MQT-TZ: Hardening IoT Brokers Using ARM TrustZone

(Practical Experience Report)

Carlos Segarra*†, Ricard Delgado-Gonzalo†, Valerio Schiavoni*
†*CSEM*, ricard.delgado@csem.ch
*Université de Neuchâtel, Switzerland, first.last@unine.ch

***Abstract*—The publish-subscribe paradigm is an efficient communication scheme with strong decoupling between the nodes, that is especially fit for large-scale deployments. It adapts natively to very dynamic settings and it is used in a diversity of real-world scenarios, including finance, smart cities, medical environments, or IoT sensors. Several of the mentioned application scenarios require increasingly stringent security guarantees due to the sensitive nature of the exchanged messages as well as the privacy demands of the clients/stakeholders/receivers. MQTT is a lightweight topic-based publish-subscribe protocol popular in edge and IoT settings, a *de-facto* standard widely adopted nowadays by the industry and researchers. However, MQTT brokers must process data in clear, hence exposing a large attack surface. This paper presents MQT-TZ, a secure MQTT broker leveraging ARM TRUSTZONE, a trusted execution environment (TEE) commonly found even on inexpensive devices largely available on the market (such as Raspberry Pi units). We define a mutual TLS-based handshake and a two-layer encryption for end-to-end security using the TEE as a trusted proxy. The experimental evaluation of our fully implemented prototype with micro-, macro-benchmarks, as well as with real-world industrial workloads from a MedTech use-case, highlights several trade-offs using TRUSTZONE TEE. We report several lessons learned while building and evaluating our system. We release MQT-TZ as open-source.**

I. INTRODUCTION

The Internet of Things (IoT) is an increasingly popular environment to deploy all kind of data sensors, gather the produced data, and process it. Examples include live heart-rate data [48], smart-grids [31], or infrastructure management systems [33]. The scale of IoT deployments is expected to grow exponentially in the next decade, with each individual to own and control several connected *things* [55]. Efficient communication between the things is hence of paramount importance.

One scalable and flexible communication pattern, commonly adopted by IoT deployers, is the publish-subscribe paradigm. Specifically, a prominent choice in pub-sub IoT contexts is the Message Queuing Telemetry Transport (MQTT) [28], a topic-based [13] publish-subscribe protocol [23] designed for environments with limited memory and reduced network bandwidth. In a nutshell, a client publishes data to a *topic* (*e.g.*, soccer, art, *etc*), while a set of *brokers* forward it to the nodes subscribed to that topic. Some deployments, including the ones that motivated our work (§III) operate in sensitive environments, for which the topics, the subscriptions, or the messages being routed require high security guarantees. Most MQTT implementations support TLS [21] for transport-level security in the client-broker link,

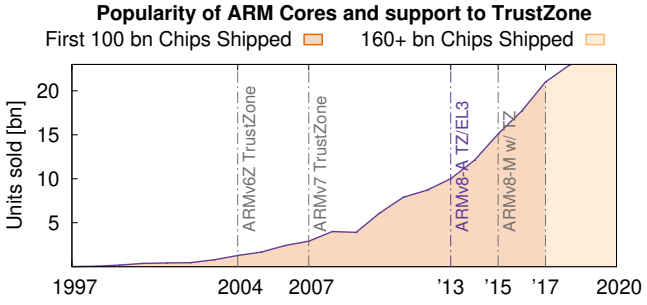

Fig. 1. Sales of chips containing ARM cores since 1997. We use filled curves with the cumulative value, and highlight the years when earlier TRUSTZONE versions were introduced. The current MQT-TZ can be deployed on ARM-v8-A chips, introduced in 2013.

preventing malicious actors from spoofing application data. However, the brokers still expose a great attack surface [53]. In particular, a powerful attacker with physical access to the broker node can intercept and tamper with all inbound and outbound traffic.

While software-based solutions could partially mitigate the risks of such attacks, for instance exploiting homomorphic encryption (HE) [56], the overhead that such solutions introduce in the systems is still unpractical, *i.e.* nowadays still several orders of magnitude slower [26].

Trusted execution environments (TEEs) are hardware-based security mechanisms that shield code and data on compromised systems. Examples include Intel Software Guard Extensions (SGX) [18], AMD Secure Encrypted Virtualization (SEV) [29] for server-grade processors, and ARM TRUSTZONE [6, 38, 41] for edge-based processors.

ARM [7] is a IoT leading manufacturer of IoT, from embedded devices to cloud appliances, as well as cloud-at-the-edge solutions. It was estimated [51] that by 2035 ARM will reach a trillion cumulative IoT devices shipped, with yearly sales of chips in the hundreds of billions, and trillion dollars annual spendings [9]. The adoption of ARM chips in the consumer market continues to grow (Figure 1). The support of ARM TRUSTZONE across all their processors makes security become a commodity rather than an additional feature.

TRUSTZONE is an edge-based TEE that enables system-wide hardware isolation against privileged processes or even malicious operating systems (*i.e.*, compromised kernel or kernel modules), without the overhead of software-oriented solutions, such as the previously mentioned HE. It is widely

This is the authors preprint version of the camera-ready article. The definitive version is published in the proceedings of the 2020 39th International Symposium on Reliable Distributed Systems (SRDS 2020).

available on billions of consumer-grade devices (see Figure 1), most of the time for IoT applications. For instance, the Raspberry Pi 3 Model B/B+ is a low-cost unit largely available in the consumer market. It embeds a Cortex-A53 ARM CPU core with native support to TRUSTZONE. Additionally, TRUSTZONE is easily accessible for prototyping via virtualization tools such as QEMU [14]. These reasons consolidate ARM TRUSTZONE's position to be the ideal tool to enhance the security guarantees of existing IoT pipelines which, most likely, already run on TRUSTZONE-compatible hardware.

This practical experience report presents the motivation, design, implementation, evaluation, and the lessons learned while building MQT-TZ, a secure edge-based publish/subscribe middleware leveraging MQTT and ARM TRUSTZONE protecting IoT systems against a variety of attacks. The MQT-TZ broker exploits TRUSTZONE's tamper-proof secure storage to store clients' keys (*i.e.*, publishers and subscribers) upon a successful handshake phase. Authenticating the data publisher is not only beneficial for a trustworthy end-to-end communication, but could also be used as a digital signature when connecting a storage back-end to our secure broker. Additionally, the re-encryption of the data—decrypting with the publisher key and encrypting with the subscriber's one—happens within the memory-protected TRUSTZONE, protecting against memory-dumps. As detailed later (§V-B), we built MQT-TZ's messaging broker on top of `mosquitto`, the standard *de facto* MQTT implementation.

In summary, our contributions are as follows:

- after describing our motivating scenarios from real-world settings (§III), we generalize and describe the deployment scenarios (see §IV) for which a secure edge-based pub/sub middleware is meaningful;
- we describe how we protect these industrial scenarios against a powerful attacker with the available hardware, minimal additional software, and no changes to the application code running at the edge;
- we provide insights regarding our open-source implementation of such design. In particular, we describe a novel caching mechanism that combines TRUSTZONE trusted application memory and persistent storage;
- we provide an in-depth evaluation of our system with real-world workloads, with the intent to highlight the performance trade-offs of MQT-TZ.

