HCE Mobile Ticketing

Providing Security for Mobile Ticketing Applications Backed by TrustZone
(extended abstract of the MSc dissertation)

Pedro S. Ribeiro
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Nuno Miguel Carvalho Santos

Keywords: Trusted Execution Environments, ARM TrustZone, Security, Android Applications, Mobile Ticketing

Abstract: ARM TrustZone technology has been widely use to enhance the security of mobile devices by allowing for the creation of Trusted Execution Environments (TEE). However, existing TEE solutions tend to struggle with a trade-off between security and functionality: they either expose a larger attack surface to favor dynamic code loading inside the TEE, or depend on the static deployment of trusted services inside the TEE which is more cumbersome and error-prone to maintain. This paper proposes the deployment of a trusted service which aims to serve a broad range of applications by offering secure database storage capability inside the TEE. We present DBStore, a TrustZone-backed database management system for mobile applications. Applications can create and operate DBStore databases inside a TEE that provides confidentiality and integrity protection of databases and respective SQL queries without depending on the integrity of the mobile OS. We present a case study where DBStore is used in order to thwart existing attacks in HCE-based mobile ticketing applications.

1 INTRODUCTION

Trusted Execution Environments (TEE) have played an increasing role within the mobile security landscape. A TEE is an isolated execution environment which can provide confidentiality and integrity protection of code and data. Its appeal comes from the fact that these properties can be enforced against a strong attacker with the ability to compromise the OS (e.g., a rootkit). For this reason, a TEE can be a very important resource for protection of sensitive code and assets. In the mobile world, a TEE tends to be created by special hardware available in ARM processor-based devices: ARM TrustZone. Examples of commercially deployed systems that use TrustZone for protection of data and code inside a TEE include Hardware-backed Android Keystore [Android, 2018] and Samsung Knox [Samsung, 2013].

Both academia and industry have pushed the development of TEE solutions that can be both functionally rich and secure. Achieving both goals, however, is challenging. In one line of work, the idea is to develop specific security services that reside inside the TEE [Liu and Cox, 2014, Sun et al., 2015b]. Hardware-backed Android Keystore is such an example for storing cryptographic keys. Since the TEE runtime system can be customized to offer a single service, the resulting trusted computing base can be very small, which favors security. Furthermore, since it provides a narrow API to OS and mobile apps, the attack surface is also reduced. A downside of this approach is that it requires firmware changes for every newly deployed service, which discourages their wider adoption by device manufacturers.

At the other end, we find TEE runtime systems which aim to allow application code to be loaded and executed inside the TEE itself. Examples of such systems include Samsung Knox, the Trusted Language Runtime (TLR) [Santos et al., 2014], and systems that implement GlobalPlatform's TEE specifications [opt,]. Because these systems allow dynamic code loading at runtime, it means that new security services can be deployed without the need to change the device firmware. However, although the application code is deployed inside independent TEE-hosted sandboxes, the system's attack surface increases. Furthermore, in some cases, such as Samsung Knox's, the trusted computing base is very large. Both these factors may weaken the overall security of the system.

Our goal is to devise an intermediate solution that allows mobile applications to take advantage of the benefits of TEEs without (i) allowing for dynamic

loading and execution of third-party application code inside the TEE, and (ii) requiring frequent firmware upgrades to mobile devices. To this end, we propose a single trusted service aimed to provide general-purpose database hosting inside the TEE. Such a service would provide a narrow interface through which mobile applications can preserve their databases securely and issue database queries to be executed inside the TEE while preserving the confidentiality and integrity of their execution state. With our solution, although mobile applications cannot run arbitrary code inside the TEE, they enjoy a great degree of flexibility in managing their data securely through the possibility to issue sophisticated database queries.

This paper develops this idea and presents the design and implementation of *DBStore*, a TrustZone-backed database management system for mobile applications. DBStore provides an SQL interface to application-custom databases. A simple-to-use API masks the security protocols designed to defend against rollback and replay attacks, and provide confidentiality and integrity protection of both databases and SQL commands, including inputs and outputs. In our current prototype, we adopted SQLite as our SQL engine running inside the TEE. By providing an SQL interface, we aim to offer a programming model familiar to application developers. DBStore was carefully designed in order to provide secure remote access to the databases. By leveraging TrustZone for isolation purposes, DBStore can protect applications' databases in the presence of a powerful attacker which can control the local operating system.

To demonstrate our system, we present a case study in which we apply DBStore in secure mobile ticketing applications. We present the design of an existing HCE-based mobile ticketing app based on Android keystore, and analyze its vulnerabilities. We then show how DBStore allows to mitigate such attacks with simple DBStore SQL transactions. Although we have focused on a single use case, we envision DBStore to be used for a wide range of applications, e.g., e-health, mobile payments, etc.

