End-to-End Trust for Privacy Enhanced Renewable Energy Certificates

by

Dimcho Karakashev

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

3Government incentives have been an important factor for the growth of the renewable energy industry. An example is Renewable Energy Certificates (RECs), which are issued to green energy producers for the amount of produced energy, usually 1 MWh. RECs can be used for claims and government compliance. They can be also bought and sold. Currently, the systems deployed for REC trading are centralized, cumbersome, and require a high level of trust in entities such as utility providers and energy producers. Thus, a healthy competitive environment is not fostered, which, as shown in recent work, results in the inability for small scale energy producers, such as households, to participate.

To address this issue, we propose a system for trading RECs that removes the requirement for trust between the parties involved. It also increases transparency and security. Our solution is based on tamper-proof smart meters and a blockchain to increase transparency. Although our system permits more fine grained RECs, which is desirable, this can have negative privacy implication. Hence, we describe how RECs can be enhanced to preserve privacy.

To show the feasibility of our proposed design, we describe two implementations based both on a permissioned Blockchain, Hyperledger Fabric, and a permissionless Blockchain, Algorand. To the best of our knowledge, this is the first work to successfully implement such an end-to-end trusted system for trading RECs.

Acknowledgements

TODO:

Dedication

TODO:

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The electricity and heat production sector contributes about a quarter of the global greenhouse emissions [77], one of the root causes for temperature rise. Government policies attempt to solve this problem by incentivizing the adoption of renewable energy. In this thesis, we describe a system that addresses issues with a particular policy to make the deployment of renewable energy sources more attractive.

The electricity and heat production sector, unlike other sectors, is amenable to emission reduction because renewable energy sources have become a viable alternative to fossil fuels. In particular, the cost of renewable energy, a major inhibitor to adoption, has dramatically decreased over the past two decades. For example, in 2011, the US Department of Energy set a goal to reduce PV-cost by 75 % [79]. Not only did they achieve the goal in 2017, three years ahead of schedule, but they also have set a new one, which would reduce the PV cost by an additional 50 % by 2030 [78]. If accomplished, solar energy would be amongst the cheapest options for energy production.

Although reduced cost is essential for the remarkable growth of the renewable energy industry, the current rate of greenhouse emissions reduction is not sufficient [26] and complex government incentives are still a crucial tool to keep the momentum [70]. Incentives are usually implemented in the form of policies and attempt to address market failures of the industry; that is, a free market does not encourage a behavior benefiting the group [51]. Beck et al. [33] conducts a comprehensive review of these complicated policies and broadly categorizes them into categories such as renewable energy promotion, emission reduction, and distributed generation. Each of these categories contains other sub-categories; for example, renewable energy promotion policies include tax breaks and quantity-forcing policies.

The government incentives effectiveness varies, and some successful policies are scheduled to be reduced. For example, end-users can sell excess energy via net-metering programs, but such a policy commonly does not offer enough compensation to be an attractive

investment [7]. The solar investment tax credit, a tax credit that can be applied to the personal income in the USA, is scheduled to be cut from 30% in 2019 to 10% in 2023 [17]. Although foreseeing the exact impact of each policy is difficult, it seems that there is a consensus that the best solution is a combination of several different policies.

In this work, we focus on *quantity-forcing*, a policy that requires either a specific amount or a portion of the total power to be produced by renewable energy sources [33]. The policy in the USA and Canada is implemented under the name Renewable Portfolio Standard (RPS), and it is a generation-based standard [33] [7]. It specifies not only the required portion of renewable energy by jurisdiction but also a deadline for this to be achieved. The policy is associated with about half of the added renewable energy generation in the USA [32].

To comply with the RPS, electricity supply companies can either produce renewable power or purchase power. Renewable Energy Certificates (RECs), which are tradeable energy commodities, are a mechanism for the latter [7]. They are issued when a certain amount, usually 1 MWh, of renewable energy power is produced; there is no connection to the physical electricity with the common *unbundled* RECs. From a renewable energy producer perspective, those RECs provide an additional income and incentive.

Not only can electricity supply companies use RECs to meet their requirements, but also any supporter of renewable energy can voluntarily purchase RECs. Such supporters, from large companies to small households, use RECs to make claims or support the industry. For example, RE100 [24], a global initiative, has managed to convince over 200 companies, including Microsoft, Goldman Sachs, GM, and eBay, to commit to 100 percent renewable energy. The majority of those companies rely on buying a mixture of RECs and physical power from renewable energy farms [67] or entirely rely on RECs [6] to support their claim of being 100 percent green. RECs in the USA are traded on multiple markets such as WECC [21], NEPOOL [15], and MRETS [13].

Although RSP has been successful, the contributed addition of renewable energy sources attributed to the policy has decreased over time [32]. There are several issues with the current REC systems.

Inconsistencies among jurisdictions are common. For example, AIB—the organization that develops and promotes a standardized REC system in Europe—has discovered that Italy has been issuing RECs for a source that is considered renewable in Italy but not in Europe [11]. Additionally, Hulshof et al. [46] also discovered inaccuracies in the AIB's database. The issue is not limited to Europe, but it is also present in, for example, USA [67].

Besides inconsistencies, the system for trading RECs has suffered from fraud. An end-user who wants to contribute to the renewable energy adoption could buy RECs that are

generated from a renewable energy source that does not exist [67]. Moreover, double-spending is also common [67]. Other frauds, such as tax evasion, have also been uncovered [50] in the past.

To prevent fraud, some providers try to enforce third-party audits. For example, Green-e, a voluntary program in the US and Canada, focuses on certifying and ensuring no REC double-spending. The participants in the project are required to comply with two annual audits [12] that involve additional fees and time, a deterrent for energy producer participation, especially for small scale producers such as households.

Apart from these shortcomings, no existing REC system addresses issues related to user anonymity. Jawurek et al. [49] discuss in depth how frequent smart meter readings can be detrimental to privacy; readings reveal too much personal information such as lifestyle and religion. We expect that small renewable energy producers, many of which might not have battery storage, will participate in our system. Since RECs are issued based on the injected energy in the grid, using estimation of how much energy would have been produced, it would be possible, for example, to determine when the owners of a household are away.

To summarize, the current systems for RECs are centralized, with inconsistencies and issues, such as double-spending, being common. Existing systems provide no REC guarantees unless a trusted third auditor is involved. However, the need for auditors discourages small-scale renewable energy producers such as households, due to cost, time, and limited investment return potential. Our observation is confirmed by Andoni et al. [30], who surveyed over 100 projects related to blockchain applications in the energy sector. They also describe PowerLedger, a project that demonstrates the market disadvantage of small renewable energy producers. Via excess energy trading implemented using blockchain, energy producer profit of households was increased from 7 c/kWh to 15 c/kWh, while consumers experienced a price decrease from 25 c/kWh to 20 c/kWh. Apart from the lack of incentives for small renewable energy producers, the survey also emphasizes that user anonymity issues are neglected.

1.2 Contributions and System Overview

In this thesis, we investigate how to improve the REC quantity-forcing policy. We speculate that it is feasible to create an end-to-end trusted market for RECs, which would eliminate the requirement for trust between the parties involved. Specifically, we make the following contributions:

- Design, to the best of our knowledge, the first system to provide end-to-end trust for RECs, as well as the first proposal to enhance RECs with anonymity.
- Define system properties and describe how our proposal satisfies those properties.
- Describe and evaluate two implementations of our system.

At a high level, in our proposed system trust is established using a trusted smart meter, and it is maintained within the trading platform. A smart meter, connected to a solar panel, has access to a building block guaranteeing the accuracy of the RECs that it generates; we refer to the building block as *secure hardware*. During operation, the device produces metadata updates—which contain a proof for authenticity and that the device is not tampered—and sends RECs to the trading platform. End-users trying to establish the trust in the system can access and verify the proofs in the REC without requiring trust in entities such as the energy producers. Additionally, our proposal enhances RECs with anonymity using ring signatures [34] and stealth addresses [40]. The trading platform, which handles REC trading, has access to a blockchain that removes the trust in a single party. Lastly, the client application provides an end-user interface to the trading platform.

To show the feasibility of this high-level system proposal, we describe two implementations. Both instantiate the smart meter using Azure Sphere, but the instantiation of the trading platform differs. In the first one, the trading platform is based on Hyperledger Fabric, a solution that could be difficult to bootstrap. To address that, we present a second solution based on Algorand, an already bootstrapped system. In the evaluation of the implementations, we argue that both provide secure REC trading functionality. To show feasibility, we estimate the initial cost—one-time expense added to each solar panel—and the monthly cost—the expense of submitting RECs in fifteen-minute intervals—per solar panel.

Although the results in this thesis are significant to continue incentivizing the growth of the renewable energy industry, our implementations have several limitations. First, our prototype has not been tested in a real-world scenario. Second, we did not implement the anonymity features described in the proposal. Lastly, we made simplifications in our implementation that need to be addressed in a real-world deployment.

Chapter 2

Background and Related Work

2.1 Background

In this chapter, we overview the technologies used in our implementation. First, we introduce the Hyperledger Fabric and Algorand blockchains, with a focus on transaction processing and parties involved. Although both blockchains are used as a black-box, the description of transaction processing enhances the understanding of the trust in the involved parties. Second, we describe Azure Sphere, a microcontroller used to instantiate a smart meter. Our goal is to summarize Microsoft's design properties and to explain how they are satisfied.

2.1.1 Blockchains

Blockchains—ledgers with tamper-proof history in which users can read and write—are one of the main building blocks in our systems. There are two types: permissioned blockchains and permissionless blockchains. The former limits the party participation to only users with known identities, while the latter does not impose any limitations on user participation. According to Androulaki et al. [31], permissioned blockchains are used when parties have common goals but do not fully trust each other; for example, two companies trading an asset. In contrast, permissionless blockchains are applicable to use cases in which parties do not have any trust in each other. We now overview Hyperledger Fabric and Algorand.

Hyperledger Fabric

Hyperledger Fabric [31] is a permissioned blockchain that provides a platform for distributed applications, also referred to as smart contracts or chaincodes. One of the main advantages of the system is its modularity, which allows for balancing trade-offs such as throughput and trust. Another advantage is that the system can support up

to 20 000 tx/s [45], a throughput permitting a wide variety of applications. Before describing the processing of transactions, we describe the nodes that participate in the system.

There are three node types in Hyperledger Fabric. First, *clients* invoke chaincode by submitting transaction proposals. The second type is *peers* who are responsible for maintaining the blockchain state and executing chaincode. They are separated into groups of nodes, called organizations; each node in the group fully trusts every participant in the group. Lastly, *ordering-service nodes* implement a consensus protocol to order and group transactions into blocks. All node types interact with a *membership service provider* that enforces the permissioned aspect of the system.

Hyperledger Fabric processes transactions by executing, ordering, and validating them, as opposed to ordering and executing them, an approach used by all previous blockchains. To execute a chaincode, a client submits a transaction proposal to a set of peers. The proposal specifies information such as chaincode to be executed, arguments to the chaincode, and identity of the client. Each transaction proposal is simulated in a Docker container using the current state of the blockchain and results in two sets. The *readset* contains accessed keys and version numbers denoting the last time the key has been updated, while the *writeset* is a key-value set of updated values and keys. The *endorser*, the peer executing the chaincode, returns the client an endorsement containing these two sets, metadata, and a signature. If the client receives enough endorsements to satisfy an *endorsement policy*[1], a transaction is created and sent to the ordering nodes.

