Towards a Rust SDK for Keystone Enclave Application Development

Jukka Julku[a] and Markku Kylänpää[b]
VTT Technical Research Centre of Finland Ltd., Espoo, Finland

Keywords: Trusted Execution Environments, Enclave, Rust, Security, Software Development.

Abstract: Secure enclaves are commonly used for securing sensitive data and computation. However, an enclave can only be trusted if the software running in the enclave is secure. Nevertheless, enclave software is often written in low-level languages that are prone to vulnerabilities. As the number of enclave application developers grows, more attention must be paid to secure software development. The use of safe programming languages could be one step towards more secure trusted software. In this paper, we discuss our work towards a Rust programming language SDK for Keystone enclave application development. In addition, we present early performance measurements of the SDK compared to the original Keystone SDK written in the C/C++ languages.

1 INTRODUCTION

Trusted Execution Environments, TEEs, have become a standard practice to protect device secrets and critical system functionality from untrusted software running on the same platform, e.g., in mobile devices (Gunn et al., 2022). Traditionally, TEEs have been closed environments, running trusted applications developed by a small and tightly controlled group of manufacturers and trusted third parties. However, in the recent years the direction has been towards opening TEEs for other, less trusted, third-party applications. At the same time, the application of the TEEs is expanding to new directions that demand richer, more complex trusted applications to be supported. For example, the confidential computing paradigm (Russinovich et al., 2021) depends on secure enclaves to protect sensitive data from untrusted cloud platforms.

Unfortunately, TEEs and trusted applications are not without vulnerabilities (Fei et al., 2021; Cerdeira et al., 2020; Van Bulck et al., 2019). In practice, the TEE firmware and the applications are still commonly written using low-level programming languages such as C and C++ that make it notoriously easy to introduce subtle vulnerabilities, such as buffer and integer overflows, that are difficult to detect during testing. As TEEs are often used to store critical information, such as device keys, vulnerabilities easily lead to full compromise of the device. For example, one byte buffer overflow was recently found in Google's Titan-M (National Vulnerability Database, 2022). The authors claim it can be used to read any memory address of chip's secure memory (Melotti and Rossi Bellom, 2022). As trusted applications become more complex and the number of developers writing them increases, so rises the number of vulnerabilities.

Vulnerabilities may be prevented using secure software development methods and tools. High-level programming languages (Oak et al., 2021) and standard APIs (Boubakri et al., 2021; Open Enclave, 2022) can minimize the newly written code and leave less chance for mistakes. They can also improve the effectiveness of testing, manual inspection, and static analysis due to more manageable code size, API specific checks, and enabling common tools to be used for different platforms. Recently, the Rust programming language has become a common alternative for C/C++ in system software development as it offers better type, memory, and thread safety guarantees. It has also been taken into use in enclave application development (Wang et al., 2019).

However, most such solutions support only one or few closed architectures, i.e., Intel SGX or ARM TrustZone. TEEs for emerging architectures have gained far less attention. Keystone is an open framework for developing TEEs for the open RISC-V architecture. It uses an interesting enclave model, where only a very minimal security monitor is provided and much of the enclave functionality is moved to untrusted user-space or left to an enclave specific runtime. Keystone too only offers a C/C++ SDK for enclave application development.

In this paper, we present our work towards a Keystone compatible SDK that allows development of en-

[a] https://orcid.org/0000-0003-4395-3832
[b] https://orcid.org/0000-0003-1092-3004

Julku, J. and Kylänpää, M.
Towards a Rust SDK for Keystone Enclave Application Development.
DOI: 10.5220/0011611900003405
In *Proceedings of the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023)*, pages 29-37
ISBN: 978-989-758-624-8; ISSN: 2184-4356

clave applications in the Rust programming language. The main contributions of our work are:

- A Rust language API for both Keystone enclave and host application development
- Required Rust programming libraries (crates) published as open-source software
- Performance comparison of our implementation against the original Keystone SDK

The rest of this paper is structured as follows. First, in Section 2 we introduce the reader to Keystone enclaves and the Rust programming language. Next, in Section 3 we present the work we have done towards the Rust SDK. In Section 4 we discuss early evaluation results in the form of a demonstrator application and performance measurements. Then, in Section 5 we discuss potential pitfalls of our work and future work still needed. In Section 6 we elaborate related approaches to developing enclave applications. Finally, in Section 7 we conclude the paper.

