Open-TEE is No Longer Virtual: Towards Software-only Trusted Execution Environments Using White-box Cryptography

Kemal Bicakci
TOBB University of Economics and Technology
Ankara, Turkey
bicakci@etu.edu.tr

Ihsan Kagan Ak
TOBB University of Economics and Technology
Ankara, Turkey
ihsankaganak@etu.edu.tr

Betul Askin Ozdemir
Middle East Technical University
Ankara, Turkey
askinbetul@gmail.com

Mesut Gozutok
HAVELSAN Inc.
Ankara, Turkey
mgozutok@havelsan.com.tr

***Abstract*—Trusted Execution Environments (TEEs) provide hardware support to isolate the execution of sensitive operations on mobile phones for improved security. However, they are not always available to use for application developers. To provide a consistent user experience to those who have and do not have a TEE-enabled device, we could get help from Open-TEE, an open-source GlobalPlatform (GP)-compliant software TEE emulator. However, Open-TEE does not offer any of the security properties hardware TEEs have. In this paper, we propose WhiteBox-TEE which integrates white-box cryptography with Open-TEE to provide better security while still remaining complaint with GP TEE specifications. We discuss the architecture, provisioning mechanism, implementation highlights, security properties and performance issues of WhiteBox-TEE and propose possible revisions to TEE specifications to have better use of white-box cryptography in software-only TEEs.**

***Index Terms*—Trusted execution environment, White-box cryptography, Open-TEE, Secure storage, SPACE algorithm.**

I. Introduction

Trusted Execution Environments (TEE) bring hardware support against threats to mobile devices by providing an execution environment isolated from the main operation. They are programmable thus have the potential to protect sensitive mobile applications and services. However, two major barriers prevent unleashing their full potential:

- Device manufacturers limit their use for their purposes. Therefore, most of the time it is not available to application developers although it is physically present.
- Even when they are available to use, it is not straightforward to develop such an application (cited difficulties include expensive and/or proprietary development tools, expensive tools or primitive techniques for debugging, etc. [1]).

This work is supported by TUBITAK (The Scientific and Technological Research Council of Turkey), Grant no: 5170076.

To overcome the second barrier, McGillion et al. presented Open-TEE, a virtual TEE implementation (software based emulator) conforming GlobalPlatform (GP) specification [1]. Using Open-TEE, the developers could easily develop GP-complaint TEE applications so-called trusted applications (TA) using only the tools they already have and are familiar with. Once the TA is ready, since it is GP-complaint, it can easily be deployed to TEE-enabled devices conforming GP standards. Through a user-study, the authors also showed that the perceived usability of Open-TEE is high among TA developers.

On the other hand, the first barrier remains. In fact, this barrier is so real that after discussions with service providers, Open-TEE developers found out that Open-TEE could be used not only for development but also put in use as a fallback mechanism after deployment i.e., the application will choose to use Open-TEE if useful TEE hardware is not detected. By this way, the service providers could present a consistent user experience both for TEE-enabled devices as well as for devices without an accessible TEE. However, as noted by the authors, Open-TEE is just an emulator, it executes on the main operating environment and does not offer any security property. The improvement of the aforementioned operation mode with respect to security remains an open problem.

In this paper, we present a solution called **WhiteBox-TEE** to fulfill this need for security. Our solution relies on white-box cryptography which aims at providing security in a white-box setting as opposed to against a black-box threat model i.e., the attacker could have a reach to the environment the encryption is performed. We describe our approach to integrate white-box cryptography with Open-TEE to protect cryptographic keys in rest (but not during usage) while still remaining complaint with the GP specification. We also discuss possible extensions to this specification to make it possible to use white-box cryptography against more powerful adversaries. We argue that WhiteBox-TEE could be a promising solution for those who would like to deploy applications simultaneously to mobile devices which have and which do not have accessible hardware TEE.

The rest of our paper is organized as follows. Section II provides background information. Section III presents our solution; WhiteBox-TEE. Section IV presents an overview for a possible revision to GP specifications enabling the use of white-box cryptography for better protection. Section V discusses our proposals further with promising directions for future work. Finally, Section VI concludes.

II. Background

In the following subsections, we provide short background information on Trusted Execution Environments (TEEs), TEE Provisioning, Global Platform Specifications related to TEEs, Open-TEE, and White-box Cryptography, respectively.

