safe languages and provide a Rust-based Software Development Kit (SDK) on compiling memory-safe TAs to prevent against memory-corruption vulnerabilities. Specifically, we resolve several challenges to develop a TA with Rust. The first challenge is that none of TrustZone-assisted TEE system and associated ARM platform has been recognized as the official support target to the Rust. Therefore, we need to integrate all the Rust fundamental support such as the standard library into the TA development. Second, TAs are required to invoke the APIs of different system services, which are typically implemented as the kernel-privileged libraries. Since some low-level libraries require specific ARM assembly instructions that are not supported in Rust, it is impractical to rewrite all the libraries in Rust. Inspired by a recent work Rust-SGX [48], we solve this challenge by providing a binding layer between the Rust application and C system. The binding provides all the necessary interfaces for the TA dependent libraries while also enforcing the Rust's memory-safe standard on the bounded interfaces. Third, we resolve a TA-specific challenge, i.e., providing a secure cross-world communication channel for the TA in the TEE world to communicate with the software in the REE world. The security of the cross-world communication is ensured by regulating the TA's usage on any shared parameters between the two worlds.

After systematically studying the architectural specification of TrustZone-assisted systems, we successfully import Rust into TA development environment, and further apply multiple security enhancements to reliably invoke system-service APIs and securely conduct the cross-world communication. We develop a prototype of RusTEE based on an open-source project OP-TEE OS [34] and provide a variety of examples to demonstrate the functionalities and efficiency of RusTEE. We have open sourced the RusTEE prototype along with the memory-safe TA examples. The system evaluation has been conducted on multiple ARM platforms, including the AArch64 simulation and a real-world development board Juno r1 [3]. According to our experimental results, RusTEE only introduces 1% performance overhead on average on the evaluated examples. Moreover, RusTEE enables the TAs to be integrated with millions of existing Rust libraries, noticeably extending the functionalities of the TAs in the TEE.

In summary, we make the following contributions.

(1) We propose RusTEE, the first memory-safe trusted application development environment with comprehensive functionalities for TrustZone-assisted systems. By utilizing the built-in security properties and benefits of the Rust programming language, our trusted application environment removes most known memory-unsafe implementation bugs in trusted applications and thus enhance the security of TEE.
(2) We address two security concerns of the TrustZone-assisted TEE systems, namely, the widely exposed system-service APIs and cross-world communication channels, to enhance the security of Rust-based trusted applications.
(3) We implement a prototype of RusTEE and evaluate its performance in both a simulation environment and a real development board. Our experimental results show that our system can comply with strictly safe Rust, and it only incurs a minimal overhead. We will open source the system prototype.

The rest of the paper is organized as follows. Section 2 provides the background of ARM TrustZone architecture and programming language Rust. Section 3 presents the motivation associated with the specific challenges of this work. Section 4 describes the overview and detailed design of RusTEE, and the implementation of RusTEE's prototype is presented in Section 5. Section 6 first evaluates the efficiency of Rust on the ARM platform, and then compares the performance of Rust-based TAs with the traditional C-based versions. Section 7 discusses the future directions of our work. Section 8 surveys the related works. Finally, Section 9 concludes the paper.

# 2 BACKGROUND

## 2.1 TrustZone Architecture

Adapting *Trusted Execution Environment (TEE)* has become a popular method for system developers to protect their security-sensitive software. To provide a reliable hardware-assisted TEE, ARM deploys TrustZone technology on its most recent application processors. TrustZone creates the TEE as an isolated environment that runs opposite to the vulnerable and untrusted *Rich Execution Environment (REE)*. From the hardware perspective, ARM relies on its AMBA BUS feature [2] to divide the entire System-on-Chip resources into two worlds, where the normal world serves as the REE, and the secure world serves as the TEE.

From the software perspective, ARM website [2] recognizes *GlobalPlatform TEE Specification (aka, GPD specification)* [20] as a widely used TEE architecture on the latest ARM processors. The GPD specification defines a clear security boundary for TrustZone-assisted TEE systems by providing a completed set of software definitions between REE and TEE. Currently, multiple real-world TEE systems, such as Linaro OP-TEE [34] and Trustonic Application Protection Solution [45], apply the design of GPD specification into their implementations.

## 2.2 GlobalPlatform TEE Specification

According to the GPD specification, an REE hosts the rich OS (e.g., Android, Linux) in association with the user-privileged applications. While most applications are deployed and used entirely in REE as normal applications, some security-sensitive applications can enable the TrustZone protection on their sensitive operations.

A security-sensitive application divides itself into two components, an REE-side component called *Client Application (CA)* and a TEE-side component called *Trusted Application (TA)*. The CA supports most non-sensitive functionalities like user interactions; however, neither the counterpart TA nor the TEE trusts the CA. Meanwhile, all sensitive operations are isolated as the TA, which usually runs on a Trusted OS inside the TEE. By leveraging TrustZone hardware-assisted isolation, the confidentiality and integrity of TAs are protected from the untrusted REE. The entire GlobalPlatform Architecture for a TrustZone-assisted device is shown in Figure 1.

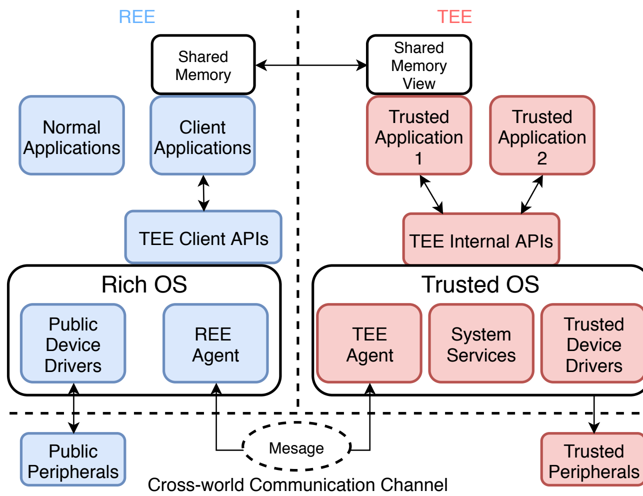


**Figure 1: GlobalPlatform TEE Architecture**

Since the CA and the TA run in two isolated environments, they perform cross-world communication in reliance upon an *REE Agent* and a *TEE Agent* for passing a command or exchanging the data. To request the trusted execution of a TA, the CA calls the *TEE Client APIs* [18] to ask the REE agent to send out the *Message* and build up the cross-world communication channel with a specific TA. Once the TEE Agent receives the Message, it initializes the corresponding TA to respond to incoming REE-side commands. The related responding APIs are defined as Cross-world Communication Channel APIs that belong to *TEE Internal APIs* [19]. To exchange data between two environments, the CA first allocates the communication memory called *Shared Memory* in the REE and then shares the memory with the corresponding TA. Since the TEE has a higher privilege on accessing the REE's memory, the TA can also operate on the shared memory in parallel with the CA.

