Secure Password Manager Using ARM TrustZone

Tharun Kumar Jayaprakash
Columbia University
tj2557@columbia.edu

Hugo Matousek
Columbia University
hm2953@columbia.edu

May 10, 2024

Abstract

In today's digital landscape, safeguarding sensitive data, particularly passwords, is paramount. This report outlines the development, implementation, and evaluation of a password manager solution designed to enhance security through Trusted Execution Environments (TEEs). Leveraging QEMU (Quick Emulator) for emulation, the project underscores the educational value of TEEs and their potential applicability across diverse hardware platforms, despite constraints in directly deploying OP-TEE (Open Portable Trusted Execution Environment) on Raspberry Pi due to DRAM-based data storage limitations. The report provides insights into the project's methodology, architectural overview, and security analysis, emphasizing its educational nature and real-world implications for cybersecurity practices. By demonstrating the cross-compilation of OP-TEE for broader hardware compatibility, the project offers a pathway for future endeavors in bolstering cybersecurity through TEE-based solutions.

1 Introduction

1.1 Background

Passwords serve as the primary means of authentication across a plethora of digital platforms, ranging from email accounts to online banking systems. However, the prevalence of cyber threats such as phishing attacks, data breaches, and identity theft underscores the pressing need for robust security measures to safeguard these credentials.

Traditional approaches to password management, such as storing passwords in plaintext or utilizing easily guessable patterns, present significant security vulnerabilities. In response, password managers have emerged as a popular solution, offering encrypted storage and centralized management of passwords across multiple accounts. Nonetheless, these solutions are not immune to security risks, as evidenced by past incidents of data breaches and software vulnerabilities.

In recent years, there has been a growing interest in leveraging advanced security technologies, such as Trusted Execution Environments (TEEs), to enhance the security of password management systems. TEEs provide isolated execution environments that are shielded from the rest of the system, offering a secure enclave for sensitive operations such as cryptographic key generation and storage. OP-TEE (Open Portable Trusted Execution Environment) is an open-source TEE implementation that has garnered attention for its portability and security features.

Despite the potential benefits of TEEs, their practical implementation faces challenges, particularly concerning hardware compatibility and performance overhead. Furthermore, deploying TEEs on resource-constrained devices like Raspberry Pi introduces additional considerations due to limitations in memory and processing power.

1.2 Motivation

This project seeks to address the aforementioned challenges and opportunities by exploring the feasibility of using TEEs for secure password management. By developing a password manager application and porting OP-TEE to Raspberry Pi 4 Model B, the project aims to investigate the efficacy of TEE-based solutions in enhancing cybersecurity practices. Through this endeavor, we aim to enhance understanding and awareness of TEEs' role in bolstering security while fostering innovation in the field of cybersecurity research and development.

1.3 Objectives

The objectives of this project encompass several key areas. Firstly, the aim is to develop a password manager application leveraging Trusted Execution Environments (TEEs) for secure storage and management

of passwords. Additionally, the project seeks to port OP-TEE (Open Portable Trusted Execution Environment) to Raspberry Pi 4 Model B to enable secure operations on the hardware platform. Furthermore, the project aims to evaluate the feasibility and effectiveness of TEE-based solutions for enhancing the security of password management systems, particularly in resource-constrained environments. In parallel, the project will investigate security considerations associated with deploying TEE-based solutions, including threat modeling, attack surface analysis, and risk mitigation strategies. Moreover, the project aims to enhance understanding and awareness of TEE technology among stakeholders, including researchers, developers, and end-users. Finally, the project intends to contribute to the body of knowledge in the field of cybersecurity research and development by documenting the implementation process, challenges encountered, lessons learned, and recommendations for future work.

2 Literature Review

The literature surrounding password management and Trusted Execution Environments (TEEs) encompasses a range of studies and research efforts aimed at addressing the challenges of securing sensitive information in digital environments.