2 BACKGROUND

First, we present some necessary background information for both our threat model and system design.

2.1 ARM TrustZone Technology

ARM TrustZone is a security extension for ARM processors that provides a hardware-level isolation between two execution domains: *normal world* (NW) and *secure world* (SW). Compared to the NW, the SW has higher privileges, as it can access the memory, CPU registers and peripherals of the normal world, but not the other way around. Isolation is enforced by checking and controlling the state of the CPU through a NS bit, and partitioning the memory address space into secure and non-secure regions.

Interaction between worlds requires a world switch operation triggered by the SMC instruction. TrustZone devices equipped with a secure boot mechanism provide integrity and authenticity protection of the SW software. The binary of the SW software must be part of the device firmware and signed by the device manufacturer. When the device is powered, the processor enters SW, jumps to the initialization procedure of the SW software, which then switches to NW and hands over control to the OS bootloader.

2.2 TrustZone-based TEE Systems

TrustZone is used in millions of ARM-based mobile devices for the creation of Trusted Execution Environments (TEE). A TEE consists of an isolation environment in which trusted applications can execute without the interference of the local (untrusted) operating system. When based upon TrustZone technology, the TEE typically resides in the secure world and the rich operating system lives in parallel in the normal world. This strict separation between both environments ensures that a potentially compromised OS cannot access the TEE's internal state.

We distinguish two types of TEE architectures: *TEE kernel* [opt, , Fitzek et al., 2015] or *TEE service* [Sun et al., 2015a]. In the first case, the TEE provides an interface to the normal world that allows for trusted applications to be installed and executed inside the TEE. On the other hand, TEE services expose only a function call interface to the normal world and implement a specific application. They do not allow for dynamic loading of third-party trusted applications inside the TEE, which reduce exposure of potential vulnerabilities to an attacker when compared to the TEE kernel architecture.

2.3 A Case for TrustZone-backed DBs

In spite of the security benefits of the TEE service architecture, an important downside lies in its inflexibility: in order to deploy a new trusted application inside the TEE, end-users' devices need to be upgraded with new firmware containing the application binary and be rebooted so that the new application can be properly instantiated inside the secure world upon boot. However, firmware upgrades tend to be error-prone

and quite cumbersome for the end-users. To alleviate this problem, the question we raise is whether there exists a class of applications that could benefit from a common general trusted service so that the applications could be executed inside the normal world by relying only on this single trusted service for protecting this sensitive state.

To answer this question, we propose a new trusted service – DBStore – aimed at maintaining and securing application databases inside a TrustZone-backed TEE. With DBStore, we decouple the persistent state of the application from the logic of the application, and provide a way to secure its persistent state only. As a result, the logic of the application can reside in the normal world thereby precluding the need to upgrade the devices' firmware. On the other hand, since databases are widely used, many different applications can benefit from DBStore. More specifically, our goal is to serve a particular class of applications in which a remote client may need to access and manipulate application data stored locally while requiring high level protection, such as secure mobile ticketing, mobile payment, and e-health applications.

2.4 Security Goals and Threat Model

In the scenarios we envision, distributed applications consist of two parts: a *client side* which is kept under the control of an application provider, and a counterpart *mobile side* which runs on users' devices. Depending on the specific application, the remote client can be implemented, e.g., by a remote cloud server, a nearby embedded device, etc. The mobile application can solicit storage space at the DBStore for storing dabatases and should interact with its databases using standard SQL statements.

DBStore must give application providers strong assurances about the integrity and confidentiality of the guest databases. It must prevent an adversary from reading or altering not only the contents of database records, but also the inputs / outputs of SQL transactions issued by the clients. It must defend against replay attacks allowing for the re-execution of past legitimate SQL, and against rollback attacks, which attempt to revert the database state to an old version.

The adversary can intercept and modify messages exchanged between client and DBStore, either while in transit over the network, or by controlling the mobile application or the mobile OS. Such control can be gained, e.g., through malware exploits, or device routing. We assume, however, that the adversary cannot perform physical attacks to the TrustZone-enabled mobile platform. Hardware and cryptographic primitives are considered to be correct and DBStore components residing in the SW belong to the trusted computing base. Side-channel attacks are not in scope.