Besides the communication functions, GlobalPlatform also defines its TEE Internal APIs to provide essential *System Services*, such as cryptography-related operation, secure storage, and big-number calculation. Since all TEE Internal APIs are provided to all TAs for calling directly, TAs are not required further to implement their own functionalities for these security services. Moreover, many of the GPD TEE Internal APIs are involved with dedicated memory-related operations, which should be thoroughly inspected before running them inside TEE.

## 2.3 Rust

Rust [35] is a programming language designed to achieve both reliability and efficiency. To achieve reliability in two distinct aspects, namely, memory-safety and thread-safety, Rust provides the following mechanisms: (1) claiming the *ownership* of each data object; (2) automatically checking the read/write permissions (*mutability*) of each object; (3) enforcing the *lifetime* managements on all objects; (4) forbidding unsafe typecasting (*type-safety*); (5) disabling dangerous *raw pointer operations* like pointer aliasing or dangling pointers. During the program compilation, if the code violates any Rust's security criteria, the Rust compiler raises errors and generates error messages to help developers correct their code accordingly. Besides improving the code security, Rust brings other benefits such as the highly efficient parallelization, the developer-friendly compiling messages, and thousands of *crates* (similar to the libraries in C language) for supporting different development requirements.

**Rust-safe vs. Rust-unsafe**. Though Rust is designed to achieve strict security criteria by default, to guarantee any program can indeed be written in Rust, it also provides the keyword `unsafe` [44] for developers to inject memory-unsafe code segments. Rust provides this unsafe option for two primary reasons: 1) allowing developers to develop some "special" functions the cannot pass the compiler's default inspection; and 2) allowing the code to interact with system/hardware components directly. A segment marked as unsafe can bypass the Rust built-in check and therefore may conduct vulnerable behaviors, such as writing on an immutable variable, conducting a non-standard typecasting, or using raw pointers directly. A typical scenario of using unsafe code segment in Rust happens when the Rust code has to invoke the C-based functions, which is defined as *Foreign Function Interface (FFI)* in Rust. Coming with the advantages of extended capabilities, unsafe Rust also introduces security risks. Several related works [6, 7, 44, 46] have revealed that unsafe Rust can introduce potential security risks.

# 3 MOTIVATION AND CHALLENGES

## 3.1 Motivation

Over the past decades, more than one hundred vulnerabilities have been reported for TrustZone-assisted TEE systems [11–13]. Among these reported vulnerabilities, most of them are software-related, which means the vulnerabilities can get exploited even if the device enables and configures TrustZone hardware components appropriately. Recently, Cerdeira et al. [9] provide a systematized summary about the vulnerabilities of existing TEE systems, and they summarize the software-related vulnerabilities in two categories, namely *implementation issues* and *architectural issues*. The implementation issues refer to the bugs triggered by specific implementation details of one TEE system, such as lacking proper security checks on the sensitive variables. Meanwhile, architectural issues include shared deficiencies or design flaws among different TEE systems, regardless of systems' implementation details.

In order to mitigate software-related vulnerabilities on TrustZone-assisted TEE systems, one critical and challenging task is enhancing the security of TAs. Nowadays, commercial TEE systems integrate more and more TA functionalities into the TEE, excessively increasing the total size and semantic complexity of the TEE. With such a large number of complicated TAs, it is impractical for the

TEE system's administrator to conduct either artificial or automatic validation on each TA's correctness. Consequently, TAs may get imported into the TEE with potential implementation issues, such as conducting sensitive operations without appropriate validations. Moreover, when TAs are developed in memory-unsafe languages like C language, these implementation issues are difficult to be fully reviewed since a memory-unsafe language can perform dangerous memory operations and cause implementation issues with many possibilities.

Besides introducing implementation issues, TA is also the critical component of two TrustZone-specific architectural issues. First, the TA's capability of invoking kernel-privileged system services can be abused to attack the TrustZone-assisted TEE system and even lead to a compromised TEE. To support the incremental functionalities of TAs, Trusted OSes deploy many system services and expose wide interfaces to TAs; however, there is no security regulation on the interactions between TAs and the Trusted OSes. Therefore, if the vulnerable TAs can be manipulated to invoke system interfaces maliciously, the entire mobile system may be compromised as well. How to govern the interface between the Trusted OS and TAs is an essential architectural challenge when deploying TEE systems. Second, most TEE systems allow TAs to accept input from the REE via the cross-world communication channel. However, since the REE is untrusted and may be fully controlled by attackers, the cross-world communication channel expands the attack surface of the TEE system.

In real-world scenarios, when both the implementation and architectural issues exist in a single TA, they may be exploited together and lead to severe consequences. For instance, a recently reported vulnerability CVE-2018-14491 [22] utilizes a vulnerable One-Time-Password TA for executing arbitrary code on Samsung S5 smart phones. Similar security issues have been reported in other CVEs such as CVE-2015-6639 [11] and CVE-2016-2431 [12]. Motivated by resolving both implementation and architectural issues, we propose to implement TAs in a strict memory-safe style and further mitigate the identified issues of TAs. In the following section, we present three particular challenges and our basic ideas for solving them.

## 3.2 Challenges

The primary object of RusTEE is to provide a secure mechanism that assists developers in building TAs with a memory-safe regulation. Specifically, there are three main challenges we need to resolve to build the required secure TAs.

*Challenge-1: Tackling memory corruptions in TA.* One fundamental attribute of a secure TA is that the TA does not contain any memory-unsafe implementation issues. In other words, our method should ensure to remove memory corruptions from TAs, such as Use-After-Free or Data Race. To address this problem, we propose to write TAs in the memory-safe programming language Rust.

*Challenge-2: Providing secure system-service APIs.* Unlike some TEE architecture (e.g., SGX) that can provide multiple hardware-enforced-isolated enclaves, the TrustZone-assisted TEE system only deploys one shared Trusted OS for executing all TAs. Therefore, any compromised TA may utilize the widely provided system-service APIs to attack the shared Trusted OS and compromise all other TAs. In order to eliminate the side-effect of exposing wide APIs to TAs, we provide a binding solution that enforces the Rust's memory-safety on the existing unsafe APIs to prevent TAs form misusing any kernel-privileged TEE system services.

*Challenge-3: Building protection on cross-world communication.* As an architectural feature of TEE systems, the cross-world communication channel is a must to support the collaboration between TEE and REE. However, this channel also provides another vehicle for the REE-side attackers to manipulate TAs' behavior, especially considering that the communication channel is connected via the untrusted REE's memory. To enhance the security of the cross-world communication channel, we redesign the cross-communication interfaces of TA, which conduct security checks on the passed-in parameters and limit the use cases of untrusted parameters.