2 BACKGROUND

Keystone (Lee et al., 2020) is a framework for building customizable TEEs that hardware and system manufacturers can use to tailor their security solutions to match their own threat model. Keystone's main design principles are:

- Leveraging platform specific isolation primitives: hardware-dependent concepts are used to implement a *Security Monitor* that provides isolation and security functions
- Decoupling security policy enforcement from resource management: Security Monitor handles mostly security checks and each enclave application has a *runtime* handling resource management
- Modularity: independent architecture layers allow designers to replace components as they see fit
- Fine-grained, minimal, Trusted Computing Base

Keystone is in many ways more flexible compared to existing commercial TEE solutions. For example, Intel SGX has fixed memory protection system and ARM TrustZone provides only one enclave, easily leading to monolithic security solutions.

Keystone implementation exists for the RISC-V (Lu, 2021) architecture and its security building blocks: processor privileged levels, Physical Memory Protection (PMP) mechanism, and process isolation between hardware threads. In the Keystone architecture, illustrated in Figure 1, the Security Monitor, runs on M-mode privilege below all other software and

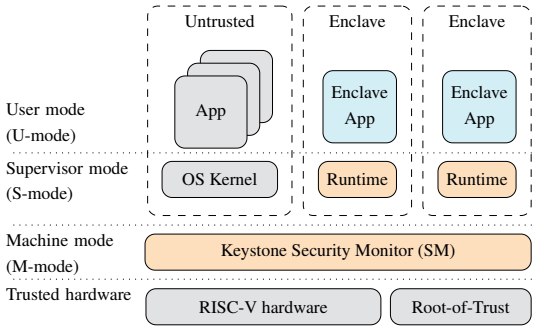

Figure 1: Keystone Security Framework architecture.

implements physical memory protection and security enforcement to partition system resources between a normal untrusted operating system and a number of isolated enclaves. In each partition, the operating system or enclave runtime runs on S-mode privilege and orchestrates applications running on U-mode privilege. Keystone provides a reference runtime called Eyrie that supports one application per enclave.

The applications of the untrusted operating system create and interact with the enclave applications using a Keystone pseudo-device and an untrusted shared memory buffer that is used to transfer data between security domains. The buffer is shared between an untrusted application and an enclave runtime. The runtime copies the data into the enclave application.

3 DESIGN

In this section we describe the design of the Rust language SDK and its main features. The implementation has been published as an open-source project (Julku and Kylänpää, 2022).

3.1 Architecture

Keystone enclaves are an interesting starting point for our work for several reasons. First, its open software and one of the few enclave systems available for the RISC-V architecture. Second, it provides a distinct enclave model, consisting only of a minimal security monitor and operating system driver, while the other parts are highly customizable. We selected Keystone's Eyrie enclave runtime as the enclave application's operating system for simplicity. However, the API can also be used with other operating systems.

The scope of our work is illustrated in Figure 2. Components presented in grey are standard system components, e.g., the hardware, the operating system, and the applications. The original Keystone components, i.e., the Security Monitor and the Eyrie

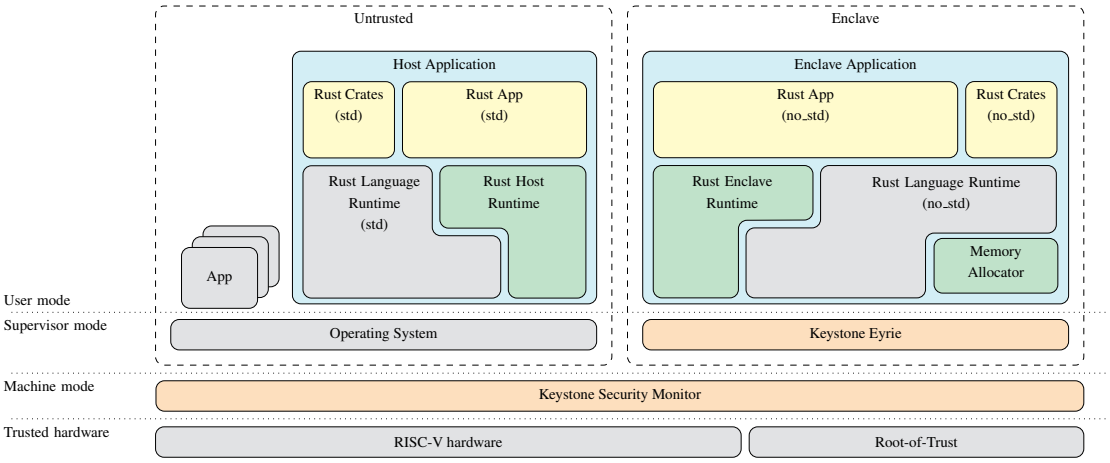

Figure 2: Scope of our work in the RISC-V and Keystone stack.