A. Trusted Execution Environments

Complexity and size are the worst enemies of security. Mobile operating systems such as Android are so large and complex that it is difficult if not impossible to secure them. A more plausible solution is having a secondary execution environment besides the more traditional one isolated using specialized hardware features. This solution called TEE has been available for more than 10 years in mobile devices [2].

TEE has a smaller Trusted Computing Base thus a limited set of functionality. But the functionality it provides is sufficient to serve Trusted Applications (TAs) which are applications running on top of TEE and provides security-critical operations. On top of a Rich Execution Environment (REE) such as Android, there is a client application (CA) which is similar to other Android applications but can invoke TAs whenever a security-related operation is needed e.g., for encrypting a message.

B. TEE Provisioning

The ecosystem of a TEE and lifecycle of TAs needs to be carefully managed to satisfy the security requirements. One of the requirements is that only authorized service providers should install, update and remove a TA. Otherwise, an attacker installing a malicious TA could easily compromise the security of the system. A recent Internet Draft presents a Trusted Execution Environment Provisioning (TEEP) architecture for this purpose [3]. The exact details of this architecture are not relevant thus not discussed here but two of the main components in the system are discussed in short because we will also propose to use them for provisioning of WhiteBox-TEE:

- TAM (Trusted Application Manager): It is usually implemented as a remote entity to perform the lifetime management activities on TAs on behalf of service providers.
- TEEP Broker: TEE and TAs usually do not have the capability to directly communicate with the TAM. The TEEP Broker is an REE application that enables this communication. CAs might carry the TEEP Broker functionality.

C. GlobalPlatform Specifications

GlobalPlatform is a non-profit consortium undertaking the initiative to standardize TEEs. Specifications are available cost-free from their web site. TEE Core API and TEE Client API [4] specifications are most related to our work since they are the primary interfaces for TEE applications.

TEE Core API is used to implement a TA and TEE Client API is used to invoke the corresponding TA by the CA. There are both open-source (e.g., OP-TEE [5]) and proprietary (e.g., Trustonic [6]) implementations of these APIs. However, once TAs and CAs are implemented using these APIs, they can easily be ported to a chosen GP-complaint TEE implementation (More information on TEE Core API is provided in Section III).

D. Open-TEE

Open-TEE is a virtual TEE (software emulator) conforming to GlobalPlatform specification [1]. It is hardware independent i.e., it does not emulate specific hardware, therefore, TAs developed with Open-TEE can be compiled to any TEE. Two of its libraries, which are of most interest to developers, are *libInternalApi.so* and *libTEE.so*. TAs are linked against the former and CAs are linked against the later library.

A user study to determine whether Open-TEE eases the burden of TA development was conducted with 14 participants [1]. The results suggest that the perceived usability of Open-TEE is higher than that of the current tools used by the experienced TA developers.

In this user study, a discussion with service providers reveals an interesting finding and is in fact the starting point for our work. Open-TEE is intended to be a developer tool but an alternative use is to ship CA and TA with Open-TEE and arrange to use it whenever a real hardware TEE is not detected. The authors noted that the question of how best to isolate it from the REE in the absence of any hardware support was their ongoing work [1]. But up to our best knowledge, no work has been published so far regarding improving the security of Open-TEE in this new usage scenario.

E. White-box Cryptography

In a black-box threat model, a usually-made assumption in cryptography, the attacker has access only to the input and output of cryptographic algorithms. Security practitioners are questioning how realistic this model is when the algorithms are deployed in applications executing on open devices like mobile phones without secure elements. In such a context, a so-called "white-box attacker" may have full access to the software implementation of cryptographic algorithms.

White-box cryptography aims to ensure the security of cryptographic algorithms against a white-box attacker. Code obfuscation is a related and complementary technique for protecting software implementations. The difference is that code obfuscation is aimed at protecting against the reverse engineering of a cryptographic algorithm in more general terms while the main motivation of white-box cryptography is the protection against key extraction.

The work by Chow et al. in 2002 is the first academic study of white-box cryptography [7], [8]. The main idea is to come up with a key-instantiated version of a cryptographic algorithm so that the key is hidden. In other words, for each secret key, a look-up table is implemented so that the key input is unnecessary after the setup [9].