# 4 SYSTEM DESIGN

In this section, we first present the threat model and overview architecture of RusTEE. Then we elaborate on the detailed security enhancements of RusTEE for resolving TA's security challenges.

## 4.1 Assumptions and Threat Model

We assume the device is equipped with ARM TrustZone technology, and the technology is can provide the hardware-enforced isolation. We assume all TEE system's software components, including the secure monitor, Trusted OS, and all TEE kernel-privileged libraries, are implemented in compliance with the GlobalPlatform TEE specification. In this case, TAs use the GlobalPlatform-defined (GPD-defined) APIs to interact with system services and the cross-world communication channel. We also assume these system components are well written, so there is no insecure flaw in Trusted OS or lower level software. As such, we focus on protecting the memory-safety of TAs that run above Trusted OS. Finally, we assume the TA developers are benign while he or she may still program a TA in a vulnerable way, which is a common scenario recognized in the recent CVEs [11–13].

## 4.2 Overview

We present the overview architecture of RusTEE in Figure 2. The main idea of RusTEE is serving as a Rust-based TA SDK in the TEE. The SDK supports most general development requirements, such as operating primitive data-types, in the strict Rust-safe style by providing Rust standard library and associated essential components to TA developers. With the assistance of the Rust compiler's built-in security checks, RusTEE ensures the TA's source code is free of known memory-corruption bugs and therefore mitigates *Challenge-1*. Since the major challenges for porting Rust standard library into ARM platforms are implementation-related, we will introduce them later in the Implementation Section 5.

Besides performing general-purpose operations, a TA also needs to invoke functions of particular TEE's system services, which are out of the scope of Rust standard library. Therefore, RusTEE integrates the extra libraries into SDK to support these requirements. There are two design options for shipping a Rust-based SDK with additional libraries. The first option is rewriting all the requested libraries in Rust. The other option is building up the Rust-based SDK based on full-fledged C-based libraries, and further providing a trustworthy binding between Rust and C components.

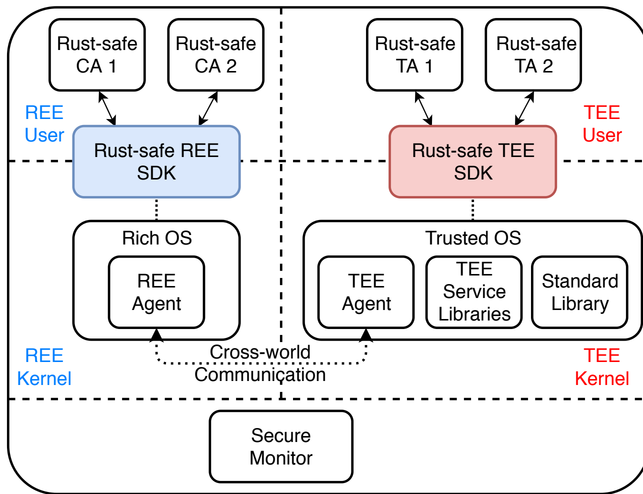


**Figure 2: RusTEE Architecture**

Though the first option offers better independence and memory-safety, it faces two non-trivial challenges when implemented on the ARM TrustZone-assisted platforms. The first challenge is that some TEE's system services involve the TrustZone-specific operations (e.g., reading a secure timer), while these operations can only be implemented with the explicit essential ARM instructions that are unavailable in the Rust's standard supports. Another challenge is that for some TEE's system services (e.g., cryptography), the C-based libraries have better performance than the Rust ones. In consideration of these challenges, we propose to provide the SDK as the binding solution. After systematized studying all critical data structure and function definitions of these additionally involved libraries, RusTEE converts all the interfaces into Rust-safe style to resolve the *Challenge-2*.

Meanwhile, TAs used to face the challenge of handling the commands and parameters that are passed-in via the cross-world communication channel, since these data are generated by the untrusted REE. By carefully reviewing the calling convention of existing cross-world communications, we redesign the connection interface between the TEE's system communication component (TEE Agent) and TAs. The redesigned communication interfaces promise that all parameters are used under secure standards and therefore handle *Challenge-3*.

Finally, RusTEE provides the REE-side SDK, which follows a similar scheme of the TEE-side SDK, as a complementary component to regulate the behaviors of CAs. Note that the security of TA does not depend on whether the REE utilizes the REE-side SDK or not, and the REE-side SDK is provided only in the case that benign CA developers want to improve a CA's memory-safety. In the following section, we focus on presenting our methodology for mitigating the architectural issues for the Rust-based TAs, particularly, securing the widely exposed system-service APIs (hereinafter referred to as "service APIs") and cross-world communication channel.

## 4.3 Secure System-service APIs

In the design of RusTEE, the Trusted OS implements TEE's system services as the C-based libraries for the best practicality, and the OS provides C-based service APIs to the upper-layer applications. To make these APIs available for Rust-based TAs, RusTEE should reliably convert these C-based interfaces into the Rust-based interfaces. We call this conversion as the binding solution. To bridge the semantic gap between Rust and C language, Rust officially provides a standard crate `std::libc`, which matches all data types and structures that are shared by two languages, such as `c_int` and `c_char`. Also, Rust provides the *Foreign Function Interface (FFI)* mechanism to allow Rust-based programs for invoking C-based functions in the Rust-unsafe way. By utilizing these two Rust's components, we can straightforwardly convert the C-based interfaces as the Rust-unsafe interfaces via FFI mechanism, and allow the upper-level TAs to interact with the low-level APIs via the parameters that are matched by `std::libc`.

However, the FFI-based bindings are not memory-safe for TAs to invoke. As we explained in the Background Section 2.3, since Rust's built-in security checks ignore any code segment marked as FFI, the bonded APIs can still contain memory-unsafe vulnerabilities. To ensure the security of these bindings, RusTEE applies multiple security-enhancements on the service APIs. In this subsection, we first introduce four general principles that are adapted as the enhancements for all bonded C-based service APIs. Then we present two particular binding principles that we propose for protecting GPD-defined service APIs.

**Secure C-based APIs.** As one close-related work of RusTEE, Rust-SGX [48] provides a secure binding for Intel SGX between Rust enclave applications and C-based SDK. More importantly, the authors conclude two common challenges for providing binding between Rust and C worlds, which are providing safe memory access of C/C++ objects and raw-bytes. The first challenge is introduced for achieving the type-safety in Rust. Ideally, every type in the Rust program has a precise definition for providing clear semantics about types' use cases. Moreover, an explicit type definition can describe all the legitimate scenarios for casting one type to another. However, in C-based libraries, many complicated data types can only refer to a pointer type `void`, and the pointers can be dangerously accessed with the wrong interfaces when the developer uses them carelessly or confused. The second challenge happens when C-based libraries access the memory chunks directly based on their pointer and length, which is considering as unsafe and not-allowed in Rust. Such pointer/length combinations frequently appear in C-based libraries.

