Annales Mathematicae et Informaticae
53 (2021) pp. 245–256
DOI: https://doi.org/10.33039/ami.2021.02.002
URL: https://ami.uni-eszterhazy.hu

TEE-based protection of cryptographic keys on embedded IoT devices*

Dorottya Papp, Máté Zombor, Levente Buttyán

Laboratory of Cryptography and System Security,
Department of Networked Systems and Services,
Budapest University of Technology and Economics
www.crysys.hu

Submitted: November 13, 2020
Accepted: February 11, 2021
Published online: May 18, 2021

Abstract

The Internet of Things (IoT) consists of billions of embedded devices connected to the Internet. Secure remote management of many of these devices requires them to store and use long-term cryptographic keys. In this work we propose to protect cryptographic keys in embedded IoT devices using a Trusted Execution Environment (TEE) which is supported on many embedded platforms. Our approach provides similar protection as secure co-processors, but does not actually require an additional secure hardware element.

Keywords: Trusted Execution Environment, cryptographic keys, key management

AMS Subject Classification: 68M25 (Computer Security)

1. Introduction

The Internet of Things (or IoT for short) consists of billions of embedded devices connected to the Internet. This new phenomenon is the basis for today's smart applications in the domains of manufacturing (Industry 4.0), transportation (Cooper-

*The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004), which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

ative Intelligent Transportation Systems), and healthcare (personalized e-Health), as well as in everyday life (smart cities, smart homes). However, in almost all application areas of IoT, we face security and privacy issues which require solutions developed for or adapted to the special characteristics of IoT systems. Security and privacy mechanisms should take into account the resource limitations of embedded devices and they should not rely on special hardware that would significantly increase the development cost of IoT applications. This leads to interesting challenges for managing cryptographic keys on IoT devices.

In many applications, IoT devices are managed remotely by system operators. Such remote management requires secure remote access to the devices, which in turn, requires the devices to store and use long-term cryptographic keys. For instance, the operator usually needs to authenticate the device before uploading configuration data or software updates on it, which may require the device to use a long-term, device-specific private key. However, as IoT devices are connected to the Internet, they may be compromised by malicious actors (aka attackers). If an attacker can obtain the long-term key of a compromised device, (s)he can impersonate and clone that device, which is undesirable. Hence, there is a need to protect long-term cryptographic keys on IoT devices such that a key remains inaccessible to the attacker even if the device itself is compromised.

A possible solution to the problem above would be to store cryptographic keys on IoT devices in secure co-processors, such as a TPM chip[1] that would never output a key, but only use it internally in cryptographic operations. However, requiring an additional co-processor on every IoT device would be too expensive in most cases.

In this work we propose a more cost efficient approach: we ensure protection of cryptographic keys by using a Trusted Execution Environment (TEE), which is mostly based on software with some minimal hardware support, and it is supported on many embedded platforms used in IoT applications. For instance, many embedded devices use ARM processors that feature the ARM TrustZone technology[2], which enables the establishment of a software-based TEE and provides some hardware-based protection mechanisms to them. TEEs usually implement a persistent secure storage service (see, e.g., the TEE specifications[3] of GlobalPlatform, a non-profit industry association aiming at enabling digital services and devices to be trusted and securely managed throughout their lifecycle), which can be used to store long-term cryptographic keys. Moreover, operations with those keys can be performed by trusted applications running within the TEE, hence, the keys would never leave the protected environment of the TEE.

[1]`https://trustedcomputinggroup.org/resource/tpm-library-specification` (last accessed: Oct 3, 2020)

[2]`https://developer.arm.com/ip-products/security-ip/trustzone` (last accessed: Oct 3, 2020)

[3]`https://globalplatform.org/specs-library/?filter-committee=tee` (last accessed: Oct 3, 2020)

2. Background

Long-term cryptographic keys have been traditionally protected using additional hardware elements, such as Hardware Security Modules (HSMs) or Trusted Platform Modules (TPMs) and secure co-processors. These hardware component provide cryptographic operations to implement secure boot, trustworthy reporting, attestation, and other components of secure computing [2]. HSMs are external hardware modules which can be attached to existing computer systems and used via PCI, USB, or network connection. They provide cryptographic functionality, as well as tamper-resistance, and are often used to securely generate, store and use cryptographic keys. Typically, HSMs implement PKCS #11[4], a platform-independent API to handle cryptographic tokens. The API itself is called Cryptoki and has header files for C and C++ applications; vendors usually have their own compliant implementations. There exists also software-based HSM implementations, for example, the SoftHSM[5], which is a well maintained open source project. It is part of the OpenDNSSEC project[6] with goal of being a complete implementation of PKCS #11.