The transparency with regard to the deployer, both in terms of hardware and application compatibility, whilst adding protection for a variety of security vulnerabilities is, to the best of our knowledge, novel in the IoT/edge-processing field, and a main novelty of MQT-TZ itself.

The rest of the work is organized as follows. We introduce some preliminary concepts on MQTT and TRUSTZONE in §II, describe our motivating scenarios in §III, and the corresponding generalized deployment settings for which MQT-TZ is most relevant in §IV. The architecture and implementation details in described in §V. We present our experimental results in §VI. Finally, we cover related work (§VII) and our lessons learned (§VIII) before presenting our future work and concluding in §IX.

II. BACKGROUND

This section presents the technological building blocks that we leverage in MQT-TZ. Namely, we introduce MQTT, `mosquitto`, as well as OP-TEE, our framework and runtime of choice. Finally, we describe TRUSTZONE in a nutshell, the TEE environment available for ARM processors, to which OP-TEE offers native access.

MQTT & mosquitto. The Message Queuing Telemetry Transport protocol (MQTT) is a lightweight client and server topic-based publish/subscribe messaging transport protocol [28, 60]. It is specially suited for constrained environments (*e.g.*, IoT) where memory and bandwidth are very scarce resources. In the protocol, a client publishes a message to a topic. Dedicated processes (*i.e.*, the brokers) forward it to every subscriber for that same topic. The protocol supports decentralized deployments, in which brokers are organized in a layout and messages are routed and forwarded along the established routing tree. The *de facto* standard open-source `C`-based implementation of MQTT is `mosquitto`, actively maintained by the Eclipse Foundation [54]. We implement MQT-TZ atop `mosquitto` and introduces additional security guarantees leveraging ARM TRUSTZONE, as described next.

TRUSTZONE & OP-TEE. TRUSTZONE [1] is a feature implemented in ARM processors since Arm1176JZ-S (2004). It implements a trusted execution environment that is primarily used to guarantee system-wide hardware isolation. Data and code are shielded from compromised environments. The TRUSTZONE architecture physically separates the device in two distinct execution environments, *i.e.*, the trusted side (TEE - secure world) and the untrusted side, *i.e.*, Rich Execution Environment REE or normal world [6]. TRUSTZONE shields against an attacker with physical access to the device, as well as higher-privileged software or malicious kernels running in the REE. The former hosts the so-called Trusted Applications (TAs). Trusted applications can leverage additional TRUSTZONE-only services, such as tamper-proof persistent storage via specialized APIs. The integrity of a TRUSTZONE-enabled device can be guaranteed by a secure boot mechanism.[1] Its root of trust builds on Hardware Unique Keys (HUK) embedded in the processor during manufacturing. A common assumption is that only trustworthy TAs are deployed inside the secure world. The Open Portable Trusted Execution Environment (OP-TEE) [36] is an open-source runtime for TEE applications sponsored by the Linaro Foundation [4] with native support for TRUSTZONE. It is designed to run together with a non-secure Linux kernel in the REE. Finally, OP-TEE is compliant with the GlobalPlatform's specifications [25].

III. MOTIVATING SCENARIOS

In this section, we describe our two main real-world scenarios behind MQT-TZ, both depicted in Figure 2. The use-cases

[1] While beneficial, we did not leverage secure boot to build MQT-TZ since our Raspberry Pi 3 Model B devices do not support it.

This is the authors preprint version of the camera-ready article. The definitive version is published in the proceedings of the 2020 39th International Symposium on Reliable Distributed Systems (SRDS 2020).

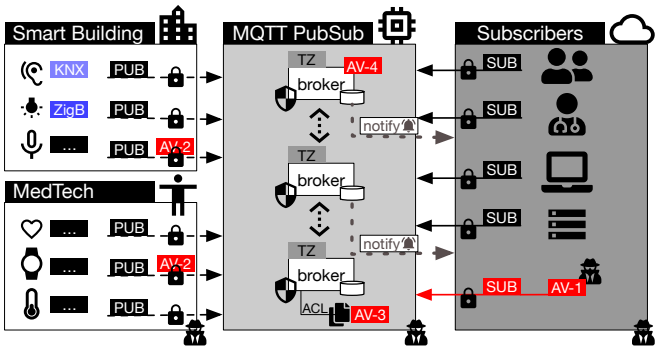

Fig. 2. Our two motivating scenarios: smart building management and vital signs monitoring (MedTech). The figure shows the interaction of the various actors with the MQTT publish-subscribe middleware as well where the identified attack vectors can be launched (§IV).

originate from two on-going research projects, in collaboration with industry-leading companies. The security vulnerabilities exposed in the following and that we tackle in the reminder are both realistic, and believed to be currently exploitable at the moment. In particular, the security and privacy concerns present in both cases are similar and can be attributed to:

- the exposure of the MQTT broker to attackers at the edge, both from attackers with physical access (on the field) to the devices, or by means of privileged malicious software running in the same node;
- the lack of authentication and authorization in the communication channel;
- the lack of transport and application layer protection;

A. Smart Building Management

The first scenario stems from TABEDE[2], 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 of communication standards and integrate innovative flexibility algorithms. The project targets the control of building mechanical and electrical equipment and services such as air conditioning, ventilation, and security systems.

From the architectural perspective, the overall system can be summarized on three key parts: *(1)* several in-house smart sensors implementing a large variety of communication protocols (*e.g.* EnOcean [34], KNX [32], Zigbee [24]); *(2)* a back-end managed by a third-party (*e.g.*, an untrusted cloud provider) where the data is streamed and stored in real time; and *(3)* a web-based front-end that visualizes and manages the network of in-house sensors.

The 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. In particular, the energy readings of an ensemble of buildings represent vital information required for the energy-balancing load algorithms that manage the internal power grid. If the readings are tampered with, even by a small amount, the grid could suffer local overloads. This method could be used to deliberately attack a building by overloading its grid (or even worse, if the attack is launched at a national scale [2]). Finally, in terms of privacy concerns, energy consumption reveals details about the habits and behaviors of individuals inside their private homes. Therefore, smart metering has to compromise between detailed energy metering and privacy protection [19, 39]. This use-case helps us identifying the following open questions:

***Q1**: What if an attacker impersonates the power meter and sends erroneous power readings to the grid?*

***Q2**: What if an attacker compromise the smart-light or smart-lock communication with the* MQTT *broker?*

B. MedTech for Vital Sings Monitoring

In the context of medical technologies (MedTech), the monitoring of vital signs is increasingly off-loaded and outsourced to third-party untrusted data centers. The main reason is to exploit the economy of scale that comes with cloud computing. However, recent data protection regulations (*e.g.*, GDPR [58]) have stressed the importance of ownership of the data and limited the scope of its use. Much research has been recently devoted to deal with such restrictions and on how to reconcile distributed systems with such legislation frameworks [16]. For instance, [49] showed that it is possible to deploy real time-processing of heart-rate variability (HRV, linked to physiological and mental stress [17, 52]) by exploiting Intel SGX TEE enclaves with reasonable overheads. However, the platform must deal with a large diversity of physiological and behavioral signals originating from wearable sensors [20, 46] such as electrocardiograms (ECG), electroencephalograms (EEG), or photoplethysmographams (PPG).