To resolve these two challenges, Rust-SGX defines four principles, which notated as `Bytes`, `ContiguousMemory`, `Sanitizable[T]`, and `Handle`$_\tau$. These four notations can regulate how to convert the challenging C-style APIs into Rust-safe style. Specifically, *O=3;4*$_\tau$ maps each C-based unsafe pointer into specific secure type in Rust. *~C4B* constructs the concrete memory in the format of arrays for securing memory accesses. The rest two notations *>=C86D>DB" 4<>A~* and *(O=8C8I01;4[])* are provided for handling the conversion between other unsafe C-based types T and the proposed *~C4B*. Moreover, Rust-SGX provides solid formalization to prove the four notations' security with the system defined in CCured [38].

Since RusTEE shares a similar binding solution as Rust-SGX, we adopt all four principles proposed in Rust-SGX. For example, we provide a specific handler for each critical data type. We further realize the other three security principles to bind the service APIs securely. Similar to the solution of Rust-SGX, the realizations of these principles require manual effort to review all libraries' critical data structures and understand the associated memory utilization. To the best of our knowledge, there is no automatic mechanism that can promise a perfect conversion from C-based APIs to Rust-based ones. Hence, we claim such a manual process is acceptable and has the most reliable security-promise for the bonded APIs.

**Secure GPD-defined APIs**. After thoroughly reviewing all APIs defined in the GPD specification, we identified two additional issues besides the four principles proposed by Rust-SGX. The first issue is that some TEE Internal APIs have complicated dependency-checks. For example, an `API-a` may only be allowed to be invoked when the `API-b` returns a specific `value-c` as the running result. To avoid the case that the developer misses any dependency-check, we enforce every depending API (e.g., `API-a`) to conduct such check automatically, and therefore promise the function of `API-a` is only executed when the required condition is met. For any case that the dependency-check fails, GPD specification defines the invocation on API should be interrupted, and we relay the unexpected status to the Rust error-handling process.

The second issue is that some GPD-defined services require multiple APIs to work in a specific sequence, especially for memory allocation and release. However, TAs can be programmed to invoke these APIs in the wrong order, or even missing some critical steps. To avoid the TA misuses any memory object, we enforce the *Resource-Acquisition-Is-Initialization (RAII)* [42] standard on such APIs. According to the RAII standard, any data structure, named as `struct` in Rust, should be promised with a correct initialization. Moreover, when the developer finished the task on the `struct`, the data structure should provide the correct function to free the resource as well. By enforcing the RAII standard on critical data structures, the memory-related APIs are promised to get execution in the correct sequence.

We present an example for applying our GPD-specific principles in List 1, which is a redesigned Rust-based data structure `OperationHandle` used in TEE's encryption-related operations. As shown in line #9 and line #10, when the structure is allocated, the TA can only move forward if the allocation's return value is `raw::TEE_SUCCESS`, while all the other return values are forwarded to `Err` handler. In this case, as long as developers utilize the redesigned API `OperationHandle::allocate` to acquire the data, the API is promised to check any "potential dangerous return value" and avoid the first issue. Furthermore, when the TA finishes using the allocated data structure, the data is freed automatically because the Rust compiler would execute the function `Drop` (from line #17 to line #23) by default. Therefore, the redesigned `struct OperationHandle` is protected from the second issue.

## 4.4 Secure Cross-world Communication

As an architectural feature of TrustZone-assisted TEE systems, the cross-world communication channel supports the TEE-side TAs to work coordinately with the REE-side CAs. According to the GPD specification, four key data structures are defined and used across the entire CA/TA cooperation process, namely *Context*, *Session*, *Command* and *Parameter*. Starting from the beginning, the CA is required to register its Context in the TEE, without requesting any specific TA to collaborate. Next, the same CA needs to set up a connected Session between it and a specific TA, and this Session is only valid under a registered Context. Once the Session has been correctly set up, the CA can make the following requests to the TA via passing different Commands. If any Command requires the usage of cross-world shared memory (e.g., sharing the plaintext/ciphertext across REE and TEE worlds), the Command can be passed with at most four pairs of Parameters. Each Parameter can represent either a numeric value or a memory chunk. For the entire process of a cross-world communication, we identify three security issues of these four data structures and propose the corresponding security enhancements.

```
1  /* Implement the details of the structure to enforce security
   ↪  principles */
2  impl OperationHandle {
3      fn allocate(algo:AlgorithmId,
4                  mode: OperationMode,
5                  max_key_size: usize) -> Result<Self> {
6          match unsafe { raw::TEE_AllocateOperation(...) }
7          {
8              /* Check the allocation result automatically */
9              raw::TEE_SUCCESS => Ok(Self::from_raw(raw_handle)),
10             code => Err(Error::from_raw_error(code)),
11         }
12     }
13     ...
14 }
15
16 /* Enforce the resource release with the assistance of the
   ↪  language s type security */
17 impl Drop for OperationHandle {
18     fn drop(&mut self) {
19         ...
20         unsafe { raw::TEE_FreeOperation(self.handle()); }
21         ...
22     }
23 }
```

**Listing 1: A Redesigned Encryption-related Data Structure**

**Secure Context's and Session's Lifetime.** One premise of successful communication is that the two fundamental data structures, namely the Context and the Session, are correctly initialized. However, this prerequisite can get challenged in several ways with the GPD specification. According to the GPD specification, these two structures are referred to as unsafe raw pointers, and the caller function has no way to tell whether the callee structure is correctly initialized or not. Moreover, a wrongly used structure may lead to a compromised communication scenario. For example, a C-based CA can get manipulated to connect its Session with another malicious CA's Context without getting any error. In such a case, any further operation may get exposed or even manipulated by the malicious CA. To protect the usages on these two structures, we redesign the Context and Session structure as Rust type-safe structures, which can promise the structures are always adequately initialized before use. Furthermore, We take advantage of Rust's `Drop` function to

promise these two structures' resources are released as the GPD-defined serialization and, hence, promise the corresponding data is erased after use.

