Empowering the Economy of Things

weeve. network

Technical Paper (Part 1/4)

Technical Paper

Empowering the Economy of Things[*]

Mathias Davidsen, Sebastian Gajek, Marvin Kruse and Sascha Thomsen

weeve.network

Abstract

The Internet of Things (IoT) paradigm is filled with opportunities due to the fact that things can mine, process, transfer data, and this way create valuable *digital assets*. An Economy of Things (EoT) is thus formed when these digital assets are autonomously traded for related digital assets, most notably Blockchain-based currencies. This idealistic economy is problematic. IoT devices are easily susceptible to be compromised and data can be manipulated and falsified. Existing IoT technologies are unable to leverage against this underlying crux and prove inadequate, as they are not able to provide a guarantee to the data's truthfulness. It is the accepted belief in the "physical world" that on the purchase of a good the product's quality is as claimed and its properties have been verified. Only then a fair price can be negotiated. In the "IoT world" the dual mechanism is lacking, as IoT data cannot yet be verifiably attested and thus holds no guarantee to its integrity and economic value.

In this summary we unpack the core technologies underneath the weeve platform. The platform enables *unfalsifiable, scalable and secure* marketplaces where IoT devices are indexing, processing and trading digital assets that are by design testified. Marketplaces leverage corporates and individuals alike the bootstrapping to private and public IoT economies, while removing the worries of the underlying technicalities involved with the configuration and orchestration of IoT devices, integration of suitable Blockchain technologies and safeguarding that the supply and demand mechanisms build upon factual data. The weeve platform ultimately lays the foundation for the Internet of Things to evolve into a new Economy of Things based on testified data, fair trade and justifiable pricing.

Keywords: Economy of Things, Trading Digital Assets, Data Testimony, Fairness, Trustworthiness, Scalability, Security by Design, Blockchain

[*]For releases of the Weeve IoT-to-Blockchain Operating System, codenamed WeeveOS, based on the security and trust principals presented in this paper, please go to to `https://github.com/weeveiot/weeveos`

Contents

1 Introduction

1.1 Motivation

The Internet of Things, commonly known as IoT, refers to an evolving phenomenon whereby computing devices are becoming connected to the Internet. This web of interconnectedness has the ability to open up communication channels between previously disconnected and isolated devices and possesses new economic and trading possibilities.

To fully grasp the possibilities of a proliferating IoT world, it is imperative to understand the magnitude that IoT is. Gartner predicts that by 2020 there will be 20.8 billion connected IoT devices. These devices will all produce data that will form a valuable resource. This data ranges from physical measurements, such as temperature and geolocation coordinates to the acknowledgment of an event, such as a payment. According to IDC, this data value will exceed a staggering 7.1 trillion USD. Thus making data a valuable resource.

The progression of IoT lets us imagine a world where data becomes tradable. IoT things, called producers, offer their data and IoT devices, called consumers, buy the data. This gives rise to an Economy of Things (EoT) where machines are autonomously indexing, producing and trading valuable IoT data for revenue. The vision is underpinned by a concurrent paradigm known as Blockchain. In a nutshell, Blockchains are append-only distributed databases leveraging the execution of smart contracts. In many cases smart contracts are linked to digital coins (e.g., Bitcoins, Ether), which have a value like a real currency. In some other cases, smart contracts trigger a transaction of digital coins after a pre-defined sequence of (oracle) events has occurred. These range from time to location or temperature events, which due to their importance, are nothing else than digital assets themselves.

The symbioses of IoT with Blockchain lays the foundation for an Economy of Things and offers not only a plethora of novel, ground-breaking use cases, but also opens up unimaginable new market opportunities. Imagine a hybrid car that has the ability to autonomously pay an induction loop for the charging of energy while waiting at a red-light crossing. In the same manner, imagine a logistic company gets paid for the transportation of a container once it has delivered its goods, provided it has taken the shortest delivery path and attained an average temperature of the goods throughout the journey.

The bottom line is that the possibilities of IoT blended with Blockchain technologies refers to a world where data will have value and be traded through machine-to-machine transaction models. Hence making the monetization of data of peak interest, in conjunction to the advances of IoT, as consumers and businesses alike realize the monetary merit that data holds.

1.2 Problem Statement

Any economic process normally comprises of the the five elementary stages:

Harvesting. At the beginning of any economic process is the resource. It is the ingredient subjected to trade. Resources typically range from goods, such as oil, coal or gold. In the imminent future, digital data will also become a resource. Currently, data such as cryptocoins are nothing else than mere bitstrings, which are being traded. Furthermore, there is a growing market for personal data that are gathered from social networks and related cloud platforms. The increasing presence of IoT devices and their tremendous production of data will provide new dimensions of economically meaningful resources. This ranges from sensing new information to confirming that some activities have taken place.

Processing. Once the resource has been harvested, it will be processed. Processing is necessary to transform the raw material into a *good*. For example, the resourced wood can be manipulated into planks. The digital dual is to compute on data. Here one can think of filtering out bad data or inferring statistics over the data. On a more general level, one runs an algorithm over the data to infer the desired information.

Transportation. Subsequently, the good must be transported. In some cases, the transportation may be delayed by dropping off the goods at a warehouse for storage or to a manufacturer for additional processing. In the same vein, digital goods are transferred to a cloud service. For instance, the digital good is stored in a data hub or a blockchain. In the latter case, dApps running on top of the blockchain may post-process the data.

Assessment. Before the good is tradable, one assesses the quality and value. By quality we mean the origin, properties and reliability of the good. Normally after the evaluation of the good, a pricing mechanism is applied that reflects the estimated value.

Trading. Lastly, the price is negotiated and finalized in relation to supply and demand. There are several models to agree upon the price, ranging from first and second price auctions to what is informally known as marketplaces, loosely speaking being fixed-price auctions. We will use 'marketplace' as an umbrella term for all the models.

The Crux with Today's EoT. The above five stages are elementary to any economy of things. They reflect the stages of turning data into digital assets (i.e. digital goods with value). Any deviation from or error in any of the stages leads to an unsatisfactory situation: *Fair and existential trading is impossible.* It is simply unlikely to determine the truthful value of the asset and the integrity of the good.

The cause lies in the inability to validate the data. Data is merely a sequence of bitstrings, harvested, processed, transported and assessed by machines. In this supply chain different computing entities are involved. It is well known, since the introduction of computers, that machines are prone to errors. Likewise it is well known that they can be compromised. Nowadays IoT devices are nothing more than lightweight computers. They build upon computing architecture supported by rich Operating System's functionalities similar to that of commodity personal computers. A vast amount of attack vectors already exist in the world, ranging from network attacks like man-in-the-middle to application attacks like buffer overflow and return-oriented programming. (Discussing the attack anatomy in more detail is out of scope. We refer the interested reader to [25].)

Indisputably IoT devices will be subjected to attacks. At this stage in time we already observe a growing number of attack vectors specifically designed to compromise IoT devices. Several arguments back up the observation. Currently, one key argument is that IoT devices already bear access to a tremendous amount of digital assets. Recall the car payment use case. The car pays in cryptocurrency for the energy charging. Thus it must utilize a wallet to do the transaction. Hence, hacking the wallet in order to “steal” the coins becomes a very lucrative goal. The monetary aspect is clearly an appealing incentive.

Moreover, when talking about EoT, where not only crypto coins are traded, but also any form of economically useful data (in fact, crypto coins are a special case of tradable digital assets), stealing the data will become a threat as well as faking the data. In an Economy of Things situations naturally arise where some data have more value than other data. The demand and supply of specific data varies, resulting in the unequal value distribution between different types of data. However, data is just a sequence of bitstrings. So, *who prevents the attacker to simply flip some bits in favor of a higher price?* Once the device is corrupted modifying the data, say in transition between the sensor and the application processing, is straightforward. One can think of an analogy of a keyboard logger. Instead of tracing and modifying keyboard events, one traces and modifies sensor inputs [27].

A critical reader might argue one can detect the modification with some prudent machine-learning algorithm (e.g., run a kernel method and report an anomaly, if the data deviates from the norm). In general this is untrue and if applicable yields a fragile countermeasure. The approach fails in various important settings, as it requires a learning phase. On the one hand, learning prerequisites a sufficient amount of “truthful” data (e.g., [23] shows how to falsify the learning process). On the other hand, learning demands some significant amount of “similar” data to identify what a good cluster of data is. In many cases, specifically the cases we are interested in, it is merely impossible to get the data with the above desiderata. Consider the payment use case again. By definition each transaction is *atomic*, meaning transaction value and destination vary and are with overwhelming probability statistically independent of previous transactions. Classifying the data in a meaningful way is futile. The same argument rigorously applies to all data being atomic per se. The crucial question that remains unanswered is

> “How can data—down to the atomic level—be harvested, processed, transferred, and assessed, such that it can be autonomously traded between devices, when various entities and/or processes are involved in between?”

Answering the question in an affirmative way is the mission of weeve. There is no single technology to reach the goal. In fact, it is the consecutive combination of different technologies at various levels of the computing stack. Starting from the hardware architecture over the Operating System and communication protocols to the backend application and Blockchain. In the following sections, we give a summary of the necessary core technologies. For technical details, we refer the reader to the forthcoming full version of the paper.

1.3 The weeve Approach

1.3.1 Goal

Our ultimate aim is to empower an Economy of Things where IoT machines (or "weeves" thereof) index, process and trade harvested data against digital assets, most notably crypto coins. We envision a public or private *marketplace*, where producers and consumers (resp., buyers and sellers) come together, escrow their supply and demand, and exchange their digital assets.

1.3.2 Requirements: Scalability, Unfalsifiability and Security

Given the fact that millions of IoT devices have continuous access to data scalability becomes a requirement of game-changing relevance. By *scalable* we mean a technology that handles offer and payment requests of, say millions of devices, at a high throughput. For example, speed energy trading (i.e. producers sell energy to consumers) requires a transaction throughput in the dimensions of Twitter or Facebook (Note that in this use case special requirements have to be fulfilled by the underlying Blockchain, if high payment throughput must be achieved.).

As argued in the previous section, different key technologies along the line of the data exploitation chain need to be combined to ensure that the digital assets are *unfalsifiable*. When digital data becomes a tradable asset the buyer needs to have some guarantees about the quality of the data and the integrity of the product. There should be no doubts regarding the origin and harvesting procedure of the data in question. Ideally when the data turns into a digital asset one must be able to verify its quality.

As data is a digital asset, the utmost care must be taken to *secure* the process of harvesting the data and turning it into a tradable asset. By secure we understand the authentic and confidential processing of data. It encompasses the linkage of device identities with harvested data and measures safeguarding data privacy. The first is a prerequisite to link the data with the device owner or an account necessary for the monetization of the data. The latter is the central ingredient for a fair trade within a marketplace. There is a strong connection between data privacy and exclusivity. Suppose, for the nonce, that data would not be kept in a private way. It thus can easily be replicated, as it is a simple sequence of bitstrings. Consequently, one can increase the supply, which leads to a reduction in price. It is clear that a price decrease due to information leakage is an undesired property in an Economy of Things.

1.3.3 Overview of the Core Technologies

In what follows is a summary of the key concepts underlying the weeve platform. We refer the reader to Section 2 for further information about the core technologies and the weeve platform, respectively. In a nutshell, our technologies can be classified as follows:

- At the bottom of our technology stack we build upon a hardware architecture, supporting a Trusted Execution Environment. We rely on the ARM Trustzone[1] security extension. Meanwhile the extension is a standard module of ARM architectures supported by all major chip manufactures.

- To leverage the ARM Trustzone, we make use of a Trusted Execution Environment enabled Operating System (TEE-OS). On the one hand, the TEE-OS allows us to isolate the execution of programs. This way, we separate the normal OS functionalities from security critical ones, including the access to cryptographic key material and algorithms. Moreover, with the aid of the TEE-OS we benefit from a secure boot process. Secure boot makes sure an IoT device is bootstrapped in a pre-configured way and the OS (and its crucial functionalities) has not been tampered with. This gives the certainty that the IoT device initially is in a trustworthy state. However, it is already noted that during run-time no statements about the trustworthiness can be made.

- One core ingredient of our stack is a lightweight secure communication protocol, called TEE-MQTTS. The goal of the cryptographic protocol is to establish an authentic and confidential connection between an IoT device and a broker (e.g. cloud backend). With the protocol we transport digital assets from the IoT device to the backend, such that digital assets are kept private (confidentiality) and are protected against manipulation, origin impersonation (authenticity) and replay in transit. Our protocol is inspired by the famous MQTT protocol [6], the de-facto standard communication protocol for IoT, extended with cryptographic features normally obtained from the computationally expensive SSL/TLS protocol [8]. This results in a highly scalable, secure publish-and-subscribe mechanism. An IoT device securely transceives a message with a communication overhead of 3 rounds (exactly the number of protocol rounds MQTTS needs) and saves up the communication overhead induced by SSL/TLS, which amounts to 4 additional rounds. To further harden security, we leverage the Trustzone approach and isolate the protocol logic from the cryptography. This measure provides additional shielding against compromise. Even in the case of the exploitation of the normal OS or its applications, it is infeasible to extract the cryptographic key material.

- The second core ingredient is a mechanism to testify the execution of particular programs. Specifically, we are interested in programs harvesting and processing data, though we note that our mechanism applies to any program. A testimony is cryptographic proof about the truthfulness of data harvesting. Similar to related non-interactive proof systems, such as NIZKs and SNARGs [12, 15], a testimony satisfies strong cryptographic soundness. It is hard to generate a valid proof for an execution of a program other than the harvesting program. The property has some direct implications for the non-falsifiability of data. A testimony assesses the data. It enables any third party to precisely verify how the data was generated and/or processed. One can think of this as a *certificate of quality*. The novelty of our technique is that the

[1]`https://www.arm.com/products/security-on-arm/trustzone`

testimony occurs at *run-time*. Prior approaches have been static per se and only were able to attest the program code, but not its execution.