All current white-box approaches of standardized cryptographic primitives such as AES and DES have been publicly broken [10]. To add further protection, external (key-independent) encodings may be used but when these encodings are applied to standard AES and DES, then the result is not standard-compliant. If standard compliance and interoperability is not a goal, another option is to design a dedicated white-box algorithm. One of these algorithms is SPACE proposed by Bogdanov and Isobe [10]. Up to our best knowledge, so far no successful cryptanalysis work is reported against SPACE.

Another contribution of the work by Bogdanov and Isobe is introducing the notion of space-hardness to quantitatively evaluate the difficulty of code lifting attacks [10]. Code lifting is one of the major practical issues regarding security of white-box implementations. It means without extracting the cryptographic keys, the entire implementation (code) is used as one big key. Code lifting attacks could be made more difficult by an incompressible large implementation hopefully beyond an attackers processing capacity to copy and distribute. Device binding countermeasures could also be used to avoid code lifting.

III. Our Solution: WhiteBox-TEE

In this section, we present WhiteBox-TEE. We begin with the adopted attacker model. We explain the provisioning and usage of WhiteBox-TEE bundled with a CA and TA. Then, we give the details of our implementation of a white-box cryptography library and the changes we made to the Open-TEE to make use of white-box cryptography that leads to our proposed WhiteBox-TEE solution. Finally, we provide performance evaluation results of WhiteBox-TEE.

A. Attacker Model

A typical hardware TEE provides many security capabilities (e.g., secure boot) against a powerful and sophisticated threat model. We believe supporting the same set of capabilities in a software-only solution while keep being conformant to GP specification is not a realistic goal. We need to adopt a weaker but still relevant model.

We consider a threat model which involves a malicious party or application attempting to access the sensitive data and cryptographic keys on permanent storage. Although unauthorized access to cryptographic keys, while they are in memory, is also a threat white-box cryptography could address, we do not include it in our attacker model because it requires us to support the white-box encryption algorithms in TEE Core API. Since TEE Core API is built around only the standard encryption algorithms (which do not have unbroken white-box implementations), this contradicts with our target of conformance to GP specification. We choose not to do so and protect cryptographic keys only in rest (while in permanent storage). In other words, secure storage for the cryptographic keys is the main target. We will discuss how we could protect keys while in use with possible extensions to GP specifications in section IV.

We also assume the attacker could not capture user input while entering sensitive information like PIN. Hardware attacks are also out of scope in our work. We also do not address attacks other than integrity and/or confidentiality e.g., denial of service via erasing the disk. The last but not the least, we assume the code running on the mobile device is protected against reverse engineering through code obfuscation and other software protection techniques.

In our attacker model (tailored from [11] to suit our needs), we want to make sure that stored sensitive data and cryptographic keys can only be accessed by authorized applications. The attacker has the following two alternative options for unauthorized access:

- Malicious app attacker: The attacker installs a malicious application and tries to access stored data and keys using its permissions.
- Root attacker: The attacker has root credentials and could run applications with root permissions.

Open-TEE, used as a fallback mechanism, could be assigned the same logical user ID as CA and TA. Then, the best practice is to store the keys in a directory that cannot be accessed by other applications [11]. Hence, we could say that a malicious app attacker could already be avoided. However, with root permissions, all data directories can be accessed. Root attacker is a serious threat against Open-TEE but not against WhiteBox-TEE as we will see.

B. Provisioning and Usage

We envision an enterprise messaging application to motivate the provisioning process. We suppose that due to security concerns, an enterprise asks its employees to use a TEE-supported in-house system instead of popular free applications like Whatsapp. But they adopt a BYOD (Bring Your Own Device) policy for economical reasons. Some of the employees have a TEE-accessible mobile phone while some others do not. Then, the later employees should follow the following steps before using the WhiteBox-TEE integrated mobile application:

1) He (or she) applies to the Company to be authorized to use the app. Once authorized, he obtains a PUK (Personal Unlocking Key)[1]. This could be done offline in person for improved security.
2) He downloads and installs the TEEP application from the app store (or from a local server).
3) Once he opens the app for the first time, TEEP checks whether a hardware TEE is available or not (in our case it is not detected).
4) The app opens a secure connection through TLS with the TAM. It informs hardware TEE is not detected

[1]We overload the term PUK, which has more digits than a PIN and used not only for unblocking but also to initialize.