TPMs, on the other hand, are chips embedded on the computer's motherboard and offer several security-relevant features in a standardized manner: protected memory and registers to securely execute commands, tamper-evident hardware module to store keys, cryptographic processing capability and a true random number generator. They are usually used as hardware roots of trust for measurement, storage and reporting, as well as to implement critical functionalities. TPM chips are commercially available on the market [3] and there is research effort [1, 14] to implement the same concepts in software.

The main disadvantage of the previously mentioned hardware-based solutions is that they are additional and often costly components of the system. By comparison, IoT devices are constrained not only in resources but in cost as well [3]. As a result, hardware-based protection for cryptographic keys is not viable economically in the IoT setting. There exists software-based implementations of the hardware concepts, but those are typically implemented as kernel modules which could be compromised by an attacker with elevated privileges.

However, there exists an emerging technology which can provide a secure and integrity-protected processing environment: the TEE. TEE runs on the same hardware as the device's main operating system (OS) but it is also isolated at the hardware-level. Many chips used in embedded devices offer the hardware support necessary to realize Trusted Execution Environments [15]. Examples include the ARM TrustZone[7], the Intel Software Guard eXtension[8] (SGX) [9], and the AMD

[4]`http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html` (last accessed: Nov 04, 2020)

[5]`https://www.opendnssec.org/softhsm/` (last accessed: Nov 04, 2020)

[6]`https://www.opendnssec.org/` (last accessed: Nov 04, 2020)

[7]`https://developer.arm.com/ip-products/security-ip/trustzone` (last accessed: Oct 12, 2020)

[8]`https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top.html` (last accessed: Oct 12, 2020)

Secure Encrypted Virtualization [5]. OP-TEE[9] and Open-TEE [7] are two TEE implementations which can be deployed on these chips.

Figure 1 shows the main components of a device with TEE capabilities. Logically, execution can be separated into the Rich Execution Environment (REE) and the TEE. Code running in the REE has access only to unprotected resources (e.g. memory). "Code" in the REE can be partitioned into the Rich operating system, usually a traditional OS such as Linux, and one or more applications, which run on top of the Rich OS. Such an application is called a Client Application (CA) in the TEE architecture. CAs implement the basic features of the device, e.g. web servers for configuration, applications for sensing physical parameters of the environment, or the actuator controlling a physical process. When necessary, CAs can request services from the TEE via the TEE Client API. This API forwards the request to a special component in the Rich OS, the REE Communication Agent, which triggers a context switch and gives control to the TEE.

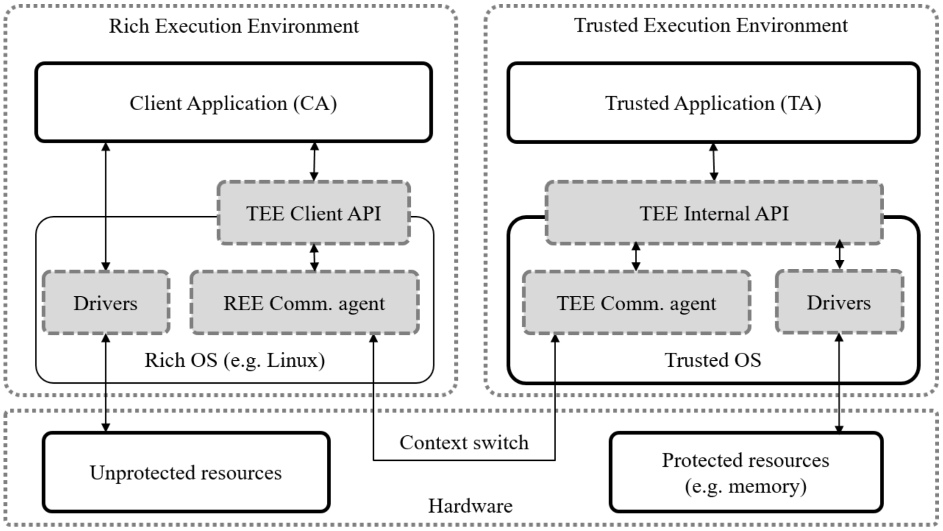

Figure 1. Logical overview of a device with Trusted Execution Environment capabilities.