- The third ingredient is the IoT Blockchain wallet. The purpose of the wallet is to manage and authenticate smart contract based transactions. It stores pre-defined, standardized smart contract templates and the cryptographic credentials to legitimate the contract. Both are very sensitive information and demand some extra protection. Otherwise, a compromise of the IoT device leads to an economic disaster. All digital assets—be it the digital coins to issue a payment or the testified data to issue an offer—are lost. Our wallet harnesses the TEE security functionalities to isolate the templates and credentials from the normal world. This way the wallet thwarts the extraction of the credentials. But the design of the wallet goes beyond. It only authorizes very history-dependent smart contract transactions, namely those being well-formed and refers to a particular offer from the marketplace. The technique hampers illegitimate signing of arbitrary smart contracts by exploiting the wallet as a black-box.
- All the described techniques serve the sole purpose of harvesting data, making it accessible to a third party in a secure way, and providing a mechanism to assess the non-falsifiability of the data. On top of all that, we present a cryptographic protocol for the fair exchange of digital assets. Our protocol can be formulated within the syntax of smart contracts, such that the exchange of data (accompanied with the necessary testimony) directly occurs over the Blockchain.

The weeve marketplace manifests from the arsenal of these technologies. In the next section we give a detailed introduction to the technologies.

2 Our Core Technologies

2.1 Trusted Execution Environment

A Trusted Execution Environment offers an execution space that provides higher security guarantees then a commodity operating system. Among the security features are isolated executions, secure storage, remote attestation, secure provisioning and trusted path, to name a few [11]. The armada of these and related security features serves the sole purpose of executing programs without modifying their program flow or compromising their inputs and outputs. Several approaches have been proposed for TEEs, ranging from hypervisor-based to hardware-based implementations.

Hypervisor-based approaches make use of virtualization techniques. A Hypervisor is a virtual machine monitor that creates and manages virtual machines. It runs with higher privileges and thus can distribute resources between the virtualized machines. This way hardware is shared between guest machines, but they remain isolated from each other. Examples include vSphere Hypervisor by VMware and the Xen Project Hypervisor. The drawback of hypervisors is that they generate some overhead for each system call. Each call needs to be handled in the hypervisor and require some modification to the OS running within the guest machine. Additional overhead is especially problematic within an embedded system where resources might be scarce and battery-life has to be taken into account.

Hardware-based approaches use a different type of hardware protection. They use some dedicated piece of hardware in conjunction with software. The role of the hardware is to offer cryptographic algorithms, secure storage for cryptographic keys or to provide a root of trust. These crypto-coprocessors can come in many variations such as a Universal Integrated Circuit Card (UICC), secure microSDs or trusted platform modules (TPM). The major drawback of these approaches, next to the additional cost for hardware, is the hard wiring of security functionalities. The latter limits the design of security applications beyond the capabilities of the crypto-coprocessor.

Recently, much emphasis has been put on reconciling the two approaches, from which the ideal and secure world paradigm has resulted. The idea is to slightly augment processor-units, such that they can switch between two execution environments. One of these environments being the secure world with higher privileges, such as higher ranked interrupts and access to privileged memory addresses, and a leaner operating system, assumed to be trusted. The other environment being the normal, non-secure world with lower privileges and a normal, fully functional operating system. The purpose is to isolate the normal from the secure world, such that security critical processes can only be executed in the secure world. Due to the isolation of the secure world, it effectively implements the functionality of a crypto-coprocessor with the full flexibility of software. Example technologies include Intel SGX, AMD SEE and ARM TrustZone. While the first two are focusing on cloud-server architectures, the latter is optimized for embedded devices.

2.1.1 ARM TrustZone—A Primer

ARMs version of a trusted execution environment is called TrustZone: a System on Chip (SoC) and CPU system-wide approach to security. Being one of today's three solutions, next to Intel and AMD, TrustZone is already implemented in various devices utilizing ARM architecture, like the Cortex-A[2] and M[3] series. A separation of the CPU's privileges into a secure and a non-secure world and different interrupts in both worlds provides us with a lot of tools to raise the security bar.

Non-Secure World. The non-secure world is the "normal" environment of the CPU, where no higher privileges are available. Only the necessary rights and interrupts are allocated to ensure that no unauthorized access to the secure world will be possible. Certain actions or system-calls that might compromise the stability or security of the system cannot be executed directly by a normal worlds' process. Instead a pre-implemented redirection of said calls to the secure monitor will be conducted, where a verification process determines the possible effects. Additionally, the communication interfaces between the secure and the

[2]`https://www.arm.com/products/processors/cortex-a`

[3]`https://www.arm.com/products/processors/cortex-m`

non-secure world are well monitored, rendering a direct attack conducted from the normal world practically impossible.

Secure World. The secure world is the main feature of ARM's TrustZone architecture. It contains the secure operating system, a secure monitor, cryptographic interfaces, secure storage and trusted applications. The implementation of the two-world-paradigm is achieved through a simple non-secure (NS) bit in the secure configuration register. While a process has higher privileges if this bit is set to 0 (secure) for the used memory cell, the privileges are accordingly lowered for a memory cell with the bit set to 1 (non-secure). This is implemented through an additional table-column containing this NS-indicator for every memory or cache cell.

2.1.2 Trusted Execution Environment enabled Operating System

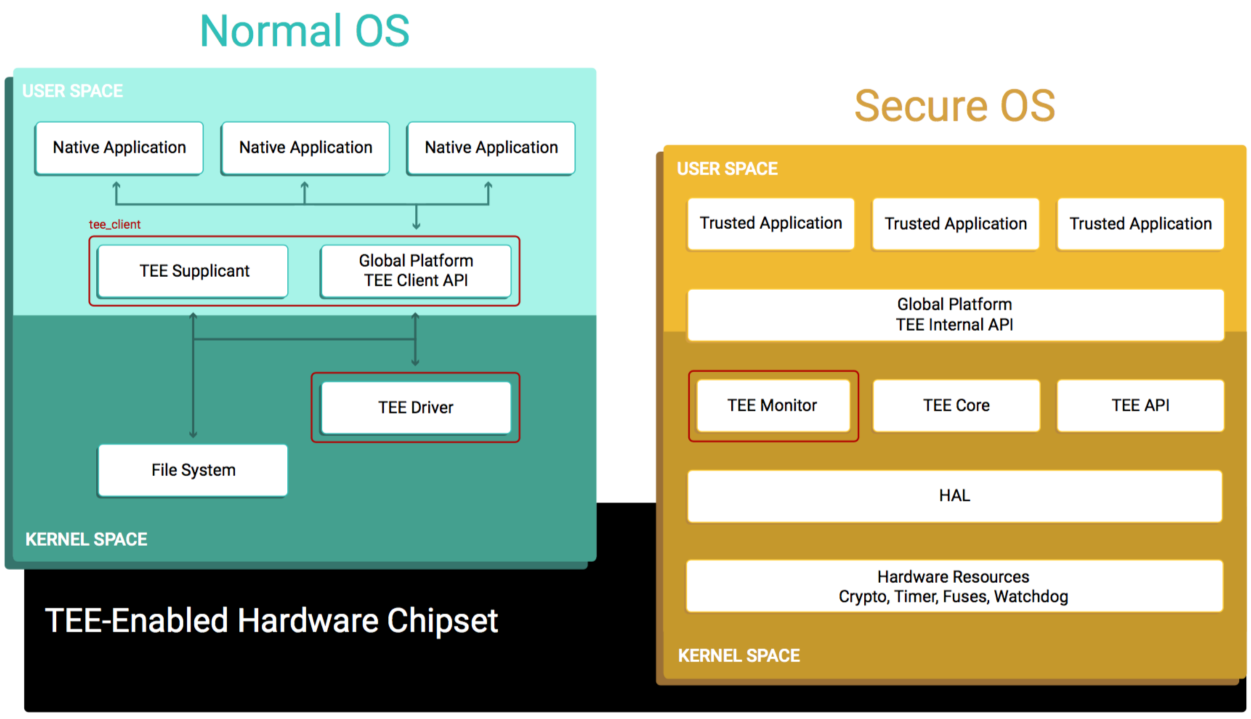

Figure 1: Schematic Illustration of a Trusted Execution Environment with core Operating System Functionalities. Red dashed lines highlight the TEE client, driver and monitor.

The Trustzone hardware alone is insufficient to grasp at all advantages of a Trusted Execution Environment. In addition to the hardware, one needs a TEE-empowering Operating System, for short TEE-OS, to activate the full security potential. In fact, a TEE comprises of (at least) two Operating Systems. The first implements the normal world and the second one implements the secure world. Although arbitrary OS choices for both worlds can be made, our approach is to choose a lean secure world OS and rich normal world OS. The argument is as follows, by reducing the complexity of the secure OS, the odds of the vulnerability being found are shorter than in the case of an OS with a comprehensive kernel and middle layer. Ideally, the secure OS constitutes some manageable thousands of lines of code amenable to peer-reviewing or formal verification [19].

As illustrated in Fig. 1, three TEE modules augment the system architecture of a commodity OS, helping with the execution of programs in the secure world known as Trusted Applications (TAs):

TEE Client: A client offers some interfaces for native applications to interact with Trusted Applications in the secure OS.

TEE Driver: A driver gives the privileged normal OS the capability of making use of the Trustzone hardware security technologies.

TEE Monitor: A (secure) monitor ensures that only valid and well-formed values are allowed as inputs for the secure world, while being the supervisor for everything that may happen in both worlds. All communication to and from the secure world goes through the monitor.

The whole TEE-OS machinery is necessary to realize security functionalities that commodity IoT Operating Systems typically lack to provide. In the following paragraphs we describe the main concepts behind program isolation, secure storage and boot, as taken from our implementation.

2.1.3 Program Isolation

Isolation is a powerful mechanism and a pivot technology in our stack. It enables to separate the processes running in the normal and secure world. The separation enforced by Trustzone is a particularly severe one and goes down to the hardware level (as opposed to, e.g., hypervisor-based isolations). The NS-bit in the memory unit helps the CPU to differentiate between normal and secure world programs. This way, programs in normal OS memory lack the privilege to access secure OS memory. Loosely speaking, one can think of running two physically distinct machines on a single system on a chip. It is worth mentioning that vice versa a secure OS program (e.g. Trusted App) does have the privilege to read and write normal OS memory. The strong isolation guarantees have some very appealing implications for shielding Trusted Apps or, more generally speaking, secure world processes against compromise through the contamination of the normal OS. Suppose a normal OS application is vulnerable and an attacker gets kernel space privileges. For off-the-shelf IoT Operating Systems the situation is devastating, as the attacker has full control over the Operating System. It can do arbitrary harm as all standard security mechanisms are subverted.

This is where program isolation comes to the rescue. When the normal OS is corrupted the separation between both worlds still guarantees that the security-critical functions are prevented from compromise. In our approach we will uncompromisingly pursue the design strategy of implementing security-critical programs and functions in the secure world. Looking ahead, we isolate the cryptographic algorithms and key material from the remainder program logic in the design of TEE-MQTTS crypto interface (Section 2.2) and the TEE-Wallet (Section 2.4). This way, the private keys to identify the IoT device or to sign a smart contract are sandboxed and protected against exposure.

2.1.4 Secure Storage

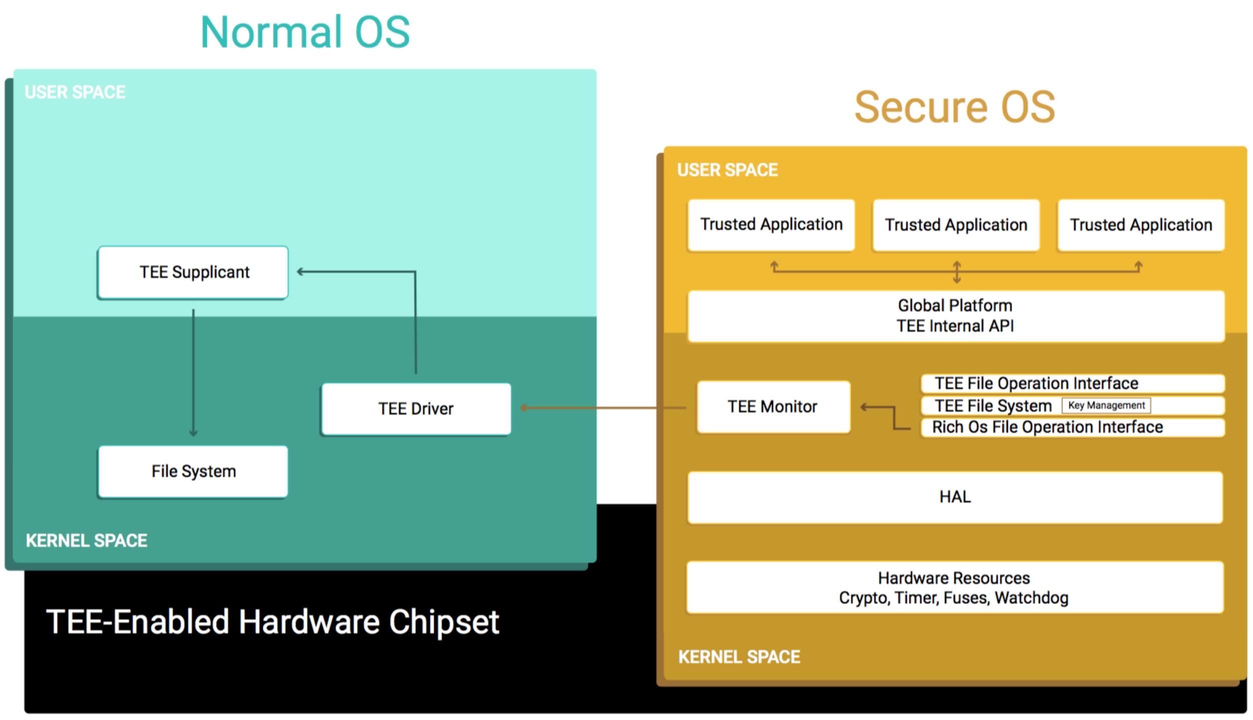

Figure 2: Secure Storage

One technological pillar in our architecture is a tamper proof mechanism to store and access the sensitive data. Several of our core technologies rely on cryptographic credentials. Being, for example, able to handle transactions in an autonomous way, brings a lot of risks if the private key to sign transactions is exposed. Secure storage makes sure that all the sensitive credentials are confidential and of integrity.

