Pearl-TEE: Supporting Untrusted Applications in TrustZone

Wei Huang Vasily Rudchenko He Shuang Zhen Huang David Lie
University of Toronto

ABSTRACT

Rising concerns about mobile security have motivated the use of architectural features such as ARM TrustZone to protect sensitive applications from compromise by malicious applications or a compromised OS. However, many TEE OSes (which run in TrustZone) currently assume all applications in TrustZone are trusted, and thus do not provide strong isolation guarantees between them. The benefit of this is that TEE OSes can be simple, allowing them to provide a high-assurance trusted computing base (TCB). However, unlike how arbitrary third-party mobile applications can be installed onto a smartphone, the need for mutual trust among all applications running inside TrustZone prevents the installation of 3rd party applications on the TEE OS. In this paper, we identify the key properties that define application code that may wish to use TrustZone and show that a standard TEE OS can be extended to support multiple, mutually distrusting applications in TrustZone with less than a 3% increase in the TCB. We realize our ideas in Pearl-TEE, a novel TEE OS prototype we have implemented that can provide mechanisms specific to the needs of TrustZone applications, including isolation for execution, secure persistent storage, and support for network communication. We find that Pearl-TEE imposes less than 20% performance overhead on applications.

ACM Reference Format:
Wei Huang Vasily Rudchenko He Shuang Zhen Huang David Lie. 2018. Pearl-TEE: Supporting Untrusted Applications in TrustZone. In *3rd Workshop on System Software for Trusted Execution (SysTEX '18), October 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3268935.3268936

1 INTRODUCTION

TrustZone is a powerful mechanism for reducing the attack surface of security-sensitive applications on the ARM platform. TrustZone implements a hardware-based Trusted Execution Environment (TEE), which reduces the attack surface of an application by enabling it to execute code that is isolated and protected from other, lower-assurance code on the system, such as the general-purpose operating system and applications. TrustZone applications execute in an isolated environment with their own TEE Operating System (TEE OS). This isolation provides strong confidentiality and integrity guarantees for applications executing in the TEE and the data processed by the TEE applications.

The existence of such a useful security mechanism might beg the question: Why isn't TrustZone used by security-sensitive applications in general? Indeed, neither iOS nor Android exposes a public API that enables application code to access TrustZone. While strategic business exclusivity may be one reason for this omission, a concrete technical reason is that TEE OSes vendors are not sure their OSes will hold up to intentional attack from applications, thus small size and simplicity are required to maintain the high-assurance level of the TEE OS. Because of this, for current TEE OSes, it is simpler just to limit which applications can run in the TEE, so that only a small number of mutually trusted TEE applications that are signed and vetted by the smartphone vendor are allowed to benefit from TrustZone.

In this paper, we propose Pearl-TEE, which removes the need for TEE applications to mutually trust each other for less than a 3% increase to the trusted computing base (TCB) size of the TEE OS. To do this, we first define an *abstract TEE application* model, which encompasses the types of applications that can benefit from TrustZone's security guarantees. We then implement a Pearl-TEE prototype TEE OS on top of an existing TEE OS to estimate the amount of extra code that needs to be added to support the abstract TEE application model. Finally, we demonstrate that our prototype is realistic by porting three mobile payment applications and one secure chat application to run in TrustZone. This paper makes the following contributions:

(1) We describe an abstract TEE application model and show how common applications map onto that model.

(2) We describe our implementation of Pearl-TEE, a prototype TEE OS that satisfies the requirements of our TEE application model.

(3) We evaluate the increase to the TCB of modifying an existing TEE OS to implement Pearl-TEE and find that it only increases the TCB by 3%. We evaluate Pearl-TEE's performance and practicality by porting four existing applications and find that Pearl-TEE imposes less than 20% performance overhead.

The remainder of this paper is organized as follows: Section 2 introduces hardware dependencies and the TEE application model with examples of secure applications. In Section 3 we give details of the Pearl-TEE system design. We present how Pearl-TEE and TEE application examples are implemented and evaluate our prototype in Section 4. We conclude this paper in Section 5.