In this scenario, the flow of information is typically mediated by MQTT brokers. More precisely, ECGs and other key signal features are sent to remote MQTT brokers for further processing like HRV monitoring and arrhythmia detection. Locating them at the edge of the network, rather than the cloud, guarantees that data ownership is not transferred, hence easing compliance with data protection regulations. If a health anomaly is detected, an alarm is relayed through an MQTT topic to which a particular doctor or emergency services is subscribed. This architecture raises similar privacy and security concerns as the smart building scenario (§III-A) since, vital signs, and in particular HRV signals, are very sensitive data as well.

Moreover, as our healthcare systems quickly transition towards personalized medicine [27], an erroneous diagnosis generated by tampered health data could result in life-threatening situations. On the privacy side, read access to this health data can be used by third parties for user profiling, which may be used by health insurance companies to raise premiums from the leaked information.

***Q3**: What if an attacker deliberately push data to one patient's HRV topic, invalidating all the further monitoring?*

[2]https://www.tabede.eu/

This is the authors preprint version of the camera-ready article. The definitive version is published in the proceedings of the 2020 39th International Symposium on Reliable Distributed Systems (SRDS 2020).

***Q4**: What if an attacker compromise the medical broker to tamper packets routed to the emergency topic?*

IV. Deployment Scenario & Threat Model

From the described use-cases, we characterize our deployment scenarios used to design MQT-TZ. This section also describes the targeted threat model, as well as to clearly identify various attack vectors (*AV*) form which MQT-TZ shields against.

Deployment Scenario. We envision a deployment scenario consisted of a large set of low-powered, memory-constrained client and server nodes.

Client nodes continuously publish live monitoring data in a streaming fashion. These nodes can typically sustain a steady network throughput of hundreds of bytes per second. For instance, a publisher streaming an ECG will send at most 350 Bytes/s, and a full fleet of publishers (*e.g.*, a floor in a hospital) will amount to around 3-5 kBytes per second (see §VI).

Server nodes in the system are placed at the edge. This choice maximizes responsiveness (minimizing latency), reduces the attack surface, and avoids the transferring of data ownership. They receive such data to process, for instance performing aggregation, averaging, or detecting statistical deviations. Increasingly common in IoT deployments—given its reduced costs, high availability in the market, popularity across developers, and hardware features—we deploy our brokers over Raspberry Pi 3 Model B units (see §VI-C). Conveniently, this device embeds an ARM Cortex-A processor with native support for TRUSTZONE. The deployment scenario includes storage services in charge of collecting data to be processed offline at a later time.

Threat Model. Figure 2 depicts the potential threats that we consider in our deployment scenario, including the several attack vectors at disposal of an attacker. Note that these are not flaws in the MQTT protocol *per se* as they may be out of scope (*e.g.*, transport layer security). We highlight flaws in the way these tools and protocols are used in industrial settings.

First, most IoT settings leveraging MQTT bypass client or broker authentication (*AV-1*). As a consequence, unauthorized parties can publish data to the brokers as long as their address is known. Similarly, attackers might try to impersonate brokers.

Second, they do not encrypt data packets before transmission. As a consequence, an attacker with the ability to intercept or spoof the client-broker link would gain access and tamper all the information processed by the broker (*AV-2*).

Third, edge-based pub/sub middleware usually lacks any mechanism of access control, *e.g.*, the topics are public. An attacker with knowledge of the publish/subscribe topics could inject carefully crafted information while receiving potentially sensitive data sent by the clients (*AV-3*). By leveraging access control mechanisms, we can also revoke access to byzantine nodes, hence providing a simple defense mechanism against replay attacks towards the broker.

Lastly, the `mosquitto` MQTT brokers run by default as non-privileged processes. This exposes brokers to higher privileged processes running in the same node (*e.g.*, in the form of malware as these brokers tend to be connected to the internet), malicious users (*e.g.*, with SSH access to the node), or even malicious operating systems loaded by an advanced attacker at boot time. For instance, a high-privilege process running on the same broker machine could intercept all its incoming and outgoing traffic, tampering with the data it processes. Note also that even when the client-broker link is encrypted (*e.g.*, via TLS channels), processing data in clear at the broker constitutes a privacy and security risk (*AV-4*). To protect against an attacker patching our broker, or rebooting with a different implementation which could trick the untrusted code to authenticate malicious clients, we measure the REE code base at secure-boot time.

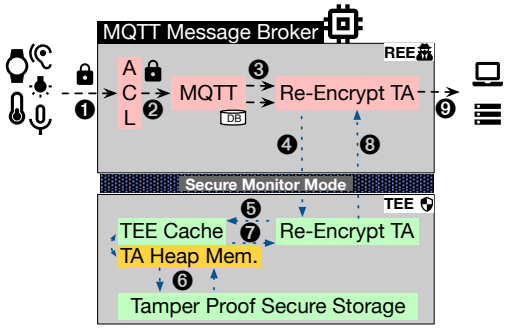

Fig. 3. MQT-TZ architecture and flow of operations.

When compared to lightweight cryptographic-based schemes implementing device authentication and access control mechanisms, MQT-TZ offers the additional in-broker security guarantees which stem from the usage TRUSTZONE.

We consider a powerful attacker with administrative rights as well as physical access to the broker node, and with the ability to exploit any of the listed attack vectors. However, we assume the client to not be compromised. While we are aware of recent TRUSTZONE side-channel attacks [45, 47], we consider those out-of-the-scope of this work. In particular, we assume that the client's cryptographic credentials (see further details in §V) cannot be extracted.

V. The MQT-TZ System

This section presents the architecture and the interaction of the various MQT-TZ components, providing additional implementation details, as well as explaining how the open questions given in §III are addressed.

A. Architecture & Component Description

The architecture of MQT-TZ is presented in Figure 3. We extend the `mosquitto` MQTT broker to ARM TRUSTZONE. Specifically, MQT-TZ's broker adds an encryption layer in MQTT's payload using client-specific keys stored inside ARM's secure storage [37]. Doing so, application data is only processed inside the TEE, where it gets re-encrypted. For the additional key-provisioning to address AV1 and AV2, we redefine the client authentication in the mutual TLS handshake to prevent the REE from gaining access to clients' keys. We also leverage Access Control Lists (ACLs) for fine-grained

This is the authors preprint version of the camera-ready article. The definitive version is published in the proceedings of the 2020 39th International Symposium on Reliable Distributed Systems (SRDS 2020).

control over the users and their topics. We now detail the various architecture components and their mutual interaction.

Note that MQT-TZ does not modify nor redefine TLS or the vanilla MQTT protocol. We leverage—and modify when needed—existing implementations of these protocols to shield our deployment scenario from possible attacks (§IV), with the minimum performance penalty, as shown next.

Two-Step Handshake. MQT-TZ defines and uses a two-step handshake that realizes broker and client authentication with end-to-end encryption from the client to the TEE. The handshake protocol requires minimal pre-provisioned cryptographic material.

A1: mutual authentication prevents from either broker or client impersonation.

The broker authentication (Figure 4, top) relies on TLS handshake, supported by default in `mosquitto`, while the corresponding client step (Figure 4, bottom) uses MQTT. We choose the latter over TLS's client side authentication to ensure that client's data is only processed in clear inside the TEE. Alternatively, we would need to install a full TLS endpoint inside TRUSTZONE, leading to a very large attack surface as code in the TEE *needs* to be trusted.