runtime, we are utilizing, but chose not to modify for backwards compatibility, are presented in orange. The two blue boxes represent an enclave application and its counterpart, a host application that drives the enclave application. Both consist of the application logic, represented in yellow, which includes application specific Rust code and possibly a set of Rust crates, i.e., programming libraries. The applications must be linked with a Rust language runtime, represented in grey. An unmodified Rust language runtime is used for both the applications. Finally, our work is represented in green. It consists of two Rust crates, one for the host application and one for the enclave application. These crates correspond to Keystone's original SDK libraries and they enable Keystone enclave operations, such as communication between the applications using the *ocall* and *ecall* operations, and attestation to verify enclave application's integrity.

3.2 Enclave Runtime

Due to restricted enclave environment, the enclave application's Rust runtime provides only the Rust Core Library, i.e., a bare metal Rust environment, but no Rust Standard library. In addition, Rust core allocation library can be optionally included for enclaves that require dynamic memory allocation. The SDK provides a safe Rust language API for enclave operations. All the unsafe features, e.g., the system calls to Keystone Eyrie and management of the raw memory for dynamic memory allocation, are hidden behind the API. As Eyrie doesn't support application threads, neither does the Rust Enclave runtime. However, the implementation is thread-safe and all access to globally shared data is guarded by spin locks.

Enclave applications are compiled for RISC-V bare metal environment. Since the enclave doesn't provide but the minimal environment by the Keystone Eyrie, all application components, including third-party crates and the Rust language runtime, must be statically linked into the application binary.

Like most standard libraries, the Rust Enclave Runtime contains the actual entry point of the application, which is used to trigger initialization routines, such as initialization of the memory heap, before the actual application specific code enters execution. A Rust macro, called *eapp_entry*, is used to define the application entry point. Similarly, *eapp_return* function is provided for terminating the enclave and returning into the host application. In addition, the runtime implements a Rust panic handler that automatically terminates the enclave in case of panic.

The runtime supports all the central Eyrie system calls, e.g., the *ocall* operation, and retrieving enclave instance's attestation report and data sealing keys from the Security Monitor. A high-level API is provided over the system calls, enabling their functionality to be used in more Rust-like conventions. For example, the *ocalls* are made using a structure that manages the underlying buffers and performs safety checks on the system call parameters. Furthermore, the *ocall* API prepends a header to each message that also allows new features not supported by the Eyrie to be implemented: the means to communicate the low-level status code and the actual response length written by the host application, as well as the means to convey the maximum response length supported by the enclave application to the host application.

Eyrie doesn't support *ecalls*, i.e., calls from the host system to an enclave. The Rust SDK adds *ecall* support by emulating them using *ocalls*. The process is discussed in Section 3.4. In the API, *ecalls* are defined as a server interface and listener trait that may be used to dispatch incoming calls to listeners.

```
#![no_std]
#![no_main]

extern crate eapp;
use eapp::{eapp_entry, Status};
use eapp::ecall::{ECall, Listener, Server};

/* Handler replies with the same data: */
struct CopyListener {}

impl Listener for CopyListener {
  fn on_ecall(&self,
              ctx: &mut ECall,
              req: &[u8]) -> Status {
    /* Check response fits into the buffer: */
    let res = ctx.response();
    if req.len() > res.len() {
      return Status::ShortBuffer;
    }
    /* Copy input buffer into output buffer: */
    res[..req.len()].clone_from_slice(req);
    ctx.response_length(req.len());
    return Status::Success;
  }
}

/* Enclave entry point function: */
#[eapp_entry]
pub fn my_eapp_entry() -> u64 {
  /* Create ecall dispatcher */
  let mut disp = Server::new();
  /* Maximum expected input size: */
  if !disp.max_input_size(64) {
    return 1;
  }
  /* Ensure attestation report fits */
  if !disp.max_output_size(2048) {
    return 1;
  }
  /* Register handler for ecall */
  let handler = CopyListener{};
  disp.register(0x1, &handler);
  /* Serve ecalls until interrupted */
  return disp.serve() as u64;
}
```

Figure 3: Example application that copies input to output.