Code in the TEE has access to protected resources, which are unavailable to the REE. For example, certain memory locations are only available to code running in the TEE. This protection is provided by the hardware components of the device. In the case of the ARM TrustZone, for example, the architecture includes a special register storing the Non-secure (NS) bit to determine whether the executed code belongs to the REE or the TEE. If the NS bit is set, signalling that the executed code belongs to the REE, access to certain protected memory locations is automatically denied. The TEE is similar to the REE in the sense that it has

[9]`https://optee.readthedocs.io/en/latest/index.html` (last accessed: Oct 12, 2020)

an operating system (the trusted OS) and several applications, which are called Trusted Applications (TAs). TAs provide those services for the REE whose computation requires strong security guarantees, for example, remote attestation [12, 13], tamper-resistant logging and storage [10, 11], or secure real-time computation for the Industrial IoT [8].

3. Architectural overview

The basic idea of our approach is to use the TEE to provide similar protection to keys as a secure co-processor but without actually requiring another processor on the device: the same processor runs a normal execution environment (the REE) and a TEE, and also implements the required hardware mechanisms that isolate these two execution environments. This isolation ensures that even if the REE is compromised, the attacker would not be able to obtain the keys stored and used within the TEE. This protection mechanism prevents attackers to clone compromised devices.

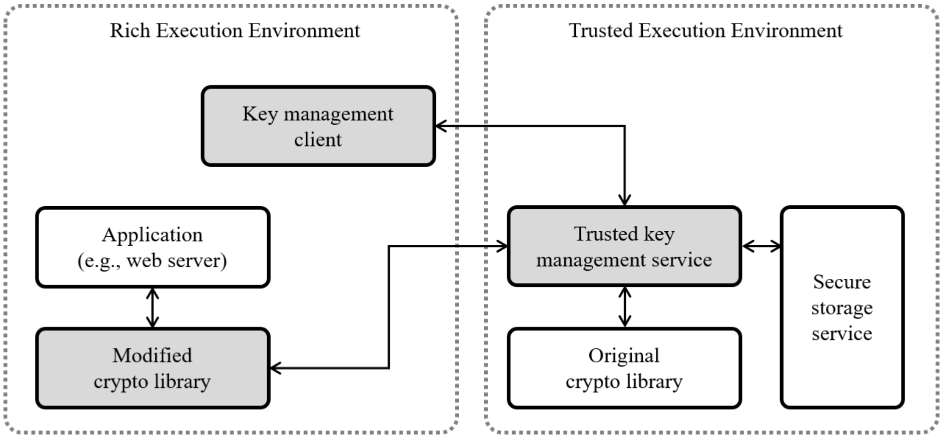
Rich Execution Environment
Trusted Execution Environment
Key management client
Application (e.g., web server)
Modified crypto library
Trusted key management service
Original crypto library
Secure storage service

Figure 2. Architecture of our TEE based key management solution. Grey boxes represent components that we developed or modified.

The architecture of our solution is illustrated in Figure 2. Private keys and private-public keypairs are stored in the secure storage of the TEE. We also store the intended use of keys, e.g. signing or decryption, in an additional attribute in the TEE. The keys could be generated by the operator off-line and loaded in the secure storage in a controlled way with the help of a key management client, or the key can actually be generated and stored in secure storage by the trusted key management service itself. In the latter case, the trusted key management service would output the corresponding public key to the key management client such that it can be made available to applications running outside of the TEE. In both cases, handles to the private keys would be output from the trusted key management service that can be used by applications in the REE to refer to the private keys when requesting operations with them.

Generating and loading keys into the TEE should only be performed by the device's operator, therefore, such requests must be authenticated. Request authentication requires the operator to set up a master password before the device is deployed. The trusted key management service allows the key management client to install the master password only once, it cannot be changed later. Requests related to key management must provide not only the invocation parameters to the underlying cryptographic library but also a salt and a message authentication code (MAC). We refer to the combination of salt and MAC as the authentication token. The master password is used together with the salt to derive a key. The derived key and the invocation parameters are input to HMAC (RFC 2104[10]) and its output is compared with the MAC value supplied in the request. The request authentication process is illustrated in Figure 3. Key management operations are only performed, if the HMAC-based authentication scheme succeeds without errors. We also log authentication tokens in the trusted key management service to prevent replay attacks with previous key management requests. If a request contains a previously used authentication token, the request is automatically denied.