Using a consensus protocol, nodes order transactions into blocks, which are then sent to the peers. To accept the block and to be added to the add-only data structure, two verifications are performed. The first ensures that each transaction satisfies the endorsing policy of the chaincode. The second verification checks if there are any conflicts in the read and write sets; in other words, the version of the keys used to execute the chaincode has not changed. Only then the peer adds the block to its history, updates the local maintained state using the write set, and the transaction is considered processed.

Algorand

Algorand [44] is a permissionless blockchain that provides security, scalability, and decentralization. The system scales to hundreds of thousands of users, while transactions are finalized in the order of seconds instead of hours, an issue common to some blockchains such as Bitcoin. To achieve such scalability, Algorand relies on a protocol based on proof

[1]The policy is specified during the creation of the chaincode and determines the peers required to execute the chaincode.

of stake. In comparison to other projects that use proof of stake, Algorand does not suffer from issues such as target compromise and denial of service.

Algorand makes several assumptions to achieve its goal [44]. First, at least two-thirds of the cryptocurrency is assumed to be owned by honest users. Second, the protocol requires strong synchrony for new blocks to be added to the blockchain. A violation of the strong synchrony assumption does not affect protocol security as long as there is weak synchrony. Lastly, users are expected to have a somewhat synchronized clock to recover from potential forks. Now that we have summarized Algorand's assumptions, we describe how transactions are processed.

Algorand is split into two phases. During the first phase, a user is selected, and this selected user proposes a block. To determine if it is selected, each user privately checks the output of a verifiable random function (VRF): the input is the user's private key and publicly available information. In case multiple users are selected, the user with the highest priority, computed using the VRF output, is chosen for block proposal. Although, with small probability, a malicious block proposer could be selected, however, the adversary can, at most, add an empty block to the chain. During the second phase, committees achieve consensus on approval of a proposed block via a Byzantine agreement protocol. The members in the committee are also selected using a verifiable random function, where each unit of Algorand can be seen as a sub-user with an equal probability to be selected. In other words, the more crypto currency users have, the bigger their voting power during the voting phase.

2.1.2 Azure Sphere

Azure Sphere [23] is a microcontroller unit (MCU) designed to provide strong security at a low cost. It can be viewed as a secure environment for custom software execution. The device is tightly coupled with Azure Cloud, which verifies the authenticity and health of each device and provides a secure channel to push software updates. In this section, first, we summarize seven properties required by Microsoft from each secure resource-constrained device. Then, we explain how Azure Sphere achieves some of those properties. Lastly, we discuss terminology used throughout this thesis related to software updates.

Hunt et al. [47] outline seven properties that are integral to the security of any low-cost device. The first property, *hardware root of trust*, requires an unforgeable cryptographic key to be inseparable from the hardware and protected from physical-side channel attacks. The second property is a *small trusted computing base* (TCB); it requires that the software

and hardware trusted to provide a secure environment be small. Resource-constrained devices satisfy the *defence-in-depth* property only when there are multiple defenses for each threat. To prevent a component vulnerability affecting the whole system, *compartmentalization* demands hardware to enforce separation between different compartments. The *certificate-based authentication* property attempts to address common issues with default or weak passwords by requiring devices to authenticate using certificates. A vulnerability would almost inevitably be found in any device, so even compromised, resource-constrained devices need to receive updates reliably; this is encapsulated in *renewable security* property. Lastly, the *failure reporting* property is described by Hunt et al. [47] using an immune system analogy; it allows system operators to identify attacker devices probing potential vulnerabilities such as software bugs.

The *Pluton* security subsystem is an Azure Sphere subsystem designed to satisfy the hardware root of trust. The subsystem features [14] include a dedicated Arm M4F Cortex, tightly coupled memory (TCM), read-only memory (ROM), e-fuses, and Pluton engine. Pluton engine [76] implements functionality such as hardware random number generator (HRNG), public key accelerator, secure hash algorithm, AES, complex command engine (CCE), shared memory, and fuse control. Another essential feature is the hardware designed to ensure that random numbers are generated with sufficient entropy by HRNG. It is important to mention that commands which require more than one cryptographic engine are handled by the complex command engine. An example of such command is remote attestation; it involves combining, hashing, and signing [63] a specially dedicated register for code measurement and challenge sent by the cloud. In Section 5.1, we describe in more depth the publicly available information related to establishing the root of trust.

Azure Sphere's TCB [23] is limited to the Pluton security subsystem, its runtime, and software referred to as Security Monitor (SM). SM runs in the secure world of the application processor subsystem, which, among other features, has an ARM A7 processor that supports Arm TrustZone. Security functionalities such as maintaining access to the security subsystem and isolating processes by controlling A7's memory management unit are the reasons why SM is part of the TCB. Lastly, process isolation by the SM satisfies the compartmentalization property.

Multiple software layers are provided to satisfy the defense-in-depth property. The first and second levels are the security monitor and a custom Linux kernel developed by Microsoft. The kernel runs in supervisor mode within ARM A7's normal world and provides functionality to upper levels such as access to peripherals [23]. Another level of software security is the software that provides OS services; it runs in user mode and handles functionality such as authenticated communication with the cloud. Lastly, applications run on the top layer and have access to the rest of the system via well-defined API, a partially

exposed POSIX standard [23]. Hunt et al. [47] explain how renewable security, certificate-based authentication, and failure reporting properties are satisfied.

Azure Spheres are split into groups to ease the deployment of new applications. At the highest level, Azure Sphere Tenant—a cloud entity representing and controlled by a service company—logically separates devices by owners. Lower-level groups are products [9]; for example, a specific model of solar panels can be a product. Lastly, at the lowest level are groups that split devices into development, testing, and production group.

There are several terms associated with deploying a new application. First, Azure Spheres are *claimed* by service companies. The procedure associates a device with a specific Azure Sphere Tenant [18], which can, for example, deploy new applications and observe the healthiness of devices. Additionally, microcontrollers are always associated with exactly one group, product, and tenant. To deploy an application, the service company creates an *image package*, a compiled binary, and metadata. Then, it distributes the package to a specific product and group via Azure Security Service.

2.2 Related Work

In this section, we describe related work. We begin with a brief introduction on how to establish trust in a remote device, and then we describe designs of trusted smart meters. Lastly, we mention existing solutions for trading RECs.

2.2.1 Trusted Smart Meters Design

We now described relevant work on designing trusted smart meters.

Trusted Smart Meters Based on TPM

Lemay at al. [56] were perhaps the first to introduce the idea of combining smart meters with trusted computing technology. In their work, they identified possible attackers and privacy concerns that were taken into account in their system design. They envisioned multiple vendors sharing the same smart meter by deploying a variety of applications; these applications were separated using virtualization. Privacy concerns were addressed with network access control policies that restrict the amount of data each application can send to parties. Since smart meters send information to parties with stake, application attestations

using TPMs were attached to the information. Unfortunately, the prototype was on a desktop computer and simulated TPM functionality using Linux-IMA, an approach not suitable for resource-constrained devices.

Zhao et al. [82] also recognized that trusted computing technology could enhance smart meters to enforce expected behavior. They identified that previous proposals only used TPM as a computing unit, while their scheme takes advantage of the TPM to verify the smart meter's configuration. In their work, the term *trusted smart meter* was coined as a meter not only having access to a module with cryptographic capabilities but also consistently behaving as expected. Attribute certificates were used to deal with configuration management and privacy issues, while ring signatures hide the electricity usage reported by the smart meter. Unfortunately, the level of abstraction in property-based attestation is not discussed in this work. Similarly to our proposal, ring signatures were used for anonymizing meter readings. However, their scheme does not address the need to demonstrate ownership of the meter readings. Lastly, they prototype their proposal on a personal computer, an implementation that is not straightforward to apply to resource-constrained devices.

Zhang et al. [80] also used trusted computing to verify smart meter's trustworthiness; their proposal was based on attribute-based attestation, a term that has the same meaning as property-based attestation. Unlike Zhao et al. [82], the granularity of the attributes was discussed; for example, they specified an attribute to be a guarantee that smart meter uses a specific version of trusted OS. The first of the five parties involved in the scheme is a trusted test agent. It generates a part of a secret key and then seals the key to a specific trusted smart meter state. The smart meter generates the other part of the key to create a *complete key pair*. Each electricity reading is encrypted and sign using the complete key pair and then sent to a regional gateway. The getaway verifies and forwards each reading to a control center. The control center decrypts the readings and obtains fine-grained electricity usage information used to calculate an electricity bill. The bill is forwarded to a service provider that maintains a mapping between smart meter pseudo-identities and real identities. Unlike Zhang et al. [80], our work uses stealth addresses instead of pseudonyms to address the need for ownership demonstration. Additionally, there is no discussion on issues such as attribute certificates management, nor implementation is provided.

Trusted Smart Meters Based on TEE

Combining TPMs with resource-constrained devices has drawbacks [52] [57]. Some of the main arguments include the increase of the already price-sensitive cost, consuming too

much power, and unsuitability for mobile devices. As an alternative to TPMs, in this section, we discuss designs based on TEEs.

Karopoulos et al. [52] used TEE to provide secure cryptographic key storage and ensure software integrity using remote attestation. Unfortunately, their work made simplifications that could be hard to overcome in a real-world scenario. For example, all devices are assumed to run the same firmware. including verifier and prover. Additionally, privacy issues were not considered, and the implementation on Raspberry Pi using Open-TEE is not open-source, so it remains unclear in terms of security.

Paverd et al. [65] focused on securing smart meters by combining TEEs and TPMs. They set security requirements such as enforcing device private keys to be only available within a specific set of applications in a specific device. To achieve the requirements, Paverd et al. [65] implement key initialization and TLS handshake as Piece of Application Logic (PAL); PALs are applications executed in a separate environment that ensures integrity and confidentiality. To initialize keys, a PAL is given as input the measurement of an application to be given access to the key. The measurement is performed by a Linux Integrity Measurement Architecture, and it is passed to the PAL in a register. Then, the PAL generates both an asymmetric key pair identifying the device and symmetric key encrypting the device key pair. The symmetric key pair is sealed to the application measurement application provided as input using the TPM, effectively restrict key access to the specific TPM and application. Similarly, the PAL implementing the TLS handshake has access to the device keys only if the PCRs match. Besides the fact that during key initialization the measurement inputted to the PAL by IMA can be tampered with, an implementation on resource constrained device is not shown. Unfortunately, anonymity is not considered and the use cases could be limited due to restricted PAL functionality.

2.2.2 Securing the link between smart and sensor

In our work, we assume that the link between the smart meter and the sensor is trusted, but in this section, we discuss how this issue might be addressed. Reciti et al. [66] designed and implemented an intrusion detection system to detect tampering with the serial communication between a sensor and a smart meter. In their proposal, data-logger records all messages exchanged between the meter and the sensor. Using those records, a data-preprocessor creates a feature vector describing the recorded events in numerical values, which are then given to two anomaly detectors. A cyber anomaly detector uses a clustering-based anomaly detection algorithm on the communication messages to detects tampering. Similarly, the physical anomaly detector detects tampering based on energy consumption

input. The difference between those two detectors is the time scale of the anomalies being detected. To assess their proposal, re-calibration, reset, and sleep mode attacks were tested against the cyber anomaly detector. According to their results, the algorithm was able to identify all intrusions with no false positives. Although details on how the attacks were performed are lacking, it would be interesting to see how such an algorithm performs in a real-world scenario.

2.2.3 Trading RECs on Blockchain

The work that is related to placing RECs on the blockchain can be divided into one that focuses on system engineering or focuses on simulation-based analysis.

