Secure Services for Standard RISC-V Architectures

Davide Bove
IT Security Infrastructures Lab
FAU Erlangen-Nürnberg
Erlangen, Germany

ABSTRACT

In systems security Trusted Execution Environments have been developed as a mean to offer additional security to existing complex system designs. In the past multiple vulnerabilities have affected TEE implementations like ARM TrustZone and Intel SGX, which is why the research community has been looking to identify and solve existing design flaws. Another branch of computer science looks at RISC-V, a modern processor architecture that allows everyone to use and extend it.

In this work, we analyze the current possibilities of the RISC-V architecture to provide TEE-related functionality while avoiding potential pitfalls and vulnerabilities early on in the design process. By looking at the current problems in established TEE frameworks, we implemented and tested actual services used by user applications and operating systems that implement common TEE features on a recent version of the standard RISC-V ISA. We found that the current technology can be used to implement file storage and cryptographic key management services without modifications to the standard. Unfortunately, our results show that RISC-V offers no solution to secure I/O communication with peripherals on a system, and therefore also no safe way to interact with the user in case of an OS compromise. We discuss potential solutions to this remaining problem.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; *File system security*; *Hardware security implementation*; *Software security engineering*.

KEYWORDS

risc-v, tee, trusted execution, secure storage

ACM Reference Format:
Davide Bove. 2022. Secure Services for Standard RISC-V Architectures. In *The 17th International Conference on Availability, Reliability and Security (ARES 2022), August 23–26, 2022, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3538969.3538998

1 INTRODUCTION

In security research we often need to make assumption on what to consider secure and what not. This is where Trusted Execution Environments (TEEs) are introduced, where software is split into "secure and trusted" components and "untrusted" ones. While TEE solutions on the market, such as ARM TrustZone and Intel SGX, are widely used today, security researchers constantly find and report flaws, bugs, and vulnerabilities in their implementations. Software and hardware vulnerabilities are sometimes caused by mistakes of developers, but often are also flaws in the design of the system itself. An example for this are the several side-channel attacks that affect Intel SGX, such as cache attacks [6], code-reuse [3], and many others [8]. Due to the widespread use of ARM TrustZone on mobile devices, research has uncovered several flaws in different vendor implementations, that the design of TrustZone does not prevent [5].

With RISC-V there is an architecture that is growing in usage in the industry, due to its open and license-free nature. It is therefore important to take a closer look at its security features, in order to evaluate and possibly shape the future developments of the architecture early on in the process. The big advantage of RISC-V is its extensibility, which allows both industry and academia to explore new ways to achieve security. For this work we are looking at standard RISC-V architectures, which **do not** include custom extensions. We take the ratified version of the RISC-V Instruction Set Architecture (ISA) and look at features that are required by the standard, such as Physical Memory Protection (PMP). Based on this, the following contributions are made:

(1) We present a technical implementation of a secure file storage based on standard RISC-V processors in section 3.
(2) We show how to use such a feature to implement a cryptographic key management service called SKeystore (see section 4).
(3) We examine the current state of Secure I/O on standard RISC-V devices and propose modifications to achieve secure communication between enclaves and peripherals in section 5.

Our work is heavily inspired by security features of current Android devices, which use ARM TrustZone for similar functionality. The presented services provide security to third-party user application while preserving the threat model assumptions of TEEs. Still, our work is not limited to mobile devices and is targeted at general purpose systems, such as desktops, servers and similar. The current implementation is not suited for most IoT devices on the market, as these devices only implement a limited subset of the RISC-V standard (see more in section 2).

2 BACKGROUND

This section provides the background to understand the concepts used in this work, the relevant details of RISC-V and the design of Keystone, which our work is based on.

ARES 2022, August 23–26, 2022, Vienna, Austria

ACM ISBN 978-1-4503-9670-7/22/08…$15.00
https://doi.org/10.1145/3538969.3538998

2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a processor feature that offers some kind of isolation inside the hardware for processing data. The goal is to have a secure execution of specific applications even in the presence of a compromised OS. Such applications, called trusted apps or trustlets, are responsible for processing security-critical operations, such as encryption or authentication. Often they also provide services that handle sensitive user data, which may need extra protection from strong attackers. These isolated environments are often called *enclaves*, which is the predominant name we will use in this work.