**Secure Parameter's Type-safety**. We discovered two security issues of the communication data Parameter. First, the Parameter is defined as type-unsafe in the GPD specification, because TAs access Parameters without a clearly defined type. In this case, a TA can use a numerical Parameter as a memory pointer, or vice versa. To provide Parameter as type-safe, we convert all existing Parameter use cases into two specific Rust-safe data types, namely `int` and `slice`, to pass the numerical value and memory chunks, respectively. With the enforced type definition, any misusing will get detected during the compilation stage. The use of `slice` can also regulate CA's behavior to share the memory chunks. Previously, REE allocates all memory buffers for a Parameter. Then REE shares the memory region with the TEE by providing the corresponding memory's raw pointer and size. This memory-sharing process is unsafe since the attacker can manipulate the pointer and size to mislead the TA to access the memory out-of-scope. By converting the Parameter as Rust `slice`, the memory pointer and associated size are guaranteed to get a securely typecasting, which can prevent TAs from further being manipulated to access the wrong memory region.

**Secure Parameter's Mutability**. Another security concern of Parameter is that a TA may access the TA with incorrect read/write permissions. The GPD specification defines three permissions of Parameters as `input`, `output`, and `inout`, and the Parameters are supposed to get accessed as read-only, write-only, and read/write, respectively. However, a GPD-defined communication channel provides these permissions as independent flags from the corresponding variables, which makes these permissions easily violated. For example, even a memory chunk is designed as a read-only `input` Parameter, a TA can still write on this Parameter as long as the developer does not manually check the Parameter's permission. In Rust, all the read and write permissions are managed via the *mutability* feature by default. By taking advantage of the mutability, RusTEE enforces the permission-check for every Parameter and therefore prevents future violations.

# 5 SYSTEM IMPLEMENTATION

We develop the prototype of RusTEE based on the project OP-TEE [34], which is one of the most well-known open-sourced TEE projects for ARM platforms. OP-TEE implements its Trusted OS and associated software interfaces in compliance with the GPD specification. Currently, the OP-TEE project is available for many ARM TrustZone-assisted devices [33], including the simulation environment QEMU [39], and experimental development boards such as HiKey family [1], Raspberry Pi 3 [40], and Juno [3]. In the following section, we present our modifications to the OP-TEE project for two aspects, namely porting Rust into OP-TEE and binding OP-TEE's Internal APIs (including service APIs and cross-world communication APIs). Meanwhile, we implement the REE-side SDK and rewrite all OP-TEE official C-based examples in Rust. Our rewritten examples demonstrate RusTEE's practicality. Note that we already release RusTEE as an open-source project on GitHub[1], and the latest version supports building both TA and CA in the Rust-safe style. Moreover, RusTEE is configurable to build applications for two most popular ARM architectures: AArch32 and AArch64.

## 5.1 Porting Rust into OP-TEE

Though Rust officially provides the compilation-support on multiple platforms, none of the OP-TEE-supported platforms is recognized by Rust yet. Moreover, in order to balance the functionalities and Trusted Computing Base (TCB) size of TEE, OP-TEE redesigns its basic library `libutil`, which makes it unmatched to the Rust official crate `std::libc`. To resolve these challenges, we first modify the Rust fundamental components `compiler-builtins` and `rust/libstd` to add OP-TEE as the supported targets, which can be further configured based on the architectural features of `arm` (AArch32) or `aarch64` (AArch64). Furthermore, we manually inspect the OP-TEE's basic library `libutil` and match it with the `libc` crate. As the `libutil` does not fully implement all featured functions presented in `libc`, the matching process is realized as a best-effort solution by acceptably sacrificing some functionalities. For example, due to the implementation limitation, a TA runs in OP-TEE OS is implemented as a single-thread task, and the kernel does not provide any multi-threading management. In this case, whenever a Rust program invokes the thread-related operations, we raise panic messages for these operations to remind the developers.

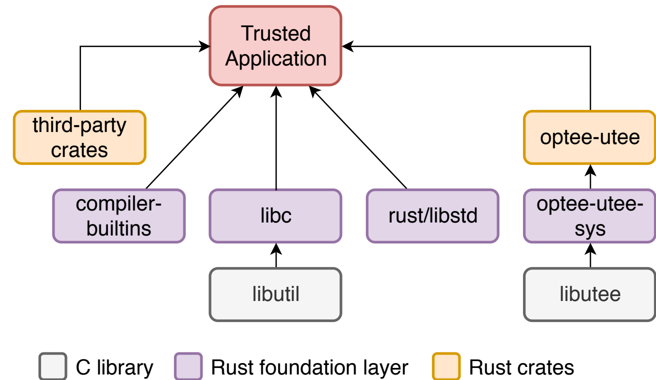


**Figure 3: Porting Rust into OP-TEE**

Besides the three discussed components of Rust's foundation layer, we also provide one extra component `optee-utee-sys` to bind OP-TEE's specific library `libutee` for providing functionalities of all Internal APIs. We further wrap the raw component `optee-utee-sys` as a safe Rust crate `optee-utee`. The details of this binding can be found in Section 5.2. By integrating all the foundation components along with `optee-utee`, RusTEE provides the comprehensive functions for the TA developers to program a TA in Rust-safe style. Finally, RusTEE also supports developers to import trusted third-party Rust crates into the TA development. The entire implementation structure is presented in Figure 3.

[1] The project has been evaluated as an ACM Reusable Artifact by ACSAC 2020. Open-source link: https://github.com/sccommunity/rust-optee-trustzone-sdk

## 5.2 Binding OP-TEE'S TEE Internal APIs

GlobalPlatform TEE Internal Core API Specification [19] defines six types of the necessary APIs for TA development. The first type *Trusted Core Framework API* defines the APIs that provide basic OS functionalities for all kinds of TAs, such as memory management, system-information retrieving, and cross-world communications. For example, each TA should call the same set of APIs to construct and maintain the communication channel with the REE. Moreover, in the current implementation of cross-world communication, we label two operations, `Parameter::as_value` and `Parameter::as_memref`, as `unsafe` operations because OP-TEE's `Parameter` are implemented as unsafe from Rust's ownership and thread-safety perspective. Specifically, whenever a TA receives the data in the shared memory, the CA and REE still have the privilege to modify the Parameter, so there exists a potential concurrent issue for using shared Parameters. Currently, these two operations are the only two exceptions that can appear in the TA source code as `unsafe` segment. Note that the `unsafe` labels here do not mean any memory vulnerability is actually introduced, while they are more to syntactical definitions to alert the developers. For example, whenever the TA is supposed to use any passed-in data array exclusively, it should copy the data from the unsafe Parameters into a safe array, and then conduct rest operations reliably.