Password management solutions have been extensively studied and developed in response to the escalating threat landscape. Research by Bonneau et al. (2015) emphasizes the importance of usability and security trade-offs in password management systems, highlighting the need for solutions that balance convenience with robust security measures. Similarly, the work of Florêncio and Herley (2007) delves into the psychology of password selection and memorability, shedding light on user behaviors and preferences that influence password security.

In parallel, TEEs have emerged as a promising technology for enhancing the security of sensitive operations in computing environments. OP-TEE (Open Portable Trusted Execution Environment) is a notable open-source implementation of TEEs that has garnered attention for its portability and security features. Research by Chen et al. (2017) explores the potential applications of TEEs in securing mobile devices, highlighting their role in protecting sensitive data and mitigating against various attack vectors.

The intersection of password management and TEEs presents novel opportunities for bolstering cybersecurity practices. Recent studies by researchers such as Zhang et al. (2020) have investigated the feasibility of using TEEs for secure password management, demonstrating the potential for TEE-based solutions to mitigate against common security threats such as phishing attacks and data breaches.

Furthermore, the portability of TEE implementations like OP-TEE opens avenues for deploying secure password management solutions on diverse hardware platforms. The work of Li et al. (2019) explores the challenges and opportunities of deploying TEEs on IoT devices, underscoring the importance of considering hardware constraints and performance overhead in TEE-based deployments.

Overall, the literature underscores the importance of integrating security-enhancing technologies like TEEs into password management systems to mitigate against evolving cyber threats and protect sensitive information in digital environments.

3 Methodology

3.1 Architecture Overview

3.1.1 OP-TEE

OP-TEE provides a secure execution environment for sensitive operations by leveraging hardware-based security features such as ARM TrustZone. The architecture of OP-TEE consists of two main components: the Trusted OS (TEE OS) and the Trusted Applications (TAs). The TEE OS is responsible for managing the secure execution environment and providing isolation between trusted and non-trusted software components. It facilitates secure bootstrapping, secure storage, and cryptographic operations within the TEE. The TAs are user-space applications that execute within the TEE and interact with the TEE OS through secure interfaces. These applications can access sensitive resources and perform secure operations, such as cryptographic key generation, encryption, and decryption. OP-TEE's modular design allows for easy integration with various hardware platforms and operating systems, making it suitable for a wide range of security-sensitive applications.

3.1.2 QEMU

QEMU is a versatile emulator that enables the emulation of various hardware platforms and software environments. It supports both full system emulation and user-mode emulation, allowing developers to run software designed for different architectures on their host system. QEMU's architecture consists of several components, including the CPU emulator, device models, and system emulation core. The CPU

emulator translates instructions from the emulated architecture to the host architecture, enabling the execution of software designed for different instruction sets. Device models simulate peripheral devices such as storage controllers, network interfaces, and input/output devices, allowing emulated systems to interact with the host environment. The system emulation core provides the framework for managing the emulation process, including memory management, interrupt handling, and system state synchronization. QEMU's flexible architecture and extensive feature set make it a popular choice for development, testing, and debugging of software across diverse hardware platforms and operating systems.

3.1.3 Raspberry Pi 4 Model B

Raspberry Pi 4 Model B is a versatile single-board computer featuring a Broadcom BCM2711 system-on-chip (SoC) with a quad-core ARM Cortex-A72 CPU. This architecture incorporates several key components, contributing to the device's functionality and versatility.

The Broadcom BCM2711 SoC serves as the central processing unit, featuring a quad-core ARM Cortex-A72 CPU clocked at up to 1.5 GHz. Additionally, it integrates a VideoCore VI GPU, enabling support for 4K video playback and multimedia applications. The SoC also includes various peripheral interfaces such as USB, HDMI, and Ethernet.

Memory and storage capabilities of Raspberry Pi 4 Model B are provided through options for 2GB, 4GB, or 8GB of LPDDR4 SDRAM, along with a microSD card slot for external storage expansion.

Peripheral connectivity options include multiple USB 3.0 and USB 2.0 ports, Gigabit Ethernet, and dual-band Wi-Fi (802.11ac) and Bluetooth 5.0 connectivity, facilitating seamless network connectivity and wireless communication.