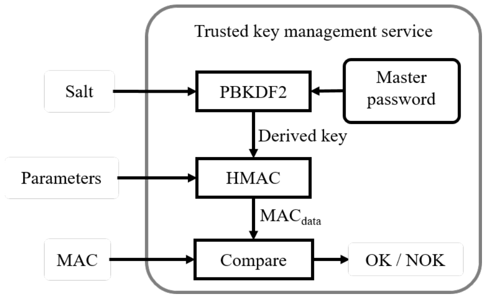

Figure 3. Process overview of authenticating requests from the key management client in the trusted key management service.

Any application (e.g., a web server that provides a remote configuration possibility for the operator of the device) that runs in the REE can be compiled with a cryptographic library that we modified such that private key cryptographic operations are delegated to the trusted key management service running in the TEE. In TEE terminology, the modified cryptographic library acts as a CA and the trusted key management service is a TA. From the application's point of view, the cryptographic library exposes functions to encrypt and decrypt data, which can be invoked similarly to API functions, as shown in Figure 4. However, instead of supplying the key itself, the application provides a handle to the private key with which to perform the cryptographic operation. The modified cryptographic library serialized the key handle and the provided parameters as a message and passes

[10]`https://tools.ietf.org/html/rfc2104` (last accessed: Nov 12, 2020)

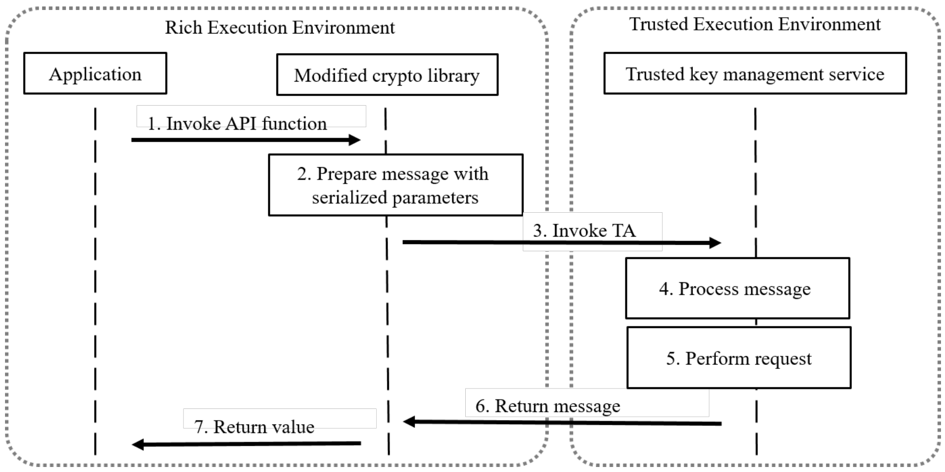

Figure 4. Interactions involved in performing cryptographic operations with private keys stored and managed in the TEE.

that to the trusted key management service. The trusted key management service, which is compiled with the original cryptographic library, processes the serialized parameters, retrieves the key referred by the provided key handle, and calls the original cryptographic library to execute the requested operation. The results are passed back to the modified cryptographic library and the modified cryptographic library provides the return value to the application.

The two components can pass parameters and values to each other via shared memory: a block of memory which is shared between the CA and the TA. Both the CA and the TA can read data from and write data to the shared memory, however, only the CA can allocate it. Therefore, the modified cryptographic library must allocate memory to hold the results of cryptographic operations. Knowing the requested operation and information about the key, the modified cryptographic library can estimate the necessary amount of memory. If the modified cryptographic library underestimated the amount of memory, the trusted key management service returns a special message requesting more memory to return the result.