In this case, we rely heavily on utilizing the Trusted Execution Environment. Secure storage is a mechanism anchored in the secure OS. The only way to read, write and erase objects is through the mediation of a TA (see Fig. 2). To be a bit more precise, when the TA is calling the write function provided by Trusted Storage API to write data to a persistent object, a corresponding syscall implemented in TEE Trusted Storage Service will be called, which in turn will invoke a series of TEE file operations to store the data. TEE file system will then encrypt the data and send rich operating system file operation commands and the encrypted data to TEE supplicant by a series of RPC messages. TEE supplicant will receive the messages and store the encrypted data accordingly to the Linux file system. Reading files is handled in a similar manner.

Only properly authorized normal OS programs are able to read, write or delete those files, since every single one of these operations can only be executed through the secure worlds API (namely a "secure storage call"). To this end, the procedure described above is reverted and the object in the file system is decrypted in the secure world. Sometimes the decrypted object does not leave the secure world, but is left to the TA. Integrity is validated by only allowing two outcomes: success and failure. In case of a failure, the file will be reverted to its previous state. With these two features all of the sensitive information in the secure storage is not only protected from unauthorized access, but also protected from errors that occur due to wrong usage or general system-related faults.

2.1.5 Secure Boot

Annulling the booting process, say through the injection of some malicious code, is a very effective approach to circumvent device security. Attacks of that kind have tremendous appeal. They tackle the Achille's heel of any TEE-OS. A boot process is a delicate procedure, as the outcome defines the very initial state of the system. Deviations from the normal boot procedure entail a fragile system configuration. Core features like program isolation, secure storage and related TEE-OS technologies can be deactivated and vulnerable.

It is of utmost significance for a reliable deployment of the Trusted Execution Environment to keep track of the booting process. The notion of secure boot goes a step further. It measures the code modules involved in the booting process, each verifying the integrity of the successor. If at any point a module fails to verify the code integrity, the boot process rigorously aborts. A chain of trusted code executions arises, provided that the root in the chain is trustworthy. To this end, the root of trust of the Trustzone architecture is a piece of code, called Trusted ROM, being stored in read-only memory (ROM). Upon cold boot, the Trusted ROM measures the codebase of the secure world bootloader and verifies its integrity. If verification succeeds, the CPU loads the bootloader in the memory space allocated for the secure world, passes control over and executes the code. In the same vein, the remaining modules of the TEE-OS are verified and executed, including in consecutive order: the secure OS, Trusted Apps, the bootloader of the normal OS and the normal OS itself. A more technical explanation can be found in [2]. Through consistent integrity measurements the slightest malicious manipulation in any of the code bases will result in failed secure boot.

2.2 Secure and Highly Scalable IoT Communication

2.2.1 Publish-Subscribe Paradigm and Why TLS is inappropriate for EoT

The standard communication model on the Internet is the *request-response paradigm.* An initiator requests some content from a provider and the provider responds with the content or an error message. Protocols like HTTP [13], WebSocket [9], REST [7] and more recently CoAPP [3] utilize this model of communication.

In some cases the roles of content consumer and provider are not fixed. Rather a party can act both as consumer and provider. This is particularly true in various IoT scenarios. Recall the energy trading running example where a car is not only a consumer of energy, but also a producer. The car is able to buy energy, as a consumer, and sell energy to other consumers, as a producer. For example, when the car is parked for a longer period of time it could sell some of its charge during peak-times and recharge during low-cost times.

The *publish-subscribe paradigm* addresses the need for a more flexible role model. In a nutshell, in the publish-subscribe paradigm a party subscribes for some topic to a broker. Each time a peer publishes content matching the topic, the broker provides the content to the subscriber. This way, multiple IoT devices receive the content while the publisher sends the message only once. The broker behaves like a bulletin board and relays messages according to their topic to corresponding subscribers. Numerous message-oriented middle-layer protocols satisfying the requirements of the Internet of Things have been introduced recently. They include DSS [4], MQTT [6] (the lighter version MQTT-SN [5]) and AMQP [1]. DDS and MQTT/AMQP share some common principles, such as parsimony, efficiency and temporal decoupling. Each technology has unique features that make it most applicable for certain use cases. The main difference is the broker concept in MQTT/AMQP, while DDS is broker-free.

MQTT and AMQP provide no security by design. Rather it is suggested to run the protocol over the TLS protocol [8]. For years TLS has been the de-facto standard for securing application-layer protocols. It is the Swiss-army-knife of Internet security protocols when it comes to the necessity of confidential and mutually authenticated channel provisioning. The inherent problem with the TLS protocol is its design. It was developed to secure Web client and server communication, making it for several reasons a bad choice for IoT.

Securing a publish message, say in MQTT, costs at least 7 communication rounds. It is well known that every communication round massively drains the battery of an IoT device. A workaround is to artificially keep the TLS session through the underlying TCP socket alive (from our point of view is wrong). The downsides of the workaround are the continuous allocation of computational resources. The IoT device and the broker must reserve computing resources to keep the communication session alive. The latter is problematic, when thousands of sessions need to be handled. It simply does not scale in IoT settings when millions of devices connect to the broker.

IoT networks are known to have a high latency. Every additional protocol round leads to a significant delay of the message delivery and easily turns into a performance bottleneck. Various EoT scenarios, such as energy trading, cannot afford the delay. They demand for a communication with high throughput, typically in the range of VISA transactions or tweets.

TLS supports a large list of cryptographic algorithms (aka cipher suites) to be compatible with various client and server implementations. Some of them are known to be obsolete, but remain in the specification for compatibility reasons. This comes at the price of a large code base. However, space is a critical aspect for many IoT devices as it directly relates to the costs of the embedded device.

From a cryptographic perspective the TLS protocol (and its predecessor versions) is unnecessarily complex and does not follow the principal of "good" cryptographic protocol design. In fact, it was a longstanding open problem to formally analyze the exact protocol security [14, 22, 18]. Developed in the mid 90ties, TLS was continuously extended with the premise of downward compatibility. A blow up of the protocol specification led to a mess in many, widely distributed implementations (e.g. OpenSSL) because developers could hardly handle the intrinsic code complexity. This evolution among others was the stepping stone for devastating attacks, such as the Poodle, Beast and Breach [26]. While TLS in the present version 1.3 is not known to be vulnerable, its complexity has not been dramatically tamed.

2.2.2 Our TEE-MQTTS Protocol

With the aim to overcome the above limitations we designed our own protocol, dubbed TEE-MQTTS. It borrows from the MQTT protocol's message reliability and adds security similar to the TLS protocol with hardening through the means of the Trusted Execution Environment. The result is the astonishing TEE-MQTTS, a protocol that is more lightweight than MQTT over TLS with a higher level of security and a bit more technical. The differences are as follows:

- **Low Latency and High Throughput.** TEE-MQTTS requires only 3 rounds to achieve the goal of exchanging a session key, encrypting and authenticating a publish or subscribe message. The round complexity is optimal, as MQTT requires at least the same amount of rounds. The result is a low latency protocol ideal for usage demanding reliable message transfer with high throughput.
- **Lean Cryptography and Code Base.** The present protocol is lean. It relies on a single "cipher suite" taking into account lightweight and contemporary cryptographic algorithms. Unnecessary features including session resumption, re-negotiation or cipher suite change have been left out from the specification to ensure the protocol's leanness.
- **Security Hardening through TEE.** A critical point for the security of the messaging protocol is when the IoT device is compromised. The adversary is then capable of intercepting, altering and forging the message content. The latter is particularly worrisome as the adversary has access to the long-term cryptographic secrets linked to the device's identity, allowing the adversary to impersonate the device (by simply copying the cryptographic key material). To mitigate the threat, we strictly isolate the cryptographic algorithms and key material from the rest of the protocol logic, leverag-

ing the program isolation feature of the TEE. This way, we make sure that in case of a normal OS compromise, no access to the cryptography is possible.

2.2.3 System Architecture

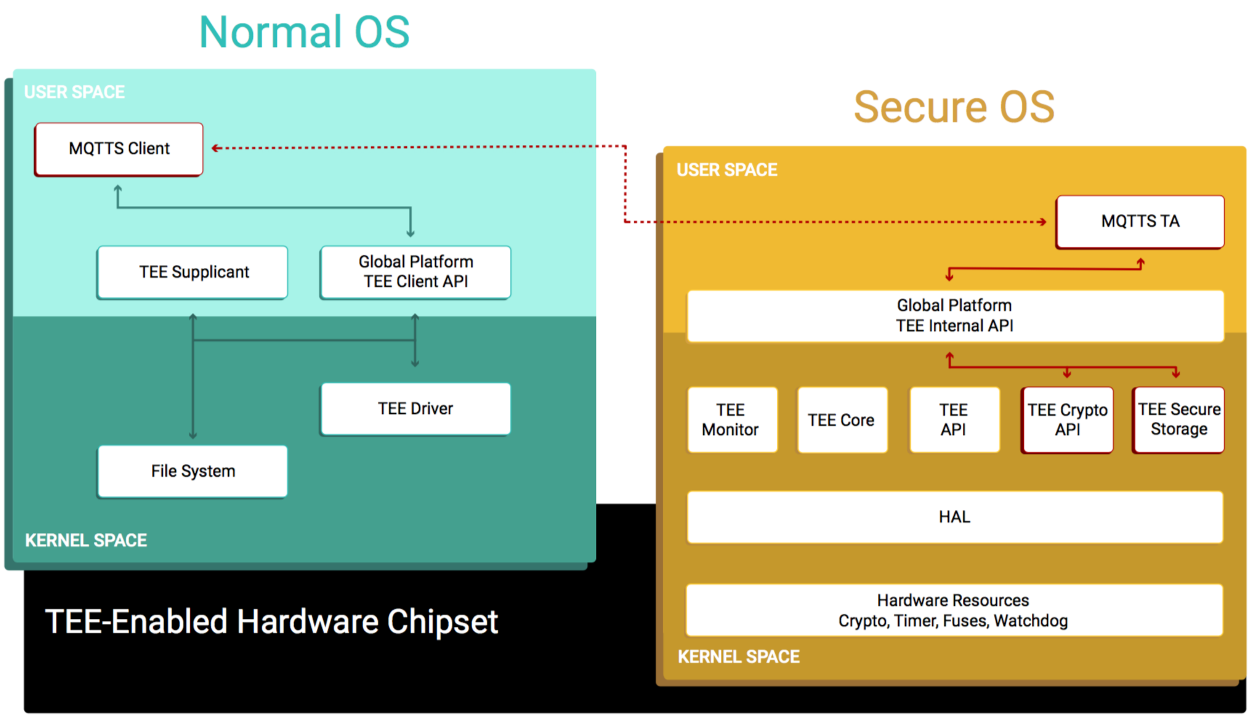

Figure 3: TEE-MQTTS Architecture. Red marks MQTTS specific components.

Fig. 3 illustrates the system architecture for the establishment of an end-to-end secure messaging mechanism between the different participating devices of the Economy of Things that we envision. The core of the idea is to isolate the non-critical system modules from the critical ones. The non-critical modules comprise the standard networking protocols, containing the whole protocol stack from the physical layer to the transport layer. They run in the normal world. The critical components are those that are responsible for executing the secure messaging protocol. They run in the secure world of the IoT device and include the following modules:

- the *TEE Crypto API* implements the cryptographic algorithms for the MQTTS secure communication protocol and the *TEE Secure Storage* stores the cryptographic key material like the long-term cryptographic device identity. We remark that the design is modular. Both modules are as well implementable in a trusted hardware element to increase security against physical attacks.

- the *MQTTS Trusted Application* receives authorized instructions from the normal worlds' *MQTTS client*, for example, to decrypt or encrypt a protocol message. These operations are performed by communicating with the *TEE Crypto API* and the *TEE Secure Storage.*

2.2.4 Specification of the Secure Messaging Protocol

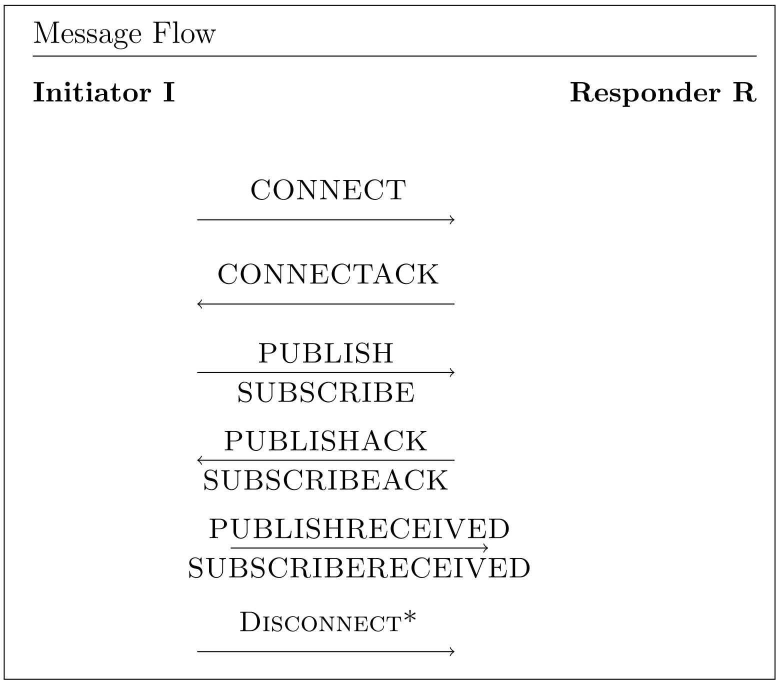

Figure 4: Overview of the Secure Messaging Protocol Flow.