Popular implementations of TEEs are ARM TrustZone and Intel SGX, based on ARM and x86 architecture respectively. For the RISC-V architecture there is MultiZone[1], a commercial TEE for IoT devices. Apart from that, there are mostly academic projects, such as Keystone [7], TIMBER-V [10] or MI6 [4].

TEEs are characterized by a hardware-imposed logical separation of code and data. This is true for TrustZone, which has separate memory regions for the *Secure World*. In our work we use the Keystone model, which relies on RISC-V features.

2.2 RISC-V

The RISC-V architecture includes several features that are needed to implement TEE features. The most important one is included in the Privileged Architecture[2], which defines different privilege levels where software can run. To achieve some security on the platform level, a RISC-V system needs at least two execution levels. The mandatory Machine Mode (M-Mode) has the highest privileges and is supported by every system. The second one is usually the User Mode (U-Mode), which is intended for traditional user applications. A third mode, the Supervisor Mode (S-Mode) is supposed to be for operating systems and mostly supported by devices that are performant enough to run a full OS (e.g., the Linux kernel).

Most academic RISC-V solutions published before 2019 use software implementations or modified hardware extensions to achieve isolation of memory and processes. Since 2019 the RISC-V ISA version has been frozen, such that vendors and developers could target a common version. This version also includes Physical Memory Protection (PMP), a hardware-based specification that describes a memory access protection mechanism for unprivileged applications. In short, PMP allows blocking or granting access to memory regions from M-Mode to the lower-privileged S- and U-mode. From the mentioned academic solutions, only Keystone uses PMP to implement enclaves, making it faster and able to run on standardized RISC-V hardware.

Keystone consists of three main components: the Secure Monitor (SM) that runs in M-Mode, an Enclave Runtime (RT) that runs in S-Mode and enclave apps that are executed in U-Mode (see Figure 1). Using PMP the SM protects an enclave process from memory access by other processes. Therefore, normal user apps and secure enclave apps can run at the same time. Since enclaves are isolated from the OS, they rely on the RT for OS-like features (e.g., file system access). Also, the RT usually includes some version of the *libc* standard C library. In general, enclave apps are split between normal code and sensitive code, such that only the sensitive functions are executed in the secure environment.

By design, enclaves in Keystone are based on a specific Root-of-Trust (RoT), which might be a secure coprocessor (e.g., a TPM), a read-only secret that is created during production or similar. A RoT ensures that an attacker cannot compromise key parts of the system, such as the firmware. Most cryptographic operations for integrity and security purposes are based on this RoT. For this work this means that we assume a secure RoT and thus a secure (and bug-free) Keystone SM and build our security assumptions accordingly.

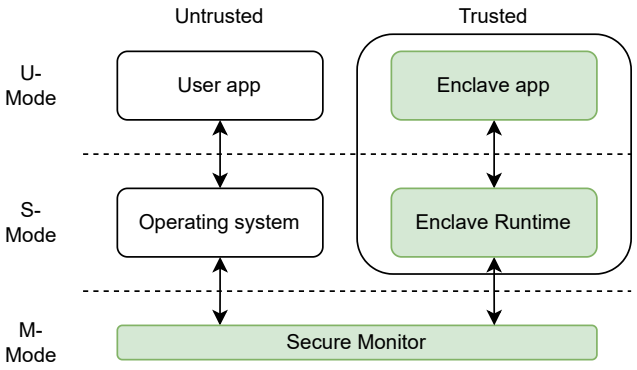

Figure 1: Architecture of enclaves based on Keystone Enclave.

3 SECURE STORAGE

The Secure Storage executes read and write operations in secure memory. Due to the lack of private storage on processors, the system needs to rely on the untrusted file system managed by the OS. The underlying hardware can be flash storage, a hard disk or any on-chip storage in the case of embedded devices. Per our definition of the threat model, the file system is managed by the OS and therefore *untrusted*. A common solution for this problem is encryption. For our implementation, we define architectural requirements that meet security standards. These requirements are then used to evaluate the security of our implementation.