(optionally, it sends the unique device ID which will be used by the TAM to bind the bundle to be downloaded to the device so that the bundle will not work on any other device, further details are not discussed here).

5) He authenticates himself with the PUK.
6) Upon successful entry of the PUK, the bundle (CA, TA, and WhiteBox-TEE) is downloaded from the TAM server.
7) Before the installation takes place, the following steps are carried out:
   (a) The user is asked to choose a PIN (shorter than PUK).
   (b) The white-box encryption algorithm is initialized i.e., the lookup tables of white-box algorithm (WB-Tables) are constructed after generating a key using PUK as a seed to a PRF [12][2]).
8) CA, TA, WhiteBox-TEE, and WB-Tables are installed (stored in local storage) to the mobile phone.
9) The PUK is erased securely.
10) The PIN is encrypted (with the white-box algorithm) and stored.

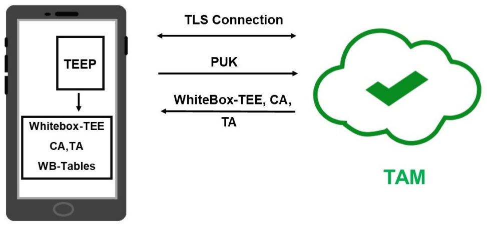

Fig. 1. Provisioning of WhiteBox-TEE integrated mobile applications.

After the successful provisioning, the WhiteBox-TEE integrated mobile application is used as follows:

1) The user opens the application.
2) He enters the PIN.
3) The PIN encrypted with the white-box algorithm is decrypted and is matched with the user-entered PIN.
4) If the PIN is correct, the execution continues. Otherwise, the PIN is re-asked for a limited number of times (e.g., two more times). If it is false in all trials, the application closes.
5) TA and CA are executed. During the execution of TA, whenever an encryption key is created or opened, it is carried out after encryption or decryption with white-box library. Any cryptographic key is never stored in plaintext.

In the following subsection, the last item is discussed further with implementation details.

C. Development of a White-Box Cryptography Library and Changes Made to TEE Core API

We developed a white-box cryptographic library as required for WhiteBox-TEE. It currently supports only the SPACE algorithm [10]. The library supports three main operations:

1) **Table generation:** As mentioned, white-box encryption algorithms including SPACE algorithm do not require key input after a lookup table is generated from a secret key. A function which takes SPACE type - the variant of the specific block cipher in SPACE family of algorithms (SPACE-8, SPACE-16 or SPACE-24) and secret key as input and returns a lookup table is implemented:

```
uint8_t **generate_table(
uint8_t space_type,uint8_t *key);
```

   For table generation, SPACE uses a block cipher, which could be a standard algorithm in the black-box model. As recommended by SPACE designers, AES-128 is preferred in our implementation. The implementation is speeded up with the AES instruction set (ARM architecture cryptographic extensions).

2) **Encryption:** Encryption function implements encryption of a variable-sized input plaintext message using SPACE algorithm and counter mode of operation [13]. Other inputs to the function are a nonce value, length of the input message, lookup table and SPACE type. The output is a pointer to the ciphertext.

```
uint8_t *encrypt_input_ctr_mode(
uint8_t *plaintext,uint8_t *nonce,
uint32_t length,uint8_t **table,
uint8_t space_type);
```

3) **Decryption:** Decryption function implements decryption of a ciphertext message using SPACE algorithm and counter mode of operation. Similar to encryption function, other inputs to the function are a nonce value, length of the input message, lookup table and SPACE type. The output is a pointer to the plaintext message.

```
uint8_t *decrypt_input_ctr_mode(
uint8_t *ciphertext,uint8_t *nonce,
uint32_t length,uint8_t **table,
uint8_t space_type);
```

As mentioned, TEE Core API and TEE Client API specifications are the primary interfaces for a CA and TA, respectively in a TEE application. Since security-sensitive operations are performed by the TA, we do not need to change *libTEE.so* in Open-TEE library and need to analyze TEE Core API to understand the modifications required for *libInternalApi.so* so that the cryptographic keys are securely stored.