The goal of the first line of work, also in which our work fits in, is to solve challenges that occur when designing and implementing a REC system using blockchain. For example, Knirsch et al. [53] propose an off-chain REC trading system in which transactions are stored on a distributed file system, and the state is periodically locked by including a hash of the state in an Ethereum transaction; an approach that resembles our Algorand implementation. Another work in the same direction is based on Predix [48], an industrial IoT platform. A significant difference in our work is that we do not assume that the smart meter hardware cannot be compromised. With such an assumption, system operators need to regularly audit smart meters [53], which introduces additional overhead and cost. We instead rely on remote attestation for inexpensive verification. Furthermore, we do not require smart meters to process the off-chain transactions or rely on third parties for the processing, which is essential since smart meters have limited resources.

The focus in the second line of work is understating the effect of different approaches to incentivizing the deployment of renewable energy sources. Zhao et al. [81] propose a new consensus protocol that takes into account the amount of generated energy. Each period, energy producers are ordered by the amount of generated energy. A random leader among the top producers is chosen to propose a block on which at least two-thirds of the nodes need to agree. An incentive scheme based on periodically computed coefficients for both individuals and communities is developed to encourage the deployment of renewable energy. A simulation of the protocol and incentive scheme shows improved effectiveness. In contrast to their work, we build on already thoroughly studied consensus protocols. We have not implemented an incentive scheme, but a similar approach could be adopted. However, it would be challenging to compute the individual coefficients since, in our proposal, RECs are enhanced with anonymity. Another work in the same direction is the simulation by Castellanos et al. [37], and it attempts to determine the effectiveness of different price strategies on RECs.

In contrast to academia, there are many industry projects related to blockchain and RECs. Perhaps the most prominent and closest to our project is the collaboration between Nasdaq stock exchange and Filament.[2] In addition to the claim that Filament's secure hardware guarantees RECs validity [25], the created platform is claimed to allow anonymous RECs [60]. Our work has the same goal, but unfortunately, the available public information related to Filament's hardware and Linq—Nasdaq's blockchain used in the project—is very scarce, and we are unable to determine the difference between our and their approach.

The Energy Origin (TEO) [29], a subsidiary of the French multinational utility provider ENGIE, has partnered with a company called Ledger. Similar to our system, their device, Ledger Origin [28], also cryptographically attests security state of devices. Meter readings are sent to Energy Web Chain [27], an open-source blockchain. Their work initially announced in 2017, is concurrent with ours. An official working solution is announced in 2020, but, to the best of our knowledge, details of Ledger Origin are not fully available, which does not allow us to identify how our work differentiates from theirs. Additionally, their project does not consider anonymity issues.

Another active company is Powerledger, which has developed a proprietary blockchain and has multiple projects in different countries. One of the functionalities offered on their platform is REC trading. Recently, they have partnered with M-RETs, Midwest North America's renewable energy registry, to launch a global REC marketplace. We believe that our work can enhance trust in their project.

Although there are many similar industry projects[3], none of them seem to address any privacy issues. Additionally, very little smart meter trust is guaranteed.

[2] As of March 2020, Filament's official website appears unreachable

[3] To name a few: LO3 Energy, WePower, ImpactPPA, SolarCoin, Enervalis, Greeneum, Suncontract, The Alva project, Pylon Token, Volt markets

Chapter 3

Preliminaries

In this chapter, we informally define the primitives used in our system. We assume the algorithm inputs are defined over finite space.

3.1 Signatures

A signature scheme is a triple of algorithms (S.Gen, S.Sign, S.Vrf) such that:

1. **Key Generation** S.Gen(1^n): is a PPT algorithm with input a security parameter and output a key pair (pk, sk).
2. **Signing** S.Sign(sk, m): is an algorithm with input a secret key sk and a message m and output a signature σ.
3. **Verify** S.Vrf(pk, m, σ): is a deterministic algorithm with input a public key pk, message m, and signature σ, and output either true if the signature is valid or false otherwise.

Definition 3.1.1. Correctness: The scheme is correct if the signature of any message signed with a secret key can always be verified using the corresponding public key. The statement is valid for any message within the finite message space and any key pair generated by the key generation algorithm.

Definition 3.1.2. Security: The scheme is secure if no adversary that is given access to a *signing oracle* can create a valid signature of a message for which the oracle has not been queried. The adversary can use the signing oracle to sign any message using any key pair.

3.2 Ring Signatures

A ring signature scheme allows a party in a group to non-interactively sign a message on behalf of the group. An additional requirement is that the signature does not reveal the

identity of the signer. We reiterate the ring signature definition by Bender et al. [34], which defines the scheme as a triple of algorithms (RS.Gen, RS.Sign, RS.Vrfy):

1. **Key generation** RS.Gen(1^n): is a PPT algorithm with input a security parameter and output a key pair (pk, sk).
2. **Signing** RS.Sign(sk, R, m): is a PPT algorithm with input a secret key sk, a set R of k public keys and a message m. The output is a signature σ. The requirements for the *ring group* R are:
   (a) The key pair (sk, pk) used to signed the message is a valid key pair generated by RS.Gen and $pk \in R$.
   (b) $|R| \geq 2$.
   (c) Each public key pk in the set R of public keys is distinct.
3. **Verification** RS.Vrf(R, m, σ): is an algorithm with input a signature σ, a set R of public keys, and a message m and output either true if the signature is valid or false otherwise.

Definition 3.2.1. Correctness: The scheme is correct if the signature of any message signed by a party in a ring group can always be verified using the ring group. The statement is valid for any message in the finite message space and for any ring size greater than two. The key pairs in the ring group are generated using the key generation algorithm.

Definition 3.2.2. Security: Following Definition 7 [34], the scheme is secure if no adversary can create a valid message signature for which a *signing oracle* has not been queried and the ring group used to sign the message does not have *corrupted parties*. The adversary is given access to a signing oracle that allows to sign any message using any ring group and corrupting oracle that allows to adaptively corrupt secret keys of any parties.

Definition 3.2.3. Anonymity: Following Definition 4 [34], the scheme is anonymous if no adversary can determine the party that signed a message, even if all keys in the ring group are compromised after the signing and the message and the ring group is chosen by an adversary that has access to a *signing oracle*. The *signing oracle* can be used to sign any message using any ring group.

Remark. The anonymity definition implies that no two signatures can be attributed to the same signer [34].

3.3 Stealth Addresses

Stealth addresses protect the anonymity of the receiver, where as ring signatures guarantee sender anonymity. It is a triple of algorithms (SA.Gen, SA.GenAddr, SA.Vrfy) such that:

1. **Key generation** SA.Gen(1^n): is a PPT algorithm with input a security parameter and output a key pair (pk, sk).
2. **Address generation** SA.GenAddr(pk): is a PPT algorithm with input **receiver's public key** pk and output a new public key pk_{new} and an auxiliary information γ.
3. **Address verification** SA.Vrf(pk_{new}, γ, sk): is an algorithm that verifies inputs: new public key pk_{new}, auxiliary information γ, and **receiver's secret** sk. The output is a tuple (out, sk_{new}), such that out is either true when the new public key was generated using the key pair (pk, sk), or **false** otherwise. If out is true, new secret key and the new public key form a valid key pair.

Definition 3.3.1. Correctness: The scheme is correct if any new public key generated using a public key of a user can always be verified with the corresponding secret key of the same user. Additionally, the verification outputs a valid new secret key part of the new key pair. The statement is only true for key pairs generated by the key generation algorithm.

Definition 3.3.2. Security: The scheme is secure if no adversary can determine the new secret key using only the receiver's public key.

Definition 3.3.3. Anonymity: A stealth address is anonymous if an adversary cannot determine if a new public key is generated using a particular public key.

3.4 Secure Hardware

We informally define minimal required secure hardware functionalities and properties to help us with system description and security analysis. The minimal functionalities are two algorithms (SH.Load, SH.Exec), where:

1. **Remote application deployment** SH.Load(app): is used by users to remotely instantiate a given application app.

2. **Trusted execution** SH.Exec(APP, in): is used by users to execute an instantiated application APP using an input in. The output contains the result of the execution and a proof ϕ that can be used to check the authenticity and security of the device.

In addition to those two functionalities, we require the hardware satisfies the following property:

Property 3.4.1. **Hardware-based root of trust**: device keys can only be accessed via hardware APIs and cannot be separated from the device. Additionally, hardware protection against physical side-channel attacks is crucial.

3.5 Remote Attestation

Remote attestation is a scheme in which a verifier—such as an external trusted party—can check the security state of a prover, an untrusted device. It is used to create the proof during trusted execution in the secure hardware. Francillon et al. [43] defines remote attestation as a triple of algorithms (RA.Setup, RA.Att, RA.Vrfy), where:

1. **Setup** RA.Setup(1^n): is a PPT algorithm with input a security parameter n and output a key pair (pk, sk).

2. **Attestation** RA.Att(sk, s): is a deterministic algorithm with input a secret key sk and a device state s and output an attestation token α.

3. **Verification** RA.Vrfy(pk, s, α): is a deterministic algorithm with output `true` only if the authentication token α was created using the corresponding device state s and private key sk, otherwise out is **false**.

Remark. Replay attacks are not considered in this definition. We alter the definition by Francillon et al. [43] to use asymmetric key pair instead of symmetric key pair. This change is introduced to enforce that only the owner of the secret key can generate valid attestation tokens.

Definition 3.5.1. Security: Similar to Definition 2 [43], remote attestation scheme is secure only if adversary that is given access to an attestation oracle cannot produce a valid authentication token for which the oracle has not been queried. The oracle allows the adversary to get valid attestation tokens for any device state and secret key.

Francillon et al. [43] explain that the attestation algorithm need to be the only algorithm able to generate valid attestation tokens α that correctly capture the security state of the device. Several properties enforced by the hardware are needed to achieve those requirements:

Property 3.5.1. **Exclusive Access:** no algorithm has access to secret key besides the remote attestation algorithm.

Property 3.5.2. **No Leakage:** no information is leaked about the secret key by the attestation token.

Property 3.5.3. **Immutability:** the remote attestation algorithm should be immutable to prevent attacks such as TOCTTOU.

Property 3.5.4. **Uninterruptibility:** all interrupts and exceptions should be disabled during the execution of the remote attestation algorithm.

Property 3.5.5. **Controlled Invocation:** the remote attestation algorithm should be only executed from expected entry points.

3.6 Blockchain

A blockchain is a ledger with immutable history. In an ideal scenario, the blockchain would support smart contracts but that it is not required, as shown in the implementation. We now define several algorithms that help with the description of our design. The blockchain interaction is done using (B.GenAddr, B.CreateTX, B.SubmitTX, B.FetchTX), where:

1. **Address generation** B.GenAddr($amount$): is an algorithm that creates a blockchain address (pk, sk) with a certain amount of trading currency

2. **Transaction creation** B.CreateTX(pk_{rec}, sk_{send}, add_info): is an algorithm that constructs blockchain transactions from a sender sk_{send} to a receiver pk_{rec}; each transaction carries additional information add_info. The additional information could be arbitrary information or input to a smart contract.

3. **Transaction submission** B.SubmitTX(tx): is an algorithm that adds an already created transaction to the blockchain.

4. **Transaction fetching** B.FetchTX($txid$): is an algorithm with input a transaction identifier $txid$ that fetches a transaction and proof that the transaction is not double-spent.

Chapter 4

Design

In this chapter, we begin with a description of the goals we set to achieve in our design. Then, we discuss the threat model and describe our system. Before concluding the chapter, we explain how our system satisfies each goal.