3.1 Requirements

Our requirements for a Secure Storage implementation are based on TEE standards by GlobalPlatform, specifically its definition of "Trusted Storage API for Data and Keys" [9]. We change the wording and include Keystone-specific terminology to avoid confusion:

(1) The Secure Storage may be backed by non-secure resources as long as suitable cryptographic protection is applied.
(2) The Secure Storage must be bound to a particular device, which means that it must be accessible or modifiable only by authorized enclave apps running in the same enclave and on the same device as when the data was created.
(3) The Secure Storage should be able to hide sensitive key material from the enclave itself.

[1] https://hex-five.com/multizone-security-tee-riscv/
[2] https://riscv.org/technical/specifications/

(4) Each enclave has access to its own storage space that is shared among all instances of that enclave but separated from the other enclaves.

In short, the Secure Storage guarantees that only the enclave that created a file can access it. Further, the content should be bound to a specific device, meaning that a copy of the enclave binary cannot open a copy of the file on a different device.

Regarding the design goals, we want to address two negative aspects of TrustZone-based solutions analyzed in the last section:

- A security vulnerability in an enclave must not expose secrets of other enclaves.
- The TCB should be reduced, therefore reusing code on the host side and allocating functional features on lower-privileged levels, wherever possible.
- The module should run in user instead of supervisor mode.

Altogether, the secure storage is meant to be a simple library for enclave applications to store sensitive data on persistent memory, while the application does not have to care about encryption and key management.

3.2 Implementation

For file-based encryption the system needs to securely store and manage encryption keys. In other TEE systems, the storage of cryptographic material is done on trusted storage, or managed by a single instance (e.g., a trusted application). Since the system is not designed to include trusted storage or enclave-exclusive system partitions, the individual private keys cannot be saved on the file system itself. For our solution we dynamically derive cryptographic keys from the Sealing Key, which enclave apps can request through their runtime. The Sealing Key never leaves the context of the SM, and due to the nature of the Sealing Key, any derived keys are bound to the actual device where the enclave is executed. This means an attacker cannot use a compromised SM to extract keys or file contents.

Using this method, the cryptographic keys only reside in volatile RAM. Using the memory protection of PMP, we can protect them during usage and flush the memory after freeing the memory space, therefore removing them from memory again. This ensures that neither a malicious user application nor the OS can access them.

The secure storage is a static library that can be included into enclave applications. It contains two functions:

- `encrypt_and_write` encrypts a memory block and writes it to disk. It is guaranteed that plaintext is not leaked from the enclave.
- `read_and_decrypt` reads an encrypted file from disk into memory.

The encryption is handled by the *libsodium* crypto library. Our implementation is based on the *XChaCha20* cipher, which allows using a random nonce in the encryption process. Therefore, the encryption of two files with identical content results in two distinct cipher texts, which helps prevent "known-plaintext" attacks.

A central aspect of the Secure Storage is that it does not require to know a secret in order to decrypt and encrypt files. While the user handles the file contents, the service does the heavy lifting and uses a derived key for encryption. This key derivation is based on the Data Sealing feature of Keystone, which derives a key that is bound to three entities:

(1) A device-specific Root-of-Trust, in general an asymmetric key pair unique for the current device.
(2) The hash of the Secure Monitor, therefore binding a key to a specific version of the SM.
(3) The hash of the enclave binary, therefore binding a key to a specific enclave version.

The first token ensures that no derived key can be copied to a second system. This is useful to prevent replay attacks. The SM hash ensures that the key is not accessed with an older, potentially vulnerable version of the SM. This can mitigate downgrade attacks, where attackers deploy a vulnerable version of secure software in order to exploit the system. The inclusion of the enclave hash makes keys specific to a specific software and prevents other enclave applications from accessing the same key. A system where multiple enclaves need to access the same key material therefore need to invoke the same enclave in order to retrieve it.