2 DEPENDENCY AND APPLICATION MODEL

2.1 Hardware Dependency

ARM TrustZone An ARM SoC with TrustZone partitions software and hardware resources, such as memory and access to peripherals, into a secure- and normal- world, with the secure world isolated from the normal world by hardware protection. In the secure world, a small secure TEE OS provides basic security operations as well as a TEE in which applications can run. The benefit of this division between a large, normal-world OS and a small TEE OS in the secure world is that applications running in the TEE have a smaller TCB, isolated from the large and complex normal-world OS. To maintain these benefits, the TEE OS must be high-assurance, meaning that it has passed a higher standard of code auditing and testing during its development. In the current ecosystem, only trusted, signed applications can run the TEE OSes, which means that all TEE applications must have their code vetted before being allowed to run in the TEE.

Secure Element To attest the integrity of software running on a device to a remote party, a trusted hardware device in the form of a Secure Element (SE) or a Trusted Platform Module (TPM) is required. Both of these are security-hardened processing elements capable of securely storing signing keys and performing cryptographic operations with those keys. An SE typically contains a private signing key, whose public verification key is certified by a well-known entity. Access to the SE is gated by a secure or verified boot procedure so that only a properly certified TEE OS image can access the SE and have it sign messages. As a result, the SE key can be used to *attest* the integrity of a TEE OS and the messages it generates to a remote party. Signatures made by SE keys can be trusted because the keys are never exposed to the normal-world OS or applications running on the application processor. In our design, we assume the presence of an SE, but our design would work equally well with a TPM.

Trusted Touch Screen A trusted touch screen provides a secure channel between the user using a device and the TEE software. Typically, the touch screen is shared with the normal-world OS, but provides exclusive access to either the normal-world OS or the TEE OS at any one time. Thus, when used by the TEE OS, it provides exclusive read and write access to the TEE OS. Content displayed by the TEE OS and input read by the TEE OS cannot be tampered with, interrupted or read by the normal-world OS.

2.2 TEE Application Model and Examples

We first describe several common classes of applications that currently use, or may benefit from, the security provided by TrustZone. We then identify shared qualities of these applications to construct our abstract model.

Payment Applications Malware targeting mobile payment applications has become more prevalent [18]; thus, some phone manufacturers now use TrustZone for application execution to isolate their payment applications from malware and vulnerabilities in the normal-world OS. Given that they must sign operations as well as verify the authenticity of responses sent to them, they must store long-term cryptographic keys securely. Examples of payment applications that use TrustZone include HuaweiPay and SamsungPay; on the other hand, PayPal and AliPay are examples of applications that do not.

Digital Rights Management Digital Rights Management (DRM) systems intend to control the use, copy and distribution of copyrighted software and data. Unlike payment applications, DRM systems do not take user input from the trusted touch screen, but only require a secure path to output video content. DRM applications also need to communicate with remote servers and must protect and use keys to decrypt content for users. For example, XBox and PlayStation encrypt game-related data so that they cannot be copied across multiple devices [2].

Authentication Authentication applications are used to provide the second factor in two-factor authentication in the form of a pseudorandom sequence of secrets that are generated by the device. The pseudorandom sequence is seeded by a secret and the server and device are synchronized so that the server knows what indexes in the sequence it might receive. As a result, such an application must be able to store secrets and operate on them in a way that is protected from tampering or observation. Many software two-factor authentication applications, such as Google Authenticator and Microsoft Authenticator, exist, but as far as we know, none currently use TrustZone to protect their execution.

Integrity Monitoring Integrity monitoring applications in TrustZone, such as Samsung Knox [5], continuously monitor normal-world applications and the normal-world OS to ensure that their execution patterns imply that the software has not been compromised. While the operation of Samsung Knox is not public, we surmise that they execute in TrustZone to protect their integrity against a possibly compromised normal-world OS and also to protect cryptographic secrets with which they may securely communicate the state of their device to a remote server for mobile device management (MDM) functionality.

Secure Chat Many mobile chat applications provide secure end-to-end chat with message encryption, e.g., Telegram and Xabber. They communicate with an external server via messages, require secure key storage for encryption keys, and need to run in isolation from one another. Secure keys can be used to encrypt and sign data (such as chat logs) stored in the normal-world OS. Furthermore, to ensure that users see correct messages, they require a secure channel to the screen. Currently, we are not aware of any secure chat applications that can use TrustZone.