4.1 System Properties

In this section, we define properties that are required from the system described in Section 4.3. We begin with a description of what it means for the system to operate correctly; in other words, we explain the required system functionality. Following that, we define the security, and more specifically, we stipulate that a REC can only be produced by authorized smart meters in a secure state, and no RECs can be double-spent. Lastly, we describe the anonymity we aim to achieve.

4.1.1 Correctness

If the system operates correctly, all end-users should be able to view all non-retired RECs. Furthermore, buyers with sufficient balance should be able to bid for any REC owned by other users, and REC owners should be able to sell or retire RECs they own. We capture the correctness property in the following definition:

Definition 4.1.1. Correctness: The system is operating correctly if any authorized non-compromised smart meter can publish RECs, and any end-user can view the non-retired RECs. Only REC owners can retire or sell, provided sufficient buyer balance, the non-retired RECs they own, and any end-user can bid for any non-retired REC owned by other users.

4.1.2 Security

In a secure system, only approved non-compromised devices should be able to produce RECs, and no user should be able to double-spend RECs. An adversary that attempts to violate the security of the system is allowed to have access to a *smart meter oracle*, which can be used to produce RECs by a smart meter of choice, Additionally, we allow the adversary to own a set of smart meters and end-users in the system. The adversary is discussed in more depth in our threat model in Section 4.2.

Definition 4.1.2. Security: The system is secure if no adversary that controls a set of smart meters and it is given access to a smart meter oracle can produce a valid REC for which the oracle has not been queried, or using either a smart meter in non-approved security state or non-authorized smart meter. Additionally, at no point in time the RECs can be tampered with and the number of RECs in the system differs compared to the number of RECs produced by approved non-compromised smart meters.

4.1.3 Anonymity

We consider the anonymity goal to be satisfied if it is impossible to distinguish RECs produced from non-corrupted smart meters in the same ring group, even if they belong to the same smart meter owner. The trading platform can act as a passive adversary by following the protocol, but attempting to learn as much information as possible. Additionally, there could be an arbitrary number of compromised end-users and corrupted smart meters, except at least one other smart meter in the same ring group.

Definition 4.1.3. Anonymity: The system is anonymous if no adversary can determine the smart meter that produced any REC nor can determine the owner of the smart meter. The adversary is given access to a smart meter oracle and can control an arbitrary set of end-users and smart meters, except at least one other smart meter in the same ring group.

4.2 Threat Model

As shown in Figure 4.1, no software and hardware on the smart meter is trusted besides a small trusted computing base (TCB) where the hardware root of trust resides. Any compromise outside of the TCB is to be detected by a security service that is trusted by end-users for remote attestation.

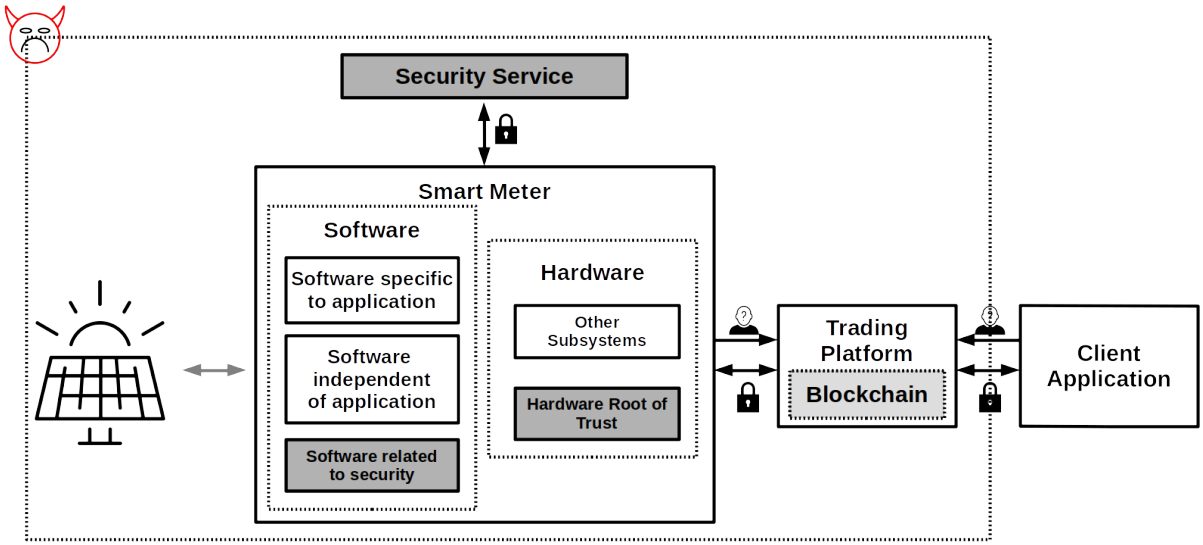

Figure 4.1: The system diagram visualizes our threat model. All trusted elements such as security service and connection between the solar panel and smart meter are in dark grey. No other elements are trusted. There are anonymous links between the smart meter and the trading platform and between the client application and the trading platform. Highlighted in light grey is the blockchain denoting that the trust depends on the choice of blockchain.

The link between the solar panel and the smart meter is currently trusted; it allows the smart meter to measure the produced green energy by the solar panel. However, potential tampering between the smart meter and solar panel will likely affect the availability of the smart meter, an indicator to be detected by the security service. Section 2.2.2 discusses possible solutions to remove the trust in the connection.

The trading platform component inherits the trust and assumptions of the underlying blockchain.

We make several assumptions in our design. First, we assume that client application, smart meter application, and trading logic are open-sourced and verifiable to implement the required functionality correctly. Second, we assume an anonymous connection between the smart meter and the trading platform and between the client application and the trading platform. Lastly, since components such as smart meter depend on the hardness of cryptographic problems such as discrete logarithm problem, we also assume a computationally bounded adversary. Now that we have explained our assumptions, we state our goals.

The goal of the system is to be secure, as described in Section 4.1.2. On the one hand, the owner of the solar panel is considered an adversary, and could attempt hardware or software attacks to increase the value of the RECs produced by the smart meter. On the other hand, an external adversary who does not have access to the hardware could attempt to perform remote software attacks, including tampering with the connections between components. Lastly, end-users might collude on the blockchain to double-spend RECs and increase their monetary possession.

The second goal of our system is to enhance RECs with anonymity, as described in Definition 4.1.3. The trading platform can act as a passive adversary by following the protocol and returning to the smart meter appropriate anonymity set during registration; however, it can attempt to identify the smart meter or owner of a REC by just observing newly created RECs. We assume an passive adversary who has a goal to learn any information about an energy producer via RECs. For example, energy usage information could reveal the lifestyle of small-scale energy producer such as household with solar panels depending on the granularity of the RECs.

4.3 System Design

We now describe a system for trading RECs based on the primitives in Chapter 3. As shown in Figure 4.2, the system includes a renewable energy source—we use a solar panel through the thesis—smart meter, security service, trading platform, and client application. First, the solar panel generates green energy that is measured by the smart meter. Meanwhile, the security service continuously verifies the smart meter's internal security state and authenticity. When enough energy is produced, the smart meter creates and sends RECs to the trading platform, where the RECs are traded. Lastly, end-users interact with the client application to interface the trading platform.

4.3.1 Security Service

Security service provides two functionalities: remotely deploying new smart meter custom software and periodically verifying the security state and authenticity of smart meters. It facilitates the detection of frauds and removes the need for annual third-party audits to help include small-scale energy producers. The component uses the remote attestation verification algorithm RA.Vrf().

- **Initialization** SS.Init(): $(pk_{SS}, sk_{SS}) \leftarrow$ GenKeys()

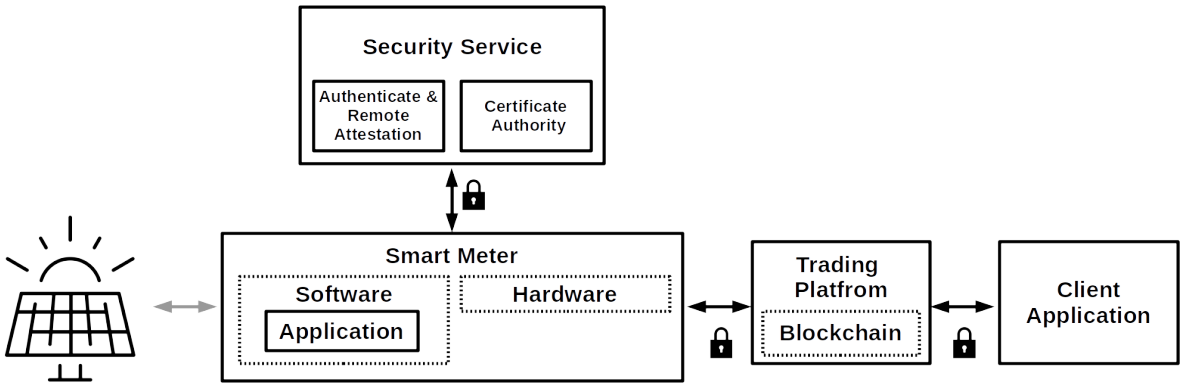

Figure 4.2: A system architecture diagram for trading RECs. The green energy produced by the solar panel is measured by the smart meter, while the security service verifies the internal state and authenticity of the hardware. The microcontroller periodically sends RECs to the trading platform, which at its core, is using a blockchain. End-users can trade RECs by interacting with the client application functionality.

- **Deploy Custom Software** SS.NewApp(sk_{SS}, new_app): The security service signs and distributes the provided custom software new_app to all smart meters. More formally, the security service creates and sends a message (NEW_APP, app) to all devices, where $app = (new\_app, \sigma_{new\_app})$ and σ_{new_app} = S.Sign(sk_{SS}, new_app). We consider user authorization and software versioning to be orthogonal issues to our work.

- **Verify Remote Attestation** SS.Att(pk_{SS}, sk_{SS}): The service periodically checks the security state and authenticity of smart meters using remote attestation token verification. If successful, the security service sends to the smart meter a signed result of the remote attestation. More formally:

  - First, the secure service initiates the procedure by sending (ATTEST) to a smart meter.
  - Upon receiving (VERIFY, pk_{Att}, pk_{SM}, s, α), the algorithm first verifies the remote attestation token α. That is, the security service invokes RA.Vrf(pk_{Att}, s, α), where s is a smart meter state and α is an attestation token. If the verification fails, then the algorithm outputs **fail** . Otherwise, it sends back a message (PROOF, ϕ), where the proof ϕ is a tuple (pk_{SM}, s, α, σ_ϕ), which can be used to determine the deployed application, security state and authenticity of the smart meter. The signature σ_ϕ is the output of S.Sign(sk_{SS}, (pk_{SM}, s, α)).

4.3.2 Smart Meter

Each smart meter has application-independent functionality to receive new applications and remotely attests, and application-dependent functionality based on the deployed custom software. In our system, the application-dependent functionality is smart meter metadata update and production of RECs. The former is periodically triggered after each remote attestation, while the latter is trigger when a condition such as producing sufficient energy is fulfilled. Since smart meters are a source of trust in the system, they use the secure hardware as defined in Section 3. Additionally, each smart meter has access to the public key of the security service pk_{SS} and the remote attestation algorithms RA.Att() and RA.Setup().

- **Initialization** SM.Init(): The algorithm creates attestation key pair (pk_{Att}, sk_{Att}) using RA.Gen() and device identity key pair (pk_{SM}, sk_{SM}) using GenKeys(). Furthermore, the blockchain address of the owner pk_{owner} is stored internally.