First, the client publishes its symmetric key, encrypted with the broker's TEE public key, to a dedicated write-only topic. This TEE key pair is generated at device start-up time (secure boot) and derived from a Hardware Unique Key (HUK). As a consequence, the private key never leaves the TRUSTZONE. Note that secure boot and HUKs are device-specific, hence the exact mechanism depends on the system on which MQT-TZ is deployed. The encrypted payload is then securely transferred to the TRUSTZONE TEE and decrypted. The client's key is stored in the secure storage (`SS` in Figure 4, bottom). An `ACK` reply is encrypted with this same symmetric key and sent back to the REE, which can forward to the client to finalize the handshake.

A2: TLS protects the communication link and prevents malicious packet interception and man-in-the-middle attacks.

Layered Encryption & Access Control Mechanisms. After the handshake, MQT-TZ uses a two-layer encryption mechanism. First, the client-broker link is protected by TLS within `mosquitto`. Second, MQTT's payload is encrypted using the clients' symmetric key. Then, data is re-encrypted in the TEE (as detailed next) and sent to its destination over a `mosquitto`-TLS channel. Doing so, we achieve end-to-end security relying on TRUSTZONE as a secure proxy. We also leverage Access Control Lists to limit the clients able to interact with the broker. These are currently stored in the REE, but if clients are defined in advance, its contents could be measured during boot and stored also in `SS`, preventing an attacker from tampering with the lists.

A3: ACLs prevent unauthorized entities to inject or subscribe to sensitive topics, and enable revoking access to clients controlled by attackers.

Payload Re-encryption. We link `mosquitto` with a trusted application that transfers the encrypted data to TRUSTZONE's secure world, where it retrieves the origin and destination keys from secure storage, and re-encrypts the information. Currently, topic subscription lists and MQTT metadata are stored in a dedicated database (MQTT DB) in the REE. We plan on shadowing these structures to keep them in the TEE since information like subscription patterns, subscriber distribution, or topic filtering can be used as a side-channel to leak sensitive data.

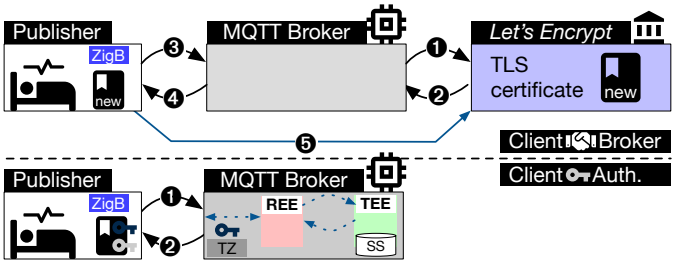

Fig. 4. MQT-TZ: broker (above) and client (below) authentication.

*A4: The re-encryption of information **inside** TRUSTZONE prevents a physical attacker or privileged process from spoofing sensitive information.*

LRU Cache in the TEE. Our evaluation (§VI) shows that retrieving the keys from secure storage is the most expensive operation when re-encrypting application data in the TEE. To mitigate this behavior, MQT-TZ embeds a lightweight LRU cache in the TEE that keeps the most recently used keys in the TA's heap memory, and evicts the least used ones (LRU policy) to persistent secure storage. Access to the heap's content is also hardware-protected by TRUSTZONE.

Dataflow. In a nutshell, data in MQT-TZ flows as follows. Data travels two-fold encrypted from the client to the broker (Fig.3-❶). Once the client access is confirmed (Fig.3-❷), the subscribers for the given topic are retrieved and the payload forwarded (Fig.3-❸). Then, encrypted data is transferred to the TEE (Fig.3-❹). The origin and destination client keys are retrieved (❺-❼). The payload is re-encrypted and sent back to the REE (Fig.3-❽) and to the subscriber (Fig.3-❾).

B. Implementation Details

MQT-TZ's broker is implemented in `C`. The current version of MQT-TZ adds 400 SLOC to `mosquitto` version 1.6.3 and the TA amounts to 1204 SLOC. The MQT-TZ TA relies on OP-TEE, version 3.5.0. The MQT-TZ prototype will be available from https://github.com/mqttz.

Client and Server Authentication. The server-side authentication is done through vanilla TLS. We deploy MQT-TZ's secure broker in a device with a static IP address. Then, we bound the address to a domain name and use a certificate. We rely on *Let's Encrypt* (https://letsencrypt.org/) to get one and to authenticate the broker. The client-side authentication uses MQTT as communication layer, and `openssl` (v1.1.1a) for cryptographic tools. The integration with `mosquitto` exploits custom callbacks for each packet processing. In addition, we use MQTT (v5.0) Request/Response (RR) features for the

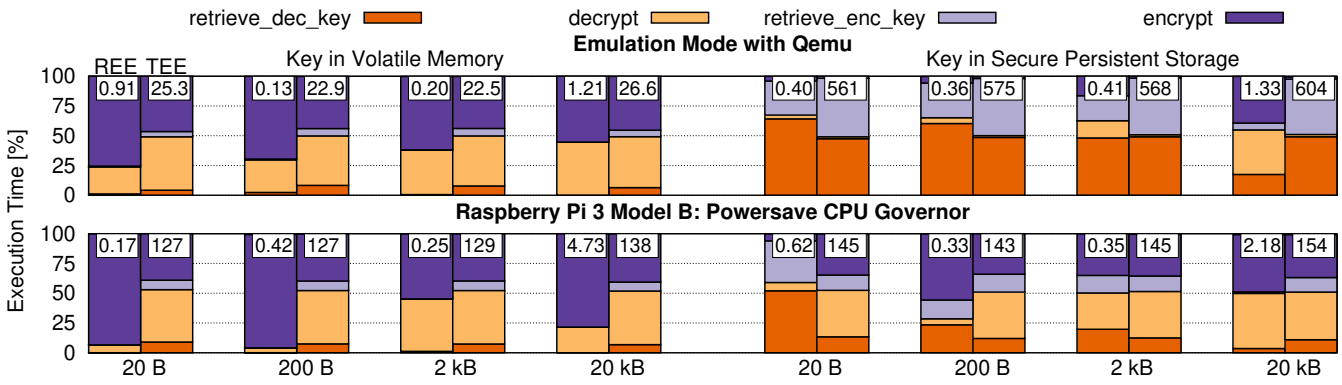

Fig. 5. Re-encryption TA microbenchmark. Percentage breakdown of the contributions of each operation in the time elapsed. For each different block size (bottom axis) we compare: re-encryption in the REE vs TEE with the key stored in memory or persistent storage. We also report average total time (in ms) boxed.

client's key exchange. To control access and R/W permissions to topics, we use `mosquitto`'s ACLs.

Trusted Application. We rely on OP-TEE APIs to implement the payload re-encryption TA. Trusted applications implemented within this framework have two parts: *(1)* a host app that runs in the REE and acts as an entry point and bridge to the TEE, and *(2)* a trusted API in the TEE that exposes different functions. MQT-TZ intercepts all MQTT packets forwarded to the recipients, and feeds our host app with both client's IDs and the encrypted data. Then, the payload re-encryption happens, using OP-TEE's storage and cryptographic libraries. TRUSTZONE not only provides isolation between worlds, but also between different TAs. Hence, we use the same secure API to store new keys during the handshake. For key retrieval, we implement a new LRU cache in the TEE to store the most frequently used keys in the TA's heap, while the remaining ones are evicted and flushed in persistent secure storage [37].