Many applications cannot benefit from TrustZone because they cannot run on existing TEE OSes, as both open-source [4, 6, 14] and closed-source [12] TEE OSes do not isolate applications from each other, nor do they isolate applications from themselves, meaning that all TEE applications on a user device must mutually trust each other. Due to this limitation, device manufacturers only run those TEE applications that come pre-installed on the device and simply disallow new applications. Recent academic work has proposed the use of a trusted language runtime for multiple applications in the TEE [13]; however, the trusted runtime does not provide a level of isolation that would prevent a malicious application from compromising another co-resident application. This trust requirement severely limits the accessibility and availability of TrustZone to applications, as a single party must be trusted by all applications in the TEE to curate and vet all the code in the TEE.

From the descriptions above, we can see that many security-sensitive phone applications share certain common traits. We tabulate these traits in Table 1 and describe them in more detail below.

Key storage All example TEE application classes have cryptographic secrets that they need to store and protect from exposure. However, none have large amounts of persistent data to store in Trustzone. Cryptographic secrets are typically small and do not change frequently. While large contents can be protected by using cryptographic keys for encryption and then stored by the normal-world OS. Thus, TEE applications only require a small amount of infrequently written persistent storage.

Table 1: Traits of each class of TEE applications

Application class	Key storage	Execution integrity and confidentiality	Secure communication with server	Secure communication with user
Payment	✓	✓	✓	✓
DRM	✓	✓	✓	✓
Authentication	✓	✓	✓	✓
Integrity Monitoring	✓	✓	✓	
Secure Chat	✓	✓	✓	✓

Execution integrity and confidentiality Because most applications process sensitive data or use cryptographic secrets for encryption/decryption or signing/verification, their execution must be protected from observation and tampering. Thus, TEE applications require an execution environment that is isolated from both the normal-world OS and other TEE applications.

Communication with a remote server Most TEE applications need to communicate with a remote server, even if infrequently. Even in the two-factor authentication example, the application still needs to receive secrets from a remote server and synchronize the pseudorandom sequence with the server.

Trusted path to user All applications except for integrity monitoring need to have secure interactions with the user of the device.

3 PEARL-TEE DESIGN

We have examined current open-sourced TEE OSes such as OP-TEE [6], TrustyTEE [4] and SierraTEE [14], and found that to provide the requirements outlined in our TEE application model, they rely on the assumption that all TEE applications are trusted. For example, they do not provide isolation between TEE applications for storage, memory or execution. If we remove the assumption that all applications are trusted and installed along with the TEE OS, as Pearl-TEE does, this raises several challenges. As in all TEE OSes, Pearl-TEE relies on TrustZone hardware to isolate it and all TEE applications from the normal-world, but also isolates itself from TEE applications, and TEE applications from each other. TEE OS uses a system call bridge to communicate with the normal-world OS, allowing it to access resources controlled by the normal-world OS as shown in Figure 1. We now describe how Pearl-TEE implements each of the requirements above in more detail.

3.1 Storage Isolation

Current TEE OSes provide shared access to persistent storage, so that any TEE application can read the values stored by any other TEE application. If TEE applications can be malicious, Pearl-TEE must provide storage isolation between TEE applications. However, unlike generic applications, TEE applications do not need to store large files or to share files with other applications, and generally write to storage once and only read it afterward. Typically TEE applications only need persistent storage for cryptographic secrets.

Pearl-TEE's storage implementation is optimized for storing keys, which are small, infrequently accessed, and written only once. To implement this, Pearl-TEE encrypts all values stored by a TEE application with an *application-specific key*, K_{AP}. The derivation of K_{AP} relies on the *Pearl-TEE master key*, K_{PT}, which is randomly chosen the first time Pearl-TEE runs and every time the device is factory-reset, and is stored in protected persistent storage that is only made accessible to Pearl-TEE via secure boot. The K_{AP} is computed by encrypting the hash of the TEE application with K_{PT}, thus ensuring that the K_{AP} is unique for each application, but at the same time unique for each device, while also revealing nothing about the K_{PT} of the device. The TEE application can then use K_{AP} to encrypt and sign any data it wants to store and then store it into the file system of the normal-world OS. In our Pearl-TEE prototype, this is achieved by a system call bridge between Pearl-TEE and the normal-world Android OS as described in Section 4.1. This method of securing data comes with the caveat that data stored in this way is vulnerable to a replay by the normal-world OS. However, since such storage is assumed to be only written once, this excludes any replay attack. In addition, any such storage must be recreated if the TEE application binary is upgraded as the hash will change, causing a new K_{AP} to be generated. This means that any old data or keys will become inaccessible. This has the benefit of automatically protecting data generated by the new version from any compromises that may have occurred against the old application, as well as making roll-back attacks ineffective for accessing data generated after the upgrade to the new application.