Remark. Similar to the remote attestation secret key, any operations with the identity secret key sk_{SM} should only be exposed via precise hardware API. That is, both keys are part of the root of trust.

Application-independent functionality

- **Receive new application** SM.NewApp(pk_{SS}): Received applications are first verified to be signed by the security service and then instantiated using the secure hardware functionality. More formally, when the smart meter receives a message (NEW_APP, *app*), it invokes S.Vrf(pk_{SS}, new_app, σ_{new_app}). If the verification does not fail, the application new_app is instantiated using SH.Load(new_app).
- **Remote attest** SM.Att(pk_{Att}, sk_{Att}, pk_{SM}, s): Remote attestation is invoked using smart meter's security state s. A successful exchange of messages concludes with storing the security proof internally. More formally:
  - When the smart meter receives a message (ATTEST), it generates an attestation token α using RA.Att(sk_{Att}, s). If successful, the algorithm sends to the security service a response (VERIFY, pk_{Att}, pk_{SM}, s, α).
  - When the smart meter receives a message (PROOF, ϕ), the proof ϕ is stored internally. The algorithm terminates by outputting the proof ϕ. If a response is not received, the algorithm outputs **fail** .

Application-dependent functionality

A remotely deployed custom software implements the application-specific functionality, in our case algorithms that interacts with the trading platform. Algorithms are executed using SH.Exec(APP, in), where in is supplied by the environment, while the output of the execution is a tuple (out, ϕ) that is sent to the trading platform. The custom software has access to the stealth address generation algorithm SA.GenAddr(), ring signature algorithm RS.Sign(), and public key of the owner pk_{owner}. Additionally, the application has a buffer $ring\_list$ that stores the anonymity set—a set of public keys—for each REC produced by the smart meter.

- **Metadata Update** APP.Update(pk_{SM}, sk_{SM}, in): Smart meter metadata update algorithm is periodically invoked with input in, such as GPS coordinates. The output is a message that is sent to the trading platform. If a trading platform reply is received, the response, an anonymity set, is stored internally. More formally, the algorithm outputs a message (UPDATE, m), where the message m is a tuple (in, σ_m) and the signature σ_m is S.Sign(sk_{SM}, in). If the smart meter receives a trading platform response (UPDATE_SUCC, L), the list L of smart meter public keys is stored in an internal buffer $ring\_list$. The algorithm outputs $ring\_list$, else outputs **fail** .

- **Create a new REC** APP.NewREC(pk_{SM}, sk_{SM}, in, $ring\_list$, pk_{owner}): The algorithm is periodically invoked with input in, a measurement of the produced energy, for example, 1 MWh or even more fine-grained to facilitate the inclusion of small scale energy producers. Then, a REC is created with an owner property set to a blockchain address generated using stealth addresses. Lastly, the REC is signed using the set of identities in $ring\_list$. More formally, the output out is set to (NEW, REC, $ring\_list$), where REC is a tuple (in, $pk_{RECOwner}$, γ, σ_{ring}). The owner address $pk_{RECOwner}$ and the additional information γ are the result of SA.GenAddr(pk_{owner}), while the ring signature σ_{ring} is computed using the invocation of RS.Sign(sk_{SM}, $ring\_list$, (in, $pk_{RECOwner}$, γ)). If all steps are successful, the algorithm outputs (REC, $ring\_list$), otherwise output **fail** .

4.3.3 Trading Platform

The trading platform exposes an API for smart meters to update their metadata or send created RECs and for end-users to trade RECs. The blockchain in the trading platform maintains the trust established by the smart meter. It prevents double-spending without

the need to rely on centralized authority and helps identify inconsistencies via increased transparency. Additionally, the combination of smart meters and blockchain results in less overhead that permits for more fine-grained RECs.

Besides being able to interact with all blockchain algorithms, the trading platform has access to the public key of the security service pk_{SS}. All smart meters that have been able to successfully update their metadata are stored in a buffer *meter_list*, while all non-retired RECs are stored in buffer *rec_list*. The trading platform has access to a list *auth_keys* of authorized smart meter identities pk_{SM} (see Section 5.2.2).

- **Initialization** TP.Init(): a blockchain address pk_{TP} that represents the trading platform is created using blockchain address generation algorithm B.GenAddr().

Interaction with Smart Meters

- **Smart Meter Update** TP.Update(pk_{SS}, *auth_keys*): Before accepting a metadata update message, the algorithm ensures that the message has been sent by an authorized smart meter and that the sender has a security state approved by the security service. If successful, the message is included in a transaction submitted to the blockchain and saved in the buffer *meter_list*. Lastly, the algorithm determines and sends to the smart meter a list of the closest physically located smart meters.[1] More formally, when the message (UPDATE, m, ϕ) is received, the algorithm verifies ϕ's signature by invoking S.Vrf(pk_{SS}, (pk_{SM}, s, α), σ_ϕ) and m's signature using S.Vrf(pk_{SM}, in, σ_m). A blockchain transaction tx_{reg} is created using B.CreateTX(pk_{TP}, sk_{TP}, (m, ϕ)) and sent to the blockchain using B.SubmitTX(tx_{reg}). Finally, the algorithm uses the transaction identifier *txid* of the sent transaction and adds the tuple (pk_{SM}, ϕ, *txid*) to the buffer *meter_list*, determines anonymity set L by selecting a set of keys from the buffer *meter_list*, and sends back a response (UPDATE_SUCC, L). If any of the verifications fail or the smart meter identity pk_{SM} is not in the list of keys *auth_keys*, the algorithm outputs **fail** ; else, it outputs L.

- **New REC** TP.NewREC($MeterList$): The algorithm first verifies that the newly submitted REC has a valid ring signature and each smart meter in the ring group has been able to successfully update their metadata. Then, the security proof of each smart meter in the ring is verified to be valid. If successful, a new transaction that includes the REC is submitted to the blockchain and stored in buffer *rec_list*.

[1] The same ring group is returned to every smart meter in the group when updating the metadata

More formally, upon receiving the message (NEW, REC, $ring\_list$), the algorithm verifies REC's ring signature σ_{ring} using RS.Vrf($ring\_list$, (in, $pk_{RECOwner}$, γ), σ_{ring}). Furthermore, the algorithm checks if every smart meter identity pk_{SM} in the list $ring\_list$ is also present in the $meter\_list$ with still valid security proof ϕ. If any of the verifications fail, output **fail** . Otherwise, a blockchain transaction tx_{REC} is created using B.CreateTX($pk_{RECOwner}$, sk_{TP}, (REC, $ring\_list$)) and submitted to the blockchain using B.SubmitTX(tx_{REC}). Lastly, the algorithm adds the tuple (REC, $ring\_list$, $txid$) to rec_list.

Interaction with Client Application Component

- **Fetch RECs** TP.Fetch(): When the algorithm receives a message (FETCH), it returns a response (ALL_RECS, rec_list), where rec_list is the buffer storing all RECs.
- **Verify REC** TP.Vrf(): Once a request with a transaction identifier is received, the algorithm fetches the blockchain transaction. Then, it creates a list of smart meter identities and their security state proofs for every smart meter in REC's anonymity set. Lastly, the fetched blockchain transaction and the created list is returned to the client application. More formally, when a request (VRF, $txid$) is received, the algorithm fetches the blockchain transaction tx and its proof ϕ_{tx} using B.FetchTX($txid$). Using the lists rec_list and $meter\_list$, the anonymity set of the REC is determine to create a new list $ring\_and\_proof$ with tuples (pk_{meter}, ϕ, $txid$). If any of the smart meters cannot be found or the transaction cannot be fetched, the algorithm outputs **fail** ; otherwise, a response (VRF_REC, tx, ϕ_{tx}, $ring\_and\_proof$) is sent back to the client application.
- **Retire REC** TP.Retire(): The algorithm forwards a retire transaction to the blockchain, which accepts the transaction if the REC owner is also the blockchain transaction sender in the retire transaction and the REC has not be retired. If successful, the REC is removed from the internal list of RECs, and a response with the retire transaction identifier is sent back. In terms of notations, when a request (RETIRE, tx_{retire}) is received, the algorithm uses B.SubmitTX(tx_{retire}) to submit the transaction to the blockchain. If successful, the algorithm removes the REC from rec_list and sends back a response (RETIRE_SUCC, $txid_{retire}$) to the client application; otherwise, it outputs **fail** .
- **Sell REC** TP.Sell(): The algorithm forwards a sell transaction to the blockchain, which accepts the transaction if the REC owner is also the blockchain transaction

sender in the sell transaction and the user-chosen buyer has sufficient balance. If successful, the REC ownership is atomically swapped in the blockchain for the bid value, and the REC ownership in the internal REC buffer is updated. More formally, upon receiving a request (SELL, tx_{sell}), the algorithm uses B.SubmitTX(tx_{sell}) to submit the transaction to the blockchain. If successful, the owner pk_{owner} and the transaction identifier $txid$ of the REC are updated in the internal buffer rec_list. Lastly, the algorithm sends to the client application a response (SELL_SUCC, $txid_{sell}$).

- **Bid REC** TP.Bid(): The algorithm forwards the bid transaction in the request to the blockchain and sends back the transaction identifier. More formally, when a request (BID, tx_{bid}) is received, the algorithm invokes B.SubmitTX(tx_{bid}) and sends back (BID_SUCC, $txid_{bid}$).

- **Check Bid** TP.CheckBid(): The algorithm checks if a bid has been successful. More formally, when the request (CHECK_BID, pk_{Bid}, tx_{bid}) is received, the algorithm returns (CHECK_BID_SUCC), if there is a tuple in the internal buffer rec_list that has an owner property pk_{Bid}. If there isn't, the algorithm outputs **fail** .

4.3.4 Client Application

The client application is the end-user interface to the system. It provides functionality such as buying, selling, retiring, and verifying RECs. It also has access to the public key of the security service, blockchain algorithms B.GenAddr() and B.CreateTX(), and stealth address algorithms SA.Gen() and SA.Vrf(). The buffers rec_own and rec_bid—as the name suggests–store the RECs owned by the user and the RECs available for bidding.

Remark. When the blockchain used is permissionless, end-user with sufficient resources can use the client application to directly interact with the blockchain.

- **Initialize** CA.Init(amount): Each client generates stealth address (pk_{owner}, sk_{owner}) using SA.Gen(), which is used by the smart meter. Additionally, another blockchain identity (pk_{Bid}, sk_{Bid}) is created using B.GenAddr(amount), specifically to bid for RECs.

Trading platform interaction

- **Fetch RECs** CA.Fetch(sk_{owner}): When invoked by the user, the algorithm requests all non-retired RECs from the trading platform. The received list of RECs is split

into a list of RECs owned by the user and a list of RECs owned by other users. More formally, the algorithm sends a (FETCH) request to the trading platform. When the response (ALL_RECS, rec_list) is received, the algorithm determines which RECs are owned by pk_{owner} using SA.Vrf($pk_{RECOwner}$, γ, sk_{owner}). For each successful REC verification in the rec_list, the algorithm stores the tuple (REC, $sk_{RECOwner}$, $txid$) in the buffer rec_own, otherwise it stores the tuple (REC, $txid$) in the buffer rec_bid. If the response from the trading platform was empty, the algorithm outputs **fail** ; otherwise, it outputs the tuple (rec_owned, rec_bid).