TEE Internal Core API provide TAs a set of core services. These core services can be classified into four broad categories [4]:

- Trusted Storage API for Data and Keys
- Cryptographic Operations API
- Time API
- TEE Arithmetic API

[2]The reason of using PUK as the seed is not security-related (the key can be independent of the PUK) but preferred to be able to recover the encrypted data if the tables are damaged and to ease porting to a new phone when needed.

To integrate the white-box crypto library with TEE Core API for secure storage, we need to modify the implementation of only the first of these services; Trusted Storage API.

According to TEE Core API Specification [4], each TA has access to a set of Trusted Storage Spaces, identified by 32-bit Storage Identifiers (however, the current version defines a single such space for each TA). The objects in this storage space are accessible only to the TA that created them and are not visible to other TAs, thus a private storage space is defined for the TA.

Trusted Storage Space contains so-called Persistent Objects, each of which is identified by an Object Identifier i.e., *storage_blob_id*. A persistent object can be of three types; a Cryptographic Key Object, a Cryptographic Key-Pair Object, or a Data Object. When the TA wants to generate or derive a persistent key, it has to first use a transient object, then write the attributes of a transient object into a persistent object. There are four persistent object functions:

(i) TEE_OpenPersistentObject
(ii) TEE_CreatePersistentObject
(iii) TEE_CloseAndDeletePersistentObject1
(iv) TEE_RenamePersistentObject

Besides these functions, there are also functions to access the data streams of persistent objects. Again, there are four data stream access functions:

(i) TEE_ReadObjectData
(ii) TEE_WriteObjectData
(iii) TEE_TruncateObjectData
(iv) TEE_SeekObjectData

As required to remain conformed to TEE Core API Specification, we did not change the interface of any of these functions. However, we made change to lower-level functions used by these functions. We briefly explain these changes, below:

In the original version of the TEE_ReadObjectData function, which is usually called after running TEE_OpenPersistentObject, the object is filled by calling *ext_read_stream()*. We modify this function so that after reading the stream first a check is performed whether lookup tables are available or not. If available, *decrypt_input_ctr_mode()* function from the white-box crypto library is called to decrypt the stream before it is returned (note that a lookup table should have already been generated from a secret key as part of the provisioning process).

In a similar fashion, *encrypt_input_ctr_mode()* function from the white-box library is called in the *ext_write_stream()* before writing the stream.

D. Performance of WhiteBox-TEE

Among the three main operations supported by our white-box crypto library, the most time consuming operation is the table generation which is fortunately required only for one-time.

In the implementation of SPACE, any block cipher algorithm can be used for the table generation. As suggested by its inventors, we instantiate the SPACE family with AES-128 as the underlying block cipher which is accelerated using hardware instructions. Performance evaluation of table generation for different SPACE variants using Android 8.0 Oreo (Octa-core Max 2.45 GHz ARMv8-A CPU and 6 GB memory) mobile phone is given in Table I. We do not find SPACE-32 feasible for our target platforms therefore did not implement it since table storage requires 51.5 GB of memory which is not available in today's most mobile phones. Furthermore, huge table sizes cause greater performance costs [14].

Space Type	Software Implementation (ms)	Hardware-accelerated Implementation (ms)	Memory Req. (MB)
SPACE-8	0.26	0.06	0.00375
SPACE-16	70.24	7.19	0.896
SPACE-24	8013.84	407.45	218

TABLE I
PERFORMANCE OF TABLE GENERATION OPERATION FOR DIFFERENT SPACE VARIANTS.

The timing values for encryption and decryption are also measured and given in Table II. Due to algorithmic details, the most efficient SPACE variant is not SPACE-8 but SPACE-16 with which encryption and decryption could be performed at a rate of around 5.40 MB/secs.

Space Type	SPACE Encryption / Decryption (MB/secs)
SPACE-8	3.49
SPACE-16	5.40
SPACE-24	0.84

TABLE II
PERFORMANCE OF ENCRYPTION/DECRYPTION OPERATIONS FOR DIFFERENT SPACE VARIANTS.