The secure messaging protocol compromises the following twelve subprotocol messages, illustrated in Fig. 4:

1. CONNECT: This message initiates the secure messaging protocol.

2. CONNECTACK: This message acknowledges the initiation of secure messaging.

3. PUBLISH: This message sends a publish message in a private and authenticated. The purpose of the message is to publish content for a topic. It contains the topic name, content and desired quality of service, abbreviated as the message *pubm*.

4. PUBLISHACK: This message acknowledges the reception of the publish message with a flag $pflag$. If $pflag = 0$, it is guaranteed that the message was delivered one time at most. If $pflag = 1$, it is guaranteed that a message was delivered at least once to the receiver. If $pflag = 2$, then it is guaranteed that the message was delivered only once. If $pflag = \perp$, then a transmission error occurred (and an error code is attached).

5. PUBLISHRECEIVED: This publish received message is the response to a PUBLISHACK where the $pflag = 2$. This guarantees that a message is received only once.

6. SUBSCRIBE: This message sends a subscription message in a private and authenticated way. The purpose of the message is to subscribe to a topic (or set thereof). The subscription request contains a list of topic name and desired quality of service, abbreviated by message *unsubm*.

7. SUBSCRIBEACK: This message acknowledges the reception of the unsubscription message with a flag $uflag$. If $uflag = 0$, then it is guaranteed that the message was delivered one time at most. If $uflag = 1$, it is guaranteed that a message was delivered at least once. If $uflag = 2$, then it is guaranteed that the message was delivered only once. If $uflag = \perp$, then an unsubscription error occurred (and an error code may be attached).

8. SUBSCRIBERECEIVED: This subscribe received message is the response to a SUBSCRIBEACK where the $uflag = 2$. This guarantees that a message was received only once.

9. UNSUBSCRIBE: This message sends an unsubscribe message in a private and authenticated way. The purpose of the message is to unsubscribe from a topic (or list thereof). The unsubscription request contains an arbitrary number of topics the sender wishes to unsubscribe from.

10. UNSUBSCRIBERECEIVED: This unsubscribe received message is the response to a SUBSCRIBEACK where the $sflag = 2$. This guarantees that a message was received only once.

11. UNSUBSCRIBEACK: This message acknowledges the unsubscription message with a flag $uflag$. If $sflag = 0$, then it is guaranteed that the message was delivered at most one time. If $sflag = 1$, it is guaranteed that a message was delivered at least once. If $sflag = 2$, then it is guaranteed that the message was delivered only once. If $sflag = \perp$, then a subscription error occurred (and an error code may be attached).

12. DISCONNECT: This final message indicates that an initiator I disconnecting cleanly from the network.

2.2.5 Notation

Before describing in detail the protocol messages we will need to introduce some notation. Let $\mathbb{G}$ be a group of prime order p and g the generator of the group. Let $(\mathsf{Sig}, \mathsf{Vf})$ be the signing and verification algorithms of a digital signature scheme. Let $(\mathsf{Enc},\mathsf{Dec})$ be the encryption and decryption algorithms of a symmetric-key authenticated encryption scheme. Further we assume that party i obtained a digital certificate $\mathsf{cert} = (\mathsf{vk}_i, \sigma_i^{CA})$ where σ_i^{CA}

is a digital signature issued by a trusted third (i.e. verifiable under the publicly available verification key vk_{CA}) over the party's signature verification key vk_i.

2.2.6 Protocol CONNECT

Secure Messaging Establishment Protocol

Common Input : vk_{CA}

Initiator's input: $\mathsf{sk}_I, \mathsf{cert}_I = (\mathsf{vk}_I, \sigma_I^{CA})$

Responder's input: $\mathsf{sk}_R, \mathsf{cert}_R = (\mathsf{vk}_R, \sigma_R^{CA})$

Initiator I **Responder R**

.... CONNECT ....

$x \leftarrow_{\$} \mathbb{Z}_p$

$\sigma_I \leftarrow \mathsf{Sig}(sk_I, g^x)$

$$\xrightarrow{g^x, \sigma_I, \mathsf{cert}_I}$$

parse cert_I as $(\mathsf{vk}_I, \sigma_I^{CA})$

abort, if

(1) $\mathsf{Vf}(\mathsf{vk}_{CA}, vk_I^{CA}, \sigma_I^{CA}) \neq 1$

(2) $\mathsf{Vf}(\mathsf{vk}_I, g^x, \sigma_I) \neq 1$

else, respond with

CONNECTACK

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. the initiator I chooses a random integer $x \leftarrow_{\$} \mathbb{Z}_p$
2. it generates a digital signature $\sigma_I \leftarrow \mathsf{Sig}(sk_I, g^x)$ by running the digital signature algorithm Sig on input its secret key sk_I and message g^x.
3. the initiator sends g^x, σ_I and his digital certificate cert_I to the responder
4. upon reception of the tuple $(g^x, \sigma_I, \mathsf{cert}_I)$, the responder R verifies
   (a) the validity of cert_I by running the verification algorithm Vf with the trusted third party's verification key vk_{CA} over the initiator's verification key vk_I.

(b) the validity of the tuple (g^x, σ_I) by running the verification algorithm Vf with the initiator's verification key vk_I over g^x.

If any of the checks fails, the responder aborts the protocol, by sending a DISCONNECT message. Otherwise, it continues to respond with a CONNECTACK message.

2.2.7 Protocol CONNECTACK

Secure Messaging Acknowledgement Protocol

Common Input : vk_{CA}
Initiator's input: $\mathsf{sk}_I, \mathsf{cert}_I = (\mathsf{vk}_I, \sigma_I^{CA})$
Responder's input: $\mathsf{sk}_R, \mathsf{cert}_R = (\mathsf{vk}_R, \sigma_R^{CA})$

Initiator I **Responder R**

.... CONNECTACK ....

Responder R:
$y \leftarrow_\$ \mathbb{Z}_p$
$\sigma_R \leftarrow \mathsf{Sig}(sk_R, g^y)$
set $\mathsf{k} = g^{xy}$

$$\xleftarrow{g^y, \sigma_R, \mathsf{cert}_R}$$

Initiator I:
abort, if
(1) $\mathsf{Vf}(\mathsf{vk}_{CA}, vk_R^{CA}, \sigma_I^{CA}) \neq 1$
(2) $\mathsf{Vf}(\mathsf{vk}_R, g^y, \sigma_R) \neq 1$
else, set $\mathsf{k} = g^{xy}$
(optionally) respond with
PUBLISH or SUBSCRIBE

Description. The protocol runs between the receiver R and initiator I and consists of following steps:

1. the responder R chooses a random integer $y \leftarrow_\$ \mathbb{Z}_p$
2. it generates a digital signature $\sigma_R \leftarrow \mathsf{Sig}(sk_R, g^y)$ by running the digital signature algorithm Sig on input its secret key sk_R and message g^y.
3. it computes the temporal key $\mathsf{k} = g^{xy}$.
4. the responder sends g^y, σ_R and his digital certificate cert_R to the responder

5. upon reception of the tuple $(g^y, \sigma_R, \mathsf{cert}_R)$, the initiator I verifies
    (a) the validity of cert_R by running the verification algorithm Vf with the trusted third party's verification key vk_{CA} over the responder's verification key vk_R.
    (b) the validity of the tuple (g^y, σ_R) by running the verification algorithm Vf with the responder's verification key vk_R over g^y.

    If any of the checks fails, the initiator aborts the protocol. Otherwise, it computes the temporal key $\mathsf{k} = g^{xy}$. It also may continue to respond with a SUBSCRIBE or PUBLISH message.

2.2.8 Protocol PUBLISH

Secure Publication Protocol

Common Input : Label l_1

Initiator's input: k and publication message *pubm*

Responder's input: k

Initiator I		**Responder R**
....	PUBLISH	....
$ctx_1 \leftarrow_\$ \mathsf{Enc}(\mathsf{k}, l_1 \vert pubm)$	$\xrightarrow{ctx_1}$	
		$(\alpha, pubm) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_1)$
		abort, if $\alpha \neq l_1$
		else store *pubm*
		and (optionally) respond with
		PUBLISHACK or terminate

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The initiator I encrypts his message *pubm* with $ctx_1 \leftarrow_\$ \mathsf{Enc}(\mathsf{k}, l_1|pubm)$ where l_1 is a pre-defined label.

2. Initiator I sends the ctx_1 to the responder R.

3. The responder R decrypts the ctx_1 with his k by running $(\alpha, pubm) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_1)$.
    - If $\alpha \neq l_1$ the responder R aborts with an error.

- Else if $\alpha = l_1$ the responder R stores $pubm$ and responds with a PUBLISHACK message or it terminates.

2.2.9 Protocol PUBLISHACK

Secure Publication Acknowledgment Protocol

Common Input : Label l_2

Initiator's input: k

Responder's input: k and publication flag $pflag$

Initiator I **Receiver R**

....PUBLISHACK..

$ctx_2 \leftarrow \mathsf{Enc}(\mathsf{k}, l_2|pflag)$

$$\xleftarrow{ctx_2}$$

$(\beta, pflag) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_2)$

abort, if $\alpha \neq l_2$

else respond with with PUBLISHRECEIVED,

DISCONNECT or terminate

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The responder R encrypts his message $pflag$ with $ctx_2 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_1|pubm)$ where l_2 is a pre-defined label.

2. Responder R sends the ctx_2 to the initiator I.

3. Upon reception, the initiator I decrypts ciphertext ctx_2 with his key k by running $(\alpha, pflag) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_2)$.

    - If $\alpha \neq l_2$ the initiator I aborts with an error.
    - Else if $\alpha = l_2$, the initiator I parses $pflag$. Depending on the value of $pflag$, the initiator (optionally) runs PUBLISHRECEIVED, DISCONNECT or terminates.

2.2.10 Protocol PUBLISHRECEIVED

Secure Publication Reception Protocol

Common Input : Label l_3

Initiator's input: k

Responder's input: k

Initiator I **Responder R**

.... PUBLISHRECEIVED ....

$ctx_3 \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{k}, l_3)$ $\xrightarrow{ctx_3}$

$\alpha \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_3)$

abort, if $\alpha \neq l_1$

else terminate

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The initiator I encrypts l_3 with $ctx_3 \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{k}, l_3)$ where l_3 is a pre-defined label.
2. Initiator I sends the ctx_3 to the responder R.
3. The responder R decrypts the ciphertext ctx_3 with his k by running $\alpha \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_3)$.
   - If $\alpha \neq l_3$ the responder R aborts with an error.
   - Else if $\alpha = l_3$ the responder R terminates.

2.2.11 Protocol SUBSCRIBE

Secure Subscription Protocol

Common Input : Label l_4

Initiator's input: k and subscription $subm$

Responder's input: k

Initiator I **Responder R**

....SUBSCRIBE..

$ctx_4 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_4|subm)$ $\xrightarrow{ctx_4}$

$(\alpha, subm) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_4)$

abort, if $\alpha \neq l_4$

else store $subm$

and (optionally) respond with

SUBSCRIBEACK or terminate

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The initiator I encrypts his subscription request $subm$ with $ctx_4 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_4|subm)$.
2. The initiator I sends the ciphertext ctx_4 to the responder R.
3. The responder R decrypts the ciphertext ctx_4 with his key k by running $(\alpha, subm) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_4)$.
   - If $\alpha \neq l_4$ the responder R aborts with an error.
   - Else, if $\alpha = l_4$ the responder R stores $subm$ and responds with a SUBSCRIBEACK message or terminates.

2.2.12 Protocol SUBSCRIBEACK

Secure Subscription Acknowledgment Protocol

Common Input : Label l_5

Initiator's input: k

Responder's input: k and subscription flag $sflag$

Initiator I | **Receiver R**

....................................SUBSCRIBEACK....................................

$ctx_5 \leftarrow \mathsf{Enc}(\mathsf{k}, l_5|sflag)$

$$\xleftarrow{\quad ctx_5 \quad}$$

$(\alpha, sflag) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_5)$

abort, if $\alpha \neq l_5$

else respond with with SUBSCRIBERECEIVED,

DISCONNECT or terminate

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The responder R encrypts his message $sflag$ with $ctx_5 \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{k}, l_5|sflag)$ where l_5 is a pre-defined label.

2. Responder R sends the ciphertext ctx_5 to the initiator I.

3. Upon reception, the initiator I decrypts ciphertext ctx_5 with his key k by running $(\alpha, sflag) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_5)$.

   - If $\alpha \neq l_5$ the initiator I aborts with an error.
   - Else if $\alpha\ = l_2$, the initiator I parses $sflag$. Depending on the value of $sflag$, the initiator (optionally) runs SUBSCRIBERECEIVED, DISCONNECT or terminates.

2.2.13 Protocol SUBSCRIBERECEIVED

Secure Subscription Reception Protocol

Common Input : Label l_6

Initiator's input: k

Responder's input: k

Initiator I | **Responder R**

.... SUBSCRIBERECEIVED ....

$ctx_6 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_6)$ $\xrightarrow{ctx_6}$

$\alpha \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_6)$

abort, if $\alpha \neq l_6$

else terminate

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The initiator I encrypts l_6 with $ctx_6 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_6)$ where l_6 is a pre-defined label.
2. Initiator I sends the ciphertext ctx_6 to the responder R.
3. The responder R decrypts the ciphertext ctx_6 with his k by running $\alpha \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_6)$.
   - If $\alpha \neq l_6$ the responder R aborts with an error.
   - Else if $\alpha = l_6$ the responder R terminates.

2.2.14 Protocol UNSUBSCRIBE

Secure Unsubscription Protocol

Common Input : Label l_7

Initiator's input: k and unsubscription *unsubm*

Responder's input: k

Initiator I **Responder R**

. . . . UNSUBSCRIBE . . . .

$ctx_7 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_7|unsubm)$ $\xrightarrow{ctx_7}$

$(\alpha, unsubm) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_7)$

abort, if $\alpha \neq l_7$

else unsubscribe *unsubm*

and (optionally) respond with

UNSUBSCRIBEACK or terminate

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The initiator I encrypts his unsubscription request *unsubm* with $ctx_7 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_7|unsubm)$.

2. The initiator I sends the ciphertext ctx_7 to the responder R.

3. The responder R decrypts the ciphertext ctx_7 with his key k by running $(\alpha, unsubm) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_7)$.