By not implementing a generic file system, Pearl-TEE is able to prevent significant bloat to the TCB of the TEE OS. For comparison, our storage-related implementation is only ~150 lines of code (LOC) excluding standard cryptographic libraries, while a generic file system such as FAT [1] has over 20K LOC and EXT4 in Linux 4.2 has over 36K LOC.

3.2 Memory and Execution Isolation

Current TEE OSes also do not isolate the address spaces of TEE applications from each other, nor do they prevent TEE applications from interrupting each other or preventing each other from running. If TEE applications can be malicious, Pearl-TEE must provide memory and execution isolation between applications, preventing them from accessing each other's memory, and ensuring that each gets a fair amount of uninterrupted execution time. Pearl-TEE must both protect its memory and execution from tampering by malicious TEE applications, and protect the memory and execution of TEE applications from each other. In addition, to prevent denial of service by a malicious TEE application, Pearl-TEE must also ensure that every TEE application gets the ability to run on the CPU.

For isolation of TEE applications, Pearl-TEE runs at a higher privilege level than TEE applications. Specifically, Pearl-TEE runs

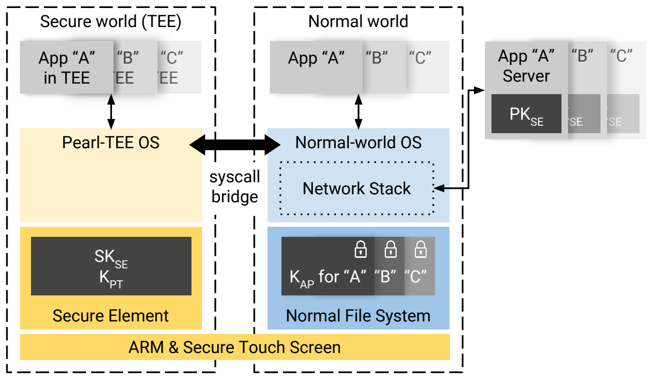

Figure 1: ***System overview of Pearl-TEE with multiple TEE-enabled applications.***

Table 2: LOC of Different System Modules in the Implementation and Comparison with Other OSes

Category	System Unit Name	Lines of Code
Server	Proxy Server	4,500
Normal-world OS	Android AOSP	25,000,000
	Normal-world user module	1,800
	Normal-world kernel module	1,500
	User application – Payment	3,000
	User application – Chat	50,000
TCB	OP-TEE	130,000
	Change to OP-TEE (C)	1,700
	Change to OP-TEE (ASM)	50
TEE User Space	TEE payment application	900
	TEE-Chat	200

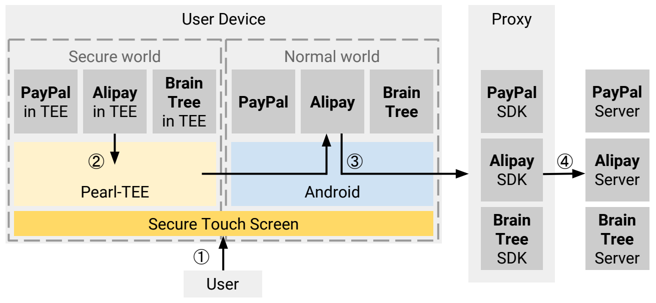

Figure 2: ***System overview of a mobile payment framework as an example for TEE applications.***

at EL1 (exception level 1) while the TEE applications run at EL0 (exception level 0). This enables Pearl-TEE to use hardware mechanisms such as the MMU to protect its code and execution state from tampering by a malicious application.

To prevent a malicious TEE application from denying other applications to run, Pearl-TEE implements batch scheduling. In the absence of user interaction, each TEE application is given a fixed amount of time, T_{batch}, before it is terminated and another TEE application to run. In contrast, normal-world OSes generally use complex interrupt-driven CPU multiplexing and scheduling to share execution resources among applications while isolating application execution state. Pearl-TEE provides API functions for both displaying output to the user and waiting for input from the user. TEE applications may avoid being terminated after T_{batch} by using T_{ui_write} and T_{ui_read}. It can display output to the user at least every T_{ui_write}, or read input from the user, giving the user T_{ui_read} to provide input. T_{ui_write} would apply, for example, to a DRM application that is displaying a movie to the user, while T_{ui_read} would allow the user ample time to input a message to a secure chat application. As a result, T_{ui_read} and T_{ui_write} are generally longer than T_{batch} (i.e., on the order of seconds or minutes, while T_{batch} may be on the order of milliseconds) because an application that wants to abuse them cannot do so silently without displaying something that the user will notice. In all cases, Pearl-TEE also provides a hard-reset option that allows the user to forcibly terminate a misbehaving application by holding down the power button on the device.