IV. Run-time Protection: WhiteBox-TEE+

In this section, we present WhiteBox-TEE+ to have better use of white-box cryptography i.e., to provide protection to cryptographic keys also while in use with a required revision to TEE specifications. Our suggestion is simple: just add SPACE (or another chosen white-box algorithm to the list of supported algorithms) in Cryptographic Operations API (as symmetric ciphers, currently supported algorithms are DES, Triple-DES and AES). Since white-box algorithms operate differently than classical algorithms i.e., use of a table instead of a key for encryption and decryption, this revision requires more than just a new entry in the list of supported algorithms. For instance, *TEE_CipherInit* function specified in the TEE Core API [4], which starts the symmetric cipher operation, is no longer an operation that must have been associated with a key. Therefore this requirement needs an exception. The exact details of all revisions to the specifications is out of scope in our current work hence not discussed further.

While it is viable to add a white-box algorithm for symmetric encryption, this is not true for asymmetric encryption, digital signature and key exchange algorithms because no drop-in replacement for conventional public-key schemes has been proposed yet [15]. In applications such as secure messaging, communication security is usually provided using a hybrid cryptosystem (combination of symmetric and asymmetric techniques). Hence, this deficiency is a serious limitation. On the other hand, the suggested revision could be sufficient for storage security and security applications such as digital rights management.

An important observation we made is that use of white-box algorithms helps not only in software-only TEEs but in hardware TEEs as well because tamper-resistance is not a requirement as per GP TEE specifications [16]. Use of white-box cryptography providing software tamper resistance may help hardware based tamper resistance to become less of a concern for certain applications.

V. Discussion and Future Work

Up to now, we consider only one of these two cases: either a TEE is available or not to execute a TA. However, in practice there is a gray area in between where a TEE is present but its use is limited i.e., only for a cryptographic key service provided by the device manufacturer that could serve for applications running in REE. The situation is even more complex due to the fact that the same phone model may or may not provide this service depending on the version of Android running on it [11].

A relevant question regarding the TEE-supported key service is whether WhiteBox-TEE is still needed or not if such a service is available to use. Cooijmans et al. analyzed the security of such a scenario and concluded that by itself it does not provide security against a root attacker [11]. We remind that WhiteBox-TEE protects against the root attacker since the key is securely erased after the lookup table is generated.

To keep things simple, a viable alternative solution against the root attacker is the encryption of all cryptographic keys with a master key derived from a user-supplied PIN. As a result, the only data accessible by the root attacker is encrypted keys useless without the master key. In fact, the PIN that is used to authenticate the user could also be used to generate the master key used to decrypt all keys before they are needed. We argue that WhiteBox-TEE has still advantage over this solution because a typical 4-6 digits PIN is vulnerable against a brute-force off-line attack[3] even though the attack could be made more time-consuming by memory-hard functions [17]. We note that a longer PUK used in our solution, only entered once (and possibly protected by a memory-hard function on the TAM server), would not have such a vulnerability.

The comparison with respect to security and compliance properties of all the solutions discussed so far is presented in Table III. Advanced security features mentioned in the table include trusted user interface, secure boot and other undiscussed properties. NA (Not Applicable) means the solution is generic and compliance is a non-issue.

[3]This attack involves obtaining a ciphertext and trying all possible combinations of PINs to see which master key derived from the PIN is successful for successful decryption.

Solution	Protection against root attacker	Brute-force attack resistance	Run-time protection	Advanced security features	GP Compliance
TEE-supported key service	-	+	-	-	NA
Encryption with PIN-derived key	+	-	-	-	NA
TA on Whitebox-TEE	+	+	-	-	+
TA on WhiteBox-TEE+	+	+	+	-	-
TA on Hardware TEE	+	+	+	+	+

TABLE III
Comparison of solutions with respect to security and compliance properties.

As mentioned, code lifting attacks are one of the serious concerns for white-box crypto implementations. Although, the attack could be made harder by choosing a SPACE variant with a large table size, there is still room for more attack resistance by the use of code obfuscation and device binding techniques. Doing research on these techniques and choosing the most appropriate ones for WhiteBox-TEE integration is the first item in our agenda. Another promising research direction is to explore other options for white-box algorithm support in TEE specifications (e.g., use of public-key schemes).

VI. Conclusion

The use of mobile phones for security sensitive applications (e.g., banking, bitcoin wallets, shopping) is on the rise. Mobile platforms will play a key role in next generation smart and intelligent systems and applications. However, the environment of Android, the most widely used mobile operating system, with its huge kernel size and hosting many potentially-malicious applications developed by different third parties is far from satisfying the security requirements for these use-cases.