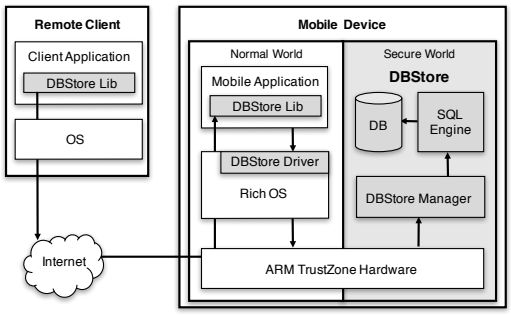

Figure 1: DBStore architecture: arrows represent invocation flow of an SRPC.

3 DESIGN

Figure 1 presents a high-level architecture of DBStore. The SW is used to implement a TEE where the DBStore resides; OS and mobile applications run in the NW. Thus, mobile applications and OS have no direct access to the DBStore state. The client application interacts with the mobile application by submitting SQL commands to the remote databases hosted inside the DBStore. The databases are securely maintained by the DBStore, which has exclusive access to the databases' state and is responsible for the execution of the SQL statements submitted by clients. The DBStore ensures that the SQL commands issued by the clients are securely forwarded to and executed on the intended database and the results routed back to the respective client.

SQL commands are issued to a database residing on a remote DBStore by invoking *SQL Remote Procedure Calls* (SRPC). To this end, an API is provided by the DBStore library. An SRPC comprises one or more SQL statements, such as "create table", "insert into", and "select" commands, and are performed as a single transaction. An SRPC can be parameterized by inputs provided by the client application, and can optionally return an output to the client application. The DBStore library and DBStore manager implement security protocols which involve the exchange of information in the form of specific Protocol Data Units (PDUs). An SRPC is translated into PDUs that are sent over the network to the mobile application. The mobile application must then issue a request to a DBStore driver in order to forward the PDUs to DBStore. This driver is required because the context switch to SW needs kernel privileges. DBStore, then, handles the PDUs, assigns the SRPC commands to the inter-

nal SQL engine, and returns the results all the way back to the client. In our system, the SMC instruction is invoked by the DBStore driver to enter the secure world and by the DBStore to exit and return to the OS. Next, we provide additional details of our system.

3.1 Initialization Protocol

Before deploying databases on a remote DBStore, a client must first verify that the remote peer runs a legitimate DBStore inside a TrustZone-enabled device. Conversely, the DBStore must be able to authenticate the client and implement an authorization policy that prevents other applications from accessing the remote client's databases. To obtain these guarantees, the client and the remote DBStore must first perform an initialization protocol by exchanging two PDUs: M_1 and M_2 (see Table 1).

To prove that the DBStore service is authentic and resides inside a valid TrustZone-enabled environment, the first message (M_1) contains a challenge, a nonce n_1, that the DBStore must sign with a valid *attestation key* (TK). The attestation key is a keypair that the device manufacturer must embed into the device firmware. The private part TK^- is accessible within the SW (if the secure boot passes). If DBStore is able to sign n_1 with such a key, then it means that the DBStore software has not been tampered with and runs inside the SW. The DBStore must then sign n_1 with TK^- and send the signature along with a certificate (C_{TK}) that contains the public key TK^+ required to check the signature. This certificate is issued by the device manufacturer in order to authenticate the attestation key. Thus, by checking the signature against a legitimately certified attestation key, the client can verify the authenticity of the remote DBStore.

To authenticate the remote client and restrict access to the local databases by alien applications, M_1 contains nonce n_1, an application ID and both are signed with the private part AK^- of *application keypair* (AK). This keypair must be generated by the application provider and the private part must be kept secret by the client. The public part AK^+ is included in the mobile application package. Upon receiving M_1, DBStore can validate the signature by obtaining AK^+ from the mobile application thus authenticating the client. To ensure that future interactions remain mutually authenticated, a *session key* (SK) is exchanged as a token of successful client authentication and service remote attestation. This is reflected in message M_2 which includes a session key SK created by the DBStore and encrypted with AK^+. This result is also included in the signature by TK^-.

This way, the DBStore can ensure the authenticity of the client endpoint, because only the legitimate client application holds the private key AK^- which will enable the session key to be decrypted. On the other hand, by checking the signature of M_2, the client can be assured about the authenticity of that session key (i.e., that it was issued within a legitimate DBStore-enabled device) and of the freshness of the session key (because the tuple includes nonce n_1 which has been chosen by the client itself). The session key is then crucial in securing SRPCs.