VI. EVALUATION

We present the experimental evaluation of the MQT-TZ prototype using micro-benchmarks and macro-benchmarks, as well as using real-world datasets from the MedTech scenario. Our intent is to validate the design of MQT-TZ, the efficiency of our implementation and to analyze the different trade-offs that the system incurs.

Evaluation Settings. We use Raspberry Pi 3 Model B units, one of the few where OP-TEE fully supports TRUSTZONE.[3] We set up the CPU in `powersave` governor mode to minimize energy consumption.

To validate the scalability of MQT-TZ, we also deploy a fleet of virtualized nodes using QEMU-V8[4], as it faithfully replicates the industrial settings planned for MQT-TZ. This emulated environment closely matches the expected hardware performance [6]. We use `mosquitto` (v1.6.3) and OP-TEE (v3.5.0). Unless otherwise specified, messages are 4 kBytes and are encrypted with 32-Byte keys. We always report average and standard deviation results for 100 executions of the described configurations. The default size of the messages reflects the setting presented in §VI-C.

[3]https://optee.readthedocs.io/en/latest/building/devices/rpi3.html#what-versions-of-raspberry-pi-will-work

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

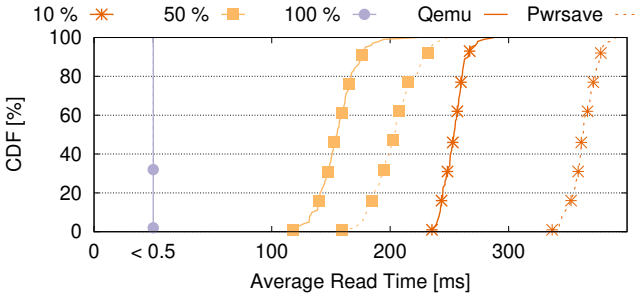

Fig. 6. Read time (CDF) from secure storage using MQT-TZ's secure cache for different numbers of cached objects. The cache capacity grows from 10% up to 100% of the total objects. Dashed lines represent hardware results and bold lines emulated ones.

A. Micro-benchmarks

Re-Encryption TA. We begin by measuring the time required to re-encrypt a block of data inside or outside the TEE, one of MQT-TZ's cornerstone operations. We include results for the hardware and the emulated environment using the `powersave` CPU governor, as it most closely matches the expected deployment settings. We breakdown these measures into four main components: the time it takes to retrieve each key (`retrieve_dec_key`, `retriev_enc_key`), and to use them (`encrypt`, `decrypt`). On the x-axis, we show results for different block sizes of data to re-encrypt, *e.g.*, from 20 Bytes up to 20 kBytes. We compare the performance of MQT-TZ decrypting inside the REE (left-side vertical bars) or in the TEE (right-side bars). For encryption, we use AES in CBC mode with 32-Byte keys. Finally, we include results for the variants of the system that maintain the keys either in volatile memory as well as in secure storage, and the overall elapsed time (in ms) boxed. Figure 5 uses a stacked bar chart representation to present these results.

This is the authors preprint version of the camera-ready article. The definitive version is published in the proceedings of the 2020 39th International Symposium on Reliable Distributed Systems (SRDS 2020).

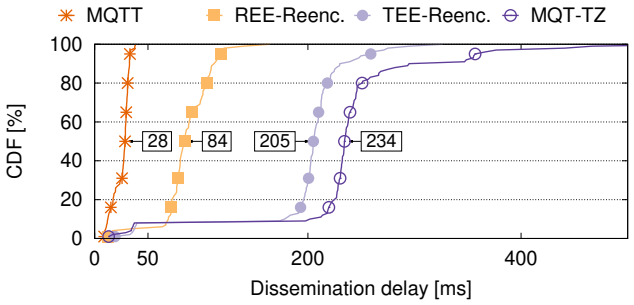

Fig. 7. Dissemination delays (CDF) for different MQTT implementations and configurations of MQT-TZ. We highlight (boxed) the delay in ms for the 50th percentile.

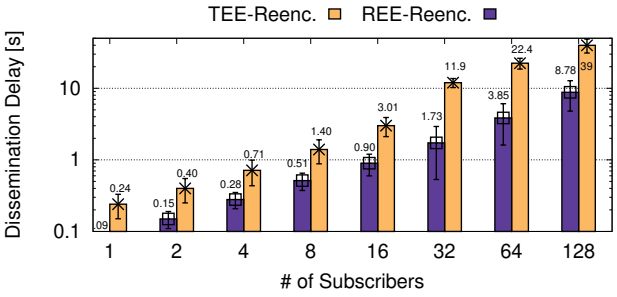

Fig. 8. Impact of total subscribers per topic on the overall message dissemination delays.

We observe that AES symmetric cryptography is two orders of magnitude slower in the TEE, both in emulation and in real hardware. For instance, for the 2kB case, we face a slowdown of 128× and 250×, respectively. This is due to the cryptographic libraries in OP-TEE not using hardware accelerators (due to security restrictions) and hence not as optimized as `openssl` used in the REE. Moreover, when switching from in-memory to secure persistent storage, we observe even higher slowdowns (*i.e.*, up to 250× and 260× in the 20 Bytes case), specifically in the time required to fetch the decryption key. These results motivate the inclusion of a LRU cache in MQT-TZ's architecture, as described next.

The emulation results show slower memory access times, but faster encryption both in the TEE and REE when compared to real hardware. In spite of that, emulation closely matches the time spent in each phase specially for the *Key in Volatile Memory Case*. When the key is stored using the Secure Storage API, QEMU overestimates the time to retrieve both objects, and underestimates the time to encrypt them. Therefore, emulation is useful for prototyping and early stages of development, but to draw conclusions on performance one must measure on the real hardware, under realistic workloads.

TEE Cache. To evaluate the performance of our LRU cache inside TRUSTZONE, we issue `get` queries to fetch entries from the cache. Figure 6 presents the cumulative distribution function (CDF) of the delays to return and read the reply. To execute this experiment, we initially preload a set of 128 256-bits AES keys to secure storage, as this matches the expected size of the system (both in terms of subscribers and publishers) that MQT-TZ will support in real-world deployments. Then, the cache is filled with keys randomly sampled from this set. The number of entries in the cache varies between 12, 64, and 128 keys, *i.e.*, 10%, 50%, and 100% of the total. The client issues 128 random queries, and measure the average latency over 100 runs for each configuration. Note that clock precision for such in-TEE measurements is 1 ms.[5] Hence, replies faster than this threshold (*e.g.*, as in the 100% case) are not reported. As expected, smaller caches lead to more the cache misses and higher average reading time. The median (50th percentile) reading value for the 50% case is 155 ms, and up to 253.85 ms for the 10% scenario. These results indicate the LRU cache is beneficial in all the tested scenarios.

When comparing hardware (dashed) with emulated (bold) values, we observe that QEMU fails to emulate the object storage and retrieval performance when the secure store is relatively full (128 keys in our case). This is specially true when using the `powersave` CPU governor mode [6].