- **Verify REC** CA.Vrf($txid$): The algorithm requests a REC blockchain transaction specified by the user input. The response by the trading platform is verified to ensure that the REC is not double-spent, it has not been tampered with, and the smart meters in the REC anonymity set have valid proofs. More formally, the algorithm sends a request (VRF, $txid$). When the response (VRF_REC, tx, ϕ_{tx}, $ring\_and\_proof$) is received, the algorithm first verifies the proof ϕ_{tx} to ensure that the transaction is not double-spent. Following that, the ring signature validity is checked using RS.Vrf($ring\_list$, (in, $pk_{RECOwner}$, γ), σ_{ring}). Lastly, for each smart meter in the list $ring\_and\_proof$, the proof ϕ and metadata update transaction $txid$ is verified to be valid. If any of the verifications fail, the algorithm outputs **fail** .

- **Retire REC** CA.Retire(rec_own, $txid$)): The algorithm creates a blockchain transaction that retires a REC specified by the user and the locally created transaction is sent to the trading platform. More formally, the user specified transaction identifier $txid$ and the buffer rec_own are used to create a retire transaction tx_{retire}, the result of B.CreateTX(pk_{TP}, $sk_{RECOwner}$, REC). Then, the algorithm sends a message (RETIRE, tx_{retire}) to the trading platform. If no response is received, then output **fail** .

- **Sell REC** CA.Sell(pk_{Bid}, rec_own, $txid$): The algorithm creates a blockchain transaction that sells a REC to a user specified in the input. The newly created transaction has a receiver set to the blockchain address of the bidder, and it contains a REC with an owner set to the same address. Lastly, the transaction is sent to the trading platform. More formally, using the user provided blockchain address pk_{Bid}, transaction identifier of the REC to sell $txid$, and buffer rec_own, the algorithm creates a sell transaction tx_{sell} using B.CreateTX(pk_{Bid}, $sk_{RECOwner}$, REC). Then the message (SELL, tx_{sell}) is sent to the trading platform. If no response is received, the algorithm outputs **fail** . Otherwise, it uses the trading platform's response to update the new transaction identifier $txid$ of the REC in rec_own and then moves the REC from list rec_own to rec_bid.

- **Bid REC** CA.Bid($pk_{RECOwner}$, sk_{Bid}, amount): Using the user provided amount, a blockchain bid transaction for the REC owned by the blockchain address $pk_{RECOwner}$ is created and submitted to the trading platform. More formally, the algorithm creates a bid transaction tx_{bid} using B.CreateTX($pk_{RECOwner}$, sk_{Bid}, amount). The transaction is then sent to the trading platform as a message (BID, tx_{bid}). Upon receiving the response (BID_SUCC, $txid_{bid}$), the transaction identifier $txid_{bid}$ is inserted in the list *rec_bid*; the REC tuple becomes (REC, ($txid_{bid_1}$, .., $txid_{bid_n}$), $txid$). If no response is received, then the algorithm outputs **fail** .

- **Check Bid** CA.CheckBid(pk_{Bid}, $txid_{Bid}$): Using the user provided transaction identifier $txid_{bid}$, a request is sent to the trading platform to check if a bid has been successful. More formally, the request (CHECK_BID, pk_{Bid}, tx_{bid}) is sent. If a response is received, a REC is moved from list *rec_bid* to *rec_own*, with an updated public key and transaction identifier $txid$; otherwise, the algorithm outputs **fail** .

4.4 User Registration

A new user that owns a renewable energy source and wants to create and sell RECs needs to submit a request to join the system. The owner first chooses a REC aggregator from a set of approved REC aggregators available in a central place, where each aggregator operates a trading platform that is verified to comply with a common standard. The standard ensures compatibility between REC aggregators so that end-users can buy any REC in the system. Using the client application of the chosen REC aggregator, the owner creates a blockchain identity and submits a request to join the market; the request has information sufficient to determine eligibility according to the local jurisdiction.

If eligible, the REC aggregator deploys the smart meter. During the deployment, the renewable energy source is confirmed to be eligible, a one time process as opposed to, for example, annual audit. Additionally, the smart meter is initialized with information such as credentials for an internet connection and smart meter's owner blockchain address, and smart meter's identity is added to the list *auth_keys* of approved devices that can create RECs. We note that we have not implemented a measurement again tampering with the link between the smart meter and the renewable energy measurement source such as an invertor or sensor. Therefore, the link needs to be physically protected.

4.5 Security Analysis

We provide an informal analysis how the high-level system in Section 4.3 satisfies the security property in Definition 4.1.2 and anonymity property in Definition 4.1.3.

4.5.1 REC Security

Theorem 4.5.1. Assuming that link between the smart meter and solar panel is secure and remote attestation, secure hardware, and blockchain are correctly implemented and satisfy the defined properties, the system is secure as defined in Definition 4.1.2.

To support the theorem, we follow the life-cycle of a REC:

Assertion 4.5.1. Authorized smart meters cannot be impersonated.

During initialization, each smart meter internally generates two key pairs, as shown in SM.Init(). An adversary attempting to impersonate the device has a goal to gain access to either attestation secret key sk_{Att} or smart meter secret key sk_{SM}.

Compromising the remote attestation key is not possible. Property 3.5.1 requires that remote attestation has exclusive access to the secret key and Property 3.5.2 mandates that attestation results do not reveal information about the secure key. Additionally, the hardware root of trust requires that the key is only accessible via hardware API and protected against physical side-channel attacks; that is, no rouge software can leak the key. Therefore, since an attacker cannot physically extract the secret key, no rough software can access it, and no information is leaked thought the results using the secret key, the adversary cannot compromise the key sk_{Att}.

The argument that an adversary cannot steal the smart meter secret key sk_{SM} is similar. The key is used for singing in algorithm SM.Update() and ring signing in SM.NewREC(), which, based on their respective security definitions, do not leak information related to the key. Additionally, Property 3.4.1 of the secure hardware limits the access to the key via well-defined hardware API. Therefore the adversary cannot steal the key.

Remark. Signing and ring signing using the smart meter secret key sk_{SM} might seem to add a significant amount of software and hardware to the TCB, which is not desirable. However, remote attestation algorithms, which are already in TCB, already need signing. Additionally, ring signatures can be implemented using similarly building blocks as signing (e.g. [68]). Therefore, we believe that the operations using the key sk_{SM} and the key sk_{SM} will be a small addition to the TCB.

Assertion 4.5.2. RECs can only be produced by a smart meter that is authorized and in an approved security state.

First, the algorithm TP.NewREC() checks that each smart meter creating a REC is authorized. However, since it is not possible to determine the exact REC creator, the algorithm checks if every smart meter in the ring group used to sign the REC is authorized. Every smart meter that can create a REC is an entry in the list *meter_list*, so the algorithm checks that all smart meters in the ring list are present in the list *meter_list*. It is equivalent to checking if all devices have been able to update metadata using the algorithm TP.Update(). A smart meter can only update metadata if the smart meter identity is in the list of the authorized smart meter identities.

Second, algorithm TP.NewREC() verifies that the smart meter is in an approved state. The security proof ϕ that is sent by the smart meter during metadata update is used for the verification. Francillon et al. [43] explain that the properties immutability (Property 3.5.3), uninterruptible (Property 3.5.4), and controlled invocation (Property 3.5.5) guarantee that the proof correctly captures the internal device security state and any malicious changes to the application.

Assertion 4.5.3. RECs cannot be tampered with and double-spent.

Every REC is signed using a ring signature algorithm, and therefore no REC can be tampered with, which is mandated by the security (Definition 3.2.2) definition of ring signatures.

The trading platform uses a blockchain, and therefore, once a REC is added to the blockchain, we default to the blockchain security assumption to enforce no double-spending of RECs.

4.5.2 Anonymity

Theorem 4.5.2. Assuming that there exists an anonymous connection between the smart meter and the trading platform and the stealth address and ring signature are correctly implemented and satisfy the defined properties, the system is anonymous as defined in Definition 4.1.3.

Assertion 4.5.4. RECs cannot be attributed to a specific smart meter nor a specific owner of a smart meter.

There are two possible sources that could reveal the smart meter that created the REC or the smart meter owner.

The first possible source is the REC signature. However, since each REC is signed using ring signature, Definition 3.2.3 requires that no adversary can determine the signer of a message. The most an adversary can learn is that the smart meter is one of the parties in the ring group. Additionally, we assume that the trading platform component is a passive adversary; that is, the trading platform does not deviate from the registration algorithm and returns to a group of smart meter the exact same group of identities, which are geographically closely located.

The second possible source that could lead to determining the smart meter that created a REC is the information in the REC. In our design, the REC includes the amount of produced energy and the owner of the REC, which is generated using stealth addresses. However, Definition 3.3.3 requires that a stealth address of the REC owner does not reveal information about the smart meter owner.

Remark. The amount of information revealed about a REC through transaction history is dependent on the blockchain used for the implementation. However, we note that each REC is considered an indivisible and non-mergeable asset, where each asset is traded separately. Therefore, a passive adversary that purely observes the blockchain transactions cannot link two RECs to the same smart meter or end-user address that owns the smart meter.

Chapter 5

Implementation

We now describe the implementation of the proposed design in Chapter 4. First, we begin with a description of the smart meter component instantiation. Then, we focus on two implementations of the trading platform and the client application. The first one is based on a permissioned blockchain, Hyperledger Fabric, a solution that is troublesome to bootstrap. To address that, we present a second solution based on already bootstrapped permissionless Blockchain, Algorand. It is important to note that our implementation focuses on guaranteeing REC security, and anonymity is left for future work.

5.1 Smart Meter Component

We define the smart meter component in the high-level design as a resource-constrained device that has several requirements. It has to have minimal functionality that supports secure remote software updates and secure software execution; the software execution outputs not only the result of the execution but also authenticity and healthiness proof. In addition to the functionality, we also required that the hardware has a hardware root of trust.

To the best of our knowledge, the only commercially available resource-constrained device that supports the required functionality and properties is Azure Sphere. The hardware root of trust is implemented by Microsoft's custom-designed Pluton security subsystem. Additionally, the trusted computing base is limited to the Pluton security subsystem, its runtime, and the security monitor. Section 2.1.2 has an in-depth explanation how additional properties are satisfied.

In addition to the high-level properties, Azure Sphere also supports all six hardware features defined by Francillon et al. [43] that are required for unforgeable and accurate proofs of authenticity and healthiness. First, custom hardware is required to enforce access to the security key. In Azure Sphere, this is implemented via specially dedicated fuse control hardware [76], which controls the access to the device keys stored in e-fuses within Pluton security subsystem. Second, a secure memory erasure feature is required [43] to

prevent leaking information about the secret device key. To address that, Azure Sphere has dedicated memory for Pluton engine, which is not accessible outside the engine, so information about the secret key cannot be leaked. Third, Microsoft's device has read-only memory in the Pluton security subsystem. The ability to atomically disable interrupts and invoke attestation only at the first instruction are another two features [43]. To the best of our knowledge, Microsoft does not discuss those features in publicly available information, but they seem to be addressed by the Pluton engine; remote attestation is handled as a single command to the complex command engine, which cannot be interrupted by adversary, nor adversary can control the invocation. Lastly, Francillon et al. [43] require a secure reset mechanism feature. Azure Sphere has an accumulation register that is dedicated for remote attestation measurement. This register can only be reset when the entire chip is reset [63]. Now that we have explained that the Azure Sphere satisfies all requirements to establish a secure environment, we explain the process of establishing a root of trust in each device.

We trust the manufacturer and Microsoft for specific actions. First, we trust the manufacturer for correctly manufacturing Azure Sphere and loading security information; we describe in Section 5.1 in more detail the exact security information. Second, we trust Microsoft for remotely attesting each Azure Sphere and enforcing the users, service company staff, who are authorized to deploy applications.