An example application using the API is illustrated in Figure 3. This application defines an entry point function for the enclave application. The function simply creates an *ecall* dispatcher and registers a custom *ecall* handler for a specific call identifier. After that it starts to serve incoming *ecalls* until interrupted. The handler responds all requests by copying the *ecall* input buffer to the output buffer, i.e., responds each call with the same payload data that it received. The runtime provides a default *ecall* handlers for serving, e.g., attestation requests, to reduce the amount of application specific code needed for basic enclave functionality.

Many common Rust features, e.g., vectors, depend on dynamic memory allocation. The enclave runtime provides a memory allocator for Rust's *liballoc*, based on the buddy system allocator (Chen, 2022). This allows developers to use the standard Rust syntax for memory allocation features in the enclave. The memory allocator can be turned on using a Rust crate feature flag. The memory available for the allocator is defined at the application link time using a linker script. Allocation failures lead to panic and termination of the enclave application.

3.3 Host Runtime

The host application is built like any other Rust application using a full Rust Standard Library Environment. The Rust SDK crate adds functionality to interact with the Keystone pseudo device in order to build and run enclave applications. The library is designed to be thread-safe and supports simultaneous interaction with multiple enclave application instances.

A distinctive feature of Keystone is that enclave application's memory image is constructed in an insecure userland process and transferred into the enclave's private memory. That is, the application is loaded into memory and the page table for the enclave private memory built by the host application. The Security Monitor only validates the result as it configures memory protection for the enclave. Attestation is used to verify that the enclave private memory matches the expected memory image. The Rust SDK provides a pure Rust implementation of the loading logic. It is exposed to the end-user in the form of a builder-pattern API that allows the enclave parameters to be configured cleanly in a step-by-step fashion. The interface also supports emulation of the enclave memory construction even when Keystone pseudo-device is not available, e.g., in order to validate applications and to compute reference integrity hashes for attestation.

The Rust API allows building new enclave instances, running instantiated enclaves, and registering *ocall* listeners for calls coming from the enclave. The *ocalls* API consist of a call dispatcher that performs safety checks for the raw memory and call parameters, e.g., buffer address and length, before wrapping the raw memory in a safe Rust byte array abstraction. Thus, the developer by default operates on safer Rust primitives. The API supports multi-threaded host applications by providing a thread-safe handle to the enclave abstraction. The handle can be passed to other threads for requesting *ecalls* while the main thread is engaged in running the enclave. The handle synchronizes the call parameters and the payload with the

main thread, which then executes the operation on the enclave at the next opportunity. In addition, the crate provides a Rust interface, e.g., for parsing and verifying Keystone attestation reports.

3.4 *Ecall* Emulation

Keystone doesn't support *ecalls* from the host application to the enclave. Nonetheless, in many applications *ecalls* are a natural way of communication: the enclave provides a secured service interfaced via a daemon process running in the host system. We designed a Rust API for the *ecall* operation, implemented by emulating *ecalls* on top of a pair of *ocalls*.

The enclave API allows to register handlers for *ecalls* and provides a server, which dispatches call requests to handlers based on user-defined identifiers, similar to how *ocalls* work in the host application. However, as the enclave application doesn't currently support threads, the main thread must either be dedicated to serving calls or multiplex between serving calls and any other processing.

The runtime internally handles certain *ecalls*, e.g., a generic attestation request and a request to terminate the enclave application gracefully. The default attestation handler serves local attestation requests by the host application in the enclave application by relaying the request, including a caller-specified nonce for freshness of the attestation report, to Eyrie's system call interface, and eventually, into the Keystone Security Monitor that generates the attestation report and signs it together with the nonce.

In the host application, *ecalls* are made using an enclave handle, as the enclave abstraction is owned by the thread dedicated to interacting with the Keystone pseudo-device and Rust thread-safety rules forbid other threads from using it simultaneously. The handle hides thread synchronization, but unfortunately, the payload of the call and any returned values must be copied once between the calling thread's memory and the untrusted memory. For larger payloads, this might cause considerable overhead.