B. Macro-benchmarks

To assess the overall performance of MQT-TZ, we measure the dissemination delay (latency) for a MQT-TZ setup with one single publisher and one single subscriber. We additionally show a larger-scale deployment scaling up the number of subscribers to the same topic. These benchmarks stress the operations occurring the most inside TRUSTZONE, *i.e.* encryption, decryption, and queries to the cache. Increasing the number of publishers (instead of subscribers) would yield the same number of cache queries, hence a symmetric effect on performance.

1 Publisher - 1 Subscriber. This is a baseline deployment, used to assess the dissemination delay performance. We compare vanilla `mosquitto` against several variants of MQT-TZ: *(1)* with re-encryption in the REE and all keys in memory, *(2)* re-encryption in the TEE and all keys in memory, and *(3)* all the features combined. Figure 7 reports the CDF of the dissemination delays. As expected, an increasing number of security features hurts the dissemination delays (up to 8× for the median values), with a long tail up to 350 ms.

1 Publisher - Many Subscribers. Next, we scale up the number of subscribers for a given topic. Figure 8 presents the dissemination delays for two configurations (REE-Reenc. and TEE-Reenc.) *e.g.*, using re-encryption in the REE and TEE respectively. We observe that the dissemination delays increase linearly with the number of subscribers. Re-encryption in the TEE (in general TA execution) being single-threaded, each subscriber has to be handled individually. As a conclusion, the biggest performance bottlenecks derive from the slow cryptographic primitives and the key retrieval routines. For the former, we rely on the primitives provided by OP-TEE. We intend to exploit Cryptographic Accelerators for TRUSTZONE (*e.g.* ARM CRYPTOISLAND [8]) to considerably speed-up these operations. For the latter, we introduced an in-TEE cache

[5]This is the default precision of the `TEE_Time` [35] offered by OP-TEE.

This is the authors preprint version of the camera-ready article. The definitive version is published in the proceedings of the 2020 39th International Symposium on Reliable Distributed Systems (SRDS 2020).

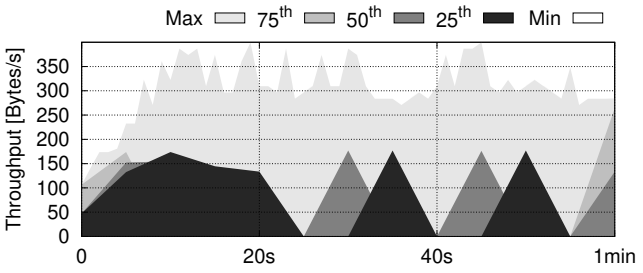

Fig. 9. Distribution of network throughput for a realistic deployment with 50 publishers emitting real-world cardiac signals.

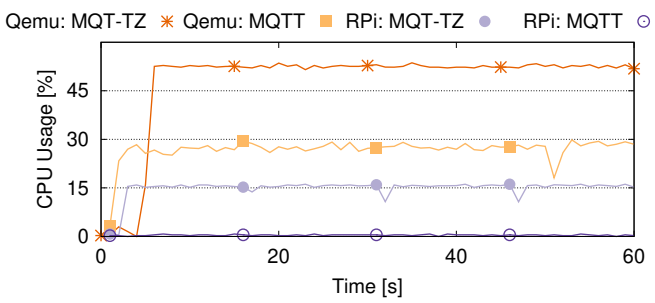

Fig. 10. MQT-TZ Broker CPU usage under the load from Figure 9.

to minimize the queries to persistent storage. Another strategy along this line would be to share one key among sets of clients (if the application's security concerns allowed to).

In terms of scalability, MQT-TZ is limited by the per-subscriber re-encryption process. This process being single-threaded (OP-TEE's limitation), all clients are serialized. A possible workaround would be to leverage `mosquitto`'s multithreading capacities combined with multiple running concurrent TAs (not in-TA threading). However, this would incur in higher CPU load and hence higher energy consumption, a possible showstopper in some constrained environments. Alternatively, establishing per-topic shared keys among client nodes, the re-encryption for some packets could be avoided. We intend to explore these options in future extensions of this work.

C. MedTech in action

Finally, we fully implement and deploy the vital signs monitoring scenario (§III-B). In this case, we are interested in understanding if in a real-world setting the MQT-TZ broker can efficiently (*e.g.*, CPU processing) sustain the injected workload. We leverage real-world ECG datasets we collected on the field. This deployment reproduces the layout of an hospital floor with 50 patients whose cardiac signals are constantly monitored. For the sake of simplicity, these signals are streamed toward one single MQT-TZ broker, although a federated deployment is also supported. We capture and measure the outbound network traffic from each publisher using `nethogs` [22]. Figure 9 depicts the outbound throughput generated by each publisher in bytes per second. We use a stacked percentile representation with shades of grey to plot the minimum, 25th, the 50th (median), the 75th and the maximum across all the publishers. We observe that at any given time only a subset of the publishers actually emits data. A single subscriber streams at 350 Bytes/s in the worst case, and the full fleet of publishers generates between 3 to 5 kBytes per second overall.

During the experiment, we use `dstat` [59] to record the CPU load at the broker (see Figure 10). We report the results for physical and virtualized environments, and against against a vanilla `mosquitto` deployment for both scenarios. Both in emulation and hardware, MQT-TZ adds between 10% to 15% to the CPU load on average, up at around 55% usage in the worst case. It is worth noting how these values are considerably higher in emulation than in HW, and the reduced amount of CPU ($\sim$ 1%) `mosquitto` requires. Overall, this suggests low energy consumption, an important factor for deployers that intend to deploy brokers in batter-powered nodes.

VII. RELATED WORK

Few attempts exist to provide secure extensions for MQTT [50], but none of them rely on TEEs. Despite, TEEs (and in particular TRUSTZONE) are used in several different domains, from cardiac signal processing over untrusted clouds [48] to control-based TRUSTZONE policy framework for air drones [57]. For the sake of conciseness, we shortly review secure messaging libraries on top of TEEs, as well as presenting a short survey of different usages of TEEs.

Secure Messaging Libraries Using TEEs. MQT-TZ leverages the native support of MQTT to establish TLS channels between the client and the broker. TaLoS [12] can establish secure TLS termination inside Intel SGX enclaves. Deploying a complete TLS stack inside the TEE is unnecessary in our context, and it would yield a larger attack surface, as the code loaded in the TRUSTZONE must be fully trusted.

PubSub-SGX [10] is a content-based publish/subscribe framework on top of SCONE [11], a compilation tool chain to securely run Linux containers inside SGX enclaves. PubSub-SGX is implemented in Python. The notion of *topic* is a first-class entity currently not supported by PubSub-SGX, making its adoption for our scenarios requiring relevant engineering efforts. Similar drawbacks exist in [43], a content-based routing mechanism for SGX on top of which privacy-preserving pub/sub framework can be implemented.

StreamBox-TZ [40] is a secure stream analytics framework for TRUSTZONE, specifically targeting telemetry data. Rather, MQT-TZ focuses on secure end-to-end packet delivery, delegating all the application-specific processing to client nodes.