This derivation system has multiple consequences for the architecture of a TEE system. First, every enclave that wants to use the Secure Storage needs to generate its enclave-specific encryption keys beforehand. This is ultimately performed by the Secure Storage library, but still requires the enclave to call it. An alternative approach would be to have a central service that handles all keys and only offer specific keys to enclaves after checking their binary hash. This last approach would be preferable if one has a system that exclusively runs vendor code and does not allow users or third-party developers to add their own software.

As described above, an update to the SM or the enclave binary invalidates all the keys, as the key derivation would need to know the previous version's hash in order to derive a key. This means that in order to update the software, there needs to be a migration procedure where old keys are regenerated with the new binary. For Secure Storage that means that every file ever stored by the service would need to be decrypted and encrypted again. Of course this is the worst case scenario for performance and usability, and the Secure Storage would need to be adapted as it does not keep track of all the files it saves. There are smart ways around this problem, such as using a (secure) static key for encryption and using the derived key to encrypt this key instead. Therefore, when the derived key changes, the enclave only needs to re-encrypt the static key instead of the whole file.

3.3 Evaluation

For the evaluation of the Secure Storage, we refer back to the requirements we defined in subsection 3.1 according to the GlobalPlatform TEE Internal API [9]. Requirement (1) is fulfilled since every file written to the untrusted world is encrypted with a derived key only accessible to the enclave. Thus, a file can only be read from the enclave that initially wrote it. (2) is fulfilled likewise. The key derivation function uses a device-specific key (namely the security monitor private key), binding a file to a particular SoC, provided that the Secure Boot procedure is bug-free.

Regarding (3), the Secure Storage library cannot hide any sensitive key material from the enclave application itself since it is part of the application and does not run in a separate process or thread.

Nevertheless, the secure storage does not save any key material in a global variable, but only in the stack frame of the encryption and decryption function, respectively. Hence, clearing these buffers before returning from the function would be sufficient to protect an enclave application from accidentally exposing any key material. Altogether, we cannot entirely hide key material from the enclave app; however, we can guarantee that an enclave does not have access to keys from another enclave.

The principle of the key derivation satisfies retirement (4). When launching multiple instances of the same enclave on the same device, they have the same hash in the SM and hence, can derive the same encryption keys.

Our implementation completely lacks a defense mechanism against replay (rollback) attacks. Although it would be sufficient to implement a rollback defense in the RT to meet requirement (5), it does not match our threat model, where we mistrust the entire RT. The authors claim that the Keystone Security Monitor can be adopted to support rollback defense though [7]. We leave this aspect to future work and hence, do not fulfill requirement (5).

3.4 Performance

In order to evaluate the performance of our approach, we set up test benchmarks and timed the execution of read and write processes. More specifically, we set up a QEMU machine with Keystone and our modified Secure Monitor and executed three different benchmarks:

- `reference_native`: Execute regular file reads and writes with increasing buffer size
- `reference_eapp`: The same as above, but executed inside a Keystone enclave (no encryption)
- `estorage_eapp`: Our Secure Storage implementation that reads and write a file (with encryption)

All executions are repeated 1000 times for every buffer size. The buffer starts with 1 byte up to 65536 bytes (0.5 megabits), the latter being a maximum limit of the Keystone Eyrie runtime which fails to page the app properly with bigger buffer sizes. The startup time of the enclave is excluded from all calculation, as well as the initial file creation (see Listing 1 and Listing 2). From the collected measurements we extract the median for every buffer size, the resulting plot is shown in Figure 2.

The graph shows that there are two types of performance overheads. The first is produced by the TEE itself, which makes file reads and writes slower, the second one is induced by our implementation. The difference between them is shown in Figure 3, where our solution added between 44.5% and 70.21% of performance overhead, while the difference between native execution and enclave execution adds significant overhead between 125% and 430.74%. In comparison to the overall overhead of the TEE, our solution is therefore not bad, considering there is also encryption involved, no hardware acceleration or compiler optimizations are used.