The emulation process is illustrated in Figure 4. The calling thread in the host application first informs the thread owning the enclave abstraction about its intent by sending a message to a Rust channel via the handle. Within the message, it includes the call parameters (#) and payload (*I*). After this, the calling thread blocks on another channel until it receives a response. In parallel, the enclave application prepares to serve *ecalls*. It issues an *ocall* with a special identifier, *EPoll*, to yield execution to the host thread owning the enclave. At the host, the enclave thread resumes to handle the *ocall* and checks the channel for

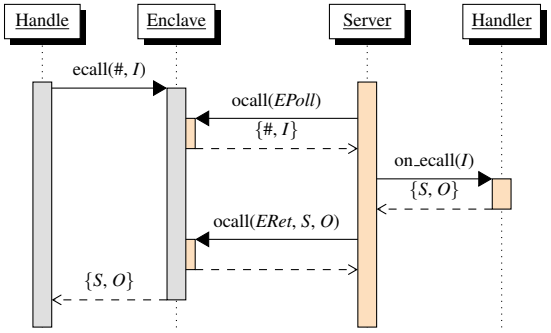

Figure 4: Emulation of ecall on top of ocalls.

pending calls. If at least one exists, then the thread validates the call parameters, copies them and payload to the *ocall* response, and resumes the enclave execution. If no calls are pending, then the thread either waits for one to appear, and in doing so blocks the enclave, or immediately resumes the enclave depending on the selected crate features.

In case a pending call was found, the enclave application's server validates the call parameters and dispatches the request to a handler assuming one is registered for the corresponding identifier. The handler then performs any application specific processing. When the handler returns, another *ocall*, called *ERet*, is made to return a status code (*S*) and any output (*O*) to the host application. At the host, the enclave thread resumes and matches the response with the original *ecall* intent and its return channel. Sending the response to the channel wakes the calling thread and completes the *ecall* emulation process.

4 EVALUATION

In this section, we present an example use case to show that the Rust SDK can be used to develop complex applications. We also discuss early performance measurements and compare them to the original Keystone SDK.

4.1 Example Use Case

We created a simple demonstrator example that implements a simplified confidential computing scenario where a remote client application exchanges data with an enclave via a secure channel created between the client and the enclave application. The application is inspired by the original Keystone demonstrator written in C (Keystone Enclave project, 2022). The client application and the enclave and the host applications on the server-side are all written using Rust programming language utilizing the Rust SDK.

The application demonstrates how to build and start the enclave instances and pass data to and from the enclave using *ecalls*. The server-side application is partitioned into host application listening for incoming connections from client applications and is starting a separate enclave instance to handle each connection. The client and the enclave use Diffie-Hellman key exchange protocol to create a shared secret that is used to establish a secure channel between them. The channel is tunnelled through the host application. Because the secure channel is established between the client and the enclave, the host-side of the server need not be trusted.

The key observations made while building the applications was that the lack of support for the Rust standard library made it difficult to utilize many existing Rust crates. Especially availability of suitable cryptographic functions was a challenge.

4.2 Measurements

We evaluated the performance of the SDK against the original Keystone SDK by measuring the communication delays between the host and the enclave application. A separate set of test programs with minimal functionality was developed to ensure that the measurements indicate the overhead caused by the SDK, not the applications using it. In addition, we used a patched version of Keystone Eyrie, since the standard version always copies data back to enclave on *ocalls*, not based on the amount the host application is sending, but the maximum size of the target buffer in the enclave application, which causes unnecessary copying on shorter payloads.

Unfortunately, we were unable to acquire hardware for running Keystone and were forced to perform the measurements using a *QEMU* (QEMU team, 2022) virtual machine emulating RISC-V architecture on top of a x86_64 platform. The virtual machine had 4 GB of memory and one CPU. The use of QEMU caused two problems for the measurements. First, binary translation and emulation affect the measurements causing extra overhead and periodical spikes. Second, QEMU is not cycle accurate. Therefore, the measurements are not comparable to real hardware, but only directional. In order to reduce the interference of emulation, we measured the performance as instructions reported by QEMU, retrieved with the *rdcycle* instruction. This approach ignores the fact that instructions have different performance overhead. However, since the payloads are identical for the compared programs, we claim that the measurements give at least a coarse estimate of the performance difference between them.

Table 1: Performance of ocalls.