The second type is *Trusted Storage API for Data and Keys*, which provides reliable storing for security-sensitive structures, and mostly applied on the cryptography keys' materials. Thirdly, *Cryptographic Operation API* defines the APIs for extensive cryptographic-related tasks such as generating the key, conducting synchronous/asynchronous encryption, and hashing calculations. Next, *Time API* can return the trusted time for TAs, where the time can be selected from different perspectives such as per-TA time, Trusted OS's unified time, or even REE's Rich OS's time. Moreover, *TEE Arithmetical API* are the essential functions that majorly serve for calculating big numbers and primes. Lastly, *Peripheral and Event API* is designed to allow TAs to interact with the hardware peripherals. Most of the peripheral-APIs are platform-specific as different platforms can equip a variety of peripherals. Since OP-TEE OS only implements the first five types of APIs, our prototype binds all of the implemented APIs, and we list the Lines-of-Code (LOC) of each type in Table 1.

## 5.3 REE and Examples

Besides the TEE-side SDK, we also implement the crate `optee-teec` as the REE-side SDK, which integrates the Rust standard library and other GPD-defined Client API-related libraries to support building secure CAs. Presently, OP-TEE provides six examples to demonstrate the CA/TA workflow in several aspects, such as basic communication functionalities, secure storage, and cryptography-related tasks. To prove the practicality of RusTEE, we completely migrate these six examples by rewriting them in Rust. Moreover, we provide six more examples to present RusTEE's capabilities of interacting with all types of TEE Internal APIs. Finally, we provide one additional example for exhibiting the case that integrates third-party Rust crate `Serde` into TA development. The detailed examples and corresponding performance evaluation are presented in the

**Table 1: RusTEE Component's LOC**

| Component | Lines of Code |
|---|---|
| **TEE** | |
| Trusted Core Framework API | 2076 |
| Trusted Storage API | 544 |
| Cryptographic Operation API | 672 |
| Time API | 52 |
| TEE Arithmetical API | 258 |
| **REE** | |
| Client API | 687 |
| **Examples** | |
| Rewritten OP-TEE Examples | 1964 |
| Newly Added Examples | 2105 |
| **Total** | 8358 |

Evaluation Section 6. The latest project's LOC[2], which includes both worlds' SDK and examples, are summarized in Table 1.

## 6 EVALUATION

In this section, we present the performance evaluation of RusTEE. Compared to the previous TA-development mechanisms, our mechanism introduces performance overhead in two aspects: the general overhead of changing programming language and specific overhead of API-related enhancements. First, since RusTEE replaces the previous programming language C with Rust, RusTEE may introduce the overhead because of using the new language. Though some existing benchmarks already presented the difference between these two languages on the x86 platform, we notice their performances vary a lot on ARM devices. Therefore, we present the language-wise difference between C and Rust for ARM devices specifically. We implement four benchmark programs in both languages and evaluate the programs' performances on the ARM-based Juno r1 [3] development board. Furthermore, we re-run the benchmark on the emulator environment QEMU [39] with the same ARM architecture to validate the observation.

Besides the differences in programming languages, RusTEE may introduce extra overhead because it performs multiple security enhancements on the TEE Internal APIs. Since the overhead of invoking APIs is tightly coupled with the real-world use cases, we evaluate this overhead based on five real-world TAs provided by OP-TEE [32]. We rewrite each TA in Rust and then compare our rewritten TA with OP-TEE's C-based TA. The difference between the two TAs' execution time can indicate the overhead of corresponding APIs.

### 6.1 Language-wise Overhead

To present the fundamental difference between languages C and Rust on ARM devices, we evaluate them with four benchmark cases of the open-source programming language benchmark-set [21]. The benchmark-set provides dozens of cases in different languages for evaluating their computation efficiencies on x86 devices. However, it is non-trivial to migrate all benchmark programs on ARM devices

[2]The LOC are counted at the time of this paper is written and may change in the future version of the open-source project.

because many programs rely on the libraries that are not supported by either C or Rust compiler on ARM platforms. Moreover, as OP-TEE OS only provides limited functionalities in the TEE, TAs are not capable of integrating any benchmark's program completely. After manually reviewing the benchmark-set, we select four cases that can get compiled and executed on ARM platforms stably for both languages. We implement the benchmark programs in the REE to get the support of the Rich OS, which equips the Linux kernel in our implementation.

Among the evaluated cases, case `n-body` models the orbits of Jovian planets as a double-precision simulation; case `fasta` generates and rewrites DNA sequences; case `fannkuch-redux` performs the indexed-access to tiny integer-sequence with the approximated time complexity = ∗ log =; case `spectral-norm` resolves the mathematical challenge [49] that requires to calculate the spectral norm of an infinite matrix. Currently, the benchmark-set already provides detailed performance of C and Rust about each case on x86 platforms, including their execution time, memory space, and CPU utilization. Also, every case can get accomplished with different algorithms.

Since previous coders and researchers already evaluated the thorough performances of two languages on x86 platforms, our experiment focuses on presenting the performances' variations after benchmark programs are migrated from x86 platforms to ARM platforms. We assume an algorithm of one case is executed as 100% time on x86 platforms, and we normalize the execution time of this algorithm on the Juno board accordingly. For each benchmark case, we evaluate all algorithms that can get compiled with both languages' ARM compiler. After collecting all algorithms' results for one case, we calculate the average value of the normalized execution time, and we present the final result in Figure 4.

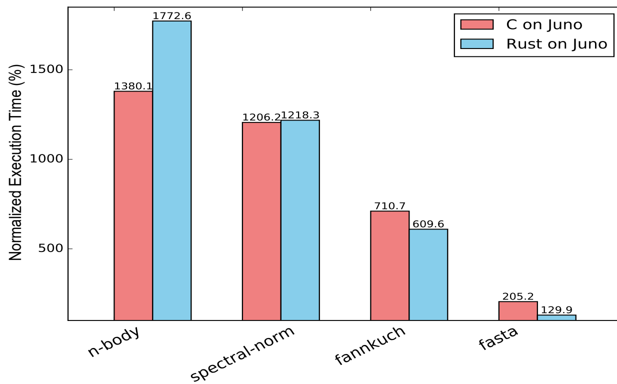


**Figure 4: C vs. Rust Performance on Juno**

According to our experiment, all the benchmark programs run slower on the Juno board than the x86 platform. The numerical difference can be introduced because of the different hardware configuration (i.e., CPU cores and total memory space). Specifically, for the first two cases `n-body` and `spectral-norm`, C language performs relatively better than Rust after normalization, while the other two cases present the contrast observation. Meanwhile, for all evaluated cases, the normalized differences between the two languages are less than 40%.