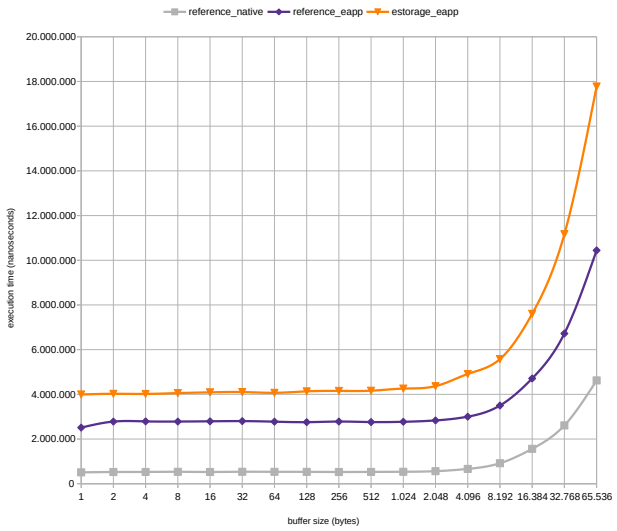

Figure 2: Performance measurements of native reference file (read/write) accesses compared to Secure Storage solution

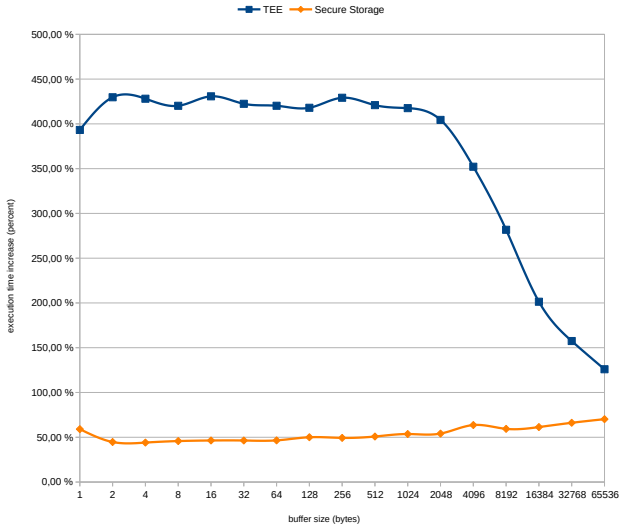

Figure 3: Performance overhead of the TEE compared to the Secure Storage implementation

4 KEY STORAGE AND MANAGEMENT

A central aspect of TEE security is based on storing and managing cryptographic keys in order to sign or encrypt messages to the outside world. This is especially important for public-key cryptography, where a private key is often deployed by the manufacturer and needs to be protected from unauthorized usage. A compromise of these keys not only allows attackers to take over a device, but can also compromise the security of a whole product line, as previous research has found that TEEs are not trivial to update and revocation of trusted apps sometimes leads to more vulnerabilities [5].

4.1 Requirements

Inspired by the Android KeyStore[3] service, we defined several requirements that a comparable service has to offer in terms of confidentiality, integrity, and authentication:

(1) **User applications may not access private keys:** Unprivileged user applications may not access sensitive cryptographic enclave keys.
(2) **The operating system may not access private keys:** The operating system or any privileged systems (e.g., kernel, device drivers) may not access sensitive cryptographic enclave keys.
(3) **Enclave apps may request cryptographic operations:** All crypto operations are requested in an enclave. The execution of the cryptographic operations, such as the key generation and encryption or decryption processes, are performed in an isolated environment. The enclave application can share public information and results to a user app, if necessary.
(4) **Enclaves may not access the keys of other enclaves:** Key pairs and key material created in one enclave are bound to the enclave and may not be accessible by other enclaves. An attacker should not be able to craft a malicious enclave to access private keys.
(5) **Private key material is exclusively stored encrypted:** When saved on the untrusted file system, an attacker should not be able to access, read or copy the key material.
(6) **Enclaves may access information to allow remote attestation by a third party:** In order for a third-party server to verify that a specific key is bound to a specific enclave and that a key pair (and its identity) was provably created in a secure enclave, an enclave should be able to request cryptographic proof of the origin of key pairs.