   - If $\alpha \neq l_7$ the responder R aborts with an error.
   - Else, if $\alpha = l_7$ the responder R unsubscribes *unsubm* from the list of topics associated with that party. It responds with an UNSUBSCRIBEACK message or terminates.

2.2.15 Protocol UNSUBSCRIBEACK

Secure Unsubscription Acknowledgment Protocol

Common Input : Label l_8

Initiator's input: k

Responder's input: k and unsubscription flag $uflag$

Initiator I **Receiver R**

.... UNSUBSCRIBEACK ....

$ctx_8 \leftarrow \mathsf{Enc}(\mathsf{k}, l_8 | uflag)$

$$\xleftarrow{\quad ctx_8 \quad}$$

$(\alpha, uflag) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_8)$

abort, if $\alpha \neq l_8$

else respond with with UNSUBSCRIBERECEIVED,

DISCONNECT or terminate

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The responder R encrypts his message $uflag$ with $ctx_8 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_8 | uflag)$ where l_8 is a pre-defined label.

2. Responder R sends the ciphertext ctx_8 to the initiator I.

3. Upon reception, the initiator I decrypts ciphertext ctx_8 with his key k by running $(\alpha, uflag) \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_8)$.

   - If $\alpha \neq l_8$ the initiator I aborts with an error.
   - Else if $\alpha = l_8$, the initiator I parses $uflag$. Depending on the value of $uflag$, the initiator (optionally) runs UNSUBSCRIBERECEIVED, DISCONNECT or terminates.

2.2.16 Protocol UNSUBSCRIBERECEIVED

Secure Unsubscription Reception Protocol

Common Input : Label l_9

Initiator's input: k

Responder's input: k

Initiator I | **Responder R**

.... UNSUBSCRIBERECEIVED ....

$ctx_9 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_9) \quad \xrightarrow{ctx_9}$

$\alpha \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_9)$

abort, if $\alpha \neq l_9$

else terminate

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The initiator I encrypts l_9 with $ctx_9 \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_9)$ where l_9 is a pre-defined label.
2. Initiator I sends the ciphertext ctx_9 to the responder R.
3. The responder R decrypts the ciphertext ctx_9 with his k by running $\alpha \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_9)$.
   - If $\alpha \neq l_9$ the responder R aborts with an error.
   - Else if $\alpha = l_9$ the responder R terminates.

2.2.17 Protocol DISCONNECT

Secure Disconnection Protocol

Common Input : Label l_{10}

Initiator's input: k

Responder's input: k

Initiator I		**Responder R**
	......................DISCONNECT......................	
$ctx_{10} \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_{10})$	$\xrightarrow{ctx_{10}}$	
		$\alpha \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_{10})$
		abort, if $\alpha \neq l_{10}$
		else release I

Description. The protocol runs between the initiator I and responder R and consists of the following steps:

1. The initiator I encrypts the label l_{10} with $ctx_{10} \leftarrow_{\$} \mathsf{Enc}(\mathsf{k}, l_{10})$.
2. Initiator I sends the ciphertext ctx_{10} to the responder R
3. The responder R decrypts the ciphertext ctx_{10} with his key k by running $\alpha \leftarrow \mathsf{Dec}(\mathsf{k}, ctx_{10})$.
    - If $\alpha \neq l_{10}$ the responder R aborts with an error.
    - Else if $\alpha = l_{10}$ the responder R releases the connection to I.

2.3 Testified IoT Data

2.3.1 Non-Falsifiability of Data and Why Attestation is insufficient

The bare bones of an Economy of Things is that data is sensed and turned into a digital asset. Technically what happens is the IoT device, in simple terms, executes a program P over some (sensor) input x, denoted as $P(x)$. In general, x is something the device *knows*, *has*, or *does*. Recall the energy charging running example. In which case, P measures the current and shuts the charging relay of the induction loop, after x electrons passed the power line. Or recall the supply chain use case. P is the program instructing a sensor to measure the temperature x in the container throughout the journey. It is exactly this data harvesting x and processing P that creates the digital asset $P(x)$.

The problem is IoT devices, like commodity computing devices, are subjected to attack. These days, plenty of IoT-specific attack methods exist, which vary in technical sophistication and adversarial effort. In the realm of IoT one normally differentiates software and physical attacks. The latter requires as the name suggests, physical access to the device. One assumes the adversary is capable of reading out the memory (or any storage medium) [17], inject faults [?], or observe leakage of sensitive data through side-channels like timing behavior, power consumption or electromagnetic emission [20, 21, 24]. Physical device access is, in most use cases we are interested in, uncommon and hardly feasible. It is quite realistic to assume the attacker exploits software vulnerabilities through remote device penetration. The intention is to inject some malicious (shell)code into the run-time execution of the vulnerable software and claim control over the Operating System and its resources. Attacks of this type are known as buffer overflows, return-to-libc and return-oriented programming [10, 28], to name a few. They take advantage of a weak programming model that lacks mechanisms to validate inputs, like the length of a string stored in a buffer.[4]

Once the attacker has corrupted the system, the capabilities are almost infinite. Depending on the exploited vulnerability, the attacker acquires privileges either at the user or kernel space. Trusted Execution Environments only partially curtail the attack. Suppose the attacker exploited a software weakness in the normal OS, such as a bug in the implementation of the TCP stack like CVE-2017-9445, and after a sequence of post exploitations finally gained root privileges. It is true that all the functions in the secure world are secure. The hardware-empowered isolation mechanism (cf. Section 2.1.3) prevents the attacker despite normal OS root privileges to carry over the attack and contaminate the secure OS. It is also true that the boot chain—at least until the point of loading the normal OS bootloader—is untampered. The reasons are the secure boot mechanisms (cf. Section 2.1.4). The attacker is infeasible of replacing the bootrom and falsifying the secure world bootstrapping. To do so, the attacker needs either physical access to the device or the capabilities to sign a malicious component of the secure world booting procedure [?]. The Trusted Execution Environment confines the harm to the normal world.

[4]The canonical example of a programming language developers are responsible for manual input validation mechanisms and thus a splendid candidate susceptible to the attacks is C. It is worth noting the C language also is the preferred language for programming IoT devices.

Nevertheless the consequences of the security breach are devastating. It is possible to manipulate the execution of normal world programs. Recalling the above running example, by manipulating program P it is possible to come up with a measurement of x' electrons, where $x' \gg x$ is much larger than the factual energy charge. In other words, the adversary delivered less energy than claimed. Clearly this is an unfair situation for the buyer of the energy and can lead to a dispute. A similar situation arises in the supply chain use case. By modifying program P it is possible to fake the temperature measurement x to x' where $x' \ll x$ may be below the negotiated average temperature. As in the previous case the situation for the buyer of the container is unbearable and unacceptable, as the delivered goods will be of a lower quality than agreed upon in the (smart) contract.

The reader might suggest to implement the program P as a Trusted App in the secure world. This way, the secure world shields the program P from compromise and harvesting wrong inputs x. The approach falls short of expectations. It necessitates not only the implementation of program P as a Trusted App, but also all the software modules to interact with the sensor hardware and related peripheries like drivers, communication protocols and kernel functions. This leads to a significant increase of the secure worlds code base. In fact, the approach turns the lean secure world OS into a rich Operating System. Tied to that is the heritage of considerably increased vulnerability susceptibility. More importantly, the approach provides no answers to the actual problem of proving to a third party the non-falsifiability of the IoT data. All one is left with is the plain belief in the soundness of the secure world. However, mechanisms to ascertain the quality of data are still lacking.

Recently, code attestation with support of a trusted hardware element (e.g., Trusted Platform Module) have gained much attention in the context of IoT. Prominent examples include Microsoft's Bletchley crypto fabric [16]. The idea is to cryptographically prove to a third party that some code is trusted. Before some program P is executed, the trusted hardware element is asked to measure the integrity of the code. A third party deduces from the signed measurement any code tampering. Although the approach goes obviously in the right direction, it still misses the point of attesting the data. As argued before, the program code at *runtime* can be modified giving a biased view of the data x. A sufficient requirement to mount the data forgery is the existence of software's vulnerability. It must not necessarily be the attested program itself. Once the attacker managed to gain control over the Operating System it can make arbitrary changes to the execution of the attested program.

2.3.2 Our Testimony

A testimony remedies the problem. In a nutshell, a testimony is a cryptographic proof over the execution of some program. In contrast to code attestation techniques being static by nature, a testimony is dynamic and linked to a particular process (program at runtime) including inputs from and to other peripherals or processes. With a testimony any proof verifying party identifies the presence of real-time attacks intending to disrupt the desired program flow. A party receiving a valid testimony has the cryptographic guarantee that the only way the data x was computed is by executing the program P on input x. A testimony scheme satisfies the following for our application's essential properties:

Completeness. Like all cryptographic proof systems, a honestly generated testimony verifies (with overwhelming probability) to true. Completeness informally says that a valid testimony implies the testified program has been executed in the expected way and no deviation from the program flow occurred.

Soundness. This property captures the security of a testimony. A malign testimony verifies (with overwhelming probability) to false. Soundness informally says that it is hard to forge a valid testimony without having executed the program P on input x.

Run-Time Verifiability. Although various non-interactive proof systems, such as NIZKs or SNARKs, exist for relevant languages (most notably $\mathcal{NP}$ and $\mathcal{P}$) and thus propound tools to verify a polynomial-time computation, the statement a testimony proves is different and cannot be proved with classical proof systems. A bit more precise, (non-interactive) proof systems relate to the computation of some *function* described, for example, as (quadratic) arithmetic program [15]. The notion of a testimony targets the execution of a *program* described in instruction code. Hence it is at the bare bone of the execution model of computers. To make the difference clearer. With a valid NIZK or SNARK proof the verifier gets the guarantee that the *result of a function* (expressed as a program) is correct whereas the testimony gives a stronger guarantee, namely that the *execution of a program through a sequence of machine instructions* is correct. The latter is of particular relevance for the non-falsifiability of data, as the instruction set of today's computing architectures goes far beyond arithmetic operations.

The Idea. The Trusted Execution Environment features the effective separation between execution environments. For example, the Trustzone-enabled TEE supports two compartments known as the normal and secure world, respectively. The underlying hardware architecture makes sure that the secure world processes are isolated from normal world processes. This means, no normal world program—be it benign or be it malign—can access and alter a secure world program. Vice versa, a secure world program has higher privileges. It can interrupt a normal world program, read the memory area where the program execution takes place and alter the program flow. It is exactly these higher privileges that we take advantage of to compute the testimony.

At the lowest abstraction level a program P executed on input x is a sequence of instructions readable by the central processor unit known as opcode. Machine instructions are stored in the memory. The processor reads the instructions from the memory cells, processes them, writes the output back to a memory cell and/or jumps to the next instruction. It may temporally store some of the outcomes in its internal memory called the register and caches. To verify the execution of a program our idea is to capture the program execution through (internal and external) memory traces. Informally a trace is a memory snapshot (or, more specifically, a snapshot over particular memory cells) at a given execution time. The execution time is typically triggered through events we refer to as *breakpoints.* To testify the program execution we ask the secure world to do the snapshotting. It computes a digital signature over the execution traces. The outcome of the digital signature scheme is the testimony.

2.3.3 System Architecture

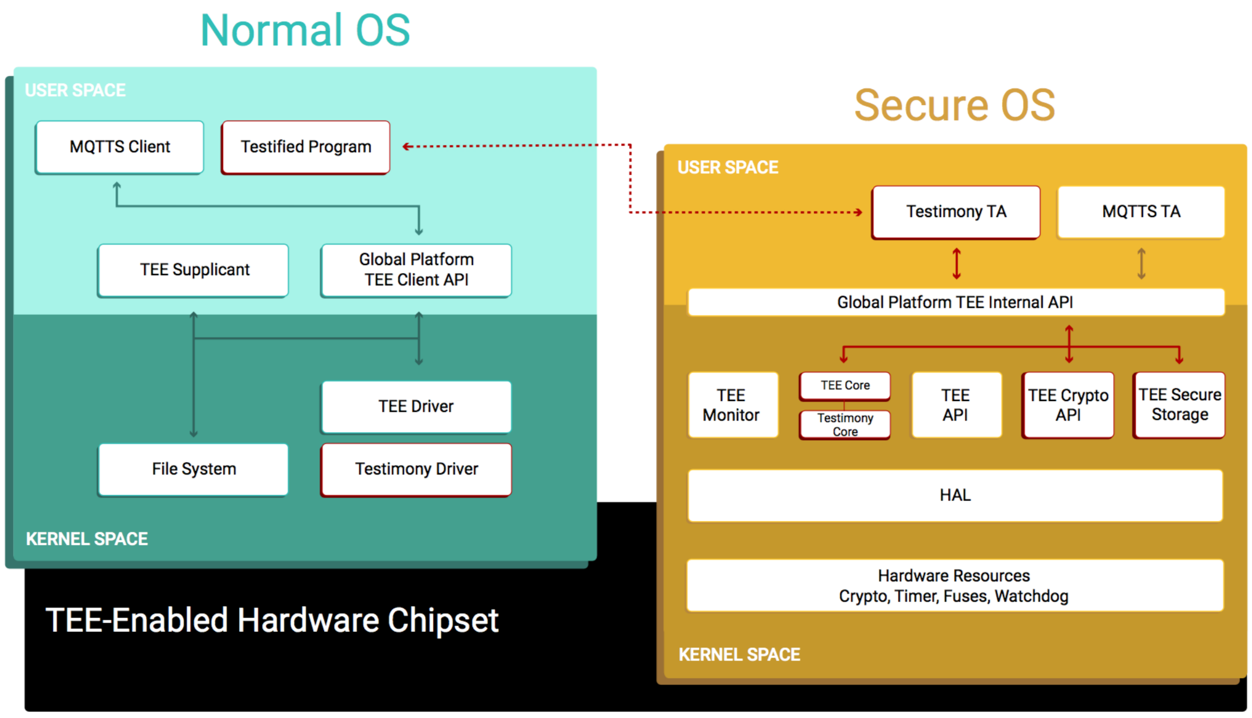

Figure 5: Testimony Architecture. Red marks specific components.