TEE-Based Applications. Due to the additional security guarantees and resilience to stronger adversarial models, TEEs are currently being deployed across a plethora diverse scenarios. One important step for all the applications is the attestation protocol, *e.g.*, a process used by participants to verity the integrity and validity of the trusted applications as well as the CPU executing those. While Intel SGX has native support for remote attestation [3], TRUSTZONE lacks clear

specifications for it. Fides [44] presents a ready-to-use Key attestation framework for Android's TEE, and we intend to look further into it. A common domain of application for TEEs is web-based systems, as well as a common way for end-users to disclose personal data. In this context, SGX can protect the identity of the users against re-identification attacks via browser extensions and privacy proxies [15, 42]. While MQTT-TZ targets scenarios where the identity of the users is not to be hidden (rather, the opposite, as in our *MedTech* scenario), a broker can indeed be compromised. Techniques such as SGX-Tor [30] could be explored in the context of TRUSTZONE and MQTT-TZ to reduce the information shared with the brokers.

VIII. LESSONS LEARNED

Through the implementation, deployment and evaluation of MQTT-TZ we acquired insights into several aspects of such a system. We highlight here the most important ones.

First, the OP-TEE framework is sufficiently mature to allow rapid development cycles to quickly test with TRUSTZONE. Trusted Applications, as our LRU cache in the TEE or the re-encryption TA, can achieve performances close to the corresponding non-secure ones. However, the crypto primitives they provide are slower than expected, especially when compared to other state-of-the-art libraries (*i.e.*, `OpenSSL` [5]). Our experimental results highlight a slowdown of up to two orders of magnitude. We hope to mitigate these drawbacks exploiting hardware support for TRUSTZONE-specific Cryptographic Hardware Accelerators such as ARM CRYPTOISLAND [8].

Second, we have shown in our evaluation that the `mosquitto` MQTT broker has a small CPU footprint. However, code executed in TRUSTZONE can only be single-threaded, negatively affecting scalability. While leveraging multi-threading in the broker is part of future work, this must be carefully handled. In fact, we expect the increased CPU usage to lead to higher memory consumption, as well as higher energy requirements. In this sense, deployers should carefully evaluate such trade-offs and decide on application-dependent requirements.

Finally, we report how the emulation accuracy for ARM processors in QEMU is sufficiently accurate to allow us to validate the design and implementation without having to deploy large (and potentially) expensive testbeds. Yet, as shown in §VI, the timing measurements from QEMU can be inaccurate, and real-hardware measurements must be planned.

IX. CONCLUSION AND FUTURE WORK

Motivated by the lack of secure-by-design communication protocols for the edge and two real-world use-cases, we built MQTT-TZ, a secure edge-based publish/subscribe middleware using MQTT and TRUSTZONE. We report on our experiences while building and evaluating our open-source prototype against a vanilla MQTT under real-world workloads. Despite the measured slowdown (up to 8× in some scenarios), our system scales and can be deployed in restricted, IoT-based settings, achieving dissemination delays in the orders of milliseconds, even when deployed in low-end devices (such as Raspberry Pi units). In addition, the motivating use-cases described in §III can benefit of the additional security guarantees provided by MQTT-TZ with no additional hardware and without changes to the client's application code.

We plan to extend this work along the following directions. Firstly, we intend to extend the MQTT-TZ evaluation and compare against other topic-based publish-subscribe systems and messaging queues as well as testing in larger-scale settings with clustered MQTT brokers. Secondly, we intend to study the energy trade-offs of our system, a key aspect for edge deployments. Thirdly, we will revise MQTT-TZ architecture to shield in TRUSTZONE some additional components (*e.g.*, ACLs, the subscription lists). Lastly, we plan to implement a proof of concept version of MQTT-TZ leveraging alternative software development kits for TRUSTZONE such as OPENENCLAVE (https://openenclave.io/sdk/) and alternative TRUSTZONE-enabled devices (*i.e.*, TRUSTBOX https://scalys.com/trustbox-industrial/).

ACKNOWLEDGMENT

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, 2019.
[2] The highly dangerous 'Triton' hackers have probed the US grid. https://www.wired.com/story/triton-hackers-scan-us-power-grid/, 2019.
[3] Intel attestation services. https://software.intel.com/en-us/sgx/attestation-services, 2019.
[4] Linaro Foundation. https://www.linaro.org/, 2019.
[5] OpenSSL. https://www.openssl.org/, 2020.
[6] J. Amacher and V. Schiavoni. On the performance of ARM TrustZone. In José Pereira and Laura Ricci, editors, *Proceedings of the International Conference on Distributed Applications and Interoperable Systems (DAIS'19)*, DAIS'19, pages 133–151, Cham, 2019. Springer International Publishing.
[7] Arm. Arm: Architecting a smarter world, 2020.
[8] Arm. Arm CryptoIsland family. https://developer.arm.com/ip-products/security-ip/cryptoisland, 2020.
[9] Arm. Record shipments of Arm-based chips in previous quarter. https://www.arm.com/company/news/2020/02/record-shipments-of-arm-based-chips-in-previous-quarter, 2020.
[10] S. Arnautov, A. Brito, P. Felber, C. Fetzer, F. Gregor, R. Krahn, W. Ozga, A. Martin, V. Schiavoni, F. Silva, M. Tenorio, and N. Thummel. PubSub-SGX: Exploiting trusted execution environments for privacy-preserving publish/subscribe systems. In *Proceedings of the IEEE 37th Symposium on Reliable Distributed Systems*, SRDS'18, pages 123–132, October 2018.
[11] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, OSDI'16, pages 689–703, Savannah, GA, November 2016. USENIX Association.
[12] P.-L. Aublin, F. Kelbert, D. O'keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch. TaLoS: Secure and transparent TLS termination inside SGX enclaves. Technical report, Imperial College London, May 2017.
[13] Roberto Baldoni, Roberto Beraldi, Vivien Quema, Leonardo Querzoni, and Sara Tucci-Piergiovanni. TERA: Topic-based event routing for peer-to-peer architectures. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*, DEBS 07, pages 2–3, New York, NY, USA, 2007. Association for Computing Machinery.
[14] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC 05, page 41, USA, 2005. USENIX Association.

[15] S. Ben Mokhtar, A. Boutet, P. Felber, M. Pasin, R. Pires, and V. Schiavoni. X-Search: Revisiting private web search using Intel SGX. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, volume abs/1805.01742 of *Middleware '17*, 2018.

[16] M. Berberich and M. Steiner. Blockchain technology and the GDPR - how to reconcile privacy and distributed ledgers. *European Data Protection Law Review*, 2:422, 2016.

[17] R. Castaldo, P. Melillo, U. Bracale, M. Caserta, M. Triassi, and L. Pecchia. Acute mental stress assessment via short term HRV analysis in healthy adults: A systematic review with meta-analysis. *Biomedical Signal Processing and Control*, 18:370–377, 2015.

[18] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[19] C. Cuijpers and B.-J. Koops. Smart metering and privacy in Europe: Lessons from the Dutch case. In *European data protection: Coming of age*, pages 269–293. Springer, 2013.

[20] R. Delgado-Gonzalo, J. Hubbard, Ph. Renevey, A. Lemkaddem, Q. Vellinga, D. Ashby, J. Willardson, and M. Bertschi. Real-time gait analysis with accelerometer-based smart shoes. In *Proceedings of the 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC'17)*, pages 148–148c, July 2017.