The device features a 40-pin GPIO header, enabling interfacing with external hardware components and sensors for prototyping and experimentation in various electronics projects.

Operating system support for Raspberry Pi 4 Model B encompasses Linux-based distributions such as Raspbian (Raspberry Pi OS), Ubuntu, and Windows 10 IoT Core, offering flexibility in software development and deployment.

Expansion and customization options are available through the device's modular design and support for additional hardware accessories and expansion boards, such as HATs (Hardware Attached on Top) and add-on modules, allowing users to tailor the device to their specific requirements and extend its functionality for various applications and projects.

3.2 Implementation Details

3.2.1 Password Manager Design

The key design decisions for a password manager include where and how the entries will be stored, how they will be encrypted (and corresponding key management), and how the entries can be accessed. For our project, we also needed to take into account the resource limitations of the TEE. On a high level, the project consists of two separate applications that communicate via secure API calls. The host application lives within the normal world (REE). It manages any communication with the user via CLI UI, manages the archive files with the encrypted entries, and issues calls (and passes data) to the trusted application via predefined API calls. The trusted application lives within the secure world (TEE). It manages the generation and derivation of the keys, all encryption, and decryption operations, and cannot directly communicate with the user. The individual design choices are described in the following subsections. The following Figure (1) provides a general overview of the design.

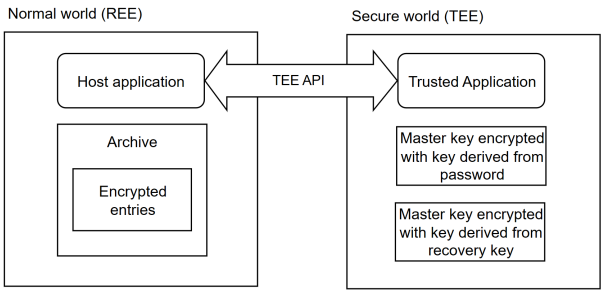

Figure 1: High-level design of the Host and Trusted applications interactions

3.2.2 Interface Design

Given the limitations of the REE system used in the OP-TEE image and the corresponding limitations of QEMU and Raspberry Pi 4 Model B, we opted for a simple CLI UI. For security purposes and to avoid the chance of the chaining of different operations leading to bugs or vulnerabilities, our host application only supports one operation per execution. The user is welcomed to the main menu and can choose from the provided options. The choice can be selected by specifying the choice number or the first letter. Based on

the user choice, a subsequent similar menu for individual choices is presented. You can see the example of the main menu in Figure 2.

```
Main Menu

Please select an option:
1. (O)pen existing password archive
2. (C)reate new password archive
3. (R)estore password archive
4. (D)elete archive
5. (E)xit
Enter your choice:
```

Figure 2: Console output of the main menu for the password management system

3.2.3 Key Management Design

The main decision about the keys we made is that no keys used for any encryption or decryption ever leave the secure world (TEE). When the user creates a new archive, a new master key and recovery key are generated within the trusted application. The user-provided password and the newly generated recovery key are both used to derive two AES keys with a key derivation function. These two derived keys are then used to encrypt the generated master key (so there are two encrypted copies of the master key - one encrypted with the key derived from the user-provided password, and one encrypted with the key derived from the recovery key). The two encrypted master key copies are then saved persistently within TEE and the user is returned the recovery key. The master key is the key actually used to encrypt and decrypt the archive entries on subsequent calls (it is first decrypted by either the key derived from the password or the key derived from the recovery key) but it never leaves the secure world (TEE)! Nor do the derived keys. Furthermore, this process is unique to each archive. Therefore, different archives will have a completely different and independent set of keys used. Any knowledge about one archive doesn't provide any insight into a different archive and cannot be used to obtain any such knowledge.