Initialization Protocol	
(M_1)	C $\rightarrow$ S: $\langle AppId, n_1 \rangle_{AK^-}$
(M_2)	S $\rightarrow$ C: $\langle n_1, \{SK\}_{AK^+} \rangle_{TK^-}, C_{TK}$
SRPC Invocation Protocol	
(M_3)	C $\rightarrow$ S: n_2, $\{SrpcReq\}_{K_i}$, hmac(K_i)
(M_2)	S $\rightarrow$ C: ($M_{4.a}$) S $\rightarrow$ C: n_2, Ok, $\{SrpcRes\}_{K_i}$, hmac(K_i)
($M_{4.b}$)	S $\rightarrow$ C: n_2, $Fail$
Resync Protocol	
(M_5)	C $\rightarrow$ S: n_3, $Resync$, hmac(SK)
(M_6)	S $\rightarrow$ C: n_3, $\{i, salt\}_{SK}$, hmac(SK)

Table 1: DBStore protocols: C: remote client, S: DBStore.

3.2 SQL Remote Procedure Calls

After the initialization, the client can invoke SQL Remote Procedure Calls (SRPC) on DBStore. Each SRPC generates a message to the DBStore (M_3), which will then return one of two possible messages: $M_{4.a}$ if the request message was well formed, or $M_{4.b}$ otherwise (see Table 1). The SRPC request message M_3 includes field *SrpcReq* which contains the SQL commands and inputs to be executed on the remote DBStore. The *SrpcRes* field is included in message $M_{4.a}$ and contains the execution output to be sent to the client. For confidentiality protection, *SrpcReq* and *SrpcRes* are encrypted with a symmetric key shared between the client and the DBStore. For message integrity protection, the same key is used to generate HMAC codes for the whole message. Nonce n_2 helps the client detect replay attacks of old M_4 messages.

Although the symmetric key shared between both endpoints is the session key SN, this key is not used in messages M_3 and $M_{4.a}$. Instead, the client creates a new symmetric key K_i for each invoked SRPC. K_i is generated by calculating a hash of SK concatenated with a counter value i, which is initialized to zero and incremented for every SRPC request issued by the client. To determine K_i, the DBStore must also keep a local counter for the i value. To calculate the key of an incoming SRPC message, all the DBStore has to do is to append i to SK, and compute the hash of the result, yielding K_i, which is then used to de-

crypt SRPC #i and encrypt the corresponding SRPC response. After submitting the response to the client, the DBStore increments its local i counter.

This mechanism serves two purposes. First, it prevents replay attacks of old SRPC requests. Since the DBStore uses a single key K_i for each request, then old messages (i.e., corresponding to request numbers less than i) cannot be decrypted. Furthermore, the i counter of the DBStore is implemented using a TrustZone monotonic counter, thereby preventing an adversary from decrementing i. This mechanism also avoids overuse of the session key SN. By generating a new symmetric key K_i, we reduce the exposure of SN thereby enhancing the overall system security.

If for some reason the client gets out of sync with the DBStore about the current counter value, and the legitimate client sends a request encrypted with K_j different than K_i expected by the DBStore (i.e., $j \neq i$), the DBStore will not be able to properly decrypt the SRPC request field of message M_3 and then returns a failure message ($M_{4.b}$). To overcome this issue, the client triggers a resync protocol by which the DBStore will return the current i value encrypted with the session key (see M_5 and M_6 in Table 1).

3.3 Protection of DBStore databases

To secure the DBStore databases at rest, we leverage the secure storage allocated to the SW. We use additional TrustZone mechanisms to prevent rollback attacks and protect data confidentiality. In particular, we use secure monotonic counters to keep track of the most updated version of DBStore databases. Every time a database is updated, its version number is incremented and written into the database, and the secure monotonic counter is also incremented. If the device is rebooted and the database is replaced with an older version, the version number of the recovered database and the current value of the counter will be inconsistent, forcing DBStore to raise an exception. To preserve confidentiality, the DBStore databases (and associated version numbers) are encrypted with a symmetric key which is then sealed, i.e., encrypted with a key which is released only if the secure boot is successful. This prevents an adversary from retrieving the encryption key and decrypt the database.

3.4 Prototype Implementation

We implemented a DBStore prototype for the Boundary Devices i.MX6 Nitrogen6x. The DBStore runs inside the SW and consists of several components. First, it comprises a small kernel responsible for memory management, thread execution, and world switch operations. To this end, we adopted the TEE implementation from Linaro's OP-TEE [opt,], which implementes Global Platform's TEE Internal Core API v1.1. The logic responsible for DBStore management is implemented in C. It is also responsible for the management of cryptographic keys, and provide cryptographic functions. The available cryptographic library provided by the TEE Internal Core API is libtomcrypt [lib,].