4.2 Implementation

The *SKeystore* is a static library that can be referenced and included in enclave applications. It offers a number of useful functions for enclaves:

- `generate_key` is used to create a key pair. With the `get_key` method, the enclave gets several public information about the key, such as the hashes of the enclave and the SM. This blob of information is signed by the Secure Monitor and can be used to establish remote attestation by a third party.
- `delete_key` to invalidate a key
- `sign_data` can be used to sign a variable-length text string (which later can be verified using the public key)
- `decrypt_data` takes a message previously encrypted through the corresponding public key and returns a decrypted message back to the enclave.

For the cryptography SKeystore uses the *libsodium* crypto library[4]. For digital signatures from the Secure Monitor we use *Ed25519* key pairs. For every enclave, SKeystore creates *X25519* key pairs. On creation, they are signed by the SM, and the signature is included into the key attestation provided for any key pair. As part of the attestation a third party can verify the integrity and authenticity of the enclave key using the public key of the SM. Any private key is securely stored using the Secure Storage library (described in section 3) in an encrypted format, such that only the SM can use them.

In addition to the library, we extended the SM through a plugin in order to generate the aforementioned key pairs. To communicate between enclave and SM plugin, we also added a system call to the Eyrie runtime.

4.3 Evaluation

From a performance perspective, we observed the same overhead as shown in section 3, since the SKeystore is based on the Secure Storage and does not introduce any different cryptographic operations compared to the functions included in Keystone. The security evaluation of the SKeystore is based on the set requirements defined above. Since the service handles the key generation, the private key is never exported or handed over to any application. By using the Secure Storage from section 3, only SKeystore can create and read private keys. Therefore, requirements (1), (2) and (5) are fulfilled. This comes with the drawback that every service needs to request cryptographic operations (requirement (3)), which may impact performance when multiple enclaves require such operations. Any enclave can only access and use keys generated from the same enclave (specifically from the same combination of enclave binary and Secure Monitor). Even if a malicious enclave manipulates the SKeystore into loading key information of other enclaves, the SKeystore is not able to derive the correct decryption key to use the private key. Even though we satisfy requirement (4), the same enclave on two different devices cannot share the same keys, even though they might use the same identical binary. Lastly, enclaves have access to metadata for performing Remote Attestation with a third party (requirement (6)). For this the enclave may send the remote party its own public key signatures as well as the public key of the Secure Monitor, forming an attestation chain that third parties can verify.

We also implemented a Secure Monitor plugin that handles the initial key generation, since the SKeystore service does not have access to the SM private key. In general increasing the size of the SM decreases the overall security [5], which is why we significantly reduced the required changes of our implementation to effectively only 41 lines of code. The cryptographic key generation reuses the crypto library and key derivation of the SM, so that impact on code size is minimized there.

The SKeystore lacks a protection against accidental file corruptions and due to the method used for key derivation, any update to the SM or the enclave binary renders keys generated with the previous versions incompatible. The latter problem concerns multiple aspects of TEE security and especially the Keystone framework, therefore any solution for a safe update system should also solve the problems with SKeystore.

5 TOWARDS SECURE I/O

Most TEE solutions provide some basic functionality to handle external peripherals. TrustZone for example implements TrustZone

[3] https://developer.android.com/training/articles/keystore
[4] https://libsodium.org

Protection Controller to mark memory regions as only accessible from the Secure World [1]. The integrity of communications between a processor and different peripherals is not particularly protected, as the latter ones are often managed by drivers in the OS, which in turn can be compromised. While there are ways to ensure confidentiality and integrity for external peripherals, the known methods often make the assumption that processor and peripheral share a common secret.

In our solution, we explore and analyze standard input/output inside a secure enclave using Keystone. Keystone uses *OpenSBI*[5] as a middleware between bootloader and Secure Monitor, and includes specific *universal asynchronous receiver-transmitter* (UART) implementations for various hardware platforms. Using these, we can write and read from standard input without relying on the operating system (which could be compromised according to our threat model). In our implementation our goals are:

- Communicate a message to the user without OS interference (preserving integrity and confidentiality)
- Receive a message from the user (through the keyboard) without OS interference (preserving integrity and confidentiality)

5.1 Secure Output

In order for enclaves and SM to print text to the console, Keystone includes a function `sbi_putchar` that communicates to the device's UART port. Therefore, we used this method to instruct an enclave to write text to the user console. Since we are directly interacting with the UART port, the host application and the OS do not receive any data while the enclave is active, even when directly reading from it. Data is only received by the OS when the enclave application is paused, closed or killed.