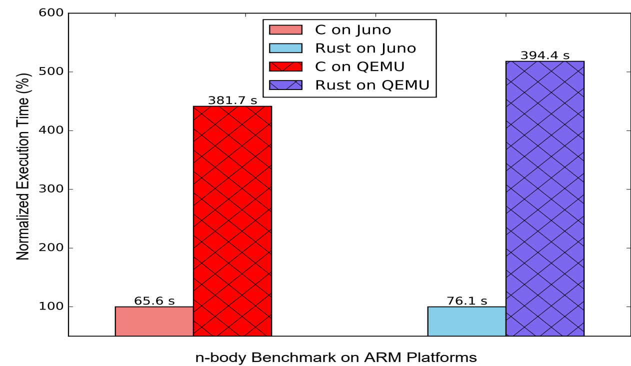


**Figure 5: QEMU vs. Juno Performance**

**Different Platforms Evaluation**. To validate the performance we evaluated on Juno is representative across different ARM devices, we provide an extra evaluation of the emulation environment QEMU. We re-implement the benchmark `n-body` in two languages on QEMU, and then evaluate the performances as presented in Figure 5. We assume the execution time of Juno board's programs are 100%, and then normalize the time of QEMU's programs accordingly. As the experiment shows, comparing to the Juno board, the emulator introduces around 3.5 times extra overhead for both C and Rust languages. Meanwhile, the extra overhead is introduced with a similar ratio for two languages, which means the relative difference between C and Rust stays at the same level on both Juno and QEMU. In conclusion, we claim that the language-wise difference we evaluate in Figure 4 is representative of the ARM architecture. Also, the evaluations on either development board or emulation environment present the same pattern of the difference.

## 6.2 Enhanced APIs' Overhead

To present the overall overhead of enhancing APIs, we evaluate TAs' performance in five real-world cases to invoke different types of APIs. For each case, we use the same CA to invoke two TAs compiled in C and Rust, respectively. Meanwhile, both C-based and Rust-based TAs are programmed to execute the same task with the same algorithm, while the major difference is that all Rust-based TEE Internal APIs are enhanced by RusTEE. Among the five cases, case `Secure_Storage` provides the functionalities for reading, creating, and deleting the secure-storage objects. We use the time of creating an empty secure-storage object to represent related tasks efficiency; case `Random` generates a 16-bytes random number; case `Hotp` generates ten HMAC-based one-time passwords according to the RFC4226 algorithm [37]; case `Aes` conducts the AES-128 encryption with CTR mode on a 4096-bytes plaintext; case `Acipher` conducts RSA Public-Key Cryptography Standards (PKCS) encryption with the 1024-bits key and the 100-bytes plaintext.

We evaluate each case 10,000 times in total, and we calculate the cases' average execution time with the data set that excludes 10% data outliers (5% largest and 5% smallest data). The comparison of C-based TA and Rust-based TA is presented in Figure 6. For each case,

Figure 6: Performances of C-based TAs vs. Rust-based TAs

we labeled the average execution time above the corresponding TA's bar. As the baseline data, the average context switch time (without conducting any task in TEE) is 676 μs for both C and Rust case, with a negligible variation. We consider the C-based TA's execution time as 100% and then normalizing the Rust-based TA's data accordingly. As the figure presents, for the five evaluated cases, RusTEE only introduces the performance degradation from 0.27% to 3.08% and four of the five cases are affected with less than 1% overhead.

# 7 DISCUSSION

**Verifying Third-party TAs' Security.** Currently, RusTEE is released as an open-sourced project to benefit any TA developer who wants to enhance the TA's security. Meanwhile, we believe that our mechanism can bring extra benefits for the mobile manufacturers to quickly review the security of third-party TAs. Before RusTEE, the manufacture can only verify a TA's security via manual inspection, which requires the verifier to understand the complicated logic inside TA. In this case, if the SDK is opened to third-party developers, many human efforts will be introduced to verify the third-party TAs. Moreover, such manual verification only provides the security promise based on personal experience, without any formal proof. After adapting RusTEE into the manufacturer's SDK, the manufacturer will have a straightforward and reliable verification method, which is checking if the TA's source code contains any unsafe segment or calls any untrusted crate. We consider providing the automatic verification script for checking third-party TAs in future work.

**Binding the Rust and C Worlds.** The major limitation of our mechanism is that we do not provide an automatic solution to bind the C and Rust worlds. Therefore, our mechanism may require extra human effort in the future if the dependent libraries are changed. Alternatively, if the system administrator wants to extend the support for a C-based library rapidly, she can utilize the RustFFI mechanism to include the library's functions with the Rust world. However, as we argued before, such a mechanism is memory-unsafe, so it does not fit RusTEE's primary objects. Moreover, since a TrustZone-assisted TEE only has a limited-size TCB and relatively stable libraries, we believe such TEE is a perfect target for manually binding with acceptable engineering effort.

# 8 RELATED WORK

## 8.1 Rust-assisted Systems

In past years, Rust language has become an attractive programming language for developers who have an interest in enhancing application security. As a memory-safe language, Rust's safety has been formally proved in RustBelt [27] in 2017. Meanwhile, lines of works [5, 29–31, 48] have been proposed to adapt Rust into the development of traditional C/C++ based systems. For example, TockOS [30] presents the idea to write a complete embedded system OS in Rust. Moreover, Rust has been integrated with TEE development [15, 16, 48]. For Intel SGX, Wang et al. [48] propose the open-source project Rust-SGX to deliver the Rust-based SDK for SGX enclave developers, and Fortanix Rust EDP [16] has implemented a similar idea. Regarding the TrustZone technology, RustZone [15] first demonstrates the possibilities to migrate Rust into TrustZone TA development, while lacking a thorough analysis of the security for each component insides TAs. To the best of our knowledge, RusTEE is the first work that presents the complete development kit set for TrustZone TA developers and provides the default features to compile TAs in Rust-safe style.

## 8.2 Security of TEE

The TEE technology has evolved rapidly during recent years. Researchers [23, 24, 36] first propose the software-only solutions that utilize the virtualization technique to create the TEEs. Nowadays, many hardware-assisted TEE architectures [2, 25, 50] have been proposed to provide isolated environments with more reliable technologies and work for different real-world scenarios. Among these recently popular hardware-assisted TEE systems, some architectures such as Intel SGX [14, 25] can provide multiple TEEs within a single device, while other architectures like ARM TrustZone [2] present the single-TEE solution to partition the device into one REE and one TEE.

The security of ARM platforms has been discussed in several aspects in the previous works [4, 17, 26, 28, 52]. Among these TrustZone-related works, one leading category is using Trusted Applications for protecting the REE's rich OS. For example, TrustZone has been exploited to conduct either integrity checking [47] or run-time rich OS activities monitoring [4, 17]. Besides the REE side, several previous works [26, 52] also present the analysis and enhancement on the security of TEE side.