As for the SQL engine implementation, we ported LittleD. Given that OP-TEE's TCB is quite small, it misses several libc/POSIX functions. As such, we would have to port a library and make it compilable for the SW. Considering that LittleD was built for arduinos and, as such, uses few libc/POSIX functions, we decided to use it as the SQL engine. For database persistence, as well as cryptographic material, we leverage the Trusted Storage mechanism provided by OP-TEE. This mechanism assures that only the TA that owns the files can access them and it protects the files against rollback attacks by storing hashes on the Replay Protected Memory Block (RPMB) of the eMMC partition.

For the NW, we used the Linux kernel, modified with OP-TEE's driver. We also had to use OP-TEE's implementation of the GlobalPlatform TEE Client API Specification v1.0, which allows an application to communicate with a trusted service running on the TEE via the TEE driver. Furthermore, we used Buildroot [bui,] to generate a filesystem.

4 CASE STUDY

In this section, we study a proprietary HCE-based mobile ticketing application for public transports with the purpose of illustrating how DBStore can thwart some of its fundamental security limitations. HCE, which stands for Host Card Emulation, was introduced in the Android OS and it allows mobile applications to maintain software emulated contactless cards and communicate with NFC readers.

4.1 HCE-based Mobile Ticketing

HCE-based mobile ticketing is penetrating the realm of public transports, such as trains and buses. In a typical workflow, a passenger installs the application, signs up with a username and password, logs in, and purchases a virtual card with credits for an amount of trips. This information is then downloaded and maintained inside an HCE-emulated card by the mobile app. To perform a trip, the passenger must validate the card by swiping the mobile device in front of the

NFC-enabled reader. The reader communicates via NFC with the app, which checks if enough credits exist in the virtual card. If so, it decrements the credits available and tells the reader to authorize the trip.

From a security point of the view, the main concern is to prevent fraudulent passengers from traveling without credit. Thus, it is necessary to prevent forging new virtual cards, stealing virtual cards from other passengers, incrementing credits of legitimately purchased virtual cards, and double spending credits of valid virtual cards. Since the passengers' mobile device may be disconnected from the network, ticket validation must be able to operate offline securely, by relying only on NFC communication with nearby readers and on the mobile devices' local resources.

Card Purchase: When a ticket is purchased, the server generates a secret serial number for the new virtual card. This secret is cryptographically signed and sent to the mobile application along with the number of associated trip credits over an authenticated HTTP connection. The signature is checked against the public key of the ticketing server, and the virtual card's data (serial number and credits) are written on a private file owned by the application. This file is encrypted with a key K protected in Android keystore.

Card Validations: By means of a handshake between card reader and mobile application, the card reader is authenticated and a session key is exchanged. The card reader sends a validation request message to the mobile application via the Android's HCE service. The app decrypts the virtual card file with key K (released by the Android keystore), and checks that enough credits exist. If so, it sends to the reader the serial number of the virtual card encrypted with the session key and updates the local file with a decremented credit. For every update, the file is encrypted with a new key K that replaces the previous one.

Card Recharge: The mobile application connects to the ticketing server over HTTPS. When the payment is ended, the mobile application learns the amount of credits just acquired and updates the local file securely like in a card validation.

4.2 Existing Security Vulnerabilities

Despite the security mechanisms described above, several attacks can be performed by an adversary with the ability to control the operating system of a passenger's mobile device. This can be achieved by accidental rootkit infection or by intentional device rooting. Next, we describe some critical attacks and their impact to the overall application security. We consider hardware-backed Android keystore implementations.

Attack 1 – Rollback keystore files and card data: The encrypted key K is encrypted with the public part of a keypair and persisted inside an Android keystore file located in /data/misc/keystore/user_0. The keypair files are encrypted with a master key that is maintained inside the TEE and that cannot be retrieved by typical means. Both encrypted items, ticket data and key K are stored on the application data folder. Thus, an adversary with root privileges can backup all these files, and replace them with older copies. This attack enables an attacker to travel using a single card for as long as its serial number is not blocked.

Attack 2 – Steal keypair with rogue app: This attack leverages the fact that Android keystore preserves the keypairs of a given application in two files: "(App's User ID)_USRPKEY_(Key Entry Alias)" and "(App's User ID)_USRCERT_(Key Entry Alias)". The owner of the keypair is identified by the (App's User ID) bit. To launch the attack, the adversary starts by creating a KeyStealer application aimed to retrieve the keypair of the mobile ticket app from the keystore file and write it elsewhere unencrypted; with that keypair, K can be obtained and hence the raw card data. Then, it is only necessary to change the User ID of files generated for the HCE mobile ticketing app to match KeyStealer's. Android keystore will be tricked into believing that the KeyStealer app owns that keypair and will decrypt it using the master key.