Azure Sphere life-cycle

The life-cycle of each Azure Sphere can be split into three phases, which we refer to as pre-deployment, deployment, and operational phase, as shown in Figure 5.1. The first two are crucial for a correct bootstrapping of the smart meter, while the last one describes the operation of the device. In the pre-deployment phase, which occurs during manufacturing, the microcontroller is supplied with information related to critical functionalities such as secure boot. Following that is the deployment phase; it begins when a solar panel is physically installed and ends right before the Azure Sphere executes any applications. It includes configurations made by a service company staff and interactions with Azure Cloud, such as remote attestation and application deployment. The last phase encompasses the microcontroller's measurement of green energy and the production of RECs. We now describe each phase in more detail.

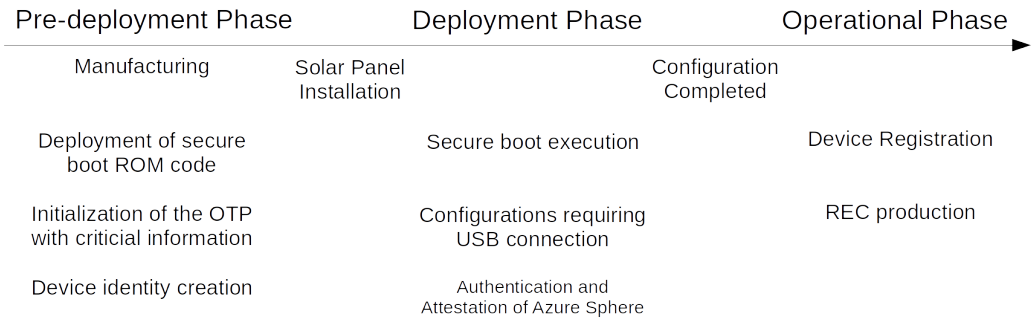

Figure 5.1: The visualization shows smart meter's life-cycle phases in chronological order from left to right. The steps below each phase occur within the phase in chronological order from top to bottom. In the space between the phases, the visualization denotes the physical events that separate the phases.

Pre-deployment Phase

The pre-deployment phase includes all steps taken to manufacture each Azure Sphere. In this section, we describe what the manufacturer is trusted for based on the publicly available information.

The manufacture is trusted for two actions besides correctly manufacturing the device. First, the UMC, a Taiwanese company, is responsible for supplying a correct read-only memory; the ROM cannot be updated, and it stores the first executed code when a device comes out of reset. The code is essential to secure boot—the feature that guarantees that only a software stack signed by Microsoft is executed. Second, the UMC is responsible for initializing a 4 kB one-time programmable (OTP) array, which is composed of e-fuses [76]. During manufacturing, OTP goes through several states, starting from a blank state. Subsequently, the UMC loads in the OTP a public key that is used by Azure Sphere to verify loaded software—whether it has been signed by Microsoft—and adds other critical to the security information. According to Stiles at el. [76], the device eventually enters a secure production state that prevents attackers from accessing the OTP.

Nonetheless, it is important to emphasize that UMC is not trusted for generating and protecting the identity of Azure Sphere. As already explained in the background chapter, the Pluton security engine has a hardware random number generator (HRNG). During one of the OTP states, the HRNG internally generates the key pair that uniquely identifies the device [76]; the process cannot be controlled by the manufacturer. In our high-level design, this key pair is referred to as (pk_{Att}, sk_{Att}). The private key is stored in OTP, and it cannot be leaked by rogue software since no software has direct access to the key. The manufacturer, however, is responsible for giving the correct public key pk_{Att} to Microsoft

Azure.

It is important to mention that in our design, we require a second key pair (pk_{SM}, sk_{SM}) that is envisioned to be generated in a similar fashion. Currently, Azure Sphere does not provide such a key pair, and to deal with that, as explained later, we generate it in the application. We believe, however, that Azure Sphere does not need a significant change to provide such a pair in the future.

Deployment Phase

The deployment phase, which begins after a solar panel is installed, starts with Azure Sphere boot up. Pluton security subsystem processor is the first to come out of reset using a secure boot [76]—one of the hardware features establishing root of trust. The code in ROM checks the initial security device state, and any subsequent code is executed if it has been signed by Microsoft. Similarly, every loaded piece of code checks any subsequent code for a signature to maintain the integrity of the device. Validated code resides and does not leave the tightly coupled memory. While this process is happening, the hardware keeps a cryptographic hash of the boot [76].

After a successful boot-up, the staff of a service company performs four configurations that require a physical USB connection. The first one is to claim the Azure Sphere(see Section 2.1.2) in the solar panel. The second configuration is setting up an internet connection. If wireless connection, staff members need to provide credentials for an access point; if wired connection, staff members need to load an additional piece of software signed by Microsoft. Third, Azure Sphere needs to be associated with an appropriate product and device group to receive software updates (see Section 2.1.2). Lastly, the device needs to be locked, so that only cloud application can be executed; the procedure stores information in flash to only trust applications signed by a particular Azure Sphere Tenant.

After a successful configuration, Azure Sphere connects to Azure Sphere Security Service, a service that instantiates to the security service component in our high-level design. The service has three sub-components [4]:

- **Device Authentication and Attestation**: the point of contact for all Azure Spheres; it provides an interface to the other two components.
- **Device ID Verification**: authenticates and remotely attests Azure Spheres.
- **Certificate Authority**: issues device certificates, based on X.509 standard, that can be presented to any external services as an authenticity and healthiness proof. The certificates correspond to the proof ϕ in our high-level design.

The first interaction between Azure Sphere and Azure Security Service is remote attestation. The cloud—more specifically, the Device ID Verification sub-component—sends a challenge to the device, which then responds with the security subsystem's measurement of the executed code and a signature of the challenge and the measurement. To sign the response, Azure Sphere uses the attestation key pair (pk_{Att}, sk_{Att}). This simple challenge-response protocol also prevents replay attacks and corresponds to SM.Att() and SS.Att() in our high-level design.

After a successful remote attestation, the Certificate Authority sub-component issues an X.509 certificate that is valid for a day. The issuer, intermediate Certificate Authority, is also deployed on Azure Cloud, and it is managed by an Azure Sphere Tenant. Any REC without a valid X.509 certificate is not trusted. Although not implemented in this work, the remote attestation measurements included in the X.509 certificate can potentially be used by end-users to verify information such as the exact kernel version running on the device.

In the last step of the deployment phase, each Azure Sphere receives an appropriate application. Users that are given permissions by the service company can upload an application to Azure Cloud and specify the product and device group to be distributed to. Once Azure Sphere receives the application successfully, the device is ready to measure the solar panel's produced green energy and produce RECs—functionality implemented by the application and described in the next section.

Operation Phase

In the final phase, each Azure Sphere continuously executes the application distributed by the cloud. The application in our implementation handles the interaction with the trading platform; that is, it handles device registration, corresponding to the high-level design algorithm SM.Reg(), and REC production, corresponding to the high-level design algorithm SM.NewREC(). The final application is composed of two sub-application:

- The **Real-Time Core Sub-Application** is deployed on one of the ARM Cortex-M real-time cores; it runs on bare metal, and it continuously measures the produced green energy via two channels on the ADC peripheral. We rely on the real-time core guarantees for precise meter readings. Although we successfully use the ADC peripheral to calculate the solar panel's energy, we simulated the readings in our prototype, since our goal is to test the system's functionality; that is, we did not attach Azure Sphere to a real solar panel. The real-time core does not have internet

access by design; that is why readings are sent to the high-level core via intercore communication. The final real-time core application is based on Microsoft's ADC and intercore communication[1] examples with tweaks to fit our use case.

- The **High Level Sub-Application** is written in C language and it is deployed on the A7 high-level core. We implement two different versions of the application: one for the trading platform based on Hyperledger Fabric and one for the trading platform based on Algorand.

  In the implementation for Hyperledger Fabric, the application receives meter readings from the real-time core, formats them in a format expected by the trading platform, and sends them to a cloud service called IoT Hub (the service is described in Section 5.2.1). Similar to the real-time core application, the high-level application is based on Microsoft's examples with tweaks to fit our use case. It is important to note that during the Hyperledger Fabric implementation, Azure Sphere did not have an exposed API to access the device certificate. As a result, this application does not implement high-level algorithm SM.Reg(). Instead, staff in the service company is trusted to add the appropriate device identities to Hyperledger Fabric.

  The second implementation of the sub-application instantiates both high-level design algorithms SM.Reg() and SM.NewREC(). As already mentioned, we focused on providing REC security, and REC anonymity is left for future work. As a result, the application does not use ring signatures and stealth addresses described in the high-level algorithm SM.NewREC(), instead RECs are singed with device secret key sk_{SM} and owned by the address pk_{owner}. We provide more details about the application as we explain the Algorand implementation in Section 5.2.2.

5.2 Trading Platform and Client Application Components

We now describe two implementations of the trading platform and client application components. The first one is based on permissioned blockchain, Hyperledger Fabric. Although the implementation provides REC security, bootstrapping a permissioned blockchain could be troublesome for energy companies. As a result, we describe a second implementation based on already bootstrapped permissionless blockchain, Algorand.

[1]Both can be found on GitHub repository: https://github.com/Azure/azure-sphere-samples/tree/master/Samples/

5.2.1 Implementation using Hyperledger Fabric

As shown in Figure 5.2, each Azure Sphere publishes RECs to Hyperledger Fabric via Azure Cloud. We envision that several energy companies maintain the blockchain, and the open-source chaincode correctly implements the trading logic. Therefore, end-users trust several energy companies to not collude and alter the chaincode. Additionally, end-users trust Microsoft and a service company—which maintains the cloud service—to not tamper with the specified throughout this thesis cloud service configurations.

Since during development Azure Sphere did not have an exposed API that can be used to access the X.509 client certificates, it is necessary to use Azure Cloud as a workaround. More specifically, we use the cloud to ensure that each Azure Sphere is authentic and in a secure state. A side benefit is the added availability and safety of cloud services. In case the component between Azure Cloud and blockchain becomes unavailable, the cloud services will retain RECs anywhere between a day to a week, depending on service plan and settings.

To facilitate a simple communication between Azure Cloud and Hyperledger Fabric, we introduce a component, which we refer to as HL client. Only HL client has access to the blockchain; in Section 5.2.1, however, we explain how this component would be removed in a real-world deployment. In the following section, we describe the cloud services used in the implementation and their purpose.

Azure Cloud

Device Provisioning Service

The device provisioning service (DPS) [2] automatically enrolls Azure Spheres to appropriate *IoT Hub*—a service described in the next section. Additionally, the process involves verification of X.509 client certificate issued based on remote attestation. If the verification is successful, DPS registers the device with IoT Hub and returns to Azure Sphere information needed to connect to IoT Hub. In Figure 5.2, this process corresponds to step 2 and 3. The verification of X.509 certificate is the main reason why this service was used, and it is the workaround to not having direct access to the X.509 certificate on Azure Sphere.