Figure 5 illustrates the system architecture of the modules participating in the program's measurement and creation of a corresponding testimony. The idea is again to separate the critical from the non-critical system modules. To make sure that the measurement itself will not be tampered with, the *Testimony Trusted Application* is located in the secure world of our system, as follows:

- the *Testimony Trusted Application* makes use of the *TEE core* functions augmented with the *Testimony Core* to access through the normal world *Testimony Driver* the memory regions of the compiled program P' objected to be testified.
- the *TEE crypto API* provides the crypto algorithms that are needed in order to create the testimony. The normal world is not included into this security-critical process in order to eliminate the risks of tampering.

- the *TEE secure storage* save the testimony under a particular identifier only accessible to authorized Trusted Applications for future processing. Looking ahead the testimony will be made accessible to the TEE Wallet for Blockchain-based smart contracting.

2.3.4 Notation

We describe a testimony scheme Θ as a tuple of three polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Testify}, \mathsf{Verify})$, such that

- the parameter generation algorithm Gen on input a security parameter 1^n and a (description of a program) P, outputs an evaluation key ek_P and a verification key vk_P.
- the testimony generation algorithm $\mathsf{Testify}$ on input the evaluation key ek_P and some input x, outputs a testimony τ for program P.
- the verification algorithm Verify on input the verification key vk_P, some input x and a testimony τ, outputs a bit b where $b = 1$ means a valid, and $b = 1$ means invalid.

Let $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a digital signature scheme and $H : \{0,1\}^* \to \{0,1\}^n$ be a cryptographic hash function mapping arbitrary length inputs to fixed-length digests.

We also define a family of compilers $\mathcal{C}_\Phi : \mathcal{P} \xrightarrow{\Phi} \mathcal{P}$ mapping a program $P \in \mathcal{P}$ into a program with breakpoints $P' \in \mathcal{P}$ where Φ is the function defining when a snapshot has to take place and over what memory cells the snapshot has to be taken. The function Φ injects breakpoints making it possible to snapshot the whole memory of the program or only certain code fragments. It is likewise possible to snapshot the memory only at the beginning of the program execution yielding a *static* code testimony or at any execution step yielding a *dynamic* code testimony. We write $\Phi(P)$ to denote the set of memory snapshots obtained from applying Φ to the program P and write $\Phi_i(P)$ for the snapshot at the i^{th} step.

2.3.5 Specification of the Testimony Scheme

When the normal world executes a compiled version of the program, that is the program with breakpoints P', on input x the following happens as illustrated in Fig. 6:

Step (1-2) program P' executes the machine-code instructions as the original program P does until a breakpoint, for short BP, is reached.

Step (3) the breakpoint event triggers the processor to execute a snapshotting program in the secure world. The snapshotting program is in a memory region exclusively accessible by the secure normal with the privilege to access the memory cells of P' in the normal world.

Step (4) The snapshotting program computes the snapshot as defined in Φ. The i^{th} snapshot is temporarily stored as $\Phi_i(P)$.

Step (5) After taking the snapshot the snapshotting program instructs the processor to return to the execution of the normal world program P'.

This procedure repeats until program P' terminates.

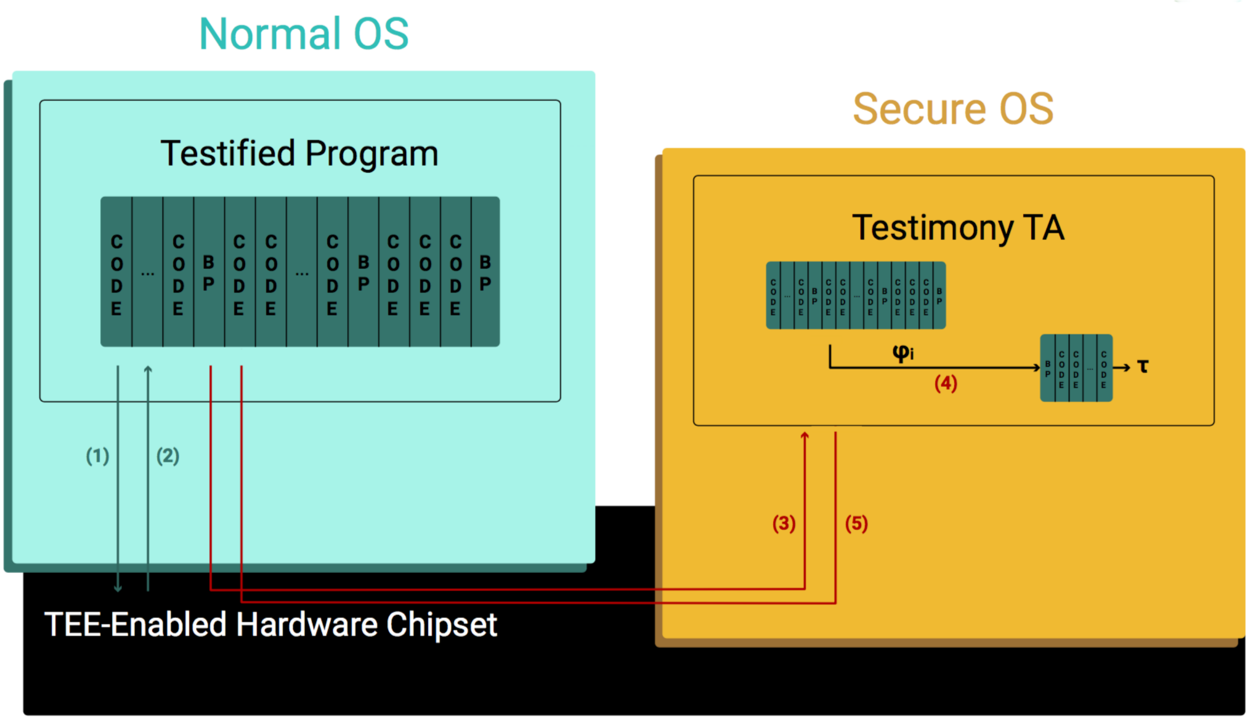

Figure 6: Illustration of the steps to snapshot a program

Computing the Testimony. Define the testimony scheme $\Theta = (\mathsf{Gen}, \mathsf{Testify}, \mathsf{Verify})$ as

Gen: On input the security parameter 1^n and the description of the program P, do:

1. Run the key generation of the signature scheme Σ to obtain a signing key and a verification key, that is $(\mathsf{ssk}, \mathsf{svk}) \leftarrow \Sigma.\mathsf{KeyGen}(1^n)$.
2. Run the Compiler $\mathcal{C}$ on Program P and snapshotting function Φ to derive the program with breakpoints P'.
3. Return the evaluation key $\mathsf{ek}_P = (\mathsf{ssk}, H, P')$ and the verification key $\mathsf{vk}_P = (\mathsf{svk}, H, \Phi, P)$.

$\mathsf{Testify}$: On input the evaluation key $\mathsf{ek}_P = (\mathsf{ssk}, H, P')$ and input x, do:

1. Run program P' on input x in the normal world. For the i^{th} snapshot in the secure world, store the hash of the snapshot, that is $h_i = H(h_{i-1}||\Phi_i(P))$ where the input to the hash function H is the previously stored digest value h_{i-1} concatenated with the present memory snapshot $\Phi_i(P)$.
2. Suppose $\Phi(P)$ consists of n snapshots. Return the testimony τ where $\tau \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}, h_n)$ is the signature computed over the last digest value h_n with the secret key ssk.

Verify: On input a verification key $\mathsf{vk}_P = (\mathsf{svk}, H, \Phi, P)$ the input x and a testimony τ, do:

1. Compile P to obtain the program with breakpoints $\mathcal{C}_\Phi(P) = P'$.
2. Run $P'(x)$ and obtain the digest h_n.
3. Accept the testimony if $\Sigma.\mathsf{Verify}(\mathsf{svk}, h_n, \tau) = 1$. Else reject the testimony.

Extension to Multiple Programs Our approach extends to the testimony of multiple programs $P_1, \ldots, P_n$ with differing inputs $x_1, \ldots, x_n$ potentially interacting with each other as well. Apply the compiler method $\mathcal{C}_\Phi(P_i)$ for $i = 1, \ldots, n$ to every program P_i and make sure breakpoints lead to the same snapshotting program.

2.4 Trusted Blockchain IoT Wallet

Cryptocurrencies are a technology of growing attraction. They are bridging the gap between the trade of digital assets and a monetization mechanism. More and more individual users, businesses and governments are delving into the depths of crypto coins and exploring applications with the same underlying concept: an immutable and publicly verifiable record of transactions known as the ledger. This ledger has one fundamentally different characteristic to existing coin-based payment methods. In order to authorize the transfer of crypto coins, in most Blockchain technologies utilizing a smart contract, one must first prove ownership of the crypto coins, either through a signature or some other (zero-knowledge) proof method.

This ownership proof bears risks. Impersonation of the owner is a dramatic consequence if the private (signing) keys are not secured. In fact, one of the greatest fears of cryptocurrency traders is to have their wallets hacked. That these hacks are real and effective has the recent parity wallet attack demonstrated. It amounted to the theft of crypto coins worth a staggering 32 million dollars[5]. As financial rewards are obviously a very strong incentive, we will observe a tremendous proliferation and professionalism of crypto coin thefts resulting in related attacks.

2.4.1 Our TEE Blockchain Wallet

We benefit from the normal-secure-world and realize the value of an IoT Blockchain Wallet, which protects the cryptographic keys against both the compromise from the normal world and unsolicited abuse:

- **Non-extractability.** With our wallet private keys reside in the secure world. The store is an extension of the secure world's storage mechanism and enables confidential and authentic persistency.

- **Abuse-resistance.** To further harden security and make the wallet thief-proof, we bind signing requests to the transaction context. Only previously obtained messages identified by a unique transaction number or smart contracts related to testified data are signed. Any arbitrary request is forbidden.

To sum the properties of the TEE Blockchain Wallet make sure that despite compromise of the rich Operating System, the attacker neither will be able to extract the private keys (i.e. steal the coins associated with the device's account) nor externally stipulate the invocation of malign payment transactions.

[5] `https://www.cryptoninjas.net/2017/07/20/parity-wallet-hacked-32m-ethereum-stolen/`

2.4.2 System-Architecture

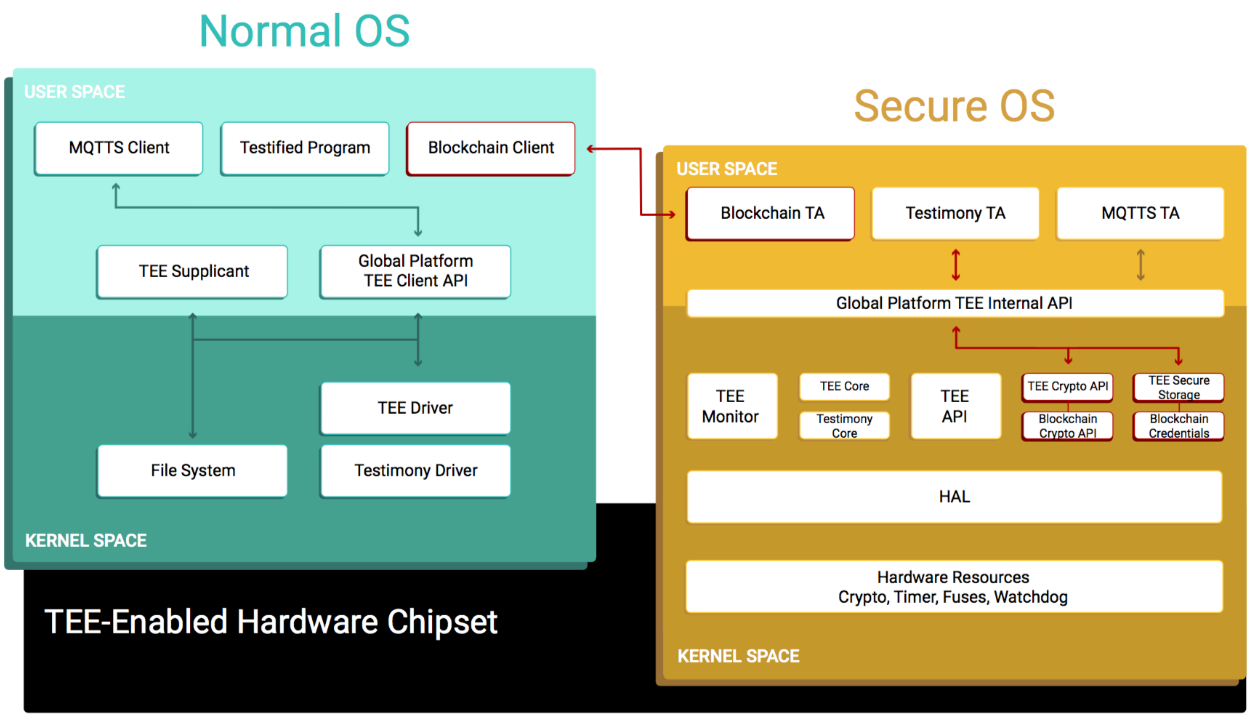

Figure 7: Testimony Architecture. Red marks specific components.

Figure 7 illustrates the system architecture for the usage of the secure blockchain wallet by making transactions or creating smart contracts without risking the confidentiality or integrity of your private keys. Again, the core idea is to separate the critical from the non-critical system modules. The core-element is our *Blockchain Trusted Application* in the secure world, the only application that gets access to the private keys held in our secure storage. The following modules are running in the secure world of the IoT device:

- the *Blockchain TA* accesses the *Blockchain Crypto API* containing ledger-specific cryptography and the *Blockchain Credential Store* where the secrets are recorded. This way, private keys will never be exposed, deleted or altered.
- the *Blockchain TA* allows for context-specific calls only. When the *Blockchain Client* requests to sign a smart contract, it can only ask to sign a particular transaction (with a unique transaction identifier) or particular asset (with a valid testimony). This way, a naive black-box use of the signing functions is impossible.