Since TrustZone only provides one TEE to run multiple TAs simultaneously, how to guarantee the security of the ported TAs becomes a critical question for ARM manufactures. Previously, multiple solutions [2, 8, 10, 41] have been proposed to answer this question. For example, TrustICE [43] executes different TAs separately in a time-slicing fashion, and the timing-based isolation prevents one vulnerable TA from accessing other TA's resources. Brasser et al. [8] propose the idea to allocate different software-generated "enclave" for every TA in REE and then utilize the shared TEE to manage all allocated enclaves. As the shared threat model, both previous works assume TAs compiled as vulnerable and focus on preventing the vulnerable TA from attacking other TEE software. Unlike these related works, RusTEE presents the capability to compile TAs with the proven memory-safety regulations. Therefore, our mechanism can promise all TAs are memory-safe

to get executed in the shared TEE without using any sophisticated isolation mechanisms.

# 9 CONCLUSION

In this paper, we presented RusTEE, a Rust-based TrustZone application SDK, which assists developers to compile the TA with the enforced memory-safety features. The TA relies on the language-wise benefit of Rust to mitigate the previously reported implementation issues. Furthermore. RusTEE redesigns the system-services APIs and cross-world communication channel of TA to resolve two architectural issues of TrustZone-assisted TEE systems. We implement RusTEE based on the existing C-based SDK OP-TEE, and evaluate the mechanism on multiple platforms that include both emulators and development boards. According to our evaluation, RusTEE introduces slight performance overhead while significantly increases the application's memory-safety in multiple aspects. Finally, we open-source the entire RusTEE with various examples.

# ACKNOWLEDGMENTS

This work is partially supported by NSF grant #1815650.

# REFERENCES

[1] 96 Boards. Accessed in June 2020. HiKey Website. https://www.96boards.org/product/hikey/.
[2] ARM. Accessed in June 2020. ARM Security Technology: Building a Secure System using TrustZone® Technology. https://developer.arm.com/ip-products/security-ip/trustzone.
[3] ARM. Accessed in June 2020. Juno Arm Development Platform. https://developer.arm.com/products/system-design/development-boards/juno-development-board.
[4] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 90–102.
[5] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System programming in Rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 156–161.
[6] Ariel Ben-Yehuda. 2015. Can mutate in match-arm using a closure. Rust issue #27282. https://github.com/rust-lang/rust/issues/27282.
[7] Christophe Biocca. 2017. std vec IntoIter as_mut_slice borrows &self, returns &mut of contents. Rust issue #39465. https://github.com/rust-lang/rust/issues/39465.
[8] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *NDSS*.
[9] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*. 18–20.
[10] Yeongpil Cho, Jun-Bum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. 2016. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices.. In *USENIX Annual Technical Conference*. 565–578.
[11] Common Vulnerabilities and Exposures. 2015. CVE-2015-6639. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6639.
[12] Common Vulnerabilities and Exposures. 2016. CVE-2016-2431. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2431.
[13] Common Vulnerabilities and Exposures. Accessed in June 2020. CVE Search Results for TrustZone. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=trustzone.
[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.
[15] Eric Evenchick. 2018. RustZone: Writing Trusted Applications in Rust. https://github.com/ericevenchick/rustzone.
[16] Fortanix. Accessed in June 2020. The Fortanix Rust Enclave Development Platform. https://edp.fortanix.com/.
[17] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747* (2014).
[18] GlobalPlatform. 2010. TEE Client API Specification v1.0. https://globalplatform.org/specs-library/tee-client-api-specification/.
[19] GlobalPlatform. 2019. TEE Internal Core API Specification v1.2.1. https://globalplatform.org/specs-library/tee-internal-core-api-specification-v1-2/.
[20] GlobalPlatform. 2019. TEE Management Framework including asn.1 Profile v1.0.1. https://globalplatform.org/specs-library/tee-management-framework-including-asn1-profile/.
[21] Gouy, Isaac. Accessed in June 2020. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.
[22] Joffrey Guilbon. 2018. Attacking the ARM's TrustZone. https://blog.quarkslab.com/attacking-the-arms-trustzone.html.
[23] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. 2013. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 265–278.
[24] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing {ARM} TrustZone. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 541–556.
[25] Intel. Accessed in June 2020. Intel Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.
[26] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2015. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment.. In *NDSS*.
[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.
[28] Hojoon Lee, Hyungon Moon, Ingoo Heo, Daehee Jang, Jinsoo Jang, Kihwan Kim, Yunheung Paek, and Brent Kang. 2017. KI-Mon ARM: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. *IEEE Transactions on Dependable and Secure Computing* (2017).
[29] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: Experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. 21–26.
[30] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 234–251.
[31] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The case for writing a kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. 1–7.
[32] Linaro. Accessed in June 2020. OP-TEE Sample Applications. https://github.com/linaro-swg/optee_examples.
[33] Linaro. Accessed in June 2020. OPTEE Device. https://optee.readthedocs.io/en/latest/building/index.html.
[34] Linaro. Accessed in June 2020. OPTEE Secure OS. https://github.com/OP-TEE/optee_os.
[35] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.
[36] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 143–158.
[37] et al. M'Raihi. Accessed in June 2020. RFC4226: HOTP: An HMAC-Based One-Time Password Algorithm. https://tools.ietf.org/html/rfc4226.
[38] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526.
[39] QEMU. Accessed in June 2020. QEMU Website. https://www.qemu.org/.
[40] Raspberry Pi. Accessed in June 2020. Raspberry Pi Serial Products. https://www.raspberrypi.org/products/.
[41] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM TrustZone to build a trusted language runtime for mobile applications. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 67–80.
[42] Bjarne Stroustrup. Accessed in June 2020. Why doesn't C++ provide a "finally" construct? http://www.stroustrup.com/bs_faq2.html.
[43] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. 2015. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 367–378.
[44] The Rust Programming Language Core Team. Accessed in June 2020. Unsafe Rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.
[45] Trustonic. Accessed in June 2020. Application Protection & Security Mobile in-app protection for critical mobile apps. https://www.trustonic.com/solutions/trustonic-application-protection-tap/.

[46] Aaron Turon. 2015. Abstraction without overhead: Traits in Rust. https://blog.rust-lang.org/2015/05/11/traits.html.
[47] Shengye Wan, Jianhua Sun, Kun Sun, Ning Zhang, and Qi Li. 2019. SATIN: A Secure and Trustworthy Asynchronous Introspection on Multi-Core ARM Processors. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 289–301.
[48] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2333–2350.
[49] Wolfram Research, Inc. Accessed in June 2020. Hundred-Dollar, Hundred-Digit Challenge Problems. http://mathworld.wolfram.com/Hundred-DollarHundred-DigitChallengeProblems.html.
[50] Fengwei Zhang and Hongwei Zhang. 2016. Sok: A study of using hardware-assisted isolated execution environments for security. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 3.
[51] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y Thomas Hou. 2016. CacheKit: Evading memory introspection using cache incoherence. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 337–352.
[52] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. 2016. Case: Cache-assisted secure execution on arm processors. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 72–90.