From a security perspective, we are preserving the integrity of enclave messages (as long as the process is not killed during text output). In addition, we noticed a second effect when simultaneously connecting to the machine directly and per SSH. When our test application (that prints several pieces of text to the console in a loop) is started through the native console (`/dev/console`), everything is printed correctly to the screen as expected. If we start it from an interactive SSH session, no text is printed to the SSH session screen. Instead, the text messages are shown on the native console again. Due to how we communicate directly with the underlying hardware, other applications on the system do not interfere with our output. Therefore, our Secure Output method is bound to the primary screen or output device and does preserve the confidentiality of enclave messages.

5.2 Secure Input

In order to allow an enclave to read data from the user, we implemented a plugin for the Eyrie runtime of Keystone that exposes a function `sbi_getchar()` to the enclave. Through this function the enclave can poll a character from the native console input and read it into the enclave memory. Since the function call is non-blocking, our test application needs to continuously read a character in a loop in order to receive whole sentences. This is a limitation of the input driver that may impact usability of our method. As before, the OS can not actively interact with this input method. It can not inject own text into any enclave buffer (since PMP is used on enclave memory), it can not read the UART while our enclave is active.

There is a second limitation to our approach, which is partly influenced by Keystone and its inner workings. Assuming that we try to get a longer input from the user (e.g., a password), the enclave app needs to continuously call the read function (which returns NULL if no character is received). Keystone includes a DOS protection inside its SM, which should prevent enclaves from starving out the other processes on the system and blocking the execution of the system indefinitely. Therefore, a continuously blocking enclave app, one which is trying to read multiple characters, will be interrupted after some time to allow the OS or other enclaves to execute. The user might not be aware of this while entering some text, as enclaves can be dynamically stopped and resumed. For our experiments, we had to deactivate this protection or else the user could be entering parts of the "confidential" message into the OS or some other user application.

Even with DOS protection disabled, we could not claim with confidence that our approach works with multicore processors. While the execution of our test enclave application would be running on one core, another application (user or OS) could be running on a second core and reading from the standard input. In multiple experiments our test enclave app could exclusively read characters from the user even when a malicious user app simultaneously tried to read input, but we were not able to explain this "desirable" outcome. Therefore, we cannot exclude that a race condition could occur where the input method from the OS is prioritized over our Secure Input method. This is a problem that should be considered in future research.

In conclusion, our Secure Input approach guarantees integrity for user input such that neither the OS nor malicious actors can inject "fake" input into the enclave application. For confidentiality our solution does not cover concurrency problems and questions around exclusive access to user input were raised during our experiments.

6 CONCLUSION

This paper demonstrates how to use standard RISC-V features to implement common security services for user applications. We show that with RISC-V it is possible to achieve secure file storage even on untrusted file systems, while fulfilling all security requirements of such a service. Based on the above, we implemented a cryptographic key management service that allows enclave apps to securely perform cryptographic operations. There is a performance overhead when introducing compute-bound cryptographic features, which might be noticeable in a real-world environment. We argue though that the overhead, which is primarily caused by the cryptographic computation, might be significantly reduced with the introduction of the RISC-V Scalar Cryptography Extensions[6], an official standard extension that promises significant hardware acceleration for cryptographic operations.

We also describe how the current standard might be sufficient to provide limited input and output capabilities, but suffers from the typical problems of TEE designs that do not explicitly consider it.

[5] https://github.com/riscv-software-src/opensbi

[6] https://riscv.org/blog/2021/09/risc-v-cryptography-extensions-task-group-announces-public-review-of-the-scalar-cryptography-extensions/

Increasing I/O support through our method means using drivers in the M-Mode, which in turn leads to an increased amount of code to be trusted. In this regard, the current hardware specification does not consider secure communication channels between the system and external peripherals. There are academic solutions to improve on this (such as CURE [2]), but there are currently no ambitions by RISC-V development groups to adopt such models, at least not publicly.