2.5 Exchange of Blockchain-based Testified Digital Assets

We now present a protocol for smart contract based digital asset exchange. We are interested in a setting where a party, called *supplier* S, has some data x, and trades the data through some intermediary, called the *marketplace* M, to some party, called *buyer* B. The marketplace implements some mechanisms to choose an appropriate supplier (or set thereof) for every buyer (or set thereof).

2.5.1 Specification of the Fair Exchange Protocol

The digital asset exchange protocol is illustrated in Fig. 8. The crux of the protocol is that supplier S, buyer B and in some cases marketplace M negotiate the contract terms, and deliver the data and payment. For the assessment of the delivery they testify their data and payment via smart contracts. That is, a testified smart contract contains a testimony stating that the supplier factually owns the data and the buyer factually owns the digital assets for the payment. We remark payments can occur with a cryptocurrency or any other digital asset.

The digital asset protocol proceeds as follows:

1. The buyer sends a DEMAND message to the marketplace. The content of the message is a description of the demanded digital good. If not previously negotiated, the message also states the terms under which the buyer is willing to trade the data. The message may contain a testimony of the demand of the digital good. The terms and the testimony may be written in a smart contract, such that a blockchain is feasible to process and store the message.

2. The supplier sends a SUPPLY message to the marketplace. The content of the message is a description of the data. If not previously negotiated, the contract also states the terms under which the supplier is willing to provide the data. The contract contains a testimony of the quality of the data. The terms and the testimony may be written in a smart contract, such that a blockchain is feasible to process and store the contract.

3. Upon reception of the SUPPLY message, the marketplace verifies the validity of the smart contract and testimony. If it fails, the marketplace aborts. Otherwise, it sends a SUPPLY* message to a potential buyer along auxiliary information. The auxiliary information augments the offer with terms necessary to complete the deal, such as the price or amount. Optionally, the marketplace stores the smart contract in the blockchain.

4. Upon reception of the SUPPLY message, the buyer verifies the smart contract terms. If the buyer agrees on the terms, it answers with a PAYMENT message. The content of the PAYMENT message is a smart contract confirming a payment relating to the previously obtained offer.

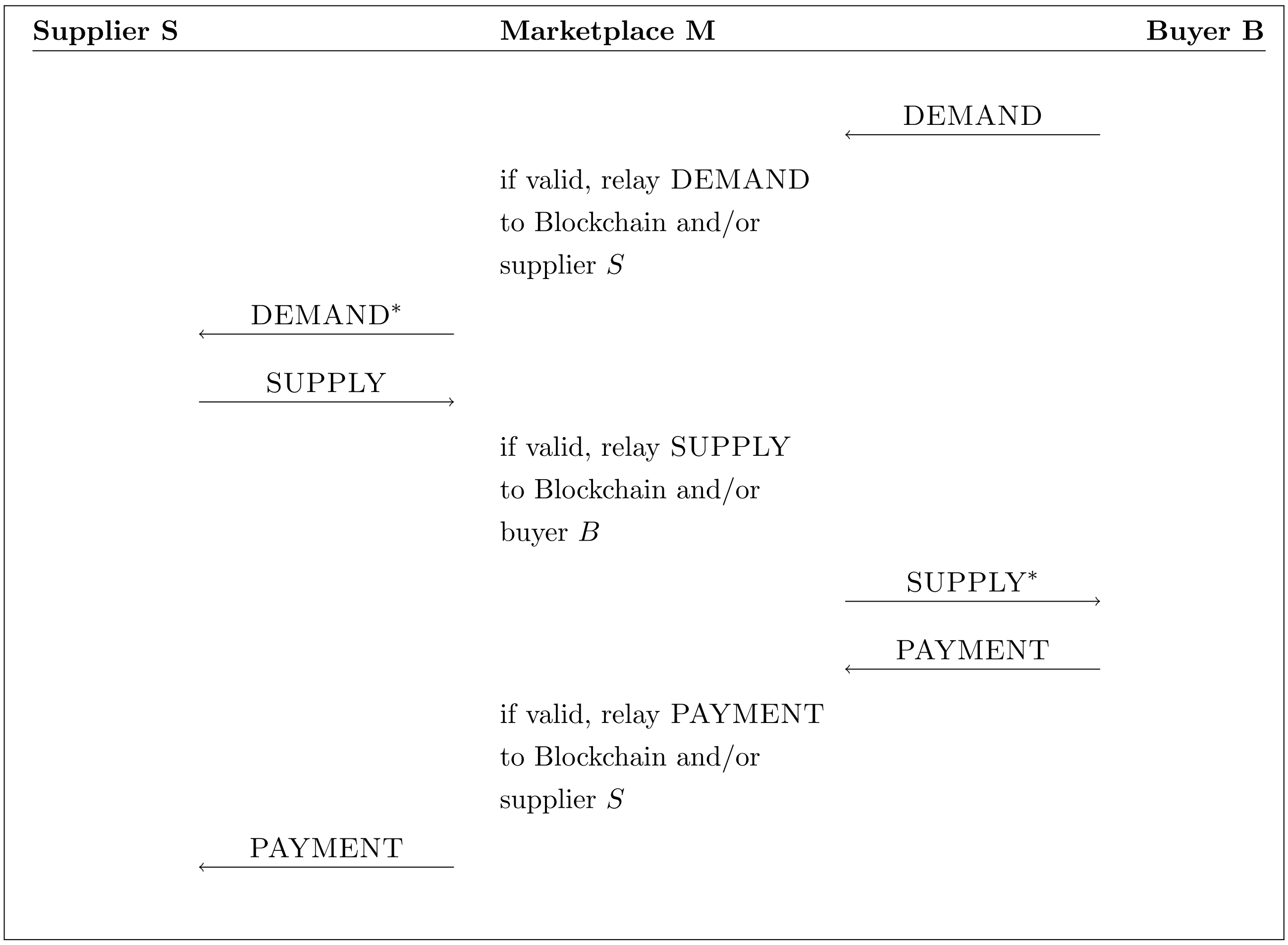
Supplier S
Marketplace M
Buyer B
DEMAND
if valid, relay DEMAND to Blockchain and/or supplier S
DEMAND*
SUPPLY
if valid, relay SUPPLY to Blockchain and/or buyer B
SUPPLY*
PAYMENT
if valid, relay PAYMENT to Blockchain and/or supplier S
PAYMENT

Figure 8: Illustration of the message flow between supplier S and buyer B through some marketplace M. (An asterisk marks a message whose content may be modified by the marketplace M.)

5. Upon reception of the PAYMENT message, the marketplace verifies the smart contract payment terms. If it fails, it aborts. Otherwise, it may forward the PAYMENT message to the supplier, informing the supplier about a valid payment. Optionally, the marketplace sends the PAYMENT message to the blockchain.

2.5.2 Extension to Multiple Entities

The protocol can easily be extended to deal with multiple suppliers. To this end, one or multiple suppliers send a SUPPLY messages. The payment occurs only if all supply message delivered data, as agreed upon in the contract terms. In the same vein, multiple payments can happen. To do so, one or multiple buyers send a PAYMENT message.

2.5.3 Protocol DEMAND

Demand Enrollment Protocol

Public Values: $\mathsf{svk}_B, \mathsf{vk}_P = (\mathsf{svk}_P, H, \Phi, P)$
Secret Values of Buyer: $\mathsf{ssk}_B, \mathsf{ek}_P = (\mathsf{ssk}_P, H, P')$

Buyer B		**Marketplace M**
	........DEMAND........	
$\mathsf{tid} \overset{r}{\leftarrow} \{0,1\}^*$		
$\tau \leftarrow \Theta.\mathsf{Testify}(\mathsf{ek}_P, x)$		
let dtrm denote the demander's contract terms		
set $dtSC := (\mathsf{dtrm}, \mathsf{tid}, \mathsf{svk}_B, x, \tau)$		
$\sigma_B \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}_B, dtSC)$		
	$\xrightarrow{\text{dtSC}, \sigma_B}$	
		if $\Theta.\mathsf{Verify}(\mathsf{vk}_P, x, \tau) \neq 1$ and $\Sigma.\mathsf{Verify}(\mathsf{svk}_B, dtSC, \sigma_B) \neq 1$ abort, else proceed with DEMAND* (optionally) send (dtSC, σ_B) to blockchain

Description. Let $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a digital signature scheme and and $\Theta = (\mathsf{Gen}, \mathsf{Testify}, \mathsf{Verify})$ be a testimony scheme. Let $\mathsf{vk}_P = (\mathsf{svk}_P, H, \Phi, P)$ be public and $\mathsf{ek}_P = (\mathsf{ssk}_P, H), P'$ be a private evaluation key. Further $(\mathsf{ssk}_B, \mathsf{svk}_B)$ is the private and public signing key of the buyer B. The protocol runs between the buyer B and marketplace M as follows:

1. The buyer samples a transaction identifier tid at random.
2. It executes the program P on input x and computes the testimony τ. To this end, it invokes the testimony generation algorithm $\mathsf{Testify}$ on input the evaluation key ek_P for Program P and input x.
3. It generates a testified smart contract $dtSC$ containing the demander's contract terms, denoted as dtrm, along the description of the data x and a testimony τ for the execution of program P on input x. The testified smart contract is signed under the buyer's signing key ssk_B. It sends the testified smart contract $dtSC$ along the signature σ_B to the market place.

4. The marketplace verifies the testimony of the smart contract by running $\Theta.\mathsf{Verify}$ on input the verification key vk_P and the to be testified input x. It also verifies the validity of the testified smart contract by running $\Sigma.\mathsf{Verify}$ on input the verification key svk_B, $dtSC$ and the signature σ_B. If any of the tests fail, the market place aborts. Otherwise, it proceeds and optionally sends the testified smart contract to the blockchain.

2.5.4 Protocol DEMAND*

Extended Demand Enrollment Protocol

Public Values: $\mathsf{svk}_M, \mathsf{vk}_P = (\mathsf{svk}_B, H, \Phi, P)$

Secret Values of Marketplace: ssk_M

Marketplace M | **Supplier S**

....DEMAND*.......................................

let etrm denote the
extended contract terms
set $edtSC := (dtSC, \sigma_B, \mathsf{etrm})$
$\sigma_M \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}_M, edtSC)$

$$\xrightarrow{\text{edtSC, } \sigma_M}$$

if $\Sigma.\mathsf{Verify}(\mathsf{svk}_M, edtSC, \sigma_M) \neq 1$
and/or $\Sigma.\mathsf{Verify}(\mathsf{svk}_B, dtSC, \sigma_B) \neq 1$
and/or $\Theta.\mathsf{Verify}(\mathsf{vk}_P, x, \tau) \neq 1$
abort, else proceed with PAYMENT

Description. Let $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a digital signature scheme and and $\Theta = (\mathsf{Gen}, \mathsf{Testify}, \mathsf{Verify})$ be a testimony scheme. Let $\mathsf{vk}_P = (\mathsf{svk}_B, H, \Phi, P,)$ be public and $\mathsf{ek}_P = (\mathsf{ssk}_P, H, P')$ be a private evaluation key. Let $(\mathsf{ssk}_M, \mathsf{svk}_M)$ be the private and public signing key of the marketplace. The protocol runs between the marketplace M and supplier S as follows:

1. The marketplace augments the testified smart contract with additional contract terms, denoted as etrm, which optionally are necessary to complete the contract. This can be, for example, the addition of a price or amount proposal. To this end it generates an extended testified smart contract, for short $edtSC$, and computes a signature σ_M over the contract using the secret signing key ssk_M.

2. It sends the extended testified smart contract $edtSC$ along the signature σ_M to the supplier.

3. The supplier verifies the validity of the extended testified smart contract by running $\Sigma.\mathsf{Verify}(\mathsf{svk}_M, edtSC, \sigma_M)$. Optionally, it also may verify the validity of the testified smart contract tSC. (In which case, the verification proceeds as in steps 3-4 of the DEMAND message.) If the verification fails, it aborts. Otherwise, it continues with the SUPPLY message.

2.5.5 Protocol SUPPLY

Asset Supply Protocol

Public Values: $\mathsf{svk}_S, \mathsf{vk}_P = (\mathsf{svk}_S, H, \Phi, P)$

Secret Values of Supplier: $\mathsf{ssk}_S, \mathsf{ek}_P = (\mathsf{ssk}_P, H, P')$

Supplier S | **Marketplace M**

.... SUPPLY ....

Supplier S:

$\tau \leftarrow \Theta.\mathsf{Testify}(\mathsf{ek}_P, x)$

let strm denote suppliers contract terms

set $stSC := (edtSC, \sigma_M, \mathsf{strm}, \mathsf{svk}_S, x, \tau)$

$\sigma_S \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}_S, stSC)$

$$\xrightarrow{\text{stSC}, \sigma_S}$$

Marketplace M:

if $\Sigma.\mathsf{Verify}(\mathsf{svk}_S, stSC, \sigma_S) \neq 1$

and $\Theta.\mathsf{Verify}(\mathsf{vk}_P, x, \tau) \neq 1$

abort, else proceed with SUPPLY*

(optionally) send (stSC, σ_S)

to blockchain

Description. Let $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a digital signature scheme and and $\Theta = (\mathsf{Gen}, \mathsf{Testify}, \mathsf{Verify})$ be a testimony scheme. Let $\mathsf{vk}_P = (\mathsf{svk}_S, H, \Phi, P)$ be public and $\mathsf{ek}_P = (\mathsf{ssk}_P, H, P')$ be a private evaluation key. Further $(\mathsf{ssk}_S, \mathsf{svk}_S)$ is the private and public signing key of the supplier S. The protocol runs between the supplier S and marketplace M as follows:

1. The supplier executes the program P on input x and computes the testimony τ. To this end, it invokes the testimony generation

2. algorithm $\mathsf{Testify}$ on input the evaluation key ek_P for Program P and input x.