Data	O0				O3			
(kB)	C (k)	Rust (k)	Δ (k)	x	C (k)	Rust (k)	Δ (k)	x
0	7.75	13.58	5.83	1.75	7.79	8.78	0.99	1.13
1	9.11	14.78	5.67	1.62	8.95	10.06	1.11	1.12
4	12.63	18.46	5.83	1.46	12.66	13.71	1.05	1.08
16	27.35	33.09	5.74	1.21	27.31	28.34	1.03	1.04
64	85.90	91.74	5.84	1.07	85.84	86.91	1.07	1.01
128	164.01	169.90	5.89	1.04	164.06	165.02	0.96	1.01
256	320.30	326.20	5.90	1.02	320.26	321.30	1.04	1.00
512	632.82	638.71	5.89	1.01	632.72	633.82	1.10	1.00

Table 2: Enclave application binary sizes.

Program	O0 (kB)	O1 (kB)	O2 (kB)	O3 (kB)	O3-lto (kB)
C	8.72	5.08	5.02	5.02	5.02
Rust	17.12	13.02	13.02	13.02	8.86
Rust / C	1.96	2.56	2.59	2.59	1.76

Table 3: Performance of ecall emulation.

Data	O0			O3		
(kB)	Host (k)	Enclave (k)	x (ocall)	Host (k)	Enclave (k)	x (ocall)
0	73.78	76.58	5.64	43.26	43.14	4.91
1	73.82	77.67	5.26	46.00	47.08	4.68
4	80.31	85.10	4.61	51.40	53.55	3.91
16	105.63	114.35	3.46	75.74	81.95	2.89
64	208.08	232.58	2.54	178.14	199.83	2.30
128	346.00	391.08	2.30	316.34	358.91	2.17
256	621.42	706.47	2.17	591.30	675.34	2.10
512	1171.73	1338.83	2.10	1141.87	1306.12	2.06

Comparison of *ocall* performance with different payload sizes is presented in Table 1 for optimization levels *-O0* and *-O3*. For both levels, the values are reported as thousands of instructions. Each call sends and receives the number of bytes indicated by the first column. All the reported values are measured as an average of 250 calls to reduce measurement error.

The difference between the measurements, indicated by the Δ columns, show that the extra overhead of the Rust SDK does not depend on the payload size, but is constant. We assume the overhead is caused by use of higher-level abstractions and extra safety checks compared to the thinner layer of the Keystone SDK. For both cases, the majority of the instructions are expected to take place in the Keystone driver and the Eyrie, when data is copied between the shared memory and enclave private memory. On larger payload sizes, the difference becomes insignificant, as the ratio of the instruction counts of the Rust and the C program, presented in column x, indicates. Variation in the instruction count for identical *ocalls* is caused by the Keystone sometimes interrupting the enclave

and passing the control back to the host application to prevent the enclave from executing forever.

Binary sizes of the enclave applications for various optimization levels are presented in Table 2. All binaries were stripped from symbols and debug information to have result comparable to production code. The C and Rust programs are compiled using different compilers. Thus, the optimizations performed for them might be different even for identical optimization levels. The results show the Rust applications are around 2-3 times larger than their C equivalents. One of the reasons for this is that the Keystone SDK only contains a very minimal C layer, while the Rust SDK contains higher-level abstractions and more checks for the incoming and outgoing data. In addition, the linker script for the applications is different, causing some program sections to be a bit larger.

Rust SDK's *ecall* performance is represented in Table 3. Measurements were made for the enclave application, i.e., instructions to serve single call, and the host application, i.e., instructions to make a single call. The *x* column shows the ratio of the instruction count to that of an *ocall* with the same amount of payload data. In the host application, the instruction count is measured from the thread initiating the *ecall*, thus thread synchronization is included. The original C/C++ SDK doesn't support *ecalls*, thus these measurements are only shown for the Rust SDK.

The results show that an emulated *ecall* is approximately 2 to 6 times slower than an *ocall*. As mentioned in Section 3.4, both all the data needs to be copied one extra time during emulation. Since on larger payload sizes the number of measured instructions is dominated by the instructions needed to copy the payload data, the performance is as expected.

5 DISCUSSION

The Rust SDK API for enclave applications is rather minimalistic and Keystone Eyrie specific. To enhance portability to other enclave architectures, a more standardized enclave API could be implemented on top of our API. To our knowledge, no such API definition exists yet for the Rust language and most existing standard APIs only define C or C++ bindings.

Most real-world enclave applications need functionality that the Rust SDK does not yet offer, although some of it can be included using existing Rust libraries. One of the most important and commonly used features is persistent secure storage that is sealed to the underlying platform. Keystone derives data sealing keys for each enclave, which the Rust API also exposes to the enclave applications. However, persistent storage must be implemented with the support of the host application or operating system to store the encrypted data on the untrusted platform. As the implementation is complex, it makes sense to provide a common API and implementation in the enclave library. Other generic functionality that could be added into the API includes derivation of enclave specific keys and other common cryptographic operations, such as data signing and verification.