3.3 Trusted Path and Communication

Similar to current TEE OSes, Pearl-TEE also provides the ability to communicate over the network and interact with the user via trusted path devices. Like current TEE OSes, Pearl-TEE assumes hardware support for the trusted path, and provides exclusive access for a TEE application to that device so that it can communicate with the user without being interrupted by another TEE application.

Network communication is implemented by forwarding network system calls to be executed by a user-space module in the normal-world OS, which contains a full network stack. To secure communication between a TEE application and a remote server, the TEE application should use an encrypted protocol (such as TLS) to protect communication. A shared secret can be established with the remote party by including a Diffie-Hellman parameter in an attestation sent to the remote server. Similarly, the remote server may sign its Diffie-Hellman parameter with a certified private key, whose certificate the TEE application should verify before combining the parameter with its local state to establish a shared secret. In this way, a TEE application can securely send and receive messages. Pearl-TEE implements a trusted path with the user by allowing the TEE application exclusive read and write access to the device touch screen. We note that Pearl-TEE currently does not implement any type of security indicator [8] to signal to the user that it is interacting with a TEE application or which TEE application the user is interacting with. This means that while a malicious normal-world OS or TEE application cannot intercept interactions between a user and a TEE application, they can masquerade as another TEE application by mimicking the target application's user interface.

4 IMPLEMENTATION AND EVALUATION

4.1 Pearl-TEE Prototype

Our implementation of Pearl-TEE is based on OP-TEE 2.0 [6], an open-source TEE that implements the GlobalPlatform TEE Internal API Specification v1.0. It runs on a LeMaker HiKey development board, which has a HiSilicon Kirin 620 SoC with an 8-core ARM

Cortex-A53 CPU at 1.2GHz and 2GB RAM. We use Android AOSP 7.1 for the normal-world OS. Pearl-TEE kernel executes at exception level EL1, while all TEE applications run at EL0, allowing Pearl-TEE to protect itself from malicious TEE applications. The GlobalPlatform API already provides a way for normal-world applications to load and execute TEE applications and invoke API functions in the TEE application. As a result, our Pearl-TEE prototype needs only to implement the following APIs:

`Device_Init()`: This API is called by the normal-world OS during device initialization. Pearl-TEE boots with secure boot and measures the state of the normal-world OS. In our implementation, Pearl-TEE checks the integrity of Android using a feature called device mapper verity (dm-verity) [3] which is similar to the way Samsung TIMA performs periodic kernel measurements [9].

`Attest()`: This API enables a TEE application to request an attestation of the application and to send the attested application hash to the remote server for verification. `Message` is an optional field which can be used by the application to fill in its own application-specific data (e.g., a Diffie-Hellman parameter to establish a secure communication channel). The application server can grant or deny access after verifying the application hash.

`Get_App_Key()`: This provides the TEE application with its K_{AP}, which is computed by Pearl-TEE based on the TEE application binary's hash and the K_{PT}, as described in Section 3.1. The application can then use its K_{AP} to encrypt/sign or decrypt/verify data it has stored in the normal-world OS.

`Secure_Display()`: Used by a TEE application to display messages to the user via secure path.

`Secure_Input()`: Used by a TEE application to receive user touch/input from the secure touch screen or provide confirmation via a secure authentication device (i.e., a fingerprint sensor).

`NKM_Syscall()`: Used by a TEE application to make system calls and transfer messages to the normal-world OS. It uses this API to send and receive network messages and to save encrypted data to the normal-world file system.