[21] T. Dierks. The transport layer security (TLS) protocol version 1.2. 2008.

[22] A. Engelens. Nethogs: Linux 'net-top' tool, 2019.

[23] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[24] Sh. Farahani. *ZigBee wireless networks and transceivers*. Newnes, 2011.

[25] Global Platform. GlobalPlatform specifications archive. https://globalplatform.org/specs-library/?filter-committee=tee, 2019.

[26] C. Gottel, R. Pires, I. Rocha, S. Vaucher, P. Felber, M. Pasin, and V. Schiavoni. Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *Proceedings of the 2018 IEEE 37th Symposium on Reliable Distributed Systems*, SRDS'18, pages 133–142, 2018.

[27] M. A. Hamburg and F. S. Collins. The path to personalized medicine. *New England Journal of Medicine*, 363(4):301–304, 2010.

[28] MQTT Version 3.1. Standard, OASIS, March 2016.

[29] D. Kaplan, J. Powell, and T. Woller. AMD Memory Encryption. AMD Developer Central, 2013.

[30] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing security and privacy of Tor's ecosystem by using trusted execution environments. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 145–161, Boston, MA, March 2017.

[31] A. Krylovskiy, M. Jahn, and E. Patti. Designing a smart city internet of things platform with microservice architecture. In *Proceedings of the 2015 3rd International Conference on Future Internet of Things and Cloud (FiCloud'15)*, pages 25–30, Aug 2015.

[32] W. S. Lee and S. H. Hong. Implementation of a KNX-ZigBee gateway for home automation. In *Proceedings of the 2009 IEEE 13th International Symposium on Consumer Electronics*, ISCE'09, pages 545–549. IEEE, 2009.

[33] Y. Lee, H. Hsiao, C. Huang, and S. T. Chou. An integrated cloud-based smart home management system with community hierarchy. *IEEE Transactions on Consumer Electronics*, 62(1):1–9, February 2016.

[34] Y. Li and S. H. Hong. BACnet–EnOcean smart grid gateway and its application to demand response in buildings. *Energy and Buildings*, 78:183–191, 2014.

[35] Linaro Security WG. GitHub: Op-TEE OS - Time definition. https://github.com/OP-TEE/optee_os/blob/6e9e277f455a70e5b7f59cd7df5da419bc0697f8/lib/libutee/include/tee_api_types.h#L163, 2019.

[36] Linaro Security WG. OP-TEE: Open portable trusted execution environment. https://www.optee.org, 2019.

[37] Linaro Security WG. OP-TEE: Secure storage. https://optee.readthedocs.io/architecture/secure_storage.html, 2019.

[38] R. Liu and M. Srivastava. VirtSense: Virtualize sensing through ARM TrustZone on internet-of-things. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, pages 2–7. ACM, 2018.

[39] P. McDaniel and S. McLaughlin. Security and privacy challenges in the smart grid. *IEEE Security & Privacy*, 7(3):75–77, 2009.

[40] H. Park, S. Zhai, L. Lu, and F. Xiaozhu Lin. StreamBox-TZ: Secure stream analytics at the edge with TrustZone. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC'19, pages 537–554, Renton, WA, July 2019. USENIX Association.

[41] S. Pinto and N. Santos. Demystifying Arm TrustZone: A comprehensive survey. *ACM Computing Surveys*, 51(6):130, 2019.

[42] R. Pires, D. Goltzsche, S. Mokhtar, S. Bouchenak, A. Boutet, P. Felber, R. Kapitza, M. Pasin, and V. Schiavoni. CYCLOSA: Decentralizing private web search through SGX-based browser extensions. In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems*, ICDCS '18, 07 2018.

[43] R. Pires, M. Pasin, P. Felber, and C. Fetzer. Secure content-based routing using Intel software guard extensions. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 1–10, New York, NY, USA, 2016. ACM.

[44] B. Prünster, G. Palfinger, and C. Kollmann. Fides Unleashing the full potential of remote attestation. In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications*, SECRYPT'19, pages 314–321. SciTePress - Science and Technology Publications, 7 2019.

[45] P. Qiu, D. Wang, Y. Lyu, and G. Qu. VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS 19, pages 195–209, New York, NY, USA, 2019. Association for Computing Machinery.

[46] P. Renevey, R. Delgado-Gonzalo, A. Lemkaddem, M. Proença, M. Lemay, J. Solà, A. Tarniceriu, and M. Bertschi. Optical wrist-worn device for sleep monitoring. In *Proceedings of the 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, EMBC'17, pages 615–618. Springer, 2017.

[47] Keegan Ryan. Hardware-backed heist: Extracting ECDSA keys from Qualcomm's TrustZone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS 19, pages 181–194, New York, NY, USA, 2019. Association for Computing Machinery.

[48] C. Segarra, R. Delgado-Gonzalo, M. Lemay, P.-L. Aublin, P. Pietzuch, and V. Schiavoni. Using trusted execution environments for secure stream processing of medical data. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems (DAIS'19)*, volume 11534 of *DAIS'19*, pages 91–107, 2019.

[49] C. Segarra, E. Muntané Calvo, M. Lemay, V. Schiavoni, and R. Delgado-Gonzalo. Secure stream processing for medical data. In *Proceedings of the 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, EMBC'19, 2019.

[50] SeongHan Shin, Kazukuni Kobara, Chia-Chuan Chuang, and Weicheng Huang. A security framework for MQTT. In *Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS'16)*, pages 432–436. IEEE, 2016.

[51] Philip Sparks. The route to a trillion devices. Technical report, ARM, 2017.

[52] G. Tan, T. K. Dao, L. Farmer, R. J. Sutherland, and R. Gevirtz. Heart rate variability (HRV) and posttraumatic stress disorder (PTSD): A pilot study. *Applied Psychophysiology and Biofeedback*, 36(1):27–35, 2011.

[53] Teserakt AG. Is MQTT secure? (A report). https://blog.teserakt.io/2019/03/04/is-mqtt-secure/, 2019.

[54] The Eclipse Foundation. Eclipse Mosquitto - An open source MQTT broker. https://mosquitto.org/, 2019.

[55] V. Turner, J. F. Gantz, D. Reinsel, and S. Minton. The digital universe of opportunities: rich data and the increasing value of the internet of things. *IDC Analyze the Future*, 2014.

[56] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EuroCrypt'10)*, EUROCRYPT'10, pages 24–43. Springer, 2010.

[57] A. Vijeev, V. Ganapathy, and C. Bhattacharyya. Regulating drones in restricted spaces. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, pages 27–32, New York, NY, USA, 2019. ACM.

[58] P. Voigt and A. Von dem Bussche. The EU general data protection regulation (GDPR). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.

[59] D. Wieers. Dstat: Versatile resource statistics tool. http://dag.wiee.rs/home-made/dstat/, 2019.

[60] M. B. Yassein, M. Q. Shatnawi, S. Aljwarneh, and R. Al-Hatmi. Internet of things: Survey and open issues of MQTT protocol. In *Proceedings*

of the 2017 International Conference on Engineering MIS, ICEMIS'17, pages 1–6, May 2017.