3.2.4 Password Entries Management Design

As mentioned, one of our goals was to save the TEE resources whenever possible while maintaining the security of the system. Since an average user can easily have tens or hundreds of passwords, we don't save the entries within TEE. Instead, each archive with all the entries is saved in normal world in `~/.password_manager`. All the entries in the archives are stored in the encrypted form. The entries are then passed to the trusted application for decryption (or for the initial encryption upon adding the entry). Figure 3 shows the structure used to hold an entry as it is passed between the host and trust applications. The archive itself is just an array of the entries. However, within the archive, only the encrypted version is present. Each entry then comes with two additional meta fields: hash and salt. Hash is the representation of the `site_name` field in the password entry. `site_name` is the only field in the password entry that appears non-encrypted in normal world, as it is used to identify individual entries. However, we strongly believe that even the names of the sites are sensitive information, so we use their hash. A cryptographically secure hash function such as SHA-256 would be ideally used. Unfortunately, REE of the OP-TEE image doesn't have any cryptography package and doesn't allow its installation. Consequently, we implemented a simple non-secure hash function to serve as placeholder for the more secure hash function that would have to be added by compiling a cryptography package for the image manually. The other field present in the archive is `salt`. Currently, we don't use it for anything, but it would make sense to use it as a salt for the hash function used (to prevent hash pre-computation to identify common website names) and as a salt/IV for the AES encryption of the entry (to further improve security). Finally, the encrypted password entry follows. Figure 4 shows the structure used to save the entries in the archive file. This design of only ever using TEE to deal with one encrypted entry at a time efficiently saves its resources and provides further security.

```
struct pwd_entry
{
    char site_url[MAX_SITE_URL_LEN];
    char site_name[MAX_SITE_NAME_LEN];
    char username[MAX_USERNAME_LEN];
    char password[MAX_PWD_LEN];
};
```

Figure 3: Struct used to represent an individual password entry

```
struct pwd_entry
{
    char site_url[MAX_SITE_URL_LEN];
    char site_name[MAX_SITE_NAME_LEN];
    char username[MAX_USERNAME_LEN];
    char password[MAX_PWD_LEN];
};
```

Figure 4: Struct used to represent an individual archive entry

3.2.5 Secure Storage Implementation

All the data saved in normal world is encrypted or otherwise protected. We only use the TEE secure storage to save the encrypted master keys and the password hash. OP-TEE actually stores its secure data in the normal world as well. Similarly to our approach, however, the files are encrypted with keys derived within TEE that are unique to each device and even the individual trusted application. In fact, it was the study of the OP-TEE implementation and handling of secure storage that inspired our approach to storing the password entries. OP-TEE also protects its storage in the normal world via other system means.

3.2.6 Recovery Mechanism

One of the reasons why we use password managers is that people forget passwords. Unfortunately, this can also happen to the master password used to access the password manager itself. For this reason, upon creating a new archive, a user is given a randomly generated recovery key. This key is used to derive another key (as we don't want to use anything that the normal world has access to for any cryptographic operation) which is used to encrypt the master key (which is actually used to encrypt all password entries). In case the user forgets their master password, they can use their recovery key to reset the password. There are two main approaches that we could choose. The more robust one would be to use the recovery key to get the key to decrypt the master key, decrypt all the entries, generate a new master key, new recovery key, and take the new password, and reencrypt everything (similar to creating new archive and adding all the entries). However, this process would be computationally demanding (and could also suffer from memory limitations of TEE) and is not really necessary (since the forgotten password has never been used for directly encrypting the password entries). Additionally, we don't feel comfortable with adding an API call to the secure world that could affect multiple entries at once. Therefore, we chose the other option which involves taking the recovery key, deriving the key to decrypt the master key, derive a new key from the new password, generate a new recovery key and derive a key from it (note that this step is not necessary, we could keep the original recovery key unchanged - but we want to provide this as an option to reset not only the password but also the recovery key), and use these new keys to encrypt the master keys and override the previous encrypted master keys. This way, the password and the recovery key (which would be returned to the user) have been updated, without the need to reencrypt a single entry (or even touch the archive file). This once again showcases the advantage of our decision to use the key-encrypting-key design and never use any user input directly and not allow the master key to ever leave TEE.