To fully implement `NKM_Syscall()`, Pearl-TEE proxies system calls to the normal-world OS in a way similar to Proxos [17]. However, while Proxos does it across a hypervisor, Pearl-TEE proxies system calls between the secure- and normal-worlds of the ARM architecture. The Normal-World Kernel Module (NKM) receives `NKM_Syscall()` requests and sends them up to the Normal-World User Module (NUM), which executes them on behalf of the TEE application. The NUM runs as a dedicated, unprivileged user program so that malicious TEE applications cannot use it to escalate privileges in the normal-world OS. This design simplifies Pearl-TEE as it can proxy most complex functionality to the normal-world OS, while at the same time providing intuitive APIs for sending and receiving network messages and storing persistent data.

The board we use to prototype Pearl-TEE does not have a Secure Element (SE) or a touch screen. We therefore emulate these using software modules. We emulate an SE according to existing documents [7] analyzing SE, and we emulate a touch screen device similar to that in [16], which emulates a touch signal to Pearl-TEE.

4.1.1 TEE Payment Applications. We implement PayPal, Alipay and BrainTree mobile payment services on our prototype Pearl-TEE system. We select these three payment services because they are commonly used, processing a total of over 1 trillion dollars in payments annually [10, 15], and because they do not currently have implementations that use TrustZone. In the mobile payment framework, a payment server works as a remote application server, serving all payment transaction requests from the client application. Since we do not have control over the actual payment servers of these three payment providers, we implement a proxy server in front of their servers that verifies the attestations and then forwards the verified payment request to the actual payment server. Our proxy servers handle Pearl-TEE specific messages and use the SDKs of the respective payment services to translate the Pearl-TEE messages into the current native payment messages of each provider, as shown in Figure 2. If Pearl-TEE is eventually deployed, any of these payment providers could support Pearl-TEE by simply running our proxy server in front of their legacy payment servers.

4.1.2 TEE Secure Chat Application. We implement a prototype TEE secure chat application: TEE-Chat, that uses the Off-The-Record messaging protocol (OTR). The basic framework of the TEE-Chat application is similar to the payment application, while the main differences are: (1) TEE-chat needs a symmetric key for encrypting and decrypting messages to be compatible with OTR [11], (2) TEE-chat needs to interact with the user and ensure confidentiality of its output to the user as well as user's input to the application, and (3) instead of an application server, the end-user's TEE application verifies the identity attestation and then displays the User ID.

We implement TEE-Chat based on top of Xabber for Android [19], which is an open-source XMPP client that supports the OTR protocol. In TEE-chat, we modify the OTR part of the original Xabber application and move all encryption tasks to the TEE application to ensure message confidentiality. In addition, in the secure chat application, users input and read their chat messages via a secure touch screen which is directly controlled by Pearl-TEE. For our prototype TEE-chat we support the *Encrypted* level of security of the OTR protocol, but implementing certification verification in our framework should be straightforward. We instrument the code whenever a cryptographic library is called for encryption or key managing functions are used, and redirect the operation to our TEE application in TrustZone. We do this, for example, in `OTRManager::getLocalKey()` where the chat application gets the key for encrypting a message.

4.2 Performance Evaluation

We evaluate the performance of Pearl-TEE with our TEE applications. In the evaluations, each TEE application type is compared with a corresponding Baseline application. The Baseline applications are executed on the same system, except outside TrustZone and without the support of Pearl-TEE. We also measure the impact of running a single TEE application using TrustZone versus having multiple different TEE application tenants.

After conducting 20 payment transactions on TEE applications and the Baseline application, we summarize average times in Table 3. We note that more than half of the time is spent communicating with the payment providers' servers. The next major component is the

Table 3: *Time cost comparison between TEE applications and their Baseline counterparts (LP% = (Local Response Time / Transaction Time) × 100%)*

Time interval (in ms)	Baseline Applications					TEE Applications				
	PayPal	BrainTree	Alipay	Average	LP%	PayPal	BrainTree	Alipay	Average	LP%
Local Response	43	45	43	43.7	8.4%	121	125	121	122.3	18.5%
Proxy Process	136	141	145	140.7		203	206	225	211.3	
Payment Process	305	310	334	316.3		305	311	328	314.7	
Transaction	495	502	533	510.0		644	658	683	661.7	

Table 4: *Secure payment system performance breakdown*

System Module	**Contributes**
Network cost & server wait time	59.3 %
Normal-world process	13.8 %
Payment server process	12.8 %
Pearl-TEE kernel operations	8.2 %
TEE application process	6.5 %
World-switching time	0.1 %

network delay and processing time of our proxy server, which could be reduced if it were directly integrated into the payment providers' servers. Thus, if taking out the network cost and server wait time, Pearl-TEE would only impose less than 20% overall overhead due to Pearl-TEE kernel scheduling, world-switching between secure- and normal-world, and extra encryption in TrustZone.