4. Prototype implementation

We implemented the proposed TEE-based architecture for protecting long-term cryptographic keys using the Trusted Firmware[11] projects OP-TEE and mbedtls. Trusted Firmware provides a reference trusted code base for the ARM platform, a widely used platform in embedded devices. OP-TEE is an open source implementation of GlobalPlatform's TEE specification, primarily maintained by Linaro, and it is usually used in conjunction with the Linux kernel in the REE. mbedtls is a

[11]https://www.trustedfirmware.org/ (last accessed: Nov 03, 2020)

cryptographic library written in C with a small code footprint. It can be used in both the REE and the TEE; OP-TEE can be complied to use mbedtls as the default cryptographic library.

For our prototype implementation, we set mbedtls as OP-TEE's default cryptographic library. We implemented a Trusted Application which fulfills the role of trusted key management service and handles incoming requests for cryptographic and key management operations. The Trusted Application stores the key pair object in the secure storage and passes it to mbedtls whenever cryptographic operations are to be performed. We also compiled a modified version of mbedtls's source code in the REE such that it includes wrapper functions to direct requests to our Trusted Application. Our prototype implementation consists of eight wrapper functions as follows:

- `tee_set_master_password`: Installs the specified master password into the trusted key management service to authenticate key management requests. This function can only be called once, we assume that it is done in a controlled environment by the device's operator.
- Key management functions: These functions perform privileged operations allowed only for the operator. Therefore, the Trusted Application performs the request authentication process described in Section 3 on their inputs.
  - `tee_generate_keypair`: Generates a long-term private-public key pair and stores it in the TEE. The function returns a handle to the key pair which can be later used for other cryptographic operations.
  - `tee_load_keypair`: Allows the operator to load an existing key pair into the TEE. The key pair must be encrypted and in PEM format. Similarly to `tee_generate_keypair`, this function also returns a handle to the key pair.
  - `tee_remove_keypair`: If a key pair becomes compromised or is considered weak, the operator can inactive it. We do not permanently delete keys because the attacker might try to reinstall old and weak keys. Instead, inactivating keys allows us to maintain a list of all previously and currently used keys. The list could be reviewed by the operator or used for attestation purposes.
- Functions available for all applications: All of these functions reference a key stored in the TEE with a key handle. In our prototype, handles are 32 bytes long and calculated as the SHA256 hash value of the key pair.
  - `tee_pk_decrypt`: Decrypts the supplied data with a given key.
  - `tee_pk_sign`: Digitally signs the input data with a given key.
  - `tee_get_keyinfo`: Returns the type, the size, and the intended usage of a given key.
  - `tee_get_publickey`: Returns the public key of a public-private key pair in plaintext.

For each function, we defined a custom message format which can hold the serialized parameters to be passed to the trusted key management service prototype. In all cases, messages start with an ID field identifying the operation requested, followed by the key handle. Depending on the function, the key handle can be an input parameter and an output parameter. For example, the function `tee_remove_keypair` expects a key handle as an input, while for the function `tee_load_keypair`, the field for the key handle is empty and must be filled with the handle assigned by the TA. The remainder of the message formats follow the length-value convention: first comes the length of the data as an 8-byte-long unsigned integer, then the data as a variable length field.

5. Evaluation

In order to measure the added overhead of TEE-based key protection, we conducted the following experiment. We set up a QEMU-based[12] environment for running our prototype implementation and manually saved an RSA long-term key pair in the TEE. We deployed two versions of mbedtls's example web server with TLS capabilities in REE: one without any modifications and another with the modification to relay cryptographic operations to our trusted key management service prototype. We used mbedtls's example client to test the connection to the web server and repeatedly send HTTP GET requests to both versions.

Our experiment was concerned with the amount of time required to perform cryptographic operations using our trusted key management service prototype. We sent 10 HTTP GET requests from the client to the webserver and measured the amount of time it took for the sign operation to complete. Communicating parties used the `TLS-ECDHE-RSA-WITH-CHACHA20-POLY1305-SHA256` chiper suite during the TLS Handshake. The communication between client and server succeeded in all 10 exchanges. In case of the unmodified mbedtls operations, all operations take place in REE memory. In case of our trusted key management service prototype, the measured amount of time includes the context switch between REE and TEE, as well as the time necessary to perform the requested operation and return the result.

The results of the experiment are shown in Table 1. Our trusted key management service prototype needed an average of 204 ms for the sign operation. This is 5x slower than mbedtls's unmodified operations which take place in REE memory. However, it is worth noting that after the first run, mbedtls's unmodified operations gain a performance boost: their required time to complete changes from 87 ms to ca. 30 ms. This performance boost is the result of mbedtls's implementation to prevent timing attacks. The authors of [6] presented timing attacks in which they measured the amount of time required to perform private key operations, consequently finding fixed Diffie-Hellman exponents and factor RSA keys. The proposed protection against such attacks involves the use of *blinding values*, a pair of ran-

[12]https://www.qemu.org/ (last accessed: Nov 10, 2020)

dom numbers (v_i, v_f) such that in the case of Diffie-Hellman, $v_f = (v_i^{-1})^x \mod n$, and in the case of RSA, $v_i = (v_f^{-1})^e \mod n$. The chosen numbers are then used similarly to blind signatures [4]: the input is multiplied by v_i and the result is corrected by multiplying it with $v_f \mod n$. However, computing the inverses is slow, therefore, mbedtls's implementation uses SSL session information to determine whether (v_i, v_f) has been chosen before and if yes, it updates their values by squaring. Unfortunately, our trusted key management service does not have access to SSL session information and must select a new random (v_i, v_f) pair for each computation.

Table 1. Comparisons between the performance of the unmodified mbedtls library and our trusted key management service prototype in the TEE. The first two columns show the performance of the operation on the server-side, while the last two columns show the amount of time required to build a secure communication channel and exchange an HTTP GET request and response between the client and the server.

	mbedtls's sign operation	Our TEE-based sign operation	Communication using mbedtls	Communication using TEE
Run 1	87 ms	208 ms	410 ms	533 ms
Run 2	30 ms	204 ms	331 ms	505 ms
Run 3	30 ms	203 ms	341 ms	516 ms
Run 4	29 ms	206 ms	319 ms	501 ms
Run 5	29 ms	204 ms	305 ms	507 ms
Run 6	30 ms	203 ms	326 ms	522 ms
Run 7	29 ms	203 ms	312 ms	504 ms
Run 8	29 ms	206 ms	315 ms	511 ms
Run 9	38 ms	202 ms	388 ms	503 ms
Run 10	33 ms	204 ms	341 ms	500 ms
Mean	**36 ms**	**204 ms**	**339 ms**	**508 ms**
Std.dev	**18 ms**	**2 ms**	**34.22 ms**	**10.59 ms**

From the client's perspective, completing a full TLS handshake and exchanging an HTTP GET request and response over the secure channel is 1.49x slower, if cryptographic operations with the long-term key are performed in the TEE. In case of the unmodified mbedtls library, the exchange takes 339 ms on average, while in case of our trusted key management service prototype, the same exchange is completed in 508 ms on average. The results in Table 1 suggest that network latency and SSL session management in both cases accounts for ca. 300 ms. Thus, the increased time necessary to complete the exchange using our trusted key management service in the TEE is the result of the overhead caused by the TEE-based sign operation.

6. Conclusion and future work

Remote administration is one of the key enabling features of IoT devices. However, remote administration requires secure communication channels, which in turn require the protection of long-term cryptographic keys. Traditionally, such keys are protected using additional hardware components, however, the cost of including such components in IoT devices is economically unviable.

In this paper we proposed Trusted Execution Environments as alternative. Their main advantage is that they are mostly software components requiring minimal hardware support for isolation. Our basic idea is to use the TEE's secure storage to protect keys in rest and run cryptographic libraries in the TEE which can protect the keys during execution thanks to access to protected resources. Our architecture includes a trusted key management service in the TEE whose task is to handle the TEE's secure storage and invoke the cryptographic library inside the TEE. Applications not running in the TEE can request operations from the trusted key management service. We created a prototype implementation of the proposed architecture using OP-TEE, an open-source TEE implementation, and mbedtls, a cryptographic library designed to run on small devices. We measured the performance overhead of performing cryptographic operations in the TEE. While there certainly was an overhead due to context switches, the overhead we measured was bearable and did not threaten the communication between client and server. Thus, we can conclude that TEEs are indeed viable alternatives to HSMs and TPMs to protect long-term cryptographic keys.

Other security-critical operations could be implemented in the TEE, as well. Our current research ideas include integrity monitoring from the TEE and using the results for remote attestation of IoT devices. One of the main challenges of remote attestation is how to ensure the trustworthiness of attestation results in the presence of an attacker. TEEs can solve this problem: even if the attacker compromises the main operating system, the device's hardware support for TEEs isolates the attestation process and cryptographic keys from the attacker.

References

[1] N. Aaraj, A. Raghunathan, N. K. Jha: *Analysis and Design of a Hardware/Software Trusted Platform Module for Embedded Systems*, ACM Trans. Embed. Comput. Syst. 8.1 (Jan. 2009), issn: 1539-9087,
doi: `https://doi.org/10.1145/1457246.1457254`.

[2] A. Avizienis, J. Laprie, B. Randell, C. Landwehr: *Basic concepts and taxonomy of dependable and secure computing*, IEEE Transactions on Dependable and Secure Computing 1.1 (2004), pp. 11–33,
doi: `https://doi.org/10.1109/TDSC.2004.2`.

[3] T. Broström, J. Zhu, R. Robucci, M. Younis: *IoT Boot Integrity Measuring and Reporting*, SIGBED Rev. 15.5 (Nov. 2018), pp. 14–21,
doi: `https://doi.org/10.1145/3292384.3292387`.

[4] D. Chaum: *Blind Signatures for Untraceable Payments*, in: Advances in Cryptology, ed. by D. Chaum, R. L. Rivest, A. T. Sherman, Boston, MA: Springer US, 1983, pp. 199–203, isbn: 978-1-4757-0602-4.

[5] D. Kaplan, J. Powell, T. Woller: *AMD Memory Encryption*, tech. rep., `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`, Last visited: 13.10.2020, 2016.

[6] P. C. Kocher: *Timing attacks on implementations of Die-Hellman, RSA, DSS, and other systems*, in: Advances in Cryptology| Crypto, vol. 96, 1996, p. 104113.

[7] B. McGillion, T. Dettenborn, T. Nyman, N. Asokan: *Open-TEE – An Open Virtual Trusted Execution Environment*, in: TRUSTCOM '15, USA: IEEE Computer Society, 2015, pp. 400–407, isbn: 9781467379526, doi: `https://doi.org/10.1109/Trustcom.2015.400`.

[8] S. Pinto, T. Gomes, J. Pereira, J. Cabral, A. Tavares: *IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices*, IEEE Internet Computing 21.1 (2017), pp. 40–47.

[9] M. Schwarz, D. Gruss: *How Trusted Execution Environments Fuel Research on Microarchitectural Attacks*, IEEE Security Privacy 18.5 (2020), pp. 18–27.

[10] D. J. Sebastian, U. Agrawal, A. Tamimi, A. Hahn: *DER-TEE: Secure Distributed Energy Resource Operations Through Trusted Execution Environments*, IEEE Internet of Things Journal 6.4 (2019), pp. 6476–6486.

[11] C. Shepherd, R. N. Akram, K. Markantonakis: *EmLog: Tamper-Resistant System Logging for Constrained Devices with TEEs*, in: Information Security Theory and Practice, ed. by G. P. Hancke, E. Damiani, Cham: Springer International Publishing, 2018, pp. 75–92, isbn: 978-3-319-93524-9.

[12] C. Shepherd, R. N. Akram, K. Markantonakis: *Establishing Mutually Trusted Channels for Remote Sensing Devices with Trusted Execution Environments*, in: Proceedings of the 12th International Conference on Availability, Reliability and Security, ARES '17, Reggio Calabria, Italy: Association for Computing Machinery, 2017, isbn: 9781450352574, doi: `https://doi.org/10.1145/3098954.3098971`.

[13] C. Shepherd, R. N. Akram, K. Markantonakis: *Remote Credential Management with Mutual Attestation for Trusted Execution Environments*, in: Information Security Theory and Practice, ed. by O. Blazy, C. Y. Yeun, Cham: Springer International Publishing, 2019, pp. 157–173, isbn: 978-3-030-20074-9.

[14] M. Strasser, H. Stamer: *A Software-Based Trusted Platform Module Emulator*, in: Trusted Computing - Challenges and Applications, ed. by P. Lipp, A.-R. Sadeghi, K.-M. Koch, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–47, isbn: 978-3-540-68979-9.

[15] F. Zhang, H. Zhang: *SoK: A Study of Using Hardware-Assisted Isolated Execution Environments for Security*, in: Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP 2016, Seoul, Republic of Korea: Association for Computing Machinery, 2016, isbn: 9781450347693, doi: `https://doi.org/10.1145/2948618.2948621`.