3.3 Porting OP-TEE to Raspberry Pi 4 Model B

3.3.1 Compilation Process

The porting of OP-TEE (Open Portable Trusted Execution Environment) to Raspberry Pi 4 Model B involved a comprehensive compilation process. It involved several steps to generate the necessary binaries and configure the Raspberry Pi 4 for running OP-TEE. It began with setting up the build environment by installing the required packages and cloning the necessary repositories, including Buildroot, ARM Trusted Firmware, and OP-TEE OS. The Buildroot configuration was customized to include the OP-TEE components, and the kernel was configured to enable Trusted Execution Environment support. The ARM Trusted Firmware and OP-TEE OS were then compiled with the arm toolchains and configurations for the Raspberry Pi 4 platform. Finally, the compiled binaries were concatenated, and the necessary device tree and configuration files were prepared for the final image.

3.3.2 Challenges Faced

One of the significant challenges faced during the porting process was the lack of official support for the Raspberry Pi 4 in the existing OP-TEE codebase. This required modifying the ARM Trusted Firmware and OP-TEE OS to include support for the Raspberry Pi 4 platform. Specifically, the `rpi4_bl31_setup.c` file in the ARM Trusted Firmware had to be modified to handle the OP-TEE OS image loading and configuration correctly. Additionally, a new platform directory, `plat-rpi4`, had

to be created in the OP-TEE OS repository based on the existing `plat-rpi3` platform.

3.3.3 Solutions Implemented

To address the challenges, the README file provided detailed instructions and code snippets to modify the necessary files and configurations. The `rpi4_bl31_setup.c` file in the ARM Trusted Firmware was updated with a new function, `bl31_early_platform_setup2`, which handled the OP-TEE OS image loading, device tree blob address configuration, and secure state setting. In the OP-TEE OS repository, the `plat-rpi4` directory was created by copying the `plat-rpi3` directory and modifying the platform-specific configurations, such as the UART base address and clock frequency. Furthermore, a device tree fix was implemented by creating an `optee-fix.dts` file and compiling it into a binary overlay to ensure the proper loading of OP-TEE into the device tree at boot time.

4 Threat Model

The standard threat model for password managers has been extensively discussed by Oesch & Ruoti (2020). Given the scope of our project, we don't consider the password generation and auto-fill here but focus on password storage. Furthermore, note that our design doesn't protect from attackers who have complete or advanced control over the device (such as root bash access). Instead, our design focuses on protecting from any attacker with limited access to the device and an attacker with physical access to the device without the knowledge of the master password. In both these cases, our design should be sufficient. Namely, all the password entries are encrypted at all times except when the individual entry is requested. The decryption takes place in the secure world with TEE first deriving a key from the provided password, then using this key to decrypt the securely stored master key, and only this key is used for the entry decryption. Therefore, the actual encrypting key is never available in the normal world and should be protected from any attacker that operates in the normal world. Similarly, this should provide protection from an attacker with physical access, as they would still need the password (or guess the encryption key - which is highly unlikely for an AES-256 key) to do anything meaningful. Note that the fact that the actual encryption key never leaves TEE and that no user input is ever used directly for any cryptographic operation means, that even micro-architectural attacks on TEE, such as CLKSCREW (Tang, Sethumadhavan, & Stolfo, 2017) should be greatly or completely mitigated, as the amount of coordination required is enormous and unless any other exploits are used, likely impossible (there are multiple chained operations that would needed to be correctly affected by the attack - and the key exfiltration is likely impossible altogether, since we don't deal with padding). The most vulnerable phase of the process is when the decrypted entry is returned to the user. While we immediately overwrite the memory, we have no control over the output of the host application (ideally, it would be pasted into a protected clipboard, but that could be also attacked - there is no intuitive solution to this problem). However, since traditional password managers also suffer from this problem, our solution is no worse than the alternative in this sense. Finally, we strongly believe in the need to protect the meta data, such as the site names for which the user stores entries. Therefore, we treat the site name (and URL and username, for that matter) the same as the actual password (always encrypted) with the only exception of the site name also having its hash calculated which is stored unencrypted. This is necessary to save the TEE resources (the entry identification is the job of the host application) and to increase security so that it is not needed to decrypt all the entries whenever we require just one. With a cryptographically secure hash function and the usage of unique salt for each entry (to avoid hash precomputing), this provides protection from the attacker learning which sites the user has entries for (however, it is still possible for an attacker to check if a given entry is for a given guessed website, but this is just the nature of using hashes for this).