Attack 3 – Reverse engineer the mobile app: By using reverse engineering tools, an attacker can modify the mobile app so as to prevent credit decrementation at each validation. Since the card reader devices have currently no means to authenticate the mobile endpoint, they would not be able to distinguish between a correct and a modified application.

4.3 Overcoming Existing Limitations

The attacks described above are possible because the security depends on the integrity of the OS. This section describes how to thwart these attacks using DBStore. In our solution, rather than preserving the virtual card data inside an encrypted file (and the respective encryption key secured in Android keystore), this data will be secured inside a DBStore database. The DBStore clients will be two: the ticketing server and the validation reader devices. An application keypair (see Section 3.1) must be generated: the public part is included in the mobile app package and the private part is secretly maintained by both ticket server and reader devices. Then, the revised mobile ticketing app works as follows.

First, upon user signing up, the ticketing server

```
-- SQL code to intialize a mobile ticketing DB
DROP TABLE IF EXISTS Tickets;
CREATE TABLE Tickets(SN INT, Type TEXT, Credits
     INT);

-- SQL instructions for card purchasing
INSERT INTO Tickets VALUES(@sn, @cardtype,
     @credits);

-- SQL instructions for card validation
UPDATE Tickets
    SET Credits = CASE
        WHEN Credits > 0 THEN (Credits - 1) ELSE
            -1 END
    WHERE Type = @cardtype;
SELECT SN, Credits FROM Tickets WHERE Type =
     @cardtype;

--SQL instructions for card recharging
UPDATE Tickets SET Credits = (Credits + @amount)
    WHERE Type = @cardtype;
```

Listing 1: SQL instructions for mobile ticketing app.

runs the initialization protocol with the mobile endpoint. Next, the ticketing server issues a first SRPC aimed at initializing the virtual card database inside the DBStore (see Listing 1). The DB contains a table of card records. Each record has a card type, a serial number, and current credits.

For card purchase and card recharge, the ticketing server only needs to issue SRPC calls for inserting and updating the DB, respectively. This is shown in Listing 1. For the validation operation, the reader issues a transaction that decrements the current credit of the virtual card if the current credit value is positive. Otherwise it sets its value to -1, indicating that credit has been exhausted. The second SQL command reads the current credit value along with the card's serial number so that the client—i.e., the reader—can determine if passage can be granted or not: -1 means it is not. Since DBStore protects against rollback attacks, Attack 1 is no longer possible. Attack 2 fails because DBStore keeps all relevant key material secured inside the TrustZone-enabled TEE. Lastly, Attack 3 is not possible because, through the initialization protocol, all SRPCs run inside the DBStore, hence outside the adversary's reach.

4.4 Performance Evaluation

To study the performance overhead of DBStore in the HCE-based mobile ticketing application, we measure the execution time of three main card operations tested on our DBStore prototype. We performed comparative measurements both in NW and SW. The reported numbers exclude network communication costs. our evaluation testbed consisted of the QEMU emulator running on a PC that features an SSD and

Initialization Protocol	
(M_1)	C $\rightarrow$ S: $\langle AppId, n_1 \rangle_{TK^+}$, $\text{Sign}_{AK^-}\langle AppId, n_1 \rangle$, AK^+
(M_2)	S $\rightarrow$ C: $\langle n_1 + 1 \rangle_{SK}$, $\langle SK \rangle_{AK^+}$
SRPC Invocation Protocol	
(M_3)	C $\rightarrow$ S: $\langle n_2, SrpcReq \rangle_{K_i}$, hmac($SrpcReq$, K_i)
($M_{4.a}$)	S $\rightarrow$ C: $\langle n_2 + 1, SrpcRes/OK \rangle_{K_i}$, hmac($SrpcRes/OK$, K_i)
($M_{4.b}$)	S $\rightarrow$ C: $\langle n_2 + 1, FAIL \rangle_{K_i}$, hmac($FAIL$, K_i)

Table 2: DBStore OP-TEE prototype protocols: C: remote client, S: DBStore.

```
-- SQL instructions for card purchasing
DROP TABLE IF EXISTS Tickets;
CREATE TABLE Card(SN INT, Type TEXT, Credits
     INT, Counter N);
INSERT INTO Card VALUES(@sn, @cardtype,
    @credits, @counter);

--SQL instructions for card recharging
SELECT * FROM Card WHERE Counter = @counterf
    ;
DELETE Card;
CREATE TABLE Card(SN INT, Type TEXT, Credits
     INT, Counter N);
INSERT INTO Card VALUES(@sn, @cardtype,
    @newcredits, @counter + 1);

-- SQL instructions for card validation
SELECT * FROM Card WHERE Counter = @counterf
    ;
INSERT INTO Card VALUES(@sn, @cardtype,
    @credits - 1, @counter + 1);
```