One motivation for our research is to decrease vulnerabilities in security critical software. Using Rust, or any other safe programming language, is not a silver bullet to increase security (Xu et al., 2021) or to guarantee that the enclave interface or applications are secure (Van Bulck et al., 2019). Safe programming languages are only enablers for writing more secure software. Nevertheless, the Rust language allows particular types of security issues, e.g., buffer overflows, to potentially be detected more easily and earlier during development. In addition, Rust language's strict type system and aliasing rules, applied also to memory and threads, enable stronger static analyses to be used for security assurance.

Thorough security analysis and hardening of our SDK is still work to be done before it can be claimed to accomplish our intended goal. Van Bulck et al. (Van Bulck et al., 2019) analyzed vulnerabilities in input sanitization at the boundary of enclave applications and untrusted code in several TEEs. They found several vulnerabilities in runtime implementations and at enclave ABI and API layer, some also in Keystone. Their insight to enclave security issues and proposed mitigations form a good starting point for our future work to ensure security of the SDK.

We limited our work to the enclave and host application APIs and their implementation. However, security of the enclave application depends on many layers, none of which can be overlooked. First, security of the enclave architecture, its ABI, and the implementation, in this case the Keystone Security Monitor, its driver for the untrusted OS, and the Eyrie runtime functioning as the OS for the enclave application. High-level security analysis of the Keystone framework was presented by Lee et al. (Lee et al., 2020). However, Keystone is still a research prototype and we are not aware if detailed security analysis and hardening of the implementation has been performed. Second, security of the API used by the applications and its implementation, i.e., in our case the Rust SDK. As said, this is still work to be done. Finally, security of the application built on top of the APIs. It is impossible to anticipate how the applications are written, and thus, to prevent or mitigate all the vulnerabilities in the runtime. Nevertheless, the

runtime layer should minimize the attack surface and help the developers to write more secure applications.

6 RELATED WORK

We are by no means the first to propose development of enclave applications in safe programming languages. Earlier work includes, e.g., Wang et al. (Wang et al., 2019) who designed a Rust API wrapper on top of the Intel SGX C/C++ SDK. Their API provides a comprehensive set of enclave functionality. The authors report around 5% performance overhead for the wrapper. Other Rust-based approaches include the Apache Teaclave SGX SDK (The Apache Software Foundation, 2022) and the Fortanix Rust Enclave Development Platform (Fortanix, 2022). However, all of these only support Intel SGX enclaves.

While most of the work revolve around Intel SGX architectures, some work exists also for the RISC-V architecture. Boubakri et al. (Boubakri et al., 2021) discuss porting of OP-TEE framework to RISC-V, enabling Trusted Applications to be written using the standardized GlobalPlatform API specifications commonly used for mobile device TEEs. Suzaki et al. (Suzaki et al., 2020) also selected the GlobalPlatform API as basis of their work that attempts to unify enclave APIs for different hardware architectures. They implemented the GlobalPlatform internal API on top of the Intel SGX's and Keystone SDK's enclave definition languages and compared their performance against the OP-TEE framework. Unfortunately, GlobalPlatform APIs are defined for the C language, and thus, retain the language problems with security.

Others have considering unifying enclave APIs using less standardized API definitions. For example, the Open Enclave SDK (Open Enclave, 2022) and Asylo (Asylo Authors, 2022) projects intend to simplify enclave development and enhance portability with enclave technology agnostic APIs. Both of these, too, only define C/C++ APIs.

Oak et al. (Oak et al., 2021) propose use of even high-level abstractions. In their approach, developers use Java annotations to partition the software between the host and the enclave and to define the interface available to non-enclave callers. In addition, they propose a type system to enforce information flow policies between the program partitions.

7 CONCLUSION

In this paper, we discussed our work towards enabling Keystone enclave applications to be developed in the Rust programming language instead of error prone C/C++ languages traditionally used for enclave application development. The aim of our work is to facilitate development of more secure enclave applications.

We discussed our Rust SDK and its implementation, published as an open-source project (Julku and Kylänpää, 2022), and presented early performance measurements. The results show that the Rust SDK incurs a small overhead compared to the original Keystone C/C++ SDK. The overhead is mostly caused by extra safety checks and higher-level abstractions.