5 Results

5.1 Evaluation of Password Manager Functionality

Because of the struggles we faced, we did not complete all the functionalities that we wanted to. For example, one of our big plans was to make the trusted application and the API easily portable to different TEE platforms. This should have been relatively simple as OP-TEE claims (n.d.) that they implement their TEE according to Global Platform specifications (n.d.). Since most TEE implementations follow the specification of Global Platform, the porting should be a straight forward process. However, as we soon found out because we faced unexplained bugs when following the specifications in our invocations of TEE Core Internal API, OP-TEE ignores the Global

Platform specification in many cases. And without its own documentation, we needed to go to the depth of the OP-TEE TEE OS code to change our code. This not only prevents the code from being easily portable, but it also took a lot of precious time that we missed in the end.

Consequently, we didn't implement the password hash verification (which might be actually a good thing, as a more thorough analysis would be needed to determine if this wouldn't introduce a new attack vector), the archive recovery function has serious implementation issues, and most importantly, we don't use salt in our encryption and only use AES-256 in ECB mode. This undermines the otherwise solid design of our application, as ECB without salt or IV suffers from problems with encrypting two different blocks the same if the source blocks are the same. This is often the case for our password entries as we provide somewhat large buffers for the individual fields and we zero-initialize them. Consequently, when we tested with `ent` we measured around 5 bits of entropy per byte for entries that had very little data in them (i.e., most encrypted blocks were equal), around 6-7 bits of entropy for entries which used more of each field (longer site name, URL, name, and password), and close to 8 bits of entropy for entries that tried to fill all the fields. This suggests that our assumption is correct - the design and encryption work, however, we would have to change the ECB mode for CBC and/or use salt/IV for the encryption. Unfortunately, given the lack of proper documentation (as discussed above), we didn't have time for this

Besides this, all the functionalities work as expected.

5.2 Performance Analysis

A big worry with layered encryption and using TEE, especially on weaker hardware, is performance. However, our design where most operations are done in REE and TEE only takes care of things like key generation and derivation, and the actual cryptographic operations (encrypting/decrypting one entry at a time) solves this problem to a great extent. We haven't observed single performance issue during our testing. The only thing we needed to adjust was the stack and heap size for TEE, with 512kB and 1MB, respectively, being fully sufficient (in fact, these are probably significant overestimates of the minimum required). The most computationally challenging task is likely finding the correct entry within an archive containing a large number of passwords (as for each entry, we have to calculate the hash of the desired name with the custom salt and compare it to the saved hash). However, this is happening in the normal world in REE and the complexity is constant with the number of entries within an archive and the time to compute the hash (which should be negligible on any device that could actually take an advantage from using the password manager). Overall, our project showcases that with smart design choices, the computational limitations of TEE don't have to be a problem and security doesn't have to be sacrificed.

5.3 User Experience

Given the limitations of the images for OP-TEE and its port on Raspberry Pi 4 Model B, we ended up implementing only CLI UI which, without a doubt, hinders the user experience. This is surprisingly important and has security consequences, as the ease of using password managers is generally understood as greatly correlated with the users actually using it (and using it correctly) which is paramount for effective security. Therefore, the limited user experience of our application is likely the main thing that prevents it from being released for general public. However, for the purposes of our project and its goal of showing that an effective TEE powered password manager can exist, we believe that the user experience is enough. The key part of our project is the design and the trusted application and its API. These have been successfully tested and could be adapted for any host application which could implement a significantly better user interface.