We study the performance overhead of the Pearl-TEE system with a measurement of a few representative events in our system. We use the TEE payment application as an example and collect the time consumed by each component by instrumenting instructions in the entry and exit functions in each component. These instructions use/call the ARM Performance Monitor Unit, which guarantees the relative accuracy of the timing data. In Table 4, we can see that most of the system execution time consists of running processes outside of the device, including server processing time, network communication time between the device and the proxy server, as well as between the proxy server and the payment servers.

We list the lines of code (LOC) of each module that we implemented or modified in Table 2. As can be seen, the majority of the functionality is implemented in the normal-world. Compared to the base OP-TEE OS, Pearl-TEE increases the TCB of the TEE OS by under 3%.

5 CONCLUSION AND FUTURE WORK

In this paper, we have proposed Pearl-TEE, a TEE OS design that enables multiple, mutually distrusting TEE applications that conform to our abstract TEE application model to execute in TrustZone. Pearl-TEE enables and assumes that arbitrary TEE applications will be installed and run in TrustZone and ensures that each application enjoys storage security, execution and memory security, and the ability to attest application integrity to remote services and access trusted path hardware. Our implementation of Pearl-TEE adds less than 3% to the TCB of the TEE OS and imposes less than 20% local processing overhead. For the future work, we plan to add more features for application usability like supporting flexible scheduling and convenient software update.

REFERENCES

[1] ChaN. 2018. FatFs - Generic FAT Filesystem Module. http://elm-chan.org/fsw/ff/00index_e.html. Last accessed: 2018-08-13.

[2] Zhu Feng and Iansiti Marco. 2011. Entry into platform-based markets. *Strategic Management Journal* 33, 1 (June 2011), 88–106.

[3] Google. 2017. Implementing dm-verity. https://source.android.com/security/verifiedboot/dm-verity. Last accessed: 2018-05-01.

[4] Google. 2018. Trusty TEE. https://source.android.com/security/trusty/. Last accessed: 2018-05-01.

[5] Uri Kanonov and Avishai Wool. 2016. Secure Containers in Android: The Samsung KNOX Case Study. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. Vienna, Austria.

[6] Linaro. 2018. OP-TEE. https://www.op-tee.org/. Last accessed: 2018-05-01.

[7] Tarjei Mandt, Mathew Solnik, and David Wang. 2016. Demystifying the Secure Enclave Processor. In *Blackhat US 2016*. Las Vegas, Nevada. https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf

[8] Claudio Marforio, Nikolaos Karapanos, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. 2013. Secure Enrollment and Practical Migration for Mobile Trusted Execution Environments. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones*. Berlin, Germany.

[9] Peng Ning. 2014. Samsung KNOX and Enterprise Mobile Security. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones*. Scottsdale, Arizona.

[10] Grace Noto. 2017. AliPay, WeChat Processed 3 Trillion in 2016). Bank Innovation https://bankinnovation.net/2017/04/alipay-wechat-processed-3-trillion-in-2016/. Last accessed: 2018-05-01.

[11] P. Saint-Andre. 2004. End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP). https://tools.ietf.org/html/rfc3923. Last accessed: 2018-05-01.

[12] D. Rosenberg. 2014. Qsee trustzone kernel integer over flow vulnerability. In *Black Hat Conference*. Las Vegas, Nevada.

[13] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Salt Lake City, Utah.

[14] SierraWare. 2018. SierraTEE Trusted Execution Environment. https://www.sierraware.com/open-source-ARM-TrustZone.html. Last accessed: 2018-05-01.

[15] Statista. 2017. PayPal's annual mobile payment volume from 2008 to 2016 (in billion U.S. dollars). https://www.statista.com/statistics/277819/paypals-annual-mobile-payment-volume/. Last accessed: 2017-09-20.

[16] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. Denver, Colorado.

[17] Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, Washington.

[18] The Hacker News. 2017. Over 420 Banking Apps Found On Google Play Store are Targeted By Android Trojan. http://thehackernews.com/2017/04/android-banking-malware.html. Last accessed: 2018-05-01.

[19] Xabber. 2018. Introducing Xabber. https://www.xabber.com/. Last accessed: 2018-05-01.