There are a few roadblocks that we want to address in order to conclude our contribution of our paper. There is not much research on the deployment of enclaves in real-world environments, as such supply-chain considerations are usually not in the scope of research efforts. But for the viability of security measures, we need to consider questions around the ability to update enclaves or secure firmware as well as provide solutions for when our main security assumption breaks and these software parts are compromised. As the usage of RISC-V increases, so does the attention of the security research community, and we can expect similar findings in vendor implementations as with the other technologies. Therefore, we would also encourage research and developments of software tools to support the development of enclave applications. With our work we hope to contribute to this in order to increase the overall security of RISC-V and its TEE capabilities, such that we might benefit from it in the near future.

ACKNOWLEDGMENTS

We would like to thank Jonathan Schmidt for the original implementation and Nils Eiling for the further development. This research was supported by the German Federal Ministry of Education and Research (BMBF) as part of the Software Campus project (Förderkennzeichen: 01IS17045).

REFERENCES

[1] ARM. 2020. About the TrustZone Protection Controller - ARM Developer. https://developer.arm.com/documentation/dto0015/a/about-the-trustzone-protection-controller. Accessed: 2020-08-13.

[2] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1073–1090. https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani

[3] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 1213–1227. https://www.usenix.org/conference/usenixsecurity18/presentation/biondo

[4] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 42–56. https://doi.org/10.1145/3352460.3358310

[5] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1416–1432. https://doi.org/10.1109/SP40000.2020.00061

[6] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*, Cristiano Giuffrida and Angelos Stavrou (Eds.). ACM, 2:1–2:6. https://doi.org/10.1145/3065913.3065915

[7] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 38:1–38:16. https://doi.org/10.1145/3342195.3387532

[8] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. A Survey of Published Attacks on Intel SGX. *CoRR* abs/2006.13598 (2020). arXiv:2006.13598 https://arxiv.org/abs/2006.13598

[9] GlobalPlatform Technology. 2021. TEE Internal Core API Specification. https://globalplatform.org/specs-library/tee-internal-core-api-specification. Last access: 2022-01-30.

[10] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/timber-v-tag-isolated-memory-bringing-fine-grained-enclaves-to-risc-v/

A APPENDIX

```
for (int i = 1; i <= 1024*64; i *= 2) {
    char test_buffer[i];
    for (unsigned int r = 0; r < NUM_REPETITIONS; ++r) {
        // start timing
        struct timespec start, stop;
        clock_gettime(CLOCK_REALTIME, &start);

        fp = fopen(TESTFILE_PATH, "w");
        fwrite(test_buffer, sizeof(char), i, fp);
        fclose(fp);

        fp = fopen(TESTFILE_PATH, "r");
        fread(test_buffer, sizeof(char), i, fp);
        fclose(fp);

        clock_gettime(CLOCK_REALTIME, &stop);
        // end timing

        uint64_t time_nsec = ((stop.tv_sec - start.tv_sec
    ) * 1000000000L + stop.tv_nsec - start.tv_nsec);
        printf("%d,%lu\n", i, time_nsec);
    }
}
```

Listing 1: Getting execution time of native file read and write instructions

```
for (int i = 1; i <= 1024*64; i *= 2) {
    char test_buffer[i];
    for (unsigned int r = 0; r < NUM_REPETITIONS; ++r) {
        size_t ret_size;
        // start timing
        struct timespec start, stop;
        clock_gettime(CLOCK_REALTIME, &start);

        encrypt_and_write(TESTFILE_PATH, test_buffer, i);
        read_and_decrypt(TESTFILE_PATH, &ret_size);

        clock_gettime(CLOCK_REALTIME, &stop);
        // end timing

        uint64_t time_nsec = ((stop.tv_sec - start.tv_sec
    ) * 1000000000L + stop.tv_nsec - start.tv_nsec);
        printf("%d,%lu\n", i, time_nsec);
    }
}
```

Listing 2: Getting execution time of Secure Storage instructions