Listing 2: SQL instructions for mobile ticketing app.

a 2.8GHz Intel Processor and the i.MX6 Nitrogen6x, which features a 1 GHz ARM Cortex-A8 Processor and 1 GB of DDR3 RAM memory. The board executed our system from a mini SD card, which was flashed with the modified OP-TEE and Linux versions. For each experiment, we do 50 runs and provide the mean and standard deviation, which is on top of each bar. We had to perform some alterations to the protocols proposed in Table 1 and to the SRPCs proposed in Listing 1. The new versions are shown in Table 2 and in Listing 2

Figure 2 presents the execution time in both NW and SW of the tested card operations: Purchase, Recharge, and Validation. The execution time is broken up into three phases: "Operation" refers to the core logic of all three cases, which is common in NW and SW; "World-Switch" refers to the logic associated with world context switch and exists in the SW alone; "I/O" refers to the time to write and read from the SQL database.

Looking at both graphics, we can easily conclude that Recharge is the slowest operation, taking 0.56 seconds on QEMU and 129.79 seconds on the Nitro-

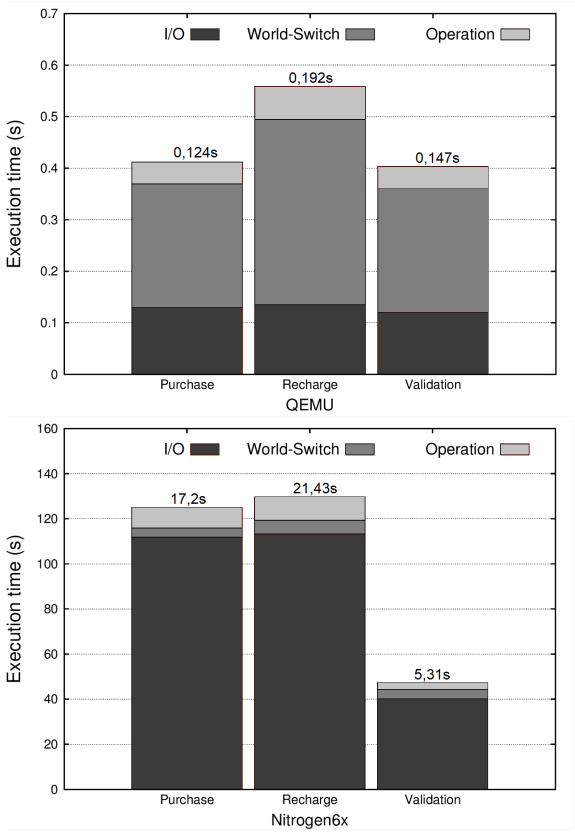

Figure 2: Execution times of the three main ticketing operations, running on QEMU (top) and the Nitrogen6x (bottom).

gen6x. This makes sense, as Recharge is comprised of four steps: first, selecting the most recent row in the 'Tickets' table; then, delete the table, followed by the creating on a new one; it ends by inserting a new row by incrementing both the credits and counter of the selected row. This can be tolerated, as Recharge is an operation the user can perform online at any given moment. The second slowest operation is Purchase, as it consists on the creation of the table 'Tickets' and the insertion of the initial state. It takes 0.41 seconds to execute on QEMU and 125.12 seconds on the Nitrogen6x. Both these operations are I/O intensive, making them a bit slower than Validation, which takes 0.40 seconds in the QEMU evaluation, and a lot slower in the Nitrogen6x evaluation, which takes 47.40 seconds. Validation is then the fastest operation, as it only requires selecting the latest row and decrementing the number of avaialable credits. This is ideal, as Validation is performed at the ticketing terminals and, in a pessimist scenario, in a case where the user has a long line behind of him, meaning that the operation must be fast in order not to anger the other passengers.

5 RELATED WORK

There is significant work on the protection of user data on mobile devices. Aquifer [Nadkarni and Enck, 2013] employs application workflow control to prevent accidental information disclosures. TaintDroid [Enck et al., 2010] leverage information flow control to trace user data movements within the operating system. MOSES [Russello et al., 2012] enforces contextual security profiles, e.g. Work, Private, etc. Another group of systems that protect data within an OS aim to enforce Digital Rights Management (DRM) [Ongtang et al., 2010]. While these systems provide effective mechanisms to protect data in a mobile device, their main downfall is their dependence on the integrity of the OS.