Trusted Execution Environments provide hardware-based isolated execution and can be used to safeguard the information processed within it. While it has a huge potential to mitigate most of the concerned mobile risks, its use by application developers is not always possible because device vendors could limit its use solely for their purposes. If this is the case, Open-TEE, a TEE emulator software, could be used as a fall-back mechanism. On the other hand, Open-TEE does not provide any security feature or isolation property.

In this paper, we aimed to answer how best to isolate Open-TEE from the Android threats in the absence of any hardware support. In our solution, we get help from white-box cryptography to design and implement WhiteBox-TEE which could provide protection against an attacker who has root credentials and could run applications with root permissions. Being compliant to GlobalPlatform (GP) specifications, we believe WhiteBox-TEE is a promising solution for developers who would like to present a consistent user experience for their clients who have or have not a TEE-enabled mobile device. We also present WhiteBox-TEE+ to provide run-time protection for TA's cryptographic operations using white-box cryptography with a revision to GP specifications.

REFERENCES

[1] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-tee: An open virtual trusted execution environment." in *IEEE Trustcom/BigDataSE/ISPA*, vol. 01. IEEE, 2015, pp. 400–407.

[2] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices." in *IEEE Security & Privacy*, vol. 12. IEEE, 2014, pp. 29–37.

[3] M. Pei, H. Tschofenig, D. Wheeler, A. Atyeo, and L. Dapeng, "Trusted execution environment provisioning (teep) architecture." 2019, last accessed 15 August 2019. [Online]. Available: http://www.potaroo.net/ietf/ids/draft-ietf-teep-architecture-03.txt

[4] "Globalplatform specifications," last accessed 15 August 2019. [Online]. Available: https://globalplatform.org/specs-library/?filter-committee=tee

[5] "Open portable trusted execution environment," last accessed 15 August 2019. [Online]. Available: https://www.op-tee.org/

[6] P. Butterworth, "Devices with trustonic tee." 2015, last accessed 15 August 2019. [Online]. Available: https://www.trustonic.com/news/blog/devices-trustonic-tee/

[7] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot, "White-box cryptography and an aes implementation," in *Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*, ser. SAC '02. Springer-Verlag, 2002, pp. 250–270.

[8] S. Chow, H. Johnson, and P. C. van Oorschot, "A white-box des implementation for drm applications." in *ACM Workshop on Digital Rights Management*. Springer-Verlag, 2002, pp. 1–15.

[9] M. Joye, "On white-box cryptography." in *Security of Information and Networks*. Trafford Publishing, 2008, pp. 7–12.

[10] A. Bogdanov and T. Isobe, "White-box cryptography revisited: Space-hard ciphers," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1058–1069.

[11] T. Cooijmans, J. de Ruiter, and E. Poll, "Analysis of secure key storage solutions on android," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM, 2014, pp. 11–20.

[12] L. Chen, "Recommendation for key derivation using pseudorandom functions." no. 800-108. National Institute of Standards & Technology, 2009.

[13] D. A. McGrew and J. Viega, "The galois / counter mode of operation (gcm)," in *NIST Modes of Operation Process*, 2004.

[14] R. Dahab, J. Lpez, C. R. Rodrigues, Flix, H. Fujii, G. Sider, and A. C. Serpa, "White box implementations of dedicated ciphers on the arm neon architecture," in *SBSeg 2018*. SBC, 2018, pp. 9–16.

[15] A. Biryukov, C. Bouillaguet, and D. Khovratovich, "Cryptographic schemes based on the asasa structure: Black-box, white-box, and public-key," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2014, pp. 63–84.

[16] C. Shepherd, G. Arfaoui, I. Gurulian, R. P. Lee, K. Markantonakis, R. N. Akram, D. Sauveron, and E. Conchon, "Secure and trusted execution: Past, present, and future-a critical review in the context of the internet of things and cyber-physical systems," in *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 2016, pp. 168–177.

[17] D. Boneh, H. Corrigan-Gibbs, and S. Schechter, "Balloon hashing: A memory-hard function providing provable protection against sequential attacks," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 220–248.