Our work is by no means complete. Intended future work includes persistent secure storage APIs and better support for cryptographic operations. Furthermore, using safe programming languages in itself doesn't guarantee that the resulting programs are more secure. Therefore, thorough security analysis and hardening of the SDK still needs to be performed in the future.

REFERENCES

Asylo Authors (2022). Asylo. Available Online: https://asylo.dev/. Accessed: March 23 2022.

Boubakri, M., Chiatante, F., and Zouari, B. (2021). Open Portable Trusted Execution Environment framework for RISC-V. In *2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 1–8.

Cerdeira, D., Santos, N., Fonseca, P., and Pinto, S. (2020). SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432.

Chen, J. (2022). GitHub - rcore-os/buddy_system_allocator: A buddy system allocator in pure Rust. Available Online: https://github.com/rcore-os/buddy_system_allocator. Accessed: August 29 2022.

Fei, S., Yan, Z., Ding, W., and Xie, H. (2021). Security Vulnerabilities of SGX and Countermeasures: A Survey. *ACM Comput. Surv.*, 54(6).

Fortanix (2022). GitHub - fortanix/rust-sgx: The Fortanix Rust Enclave Development Platform. Available Online: https://github.com/fortanix/rust-sgx. Accessed: March 18 2022.

Gunn, L. J., Asokan, N., Ekberg, J.-E., Liljestrand, H., Nayani, V., and Nyman, T. (2022). Hardware platform security for mobile devices. *Foundations and Trends® in Privacy and Security*, 3(3-4):214–394.

Julku, J. and Kylänpää, M. (2022). Rust-based Enclave SDK. Available Online: https://github.com/vector-sdk. Accessed: December 15 2022.

Keystone Enclave project (2022). Keystone Enclave · GitHub. Available Online: https://github.com/keystone-enclave. Accessed: March 18 2022.

Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., and Song, D. (2020). Keystone: An Open Framework

for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA. Association for Computing Machinery.

Lu, T. (2021). A Survey on RISC-V Security: Hardware and Architecture. *CoRR*, abs/2107.04175. Accessed: March 18 2022.

Melotti, D. and Rossi Bellom, M. (2022). Attacking Titan-M with Only One Byte. Available Online: https://blog.quarkslab.com/attacking-titan-m-with-only-one-byte.html. Accessed: August 16 2022.

National Vulnerability Database (2022). NVD - CVE-2022-20233. Available Online: "http://nvd.nist.gov/nvd.cfm?cvename=CVE-2022-20233. Accessed: August 16 2022.

Oak, A., Ahmadian, A. M., Balliu, M., and Salvaneschi, G. (2021). Language Support for Secure Software Development with Enclaves. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16.

Open Enclave (2022). GitHub - openenclave/openenclave: SDK for developing enclaves. Available Online: https://github.com/openenclave/openenclave. Accessed: March 18 2022.

QEMU team (2022). QEMU. Available Online: https://www.qemu.org/. Accessed: August 29 2022.

Russinovich, M., Costa, M., Fournet, C., Chisnall, D., Delignat-Lavaud, A., Clebsch, S., Vaswani, K., and Bhatia, V. (2021). Toward Confidential Cloud Computing. *Commun. ACM*, 64(6):54–61.

Suzaki, K., Nakajima, K., Oi, T., and Tsukamoto, A. (2020). Library Implementation and Performance Analysis of GlobalPlatform TEE Internal API for Intel SGX and RISC-V Keystone. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1200–1208.

The Apache Software Foundation (2022). GitHub - apache/incubator-teaclave-sgx-sdk: Apache Teaclave (incubating) SGX SDK. Available Online: https://github.com/apache/incubator-teaclave-sgx-sdk. Accessed: March 18 2022.

Van Bulck, J., Oswald, D., Marin, E., Aldoseri, A., Garcia, F. D., and Piessens, F. (2019). A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1741–1758, New York, NY, USA. Association for Computing Machinery.

Wang, H., Wang, P., Ding, Y., Sun, M., Jing, Y., Duan, R., Li, L., Zhang, Y., Wei, T., and Lin, Z. (2019). Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2333–2350, New York, NY, USA. Association for Computing Machinery.

Xu, H., Chen, Z., Sun, M., Zhou, Y., and Lyu, M. R. (2021). Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Trans. Softw. Eng. Methodol.*, 31(1).