Some existing work has focused on the security of the Android keystore. Sabt et al. [Sabt and Traoré, 2016] formally prove the failure of the keystore's encryption scheme in providing integrity guarantees, and exploit this flaw to define a forgery attack to breach its security. Cooijmans et al. [Cooijmans et al., 2014] assess the keystore's security, particularly in mobile devices where keystores are protected by ARM TrustZone. Given the importance of Android keystore for mobile security, we propose an analogous TrustZone-backed service for the protection of sensitive user-level databases.

The research on ARM TrustZone for mobile security has been prolific. Some systems provide provide specific secure services, such as secure authentication [Liu and Cox, 2014], one-time-password [Sun et al., 2015b], and trusted I/O channels [Li et al., 2014]. Others allow for the execution of application code instantiated inside SW-hosted sandboxes [Santos et al., 2014]. In contrast, DBStore does not allow for the execution or arbitrary code inside the SW, therefore reducing a potential attack surface, and yet provides a rich SQL interface that allows for secure database hosting.

The idea of using a TrustZone-based TEE for protection of ticket data has appeared in prior work. In one case, this idea was sketched at a very high-level in a position paper [Hussin et al., 2005] without any actual implementation. In other cases, a software component of the mobile ticketing application must run inside a TEE sandbox, which is either supported by some small trusted kernel [Tamrakar and Ekberg, 2013] or by a TEE-emulated HCE environment [Merlo et al., 2016]. We demonstrate an alternative and yet more general way for securing mobile ticketing based on TEE-backed SQL commands.

6 CONCLUSIONS

This paper presented DBStore, a system that allows mobile applications to create and operate databases inside a TrustZone-backed TEE. It preserves the confidentiality and integrity of the data against a powerful adversary that can control the OS. To foster portability, applications interact with DBStore through a standard SQL interface. We show that DBStore can be easily adopted in order to secure an HCE-based mobile ticketing application for public transports.

REFERENCES

LibTomCrypt. https://github.com/libtom/libtomcrypt.

OP-TEE. https://www.op-tee.org.

The Buildroot user manual. https://buildroot.org/downloads/manual/manual.html#_getting_started.

Android (2018). Android Key Store. https://developer.android.com/training/articles/keystore.html.

Cooijmans, T., de Ruiter, J., and Poll, E. (2014). Analysis of Secure Key Storage Solutions on Android. In *Proc. of SPSM*.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2010). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of OSDI*.

Fitzek, A., Achleitner, F., Winter, J., and Hein, D. (2015). The ANDIX Research OS – ARM TrustZone Meets Industrial Control Systems Security. In *Proc. of INDIN*.

Hussin, W. H. W., Coulton, P., and Edwards, R. (2005). Mobile ticketing system employing trustzone technology. In *Proc. of ICMB*.

Li, W., Ma, M., Han, J., Xia, Y., Zang, B., Chu, C.-K., and Li, T. (2014). Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *Proc. of APSys*.

Liu, D. and Cox, L. P. (2014). VeriUI: Attested Login for Mobile Devices. In *Proc. of HotMobile*.

Merlo, A., Lorrai, L., and Verderame, L. (2016). Efficient Trusted Host-based Card Emulation on TEE-enabled Android Devices. In *Proc. of HPCS*.

Nadkarni, A. and Enck, W. (2013). Preventing accidental data disclosure in modern operating systems. In *Proc. of SIGSAC*.

Ongtang, M., Butler, K., and Mcdaniel, P. (2010). Porscha: Policy Oriented Secure Content Handling in Android. In *Proc. of ACSAC*.

Russello, G., Conti, M., Crispo, B., and Fernandes, E. (2012). Moses: Supporting operation modes on smartphones. In *Proc. of SACMAT*.

Sabt, M. and Traoré, J. (2016). Breaking Into the KeyStore: A Practical Forgery Attack Against Android KeyStore. In *Proc. of ESORICS*.

Samsung (2013). White Paper: An Overview of Samsung KNOX.

Santos, N., Raj, H., Saroiu, S., and Wolman, A. (2014). Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proc. of ASPLOS*.

Sun, H., Sun, K., Wang, Y., and Jing, J. (2015a). Reliable and Trustworthy Memory Acquisition on Smartphones. *Transactions on Information Forensics and Security*, 10(12):2547–2561.

Sun, H., Sun, K., Wang, Y., and Jing, J. (2015b). TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proc. of CCS*.

Tamrakar, S. and Ekberg, J.-E. (2013). Tapping and Tripping with NFC. In *Proc. of TRUST*.