3. It generates a testified smart contract *stSC* containing the contract terms, denoted as strm, along the description of the data x and a testimony τ for the execution of program P on input x. It appends the previously received *edtSC* (or a reference to it) and its own identifier svk_S. The testified smart contract is signed under the supplier's signing key ssk_S. It sends the suppliers testified smart contract *stSC* along the signature σ_S to the market place.

4. The marketplace verifies the validity of the supplier's testified smart contract by running $\Sigma.\mathsf{Verify}$ on input the verification key svk_S, *stSC* and the signature σ_S. It also verifies the testimony of the smart contract by running $\Theta.\mathsf{Verify}$ on input the verification key vk_P and the to be testified input x. If any of the tests fails, the market place aborts. Otherwise, it proceeds and optionally sends the supplier's testified smart contract to the blockchain.

```
"topic": "electricity",
"tid": 2213467217238,
"seller_address": "0x52Af54efF3a43E0CEfD5Cb1472601caB007c0232",
"date": "2017-07-26 15:30:12 UTC",
"currency": "ETH",
"price": 0.0106,
"amount": 12.5,
"additional_conditions": "none"
```

Figure 9: Sample code for a supply message

2.5.6 Protocol SUPPLY*

Description. Let $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a digital signature scheme and ¡¡¡¡¡¡¡ HEAD and $\Theta = (\mathsf{Gen}, \mathsf{Testify}, \mathsf{Verify})$ be a testimony scheme. Let $\mathsf{vk}_P = (\mathsf{svk}_S, H, \Phi, P, P')$ be public and $\mathsf{ek}_P = (\mathsf{ssk}_P, H)$ be a private evaluation key. Let $(\mathsf{ssk}_M, \mathsf{svk}_M)$ be the private and public signing key of the marketplace. The protocol runs between the marketplace M and buyer B as follows: ======= and $\Theta = (\mathsf{Gen}, \mathsf{Testify}, \mathsf{Verify})$ be a testimony scheme. Let $\mathsf{vk}_P = (\mathsf{svk}_S, H, \Phi, P)$ be public and $\mathsf{ek}_P = (\mathsf{ssk}_P, H, P')$ be a private evaluation key. Further $(\mathsf{ssk}_S, \mathsf{svk}_S)$ is the private and public signing key of the supplier S. The protocol runs between the supplier S and marketplace M as follows: ¿¿¿¿¿¿¿ a95e2d480cecbdf7e8880ba597a84ad4bb4109f7

1. The marketplace augments the supplier's testified smart contract with additional contract terms, denoted as estrm, which optionally are necessary to complete the contract. This can be, for example, the addition of a price, an amount proposal or the current gas price. To this end it generates an extended supplier's testified smart contract, for short *estSC*, and computes a signature σ_M over the contract using the secret signing key ssk_M.

2. It sends the extended supplier's testified smart contract, *estSC*, along with the signature σ_M to the buyer.

3. The buyer verifies the validity of the extended supplier's testified smart contract by running $\Sigma.\mathsf{Verify}(\mathsf{svk}_M, estSC, \sigma_M)$. Optionally, it also may verify the validity of the supplier's testified smart contract *stSC*. (In which case, the verification proceeds as in step 3 of the SUPPLY message.) If the verification fails, it aborts. Otherwise, it continues with the PAYMENT message.

Extended Asset Supply Protocol

Public Values: $\mathsf{svk}_M, \mathsf{vk}_P = (\mathsf{svk}_S, H, \Phi, P, P')$

Secret Values of Marketplace: ssk_M

Marketplace M		**Buyer B**
	..........SUPPLY*..........	
let estrm denote the extended supplier's contract terms		
set $estSC := (stSC, \sigma_S, \mathsf{estrm})$		
$\sigma_M \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}_M, etSC)$		
	$\xrightarrow{\text{estSC}, \sigma_M}$	
		if $\Sigma.\mathsf{Verify}(\mathsf{svk}_M, estSC, \sigma_M) \neq 1$
		and/or $\Sigma.\mathsf{Verify}(\mathsf{svk}_S, stSC, \sigma_S) \neq 1$
		and/or $\Theta.\mathsf{Verify}(\mathsf{vk}_P, x, \tau) \neq 1$
		abort, else proceed with PAYMENT

2.5.7 Protocol PAYMENT

Description. Let $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a digital signature scheme. Let $(\mathsf{ssk}_B, \mathsf{svk}_B)$ be the private and public signing key of the buyer. The protocol runs between the buyer B and marketplace M as follows:

1. The buyer generates a payment smart contract, denoted as pSC. To this end it refers to the exchange identified under the transaction identifier tid and identity of the supplier svk_S. It adds its own payment terms pay along its identity svk_B. The buyer signs the payment smart contract by running the signature generation algorithm with its secret signing key ssk_B and pSC as its input.

2. It sends the payment smart contract pSC along the signature σ_B to the marketplace.

3. The marketplace verifies the validity of the payment smart contract by running the signature verification algorithm $\Sigma.\mathsf{Verify}(\mathsf{svk}_B, pSC, \sigma_B)$. If the verification fails, it aborts. Otherwise, it forwards the PAYMENT message to the supplier identified under svk_S and optionally pushes the contract to the blockchain.

Payment Protocol

Public Values: svk_B

Secret Values of Buyer: ssk_B

....................................PAYMENT....................................

Buyer B | **Marketplace M**

let pay denote the
payment terms
set $pSC := (\mathsf{tid}, \mathsf{svk}_B, \mathsf{svk}_S, \mathsf{pay})$
$\sigma_B \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}_B, pSC)$

$$\xrightarrow{\text{pSC}, \sigma_B}$$

if $\Sigma.\mathsf{Verify}(\mathsf{svk}_B, pSC, \sigma_B) \neq 1$
abort, else forward PAYMENT
to supplier and (optionally)
send (pSC, σ_B) to blockchain

3 Putting it all together — The weeve Platform

While all the previously discussed technologies are interesting on their own, they solely are means to the empowering of an Economy of Things. However, when one combines all those bits and pieces into a *single* and *unified* computing platform the Economy of Things becomes reality. To this end, we have designed the *weeve platform*. Its aim is to provide every asset owner—be it a company or be it an individual—with the technology to bootstrap, configure and orchestrate marketplaces where data is indexable and where supply and demand is publicly or privately tradeable. It is worth mentioning the weeve platform entitles *every data owner* irrespectively of location, (technical) expertise or human and computing resources to trade digital assets. With the weeve platform every data owner (or group thereof) shall have the ability to launch and manage in an ease-of-use way her own marketplace or participate in already accessible marketplaces. This way, it is our innermost belief, IoT data trading democratizes: Every device owner has the freedom of choice to decide whether the harvested data is worth to be made tradeable.

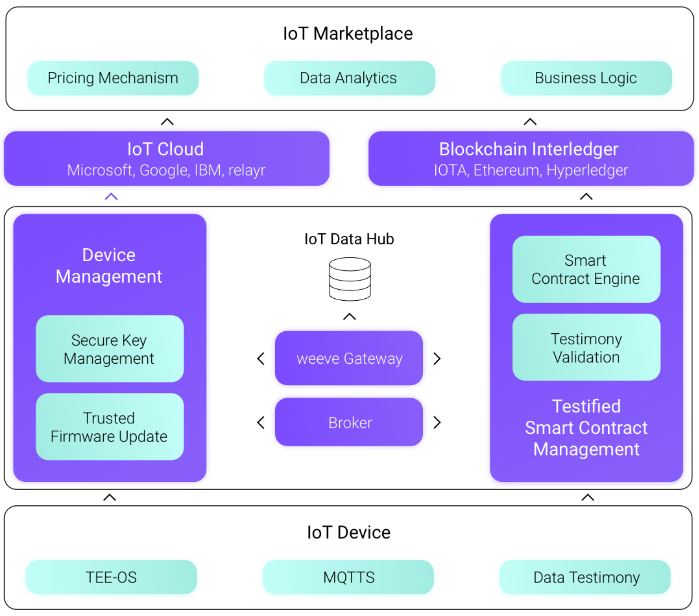

Figure 10: Illustration of the weeve Platform.

The weeve platform puts all the previously described building blocks together, such that IoT devices can be instrumented to connect witblic or private Blockchain-based marketplaces and trade testified and exclusive digital assets. To get a more technical understanding of the weeve platform, one can think of our platform as being divided into layers, as shown in Fig. 10:

- **IoT Device Layer.** Right at the bottom we have the IoT devices with a Trusted Execution Environment based Operating System augmented with the TEE-MQTTS protocol, TEE-Wallet and the necessary protocols to setup and administrate the device. By combining the features with our data testimony mechanism, we provide a new security and falsifiability standard for IoT data asset harvesting.

- **Middle Layer.** On top of the IoT device layer is the weeve middle layer located. It is the core backend IoT device connect to through the TEE-MQTTS broker. Following the publish-subscribe-paradigm the broker relays messages according to topics to weeve gateway which handles all the communicatory tasks. The weeve gateway provides secure device management featuring secure key management, access control, accounting, real-time liveness checks and trusted firmware updates. These lay the foundation for simple yet highly secure management of millions of different IoT devices.

  A very important feature of the the gateway is the validation of testified data (typically wrapped in a smart contract), before it is post-processed. The gateway makes sure this way that only honest devices and truthful data is passed to the higher layers. In some loose sense, the mechanism is similar to a firewall with the caveat that economically useless data is filtered out.

- **Infrastructure Layer.** Already today we observe a vast number of cloud and blockchain infrastructures. Due to the immaturity of the technologies one can hardly gauge by now which will prevail and become community standard. We put much emphasis on an infrastructure-agnostic approach. With the infrastructure layer the weeve platform provides interfaces to interconnect with major cloud and blockchain infrastructures. Through an API concept, device owners configure the cloud for, e.g., private storage and data mining, and the blockchain for, e.g., immutable storage and payments. This way, the greatest amount of technological openness and interoperability can be assessed.

- **Application Layer.** At the top of all layers resides the marketplace concept. This is where the Economy of Things effectively happens. The notion of marketplaces implies a great deal of customisability. The reason for that is simply argued. There is no single marketplace to optimally address the requirements of all businesses. Rather a marketplace is specifically tailored towards the use case. For example, an autonomous trading of IoT devices necessitates a pricing mechanism typically implemented by an artificial intelligence algorithm. To achieve a nash equilibrium between supply and demand (and thus maximize the utility of buyers and sellers), the algorithm has to take into account factors like quality of the good, amount, time and urgency of the buyer. These qualities heavily vary from use case to use case. The consequence is that an optimal pricing mechanism for, say, solar energy is far from optimal for temperature data. In order to guarantee the greatest flexibility we take a modular approach. A weeve marketplace consists of core components, comprising the logical parts common to many use cases, and composable parts, such as the pricing mechanism, which greatly add functionality to the business logic in order to complete the marketplace. The approach has two advantages. First, one only has to customise the marketplace. The required components are mostly provided by the weeve platform. Obviously such an approach reduces the costs of “building” the marketplace from scratch and dramatically accelerates the time-to-market. Second, the weeve platform solicits and honours contributions from the (open and closed source) community to implement extensions and apps providing additional functionality and value for, e.g., particular business logics. This notion of a third-party app ecosystem, as seen for example in the case of mobile apps, gives raise to an economy by its own.

References

[1] Amqp specification. `http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html`.

[2] Arm trusted firmware design. `https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/firmware-design.rst`.

[3] The constrained application protocol (coap). `https://tools.ietf.org/html/rfc7252`.

[4] Documents associated with dds-security, version 1.0. `http://www.omg.org/spec/DDS-SECURITY/1.0/`.

[5] Mqtt for sensor networks (mqtt-sn) protocol specification, version 1.2. `http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf`.

[6] Mqtt version 3.1.1. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf`.

[7] Rest specification. `https://www.w3.org/2001/sw/wiki/REST`.

[8] The transport layer security (tls) protocol version 1.2. `https://www.ietf.org/rfc/rfc5246.txt`.

[9] The websocket protocol. `https://tools.ietf.org/html/rfc6455`.

[10] C. Cowan. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In A. D. Rubin, editor, *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*. USENIX Association, 1998.

[11] J. Ekberg, K. Kostiainen, and N. Asokan. The untapped potential of trusted execution environments on mobile devices. *IEEE Security &amp; Privacy*, 12(4):29–37, 2014.

[12] A. Fiat and A. Shamir. *How To Prove Yourself: Practical Solutions to Identification and Signature Problems*, pages 186–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[14] S. Gajek, M. Manulis, O. Pereira, A. Sadeghi, and J. Schwenk. Universally composable security analysis of TLS. In J. Baek, F. Bao, K. Chen, and X. Lai, editors, *Provable Security, Second International Conference, ProvSec 2008, Shanghai, China, October 30 - November 1, 2008. Proceedings*, volume 5324 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2008.

[15] R. Gennaro. *Quadratic Span Programs and Succinct NIZKs without PCPs.; EUROCRYPT.* 2013.

[16] M. Gray. Bletchley – the cryptlet fabric & evolution of blockchain smart contracts.

[17] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In P. C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60. USENIX Association, 2008.

[18] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. Authenticated confidential channel establishment and the security of TLS-DHE. *J. Cryptology*, 30(4):1276–1324, 2017.

[19] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an OS kernel. In J. N. Matthews and T. E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[20] P. C. Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems.; CRYPTO.* 1996.

[21] P. C. Kocher. *Differential Power Analysis.; CRYPTO.* 1999.

[22] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448. Springer, 2013.

[23] P. Laskov. Machine learning in adversarial environments. *Machine Learning*, 81(2):115–119, 2010.

[24] J. Longo, E. D. Mulder, D. Page, and M. Tunstall. Soc it to EM: electromagnetic side-channel attacks on a complex system-on-chip. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 620–640. Springer, 2015.

[25] A. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[26] C. Meyer and J. Schwenk. Lessons learned from previous SSL/TLS attacks - A brief chronology of attacks and weaknesses. *IACR Cryptology ePrint Archive*, 2013:49, 2013.

[27] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. Iotpot: Analysing the rise of iot compromises. In A. Francillon and T. Ptacek, editors, *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015.* USENIX Association, 2015.

[28] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, 2012.