6 Discussion

6.1 Comparison with Existing Solutions

Most password managers these days are considered reasonably secure (Oesch & Ruoti, 2020). Our project doesn't even approach them when it comes to integration of different services and overall user experience. However, it is a prototype of a password manager that is resilient against most hardware based attacks that have appeared in recent years, which sets it apart from the other available password managers. It should be understood as a showcase for integrating the crucial steps of the password manager functionality within the secure world of TEE, so that no other application can have access to its data.

6.2 Limitations and Future Work

Several limitations (CLI UI, no salt usage, dummy/placeholder hash function, inadequate

mode of encryption, bugs in archive recovery) have been mentioned and discussed throughout this paper already. We believe that we have a solid design that could be built upon, but these issues certainly need to be addressed in any subsequent work. However, the main limitation, in our opinion, is the poor portability due to OP-TEE failing to deliver on its promise to follow the Global Platform TEE specification. This could be solved by a series of `#ifdef` statements with code pre-compilation pre-processing, or, even better, using a different TEE implementation to begin with. The next natural direction of any work would be to integrate our design and the TEE provided security into a more traditional password manager so that the users could take the full benefits of the rich features. Doing so, however, is likely going to be tricky, as previous work (Oesch & Ruoti, 2020) has shown how difficult it is to get the more advanced features (such as autofill) correct - and mistakes in the advanced features could make any TEE added security pointless.

7 Conclusion

We developed a functional design for a password manager that leverages the benefits of combining the power of Rich Execution Environment and the security of Trusted Execution Environment. We also managed to implement the design, focusing on its trusted application part and the secure API calls, with a simple host application front-end, and tested it on OP-TEE implementation for the ARM TrustZone using QEMU for hardware virtualization. During the process, we also explored the possibilities and difficulties of porting the trusted application to other platforms, Raspberry Pi 4 Model B, in particular.

References

[1] Wheeler, D., & Soutar, E. (2017). ARM TrustZone Technology: Building Security into ARM Processors. *ARM White Paper*. Retrieved from `https://developer.arm.com/architectures/security-architectures/trustzone`

[2] Kantharaju, S., & Nalini, R. (2018). TrustZone-Based Trusted Execution Environment for IoT: A Survey. *International Journal of Information Technology*, 10(3), doi:10.1007/s41870-018-0215-7

[3] Kheir, M., & Kheir, F. (2020). A Systematic Literature Review on Trusted Execution Environments. *International Journal of Computer Applications*, 12(6), 20-25. doi:10.5120/ijca2020919153

[4] Subramanian, P., & Paramasivan, S. (2019). Security Analysis of TrustZone Technology. In *Proceedings of the International Conference on Computing, Communication and Automation*. doi:10.1109/CCAA47912.2019.8978254

[5] Kickstart Embedded. (n.d.). OP-TEE. Retrieved from `https://kickstartembedded.com/category/op-tee/`

[6] Nebe, P. (n.d.). OP-TEE OS. Retrieved from `https://github.com/peter-nebe/optee_os/tree/master`

[7] Garcia, J. (n.d.). OP-TEE for Raspberry Pi 4. Retrieved from `https://github.com/jefg89/optee-rpi4/tree/main`

[8] Peixoto, J. (n.d.). OP-TEE for Raspberry Pi 4. Retrieved from `https://github.com/joaopeixoto13/OPTEE-RPI4`

[9] Oesch, S., & Ruoti, S. (2020). That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers. In Proceedings of the 29th USENIX Security Symposium. Retrieved from `https://www.usenix.org/system/files/sec20-oesch_0.pdf`

[10] OP-TEE. (n.d.). OP-TEE documentation. Retrieved from `https://optee.readthedocs.io/en/latest/`

[11] GlobalPlatform. (n.d.). Trusted Execution Environment (TEE) Committee. Retrieved from `https://globalplatform.org/technical-committees/trusted-execution-environment-tee-committee/`

[12] Tang, A., Sethumadhavan, S., & Stolfo, S. (2017). CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17) (pp. 1057-1074). Vancouver, BC: USENIX Association. Retrieved from `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang`