KeVlar-Tz: A Secure Cache for Arm TrustZone

Oscar Benedito, Ricard Delgado-Gonzalo, Valerio Schiavoni

KEVLAR-TZ: a Secure Cache for ARM TRUSTZONE
(Practical Experience Report)

Oscar Benedito[1], Ricard Delgado-Gonzalo[1][0000−0002−7183−6257] and Valerio Schiavoni[2][0000−0003−1493−6603]

[1] CSEM, Neuchâtel, Switzerland, obo,rdg@csem.ch
[2] University of Neuchâtel, Switzerland, valerio.schiavoni@unine.ch

Abstract. Edge devices are increasingly in charge of storing privacy-sensitive data, in particular implantables, wearables, and nearables can potentially collect and process high-resolution vital signs 24/7. Storing and performing computations over such data in a privacy-preserving fashion is of paramount importance. We present KEVLAR-TZ, an application-level trusted cache designed to leverage ARM TRUSTZONE, a popular trusted execution environment available in consumer-grade devices. To facilitate the integration with existing systems and IoT devices and protocols, KEVLAR-TZ exposes a REST-based interface with connection endpoints inside the TRUSTZONE enclave. Furthermore, it exploits the on-device secure persistent storage to guarantee durability of data across reboots. We fully implemented KEVLAR-TZ on top of the OP-TEE framework, and experimentally evaluated its performance. Our results showcase performance trade-offs, for instance in terms of throughput and latency, for various workloads, and we believe our results can be useful for practitioners and in general developers of systems for TRUSTZONE. KEVLAR-TZ is available as open-source at https://github.com/mqttz/kevlar-tz/.

Keywords: caching · edge devices · TrustZone · TEE · OP-TEE

1 Introduction

Wearable and Internet-of-Things (IoT) devices are becoming increasingly pervasive in modern society. It is predicted that by the year 2025 there will be more than 600 million wearable devices deployed and connected worldwide [9], and according to Cisco up to 500 billion IoT devices by 2030 [4]. These devices continuously produce data from a wide range of sensor types: inertial sensors (*e.g.*, accelerometers, gyroscopes) [12], biopotential (*e.g.*, electrocardiography) [14], optical (*e.g.*, photopletysmography) [42], biochemical (*e.g.*, pH) [17], *etc*. Combinations of such sensors allow for the monitoring of the health statuses of the users, ranging from the user's physical activity [19] to the detection of cardiac abnormalities [21]. The nature of this data is intrinsically privacy-sensitive. Applications and system designers must protect it from malicious attackers, including those with physical access, from accessing and possibly unveiling them. Similarly, IoT devices are regularly used to monitor and record privacy-related data. Examples include motion sensors (*e.g.*, in the case of a smart-home deployment,

revealing for instance the presence of humans indoors [34]), power-consumption meters (*e.g.*, potentially revealing the habits of a household), weather sensors (*e.g.*, a key asset in farming used to decide on optimal irrigation levels [35]), *etc*. The vast majority of such applications deal with the insertion and retrieval of data from/to a dedicated, and preferably persistent, memory area. The mentioned operations are typically offered by key-value stores, *e.g.*, software libraries or services that allow to `put` and `get` values associated with unique identifiers (*i.e.*, the keys), for later retrieval, similar to a *caching* mechanism. Note that such libraries are vastly known in literature (*i.e.*, [13,27], *etc*.), extensively studied [24] and find usage in several and diverse application domains. Noteworthy, the result of confidential computations (*e.g.*, edge-based privacy-preserving machine-learning model training, just to name one) must also be stored and retrieved following the same access patterns. Hence, the content of such memory area must be shielded.

The need for stringent data privacy guarantees, such as the mentioned shielding, usually comes at the cost of computational overhead. This is the case of full homomorphic encryption (HE) [23], a purely software-based approach to compute and operate over encrypted data. However, recent work [25] has shown how state-of-the-art HE implementations [26] still result in orders of magnitude slowdown even for simple arithmetical operations, and major breakthroughs are yet to be seen for HE to become a viable solution.

The introduction and widespread adoption in the last few years of trusted execution environments (TEE) for consumer- and server-grade devices offers an opportunity to combine the need for privacy with the ones of viable performance. TEEs provide a hardware-supported mechanism to maintain the privacy and integrity of data while allowing for efficient and transparent protection from malicious attackers or compromised operating systems. Such protected areas are commonly referred to as *enclaves*, and they represent the main programming abstraction supported by the large majority of available TEEs. Notable examples include Intel®SGX [16], AMD Secure Encrypted Virtualization (SEV) [30] for server-grade as well as cloud-based deployments [6,5] and ARM TRUSTZONE [11,37] for more edge-centric scenarios, the focus of this work.

In this practical experience report paper, we present KEVLAR-TZ, an efficient trusted cache application for ARM TRUSTZONE with support for non-volatile secure storage. KEVLAR-TZ exposes an easy-to-use REST interface to facilitate the integration with existing systems, protocols, and third-party devices. The network connection endpoints are established within the TRUSTZONE enclave. Finally, KEVLAR-TZ is designed to exploit the secure storage implemented by some TRUSTZONE-enabled systems, allowing for secure data durability.

The main **contributions** of this work are twofold. First, we present the design and implementation of KEVLAR-TZ, a secure cache for ARM TRUSTZONE. Second, we describe in detail our implementation and evaluate it using real-world data, including a performance comparison with an emulator, showcasing the performance-tradeoffs that practitioners must face.

Roadmap. The rest of this paper is organized as follows. Section 2 provides relevant background material on TEEs, TRUSTZONE and trusted applications in general, including some of the underlying libraries and systems used in our evaluation. We present the

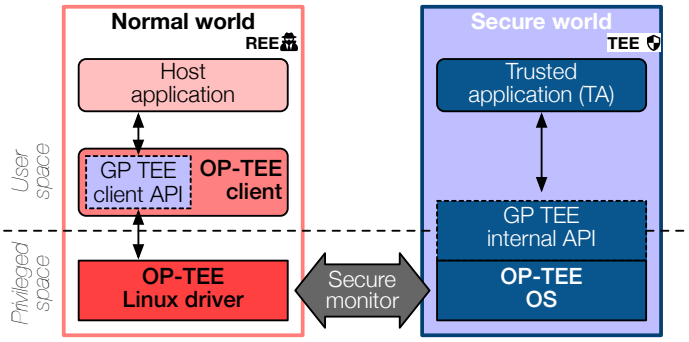

Fig. 1: Architecture of OP-TEE to realize trusted applications leveraging TRUSTZONE. GP: GlobalPlatform [7], a standardization effort for TEEs.

architecture of KEVLAR-TZ in Section 4, detailing some of its implementation details in Section 5. Section 6 presents our in-depth performance evaluation of the KEVLAR-TZ prototype, using micro- and macro-benchmarks as well as real-world data. We survey related work in Section 7, before concluding and devising future work in Section 8.

2 Background

Trusted Execution Environments (TEEs). A trusted execution environment is a hardware-protected part of the processor. Depending on the specific version and implementation, a TEE can guarantee confidentiality, integrity and protection against several types of attacks for code and data executed and processed within it. Currently, there exist several hardware-based technologies that enable physical isolation of different execution environments available in a wide range of CPUs, including ARM TRUSTZONE [1,37,11], Intel®SGX [16], AMD SEV [30], and RISC-V Keystone [31]. While we expect more TEE options to surface in the coming years, we focus on ARM TRUSTZONE in the reminder of the paper, highlighting its main features and programming framework.

TRUSTZONE. TRUSTZONE is a hardware feature implemented in ARM processors since 2004 [10]. It enables physical separation between two different execution environments: the trusted side (known as the TEE or *secure world*) and the untrusted side (known as the REE or *normal world*). The TRUSTZONE protects the integrity and confidentiality of the code run inside the *secure world* from an attacker with physical access to the device, a malicious kernel or a high-privileged software. Programs hosted inside the TRUSTZONE, known as Trusted Applications, can leverage additional TRUSTZONE functionalities such as secure persistent storage with the use of APIs.

OP-TEE. The Open Portable Trusted Execution Environment (OP-TEE) is an open source operating system with native support for the TRUSTZONE. OP-TEE implements two APIs compliant with the GlobalPlatform [7] specifications: the TEE Internal Core API [3], which is exposed to the Trusted Applications, and the TEE Client API [2], which defines how a client in the REE should communicate with the TEE. The TEE can run alongside a Linux-based operating system (such as a GNU/Linux distribution or AOSP) as the untrusted OS.

Trusted Application. Trusted Applications (TAs) run inside the *secure world*, making use of the TEE kernel to access system resources. TAs can act as a service for applications running on the *normal world* as well as for other TAs. When using OP-TEE, Trusted Applications are implemented in C and they can leverage the TEE Internal Core API implemented by OP-TEE, which offers several services, including trusted storage and cryptographic, time and arithmetical operations. KEVLAR-TZ is an application that runs on the TEE, so it is a Trusted Application. When using KEVLAR-TZ, we can do so from another TA (if we are running it on the TEE) or from a normal application running in the REE. The design of trusted applications for OP-TEE is depicted in Figure 1.

Trusted Persistent Storage. OP-TEE provides the Trusted Storage API for Data and Keys as part of the TEE Internal Core API [3]. This API can be used by Trusted Applications to access a secure storage which is only accessible to that particular TA and that is persistent between reboots. The data is stored encrypted and signed on the disk, to prevent it from being accessed or tampered with by any other application. The data can later be transparently accessed in cleartext by the TA. KEVLAR-TZ exploits this by saving the encryption keys using a public ID, which are the value and key (respectively) in the key-value storage. When an untrusted application needs to use an encryption key, it sends the key's ID and KEVLAR-TZ retrieves it.

MQTT & `mosquitto`. The Message Queuing Telemetry Transport (MQTT) is a lightweight, publish-subscribe network protocol, suited for communication in environments with few resources and low network bandwidth. MQTT has two types of entities: the broker and the clients. The clients can publish messages to a topic or subscribe to one of them, while the broker is a server that forwards each incoming message to all the subscribers of its topic. `mosquitto` is an open source implementation of the MQTT broker developed and maintained by the Eclipse Foundation, which also provides a C library for implementing MQTT clients, as well as one implementation of both a subscriber client and a publisher client.

MQTT-TZ. MQT-TZ [40] is a fork of `mosquitto`, a topic-based publish-subscribe framework for IoT. It allows brokers and the clients to leverage the TRUSTZONE TEE, by encrypting the messages sent on the network to prevent the broker from being able to read them, while maintaining the publish-subscribe pattern. Similarly it allows full decoupling between publishers and subscribers, shielding the subscriptions inside the TEE.

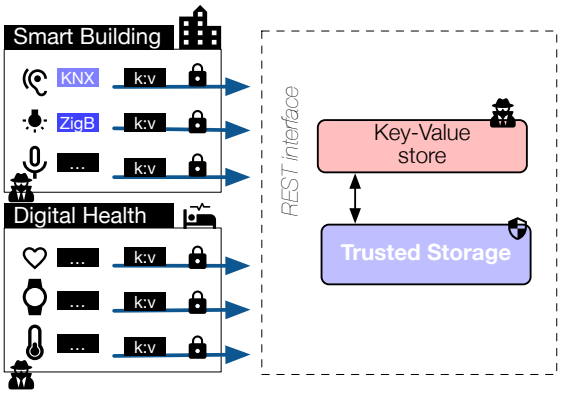

Fig. 2: Two possible application scenarios where a trusted key/value storage system is valuable. Clients (on the left-side) issue requests to store key-value pairs into the key-value store, which stores those into a trusted storage.

3 Motivating Scenarios

In this section, we describe our two main real-world scenarios behind KEVLAR-TZ, also depicted in Figure 2. The use-cases originate from two ongoing EU H2020 projects, in collaboration with industry-leading companies, which we details next.

3.1 Digital Health

The first scenario stems from the H2020 project MOORE4MEDICAL.[3] One of its objectives is to use wearable sensors and remote sensing technologies to reduce hospitalization, resulting in more comfort for the patient and less costly clinical trials in drug development. In this context, the monitoring of vital signs is increasingly off-loaded and out-sourced to third-party untrusted data centers. The main reason for such off-load is to exploit the economy of scale that comes with cloud computing. The flow of data is mainly generated from smart medical devices and sensors and it is composed of a mix of physiological signals (*e.g.*, electrocardiograms, photopletymograms) and vital signs (*e.g.*, heart rare, respiration rate, stress levels). The data streams are highly heterogeneous, since the physiological signals can reach high sampling rates (*e.g.*, Holter operate at 1 kHz) and the vital signs have typically much lower sampling rates ($\sim$1 Hz).

3.2 Smart Building Management

The second scenario stems from TABEDE[4], an EU H2020 project with the aim to integrate energy grid demand-response schemes into buildings through low-cost extenders for Building Management Systems or as a standalone system, which is independent from communication standards and integrates innovative flexibility algorithms. The

[3] https://moore4medical.eu/
[4] http://www.tabede.eu/

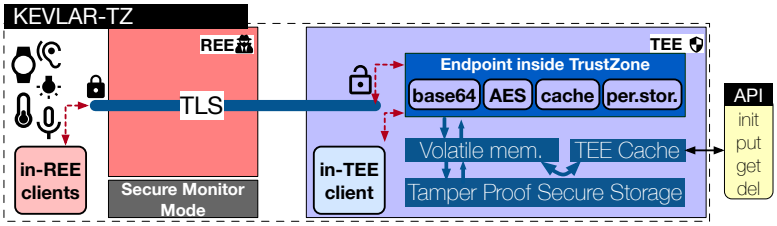

Fig. 3: Architecture of the KEVLAR-TZ TA.

flow of information relies on MQTT brokers deployed at the edge to minimize latency and limits the physical access from untrusted entities. However, it directly raises several privacy and security concerns. The flow of data is mainly generated from home appliances and sensors and it is composed of physical magnitudes such as electric current, temperature, or humidity. The data streams are generated at a slow frequency (<1 Hz) and are transfered via a large variety of communication protocols (*e.g.*, EnOcean [33], KNX [32], Zigbee [20]).

4 Architecture

KEVLAR-TZ implements a secure key-value storage with non-volatile entries (*i.e.*, available across reboots). To do so, we leverage OP-TEE's Trusted Storage API [8] to store keys to a secure persistent storage, while implementing a cache in volatile memory to minimize the number of requests made to the persistent storage. The key idea is to limit as much as possible operations (*i.e.*, `read`/`write`) involving the persistent storage, as they are considerably slower (see our evaluation in Section 6).

The architecture of KEVLAR-TZ is depicted in Figure 3. In the remainder of this section, we describe the designing principles behind its various components as well as their interaction. Finally, we detail the typical workflow of a single `write` operation, a keystone operation of KEVLAR-TZ.

Secure Persistent Storage. The persistent storage area is a dedicated hardware component that guarantees data durability, confidentiality and integrity. OP-TEE supports two modes for secure storage: *(1)* using the REE file-system (the default option), or *(2)* relying on a Replay Protected Memory Block (RPMB) partition of an eMMC device [38]. KEVLAR-TZ uses the REE file-system.

KEVLAR-TZ implements a wrapper around the Trusted Storage API to access directly to writing and reading operations, which otherwise requires the management of several internal OP-TEE components (omitted from the Figure 3 for the sake of clarity).

This wrapper exposes two functions:

- `read_ss(const char *key, char *value, uint32_t *value_sz)`: reads the data mapped to a given `key`, which is bound to the array pointed by `value`;
- `write_ss(const char *key, const char *value, uint32_t value_sz)`: writes the data in `value` mapped to the `key` into persistent storage.

Finally, we note that the available storage memory dedicated to this component is only limited by the underlying hardware.

Volatile memory – Cache. This component is the secure caching component of KEVLAR-TZ. Our design supports a few cache eviction policies (currently limited to Least Recently Used LRU and FIFO). The implementation uses structs inserted in a queue and a hash table. These are used to handle the key and value of each entry. The queue is used to remember the order of deletion of entries when new entries are to be added to a full cache; the hash table is used to access entries in average constant time. The cache is `write-through` [29], so that if the trusted application is stopped unexpectedly, no data is lost.

API for Trusted Applications. KEVLAR-TZ provides a very simple API for applications running inside the TEE with four operations:

- Initialize a cache with a given configuration consisting of cache size, hash output size and eviction policy;
- Delete a cache, freeing all space used in volatile memory. Objects in persistent storage are left untouched.
- Query a cache, to fetch the value associated to a given key. For instance, when using MQT-TZ [40], for a given ID, the cache will return the corresponding encryption key.
- Save a new key/value pair in volatile and persistent memory.

TCP interface for applications of REE. The TEE and REE are two different systems and, as such, programs can't communicate (*i.e.*, share data) between each other as if they where running on the same machine. However, KEVLAR-TZ can be useful as a secure cache service to an application running in the *normal world* (*e.g.*, in the MQT-TZ broker scenario [40]). To expose KEVLAR-TZ to the *normal world*, we designed and implemented a TCP interface, protected by TRUSTZONE, that allows to communicate KEVLAR-TZ with any other application reachable on the network.

The establishment of the TCP connection works as follows. First, an application in the REE opens a server TCP socket. Secondly, KEVLAR-TZ connects to such socket and waits (*i.e.*, blocks) for new messages. Once a new message is received, KEVLAR-TZ will execute the requested operation and return the desired value.

The workflow of a write. To conclude the description of the architecture, we take a step-by-step walkthrough for a `write` operation to insert a new key/value pair into KEVLAR-TZ. When an REE-based application needs to store a new key/value, it must first connect to KEVLAR-TZ via TCP, and pass over the content of the key/value pair. For the sake of simplicity, we assume those to be encoded using `base64`. Once received by the KEVLAR-TZ TA, they get base64-decoded, and saved to the persistent storage. The architecture allows to attach additional application-specific processing operations to the inserted key/value pairs, both before or after the value is retrieved. For instance, one might send a cipher that will be decrypted with one value and encrypted with another [40], securely changing the encryption key of a cipher without the REE ever getting ahold of any of them. This post-retrieve operations can be changed to any operation needed for the application that is using KEVLAR-TZ.

If an application in the *secure world* uses KEVLAR-TZ to store a new key, it can directly leverage the functions exposed by the KEVLAR-TZ API (`cache_save_object(Cache *cache, char *id, char *data)`), which takes the binary values and stores them to the persistent storage.

5 Implementation

This section describes some of the internal details and implementation choices of KEVLAR-TZ. The system itself is implemented in C, and consists of 791 LoC, released as open-source from https://github.com/mqttz/kevlar-tz/. We note that applications implemented using the OP-TEE framework are basically organized as two distinct components: the Host Application (HA) and the corresponding Trusted Application (TA). The host application runs on the *normal world* and initilizes and finalizes the TEE context using the TEE Client API. Moreover, the HA is in charge of invoking functions over the Trusted Application, and can do so multiple times, dividing the work between the *normal* and *secure world*. However, the TEE's volatile memory is lost between calls, hence KEVLAR-TZ's Host Application only invokes the TA once. The TA acts as a daemon that receives queries. We detail the TA components next.

5.1 KEVLAR-TZ Trusted Application

The KEVLAR-TZ trusted application is split into several modules. The implementation of a particular module is independent of the rest, to facilitate future evolutions of the code in a loosely coupled manner (*e.g.*, you can change the communication module to work with UDP instead of TCP, different symmetric encryption algorithm, *etc*.). The KEVLAR-TZ TA is composed of the following modules, which we evaluate individually and as a part of micro-benchmarks in Section 6.

Persistent storage module. This module implements the functions `read_ss` and `write_ss`, which read and write persistent storage, respectively. Our implementation follows the guidelines from the Linaro Security Working Group. [5]

Cache module. The KEVLAR-TZ cache module directly implements the proper cache API, namely: `init_cache`, `free_cache`, `cache_query`, and `cache_save_object`. In our implementation, the cache is made of nodes that are part of both a queue and a hash table, which enables accessing objects in constant time (on average). The cache module also interacts with the persistent storage (using the pertinent module). Any access to the key-value storage can be done through the cache, whether the value was stored on volatile memory or not.

AES module. Our prototype includes a symmetric cipher module on top of AES. It directly exposes two functions: `encrypt`, which encrypts data with a given key, and `reencrypt`, which given two keys and a cipher, decrypts it with one key and encrypts it with the other. Our implementation uses the Cryptographic Operation API implemented by OP-TEE to encrypt and decrypt the data. We extended it to support PCKS padding, *i.e.*, the default padding used by OpenSSL and MQT-TZ.

Base64 module. This module implements standard Base64 encoding/decoding operations (*i.e.*, `base64_encode`, `base64_decode`), as well as auxilary ones (*i.e.*, `base64_decode_length` to return the length an encoded string after the decoding). The encoding and decoding implementations leverage an open-source library.[6] This module is used to encode and decode data transmitted to other applications over the network

[5] https://github.com/linaro-swg/optee_examples/tree/master/secure_storage
[6] https://web.mit.edu/freebsd/head/contrib/wpa/src/utils/base64.c

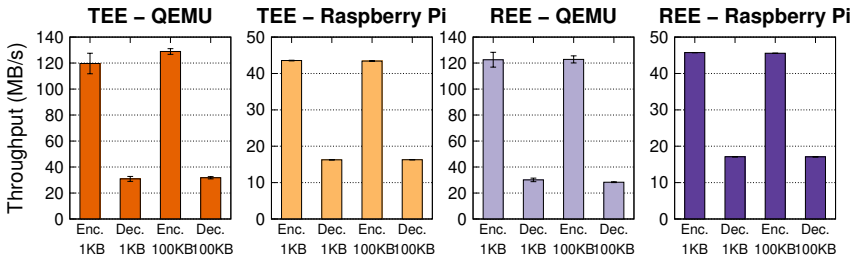

Fig. 4: Base64 encoding and decoding throughput for randomly generated data.

layer to simplify the parsing of data, in particular when dealing with multipart binary messages.

Trusted TCP module. KEVLAR-TZ uses TCP to communicate with untrusted applications. It exposes functions to initialize and close a connection (*i.e.*, `net_connect` and `net_disconnect`) as well as send and receive packets (*i.e.*, `net_send` and `net_receive`). The implementation uses the socket library that OP-TEE exposes, and while we use TCP, the code can easily be adapted to use other protocols without any changes on any other part of the application.

6 Evaluation

This section presents our experimental evaluation of KEVLAR-TZ using both micro- and macro-benchmarks. Our goal is to define the overheads of running KEVLAR-TZ to further assess whether the trade-offs to have a secure storage system are reasonable for a real-world scenario.

Evaluation settings. We deploy KEVLAR-TZ on a Raspberry Pi 3 Model B+ as well as on an emulated environment using QEMU version 8 [7] to test the application. QEMU is a tool that has been proven useful, despite its limitations, in validating design and implementation in ARM processors without having to deploy large (and potentially) expensive testbeds. The QEMU runtime is deployed on a Lenovo ThinkPad with Intel® Core™ i7-5600U CPU @ 2.60GHz. We rely on OP-TEE version 3.11.0.

6.1 Micro-benchmarks

We begin by micro-benchmarking two of the subcomponents of the KEVLAR-TZ TA, namely the Base64 encoder and the one in charge of cryptographic operations.

Base64 encoding and decoding. We measure the throughput of the base64 encoding and decoding operations. The measurements have been done by measuring the encoding and decoding of randomly generated data of 1KB and 100KB, both for the hardware deployment as well as under emulation. In both cases, the component is deployed in the TEE. We show average and standard deviation for each configuration, which is executed 200 times. Our results are shown in Figure 4. First, we observe the

[7] https://www.qemu.org

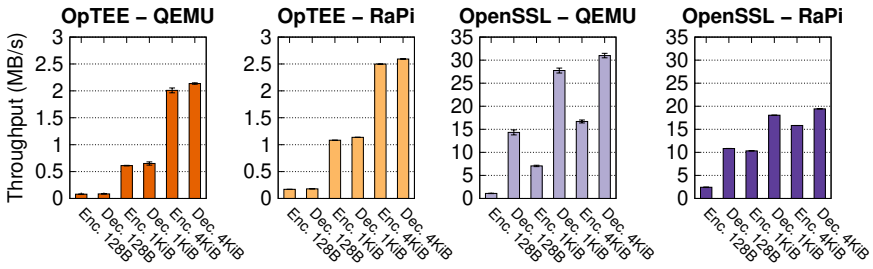

Fig. 5: Encryption and decryption throughput. We compare the built-in OP-TEE library for cryptographic operations against OpenSSL. For both cases we deploy them in TEE, and compare the throughput to encode and decode randomly generated data.

size of the data payload does not negatively affect the observed throughput, whereas we do observe differences between the emulated and hardware environment. For instance, encoding 100KB of data reaches 43 MB/s, while the QEMU is approximately 4× faster, reaching 130MB/s. Similar differences can be observed for smaller data and decoding. We also report the results obtained when executing the same operations in the REE. As we see, encoding and decoding throughputs are similar, due to the operations being GPU-intense, instead of memory-intense.

Cryptographic operations. Next, we measure the throughput of the cryptographic operations run by KEVLAR-TZ, *i.e.*, symmetric encryption and decryption. We generate random data of different sizes: 128B, 1kB and 4kB. Figure 5 depicts our results. We observe that encryption and decryption achieve similar encoding and decoding throughputs in each of the two environments (QEMU and the Raspberry Pi). For instance, we observe an average of 2 MB/s encrypting a payload of 4kB in QEMU, and a 25% improvement for the same test in hardware. Expectedly, decryption operations are slightly faster (by 6% on average). We compare our results with OpenSSL version 1.1.1f, running on the REE of both QEMU and Raspberry Pi. We observe that OpenSSL in the REE is much faster, especially for decryption operations which can be parallelized: this is expected, as OpenSSL optimizes the compiled binary for the underlying hardware. We leave as future work further investigation and porting of a (subset of) OpenSSL to run in the TEE.

TCP communication. The TPC sockets handle the communication between KEVLAR-TZ and untrusted applications. In this benchmark, we measure the throughput of our trusted TCP channels, whose endpoints terminate into the TRUSTZONE area. We measure the throughput for messages of different sizes: 1B, 245B (*i.e.*, 128B once encrypted and encoded in base64), and 757B (*i.e.*, 512B plus AES encryption and base64 encoding), and 1024B. We use these values since they represent a reasonable range of values found in real-world deployments. Figure 7 reports our results for the two testing environments. We observe that the throughput is significantly higher for larger amounts of data. Concerning the system used, we see that the Raspberry Pi is much faster than the emulated environment.

Cache module. The last in our series of micro-benchmarks focus on the throughput of the cache module itself. First, we fill up the persistent storage with 200 keys, using

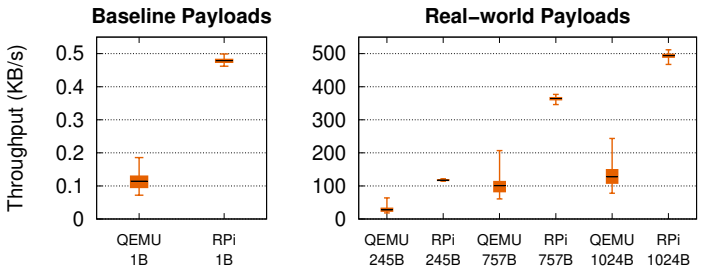

Fig. 6: Trusted TCP server: incoming throughput.

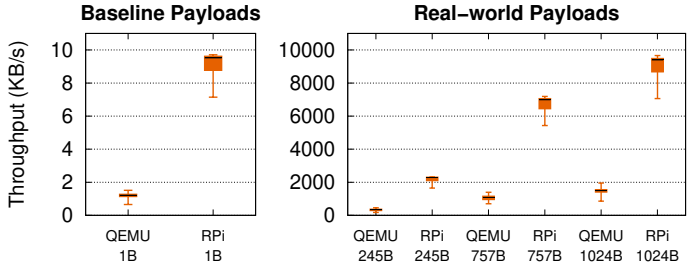

Fig. 7: TCP throughput REE to REE

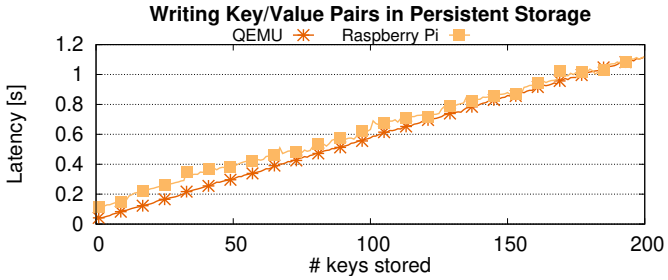

Fig. 8: Time to store a 32B encryption key (value) with a 12B ID (key) to persistent storage.

a payload of 32B. Figure 8 indicates that as the number of keys increase, there is a corresponding increase in the time to insert the key and value in the persistent storage.

Finally, we query the previously stored keys by issuing request queries for random keys to the cache module. Each cache request can result in a *hit* or *miss*. These results are shown in Figure 9. We can observe 5 orders of magnitude between the hits and miss throughputs, both in the case of emulated as well as hardware deployments.

6.2 Macro-benchmark: Digital Health

We conclude our evaluation by demonstrating the overall performance of KEVLAR-TZ. We setup the Digital Health scenario (Section 3.1), where heart-rate monitoring data streams are pushed toward KEVLAR-TZ, leaving the Smart Building use-case to future work. For the considered workload, we use a database obtained from CSEM's

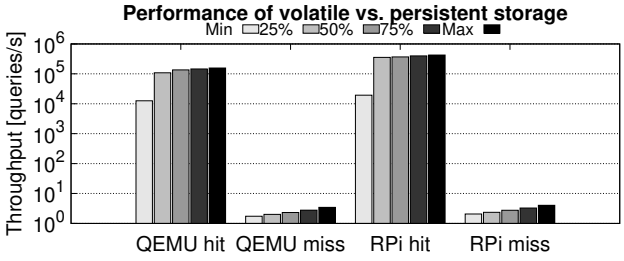

Fig. 9: Throughput of queries. We compare the difference between volatile memory (cache hits) and persistent storage (cache miss) on both hardware (Raspberry Pi) and emulated (QEMU) environments. The percentiles of the distribution are represented with shades of gray. From the brightest to the darkest: minimum, 25th, the 50th (median), the 75th and the maximum percentile.

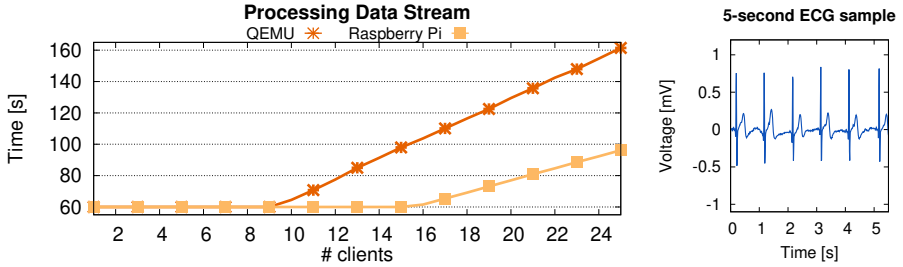

Fig. 10: Time to process a 60-second data stream from ECG sensors.

proprietary wrist-located sensors and chest-located dry electrodes [15]. In particular, cardiac data is obtained following a standardized protocol in which they perform a range of physical activities from sedentary to vigorous [18]. A 5-second sample of the ECG used is shown on Figure 10. In this scenario, the sensors inject 10 electrocardiogram data points every 93.4ms. Each data point embeds the following information: the time when it was taken and the voltage measured. Plot 10 compares the time taken to process 60 seconds of such streaming for different amounts of clients. We conclude that the Raspberry Pi takes approximately 0.064 seconds to process 1 second of streaming of 1 client, while the emulated environment takes 0.107 seconds for the same amount of data.

7 Related Work

The problem of executing software inside TEEs in general, and TRUSTZONE in particular, has attracted several research groups. CaSE [44] is a cache-assisted secure execution framework for the TRUSTZONE to defend against multiple attacks. Others [28,36,41] have implemented frameworks for TEEs to securely process data streams that could benefit from KEVLAR-TZ. However, while such projects have implemented full-fledged frameworks, KEVLAR-TZ provides a lean and resource-efficient cache with an easy-to-use API for applications that need fast access to persistent data.

A service like KEVLAR-TZ is available within MQT-TZ [40], a publish-subscribe framework optimized for IoT and TRUSTZONE deployments and backward compatible with `mosquitto`, a well-known MQTT messaging framework supporting TLS. In that context, a secure cache like the one developed in KEVLAR-TZ protects data against eavesdroppers or untrusted brokers. KEVLAR-TZ offers a more generic approach, including an API to use it from inside the TEE and a modular design to choose a specific cache eviction policy or some of its internal subcomponents. Recently [39,43], authors tried to hardening TEE applications against a broad set of attacks, including side-channels or against known weaknesses of the implementation language. While some of the countermeasures developed there could be beneficial for KEVLAR-TZ, we consider those out of scope. In [22], authors implement a cache to speed up operations on a secure Bitcoin wallet, while using the TRUSTZONE's persistent storage. While the focus is on the security of the private keys used to unlock the cryptocurrency wallet, the approach is similar to KEVLAR-TZ.

To the best of our knowledge, KEVLAR-TZ is the first application specifically designed to run on a TRUSTZONE and provide a lightweight cache to leverage the TRUSTZONE's persistent storage while maintaining a minimal read/write latency. KEVLAR-TZ implements a generic cache that can be easily embedded into other Trusted Applications or used as a secure storage for an untrusted application in the *normal world* without significantly increasing the trusted computing base.

8 Conclusion and Future Work

Motivated by the increasing attack surface of today's edge devices, KEVLAR-TZ addresses an integral part for storing and performing computations over the collected/-transmitted data in a privacy-preserving fashion. This becomes critical when the data is highly sensitive and personal, which is the case for nowadays medical implantables, wearables, and nearables. For instance, in scenarios where such IoT devices interact by means of publish/subscribe frameworks, as is typical in real-world deployments, protecting the brokers with a minimal increase in power consumption is necessary in order to preserve the ubiquity of such network of sensing devices.

We intend to extend KEVLAR-TZ along the following lines. First, we intend to extend the API exposed to the Trusted Application so that it is easier to implement new functions (similar to the already implemented reencryption. Second, standardizing the length of the messages going through TCP when communicating with untrusted applications, so that binary data can be sent and easily parsed, this will reduce the amount of bytes sent as well as eliminate the base64 dependency. Finally, we intend to compare KEVLAR-TZ to other TRUSTZONE cache implementations.

9 Acknowledgements

This work is supported in part by Moore4Medical, which has received funding within the Electronic Components and Systems for European Leadership Joint Undertaking (ECSEL JU) in collaboration with the European Union's H2020 framework Programme (H2020/2014-2020) and National Authorities, under grant agreement H2020-ECSEL-2019-IA-876190. Moreover, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 766733.

References

1. ARM TrustZone Developer. https://developer.arm.com/technologies/trustzone, visited on 2021-02-15
2. TEE Client API Specification v1.0 (GPD_SPE_007). https://globalplatform.org/specs-library/tee-client-api-specification/, visited on 2021-02-15
3. TEE Internal Core API Specification v1.2.1 (GPD_SPE_010). https://globalplatform.wpengine.com/specs-library/tee-internal-core-api-specification-v1-2/, visited on 2021-02-15
4. Digital Impact How Technology is Accelerating Global Problem Solving. https://www.cisco.com/c/dam/assets/csr/pdf/Digital-Impact-Playbook.pdf (2018)
5. AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/ (2021)
6. Confidential VM and Compute Engine. https://cloud.google.com/compute/confidential-vm/docs/about-cvm (2021)
7. Global platform. http://www.globalplatform.org. (2021)
8. OP-TEE Secure Storage API. https://optee.readthedocs.io/en/latest/architecture/secure_storage.html (2021)
9. Wearable computing devices market - growth, trends, covid-19 impact, and forecasts (2021 - 2026). https://www.researchandmarkets.com/reports/4787502/wearable-computing-devices-market-growth (2021)
10. Alves, T., Felton, D.: TrustZone: Integrated hardware and software security. ARM Information Quarterly **3**(4), 18–24 (2004)
11. Amacher, J., Schiavoni, V.: On the performance of ARM TrustZone. In: Proc. DAIS'19. pp. 133–151. Springer (2019)
12. Bennett, T.R., Wu, J., Kehtarnavaz, N., Jafari, R.: Inertial measurement unit-based wearable computers for assisted living applications: A signal processing perspective. IEEE Signal Processing Magazine **33**(2), 28–35 (2016)
13. Cao, Z., Dong, S., Vemuri, S., Du, D.H.C.: Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In: Proc. USENIX FAST 20. pp. 209–223. USENIX Association (2020)
14. Chaudhuri, S., Pawar, T.D., Duttagupta, S.: Ambulation analysis in wearable ECG. Springer (2009)
15. Chételat, O., Ferrario, D., Proença, M., Porchet, J.A., Falhi, A., Grossenbacher, O., Delgado-Gonzalo, R., Della Ricca, N., Sartori, C.: Clinical validation of LTMS-S: A wearable system for vital signs monitoring. In: Proc. IEEE EMBC'15. pp. 3125–3128 (2015)
16. Costan, V., Devadas, S.: IntelSGX explained. IACR Cryptol. ePrint Arch. **2016**(86), 1–118 (2016)
17. Coyle, S., Curto, V.F., Benito-Lopez, F., Florea, L., Diamond, D.: Wearable bio and chemical sensors. In: Wearable sensors, pp. 65–83. Elsevier (2014)
18. Delgado-Gonzalo, R., Renevey, P., Calvo, E.M., Solà, J., Lanting, C., Bertschi, M., Lemay, M.: Human ernergy expenditure models: Beyond state-of-the-art commercialized embedded algorithms. In: Proc. DHM '14. pp. 3–14 (2014)
19. Delgado-Gonzalo, R., Renevey, P., Lemkaddem, A., Lemay, M., Solà, J., Korhonen, I., Bertschi, M.: Seamless healthcare monitoring, chap. Physical Activity, pp. 413–455. Springer (2017)
20. Farahani, S.: ZigBee wireless networks and transceivers. Newnes (2011)
21. Faraone, A., Delgado-Gonzalo, R.: Convolutional-recurrent neural networks on low-power wearable platforms for cardiac arrhythmia detection. In: Proc. IEEE AICAS'20. pp. 153–157 (2020)

22. Gentilal, M., Martins, P., Sousa, L.: TrustZone-Backed Bitcoin Wallet. In: Proc. CS2'17. pp. 25—-28. CS2 '17 (2017)
23. Gentry, C., et al.: A fully homomorphic encryption scheme, vol. 20. Stanford university Stanford (2009)
24. Gokhale, S., Agrawal, N., Noonan, S., Ungureanu, C.: KVZone and the Search for a Write-Optimized Key-Value Store. In: HotStorage (2010)
25. Göttel, C., Pires, R., Rocha, I., Vaucher, S., Felber, P., Pasin, M., Schiavoni, V.: Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In: Proc. SRDS'18. pp. 133–142. IEEE (2018)
26. Halevi, S., Shoup, V.: Design and implementation of a homomorphic-encryption library. IBM Research (Manuscript) **6**(12-15), 8–36 (2013)
27. Han, J., Haihong, E., Le, G., Du, J.: Survey on NoSQL database. In: Proc. PerCom'11. pp. 363–366. IEEE (2011)
28. Havet, A., Pires, R., Felber, P., Pasin, M., Rouvoy, R., Schiavoni, V.: SecureStreams: A reactive middleware framework for secure data stream processing. In: Proc. ACM DEBS'17. pp. 124—133. DEBS '17, Association for Computing Machinery (2017)
29. Jouppi, N.P.: Cache write policies and performance. ACM SIGARCH Computer Architecture News **21**(2), 191–201 (1993)
30. Kaplan, D., Powell, J., Woller, T.: AMD memory encryption. White paper (2016)
31. Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., Song, D.: Keystone: An open framework for architecting trusted execution environments. In: Proc. EuroSys'20. pp. 1–16 (2020)
32. Lee, W.S., Hong, S.H.: Implementation of a KNX-ZigBee gateway for home automation. In: Proc. IEEE ICCE'09. pp. 545–549. ISCE'09, IEEE (2009)
33. Li, Y., Hong, S.H.: BACnet–EnOcean smart grid gateway and its application to demand response in buildings. Energy and Buildings **78**, 183–191 (2014)
34. Lin, H., Bergmann, N.W.: IoT privacy and security challenges for smart home environments. Information **7**(3), 44 (2016)
35. Padalalu, P., Mahajan, S., Dabir, K., Mitkar, S., Javale, D.: Smart water dripping system for agriculture/farming. In: Proc. I2CT'17. pp. 659–662. IEEE (2017)
36. Park, H., Zhai, S., Lu, L., Lin, F.X.: StreamBox-TZ: Secure stream analytics at the edge with TrustZone. In: Proc. USENIX ATC'19. pp. 537–554. USENIX Association (2019)
37. Pinto, S., Santos, N.: Demystifying Arm TrustZone: A comprehensive survey. ACM Computing Surveys (CSUR) **51**(6), 1–36 (2019)
38. Reddy, A.K., Paramasivam, P., Vemula, P.B.: Mobile secure data protection using eMMC RPMB partition. In: Proc. CoCoNet'15. pp. 946–950. IEEE (2015)
39. Sasaki, T., Tomita, K., Hayaki, Y., Liew, S.P., Yamagaki, N.: Secure IoT device architecture using TrustZone. In: Proc. IEEE SECON'20. pp. 1–6 (2020)
40. Segarra, C., Delgado-Gonzalo, R., Schiavoni, V.: MQT-TZ: Hardening IoT brokers using ARM TrustZone. In: Proc. SRDS'20 (2020)
41. Segarra, C., Delgado-Gonzalo, R., Lemay, M., Aublin, P.L., Pietzuch, P., Schiavoni, V.: Using trusted execution environments for secure stream processing of medical data. In: Proc. DAIS'19. pp. 91–107. Springer International Publishing (2019)
42. Tamura, T., Maeda, Y., Sekine, M., Yoshida, M.: Wearable photoplethysmographic sensors—past and present. Electronics **3**(2), 282–302 (2014)
43. Wan, S., Sun, M., Sun, K., Zhang, N., He, X.: RusTEE: Developing memory-safe ARM TrustZone applications. In: Proc. ACSAC'20. pp. 442—453. ACSAC '20, Association for Computing Machinery (2020)
44. Zhang, N., Sun, K., Lou, W., Hou, Y.T.: CaSE: Cache-assisted secure execution on ARM processors. In: Proc. IEEE SP'16. pp. 72–90 (2016)