The setting up process involves several steps. First, the service company needs to add its Azure Sphere Tenant certificate to the service. Then, the company needs to submit a response to a cloud challenge that demonstrates access to the private key corresponding to the tenant's certificate. Lastly, the company needs to set up an enrollment group that

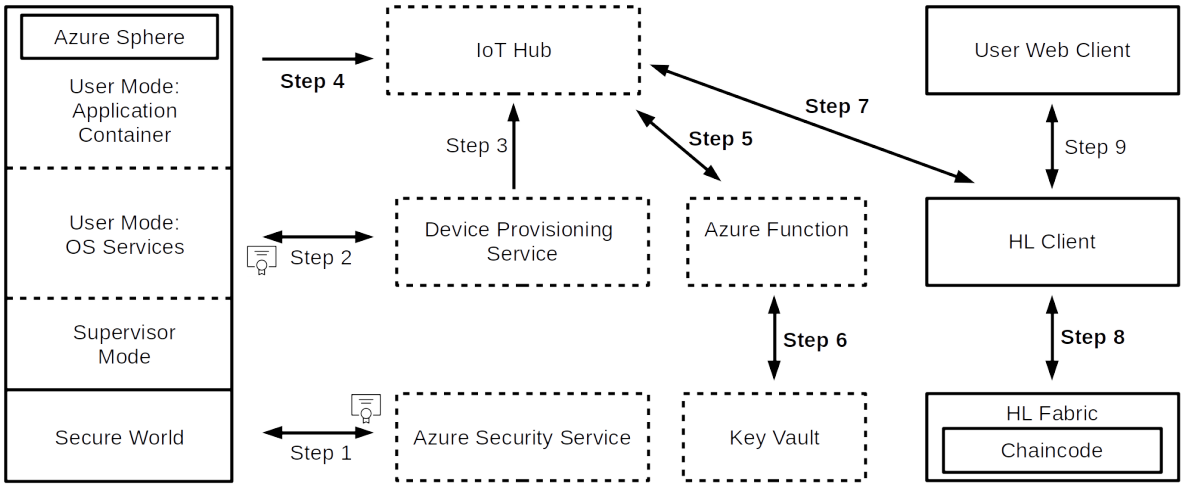

Figure 5.2: The figure visualizes a Hyperledger Fabric implementation of the high-level design. Steps 1 through 9 chronologically show the information flow. The first three steps establish the authenticity and security of Azure Sphere, while the highlighted in bold steps 4 through 8 show the continuous REC production. Lastly, in dashed squares we differentiate Azure Cloud services.

integrates IoT Hub and DPS [2]. Once the DPS is configured, each Azure Sphere claimed by the Azure Sphere Tenant is automatically associated with the appropriate IoT Hub.

IoT Hub

IoT Hub [3] is a service that provides low-latency and highly reliable communication between IoT devices and cloud.

We make several IoT Hub configuration, which are important to replicability. Although several underlying protocols can be chosen in the service, we use MQTT—a pub/sub messaging protocol commonly used for IoT devices. Furthermore, each Azure Sphere is given only DeviceConnect permissions to the service, a configuration that permits Azure Spheres to only send device-to-cloud messages. No other permissions are accessible to the smart meters. For example, the ability to change the identity registry—a place where all device information is stored—is only accessible to the service company. Lastly, IoT Hub is configured to also allow HL Client connections, which is a Node.JS application. More specifically, the application is issued a SaS token and it only receives IoT Hub messages.

Azure Function

Azure Function [1] is an environment for running small pieces of code called functions. These functions are triggered by external events; in our case, the events are new messages in IoT Hub.

The function in our system is an algorithm that sends RECs to HL Client and it is shown in Algorithm 1. The inputs to the algorithm are IoT Hub messages–telemetry sent by Azure Sphere—and context that contains information such as Azure Sphere identifiers and time when telemetry was sent. Each IoT Hub message is signed using a key pair that is stored in *Azure Key Vault* (service is described in Section 5.2.1) and represents the service company. The signature is ensured to pass malleability checks, a measurement that prevents the ability to modify a REC without invalidating it's signature. Signature's non-repudiation properties are used to testify that Azure Cloud has verified each telemetry to originate from a device that is authentic and in a secure state. Lastly, the REC is sent to the HL Client, an identifier which is manually initialized, via IoT Hub.

Algorithm 1 Function signing readings on Azure Function

```
1: function SINGTELEMETRY(context, messages)
2:     for each message in the messages do
3:         RECMessage ← create_REC(context.timestamp, context.smartMeter, message)
4:         while true do
5:             RECsignature ← sign_with_KeyVault(RECMessage)
6:             if check_signature_malleability(RECsignature) then
7:                 break
8:         REC ← create_REC(RECMessage, RECsignature)
9:         send_IoTHub_message(HLClientID, REC)
```

Key Vault

Azure Key Vault service is used for key generation and key management. According to Microsoft [22], the service uses nCipher hardware security modules (HSM) that are designed in a way that Microsoft does not have access to the keys in the modules. Using the HSM, we generate a P-256 ECC key pair; the private key does not leave the hardware boundary. Additionally, the access to the key pair is restricted using Azure Active Directory and limited to Azure Function. That is, Azure Function is the only identity that is able to query for the public key or sign with the private key of the generated key pair.

Hyperledger Fabric

We make several Hyperledger Fabric configurations to test the correctness of the secure REC trading functionality. In particular, our chaincode environment is configured as two organizations, which correspond to two energy companies, and two peers per organization, which operate as docker containers. The staff in the energy company installs and instantiates the chaincode on the peers via TLS, with a chaincode endorsement policy of at least one peer per organization.

Now that we have covered the basic blockchain configurations, we discuss the HL Client and then focus on a description of the Hyperledger Fabric chaincode that implements the REC trading logic.

HL Client

HL Client is a Node.JS application that facilitates the interaction with Hyperledger Fabric. It is the only component that has access to Hyperldeger Fabric via user credentials and can invoke and query the chaincode algorithms describe in Section 5.2.1. We had to introduce HL Client because the network used to run the blockchain did not easily permit direct external connections. The component has two responsibilities.

The first one is to expose chaincode functionality to a client application. In this implementation, the client application is a web application that was developed by an undergraduate student, and it provides a user-friendly web interface. Figure 5.3 demonstrates some of the functionality.

The second responsibility is to relay RECs from IoT Hub to Hyperledger Fabric. Each IoT Hub message triggers an event handler in HL Client that forwards the REC by invoking a chaincode algorithm (see Algorithm 3).

In a real-world deployment, however, HL Client would be removed from the system. Instead, the function in Azure Function would have its own wallet and user credentials that can be used to directly invoke or query Hyperledger Fabric. Each client application would also need to have its own wallet and user credentials as well. Those credentials would be issued by Fabric's membership service maintained by the energy companies using Fabric's Attribute-Based Access Control to enforce permissions.[2]

It is important to note that although HL Client has access to IoT Hub, the application is not allowed to trigger Azure Function and create RECs out of thin air. It can only affect availability.

[2]How ABAC can be used: https://hyperledger-fabric-ca.readthedocs.io/en/latest/users-guide.html

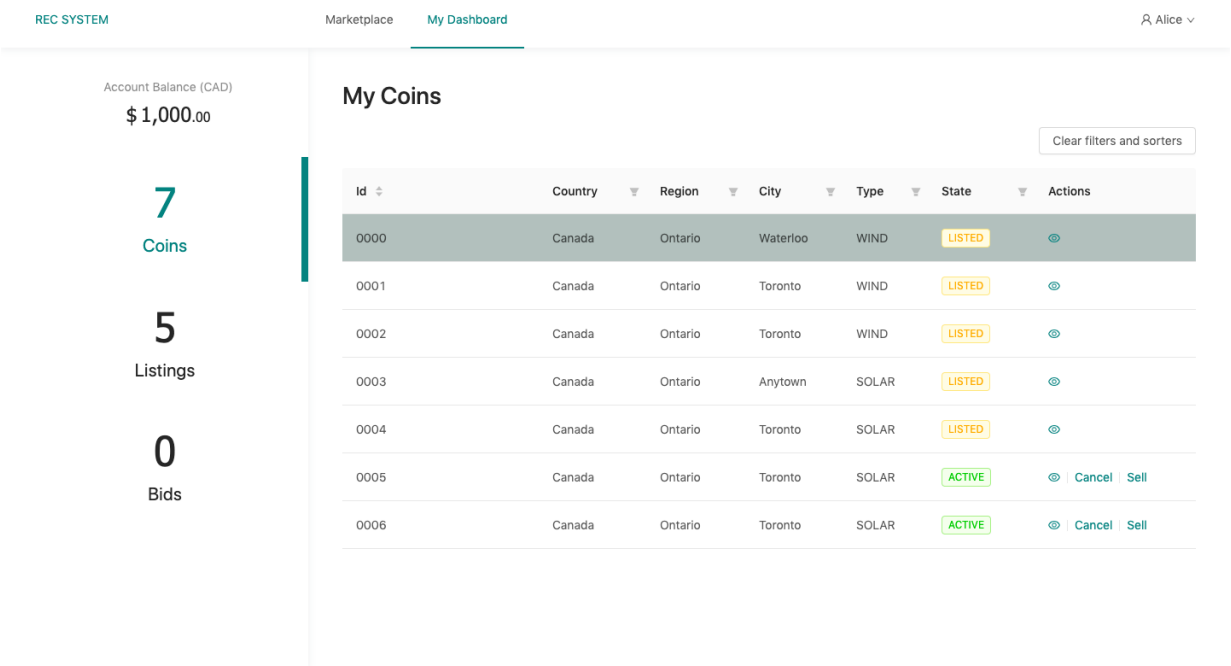

Figure 5.3: Web application that end-users use to trade RECs. The cancel button is equivalent to what we refer to as retiring a REC.

Hyperledger Fabric Chaincode

In this section, we describe the chaincode that implements the REC trading logic. We begin by providing additional information about the chaincode environment. Then, we describe the objects that are manipulated by the chaincode. Lastly, we detail each algorithm in the chaincode.

Hyperledger Fabric peers are configured to store the blockchain state in couchDB. When interacting with couchDB, the chaincode uses only simple queries such as GetState and PutState. The former retrieves data from a single key, and the latter stores data into a single key. Although Hyperledger Fabric and couchDB support more complex queries that might optimize the chaincode, these queries do not guarantee the stability of the result [8], which is why we opted to stick with the simpler queries.

Five objects are manipulated in the chaincode. The first one represents an end-user and has properties such as first name, current balance, bids and listings made by the user, and smart meters owned by the user. The second object, which we refer to as producing unit, represents a smart meter. It has properties that store information such as owner, capacity, and GPS locations; these properties mimic a similar project by Energy Web Foundation.[3]

[3]The project can be found: https://github.com/energywebfoundation/ew-helper-demo

Each REC is also an object; we refer to it as a coin and specifies the owner and amount of energy it represents. A coin listing object is created to sell a coin; it stores information such as minimum price and current bids. Lastly, each bid is represented as an object.

Before new RECs can be added to the blockchain, the chaincode needs the public key of the service company. This is handled by an initialization algorithm, which is always the first algorithm invoked after a chaincode is installed and instantiated. The algorithm submits a TLS request to Azure Function; the response, the public key of the service company, is then stored in the blockchain state.

Every algorithm input is verified using ajv module, a Node.JS module. We define schemas that test the input for sufficient and valid information.[4]

Algorithm 2 is only used by service staff to add end-users and their smart meters. Before adding a new user and producing unit to the blockchain state, the algorithm verifies that the smart meter is not already owned by another user; that is, each smart meter can be associate with at most one user.

Algorithm 2 A chaincode algorithm creating users in the system

```
1: function CREATEUSER(userInfo)
2:     user ← create a user using userInfo
3:     prodUnit ← create a producing unit using userInfo
4:     for each user in the blockchain state do
5:         if a smart meter identifier in userInfo matches an identifier in user then
6:             Throw an error
7:     Store user and prodUnit
```

Algorithm 3 creates coins using RECs produced by the smart meter. As already mentioned, HL Client invokes this algorithm for every received IoT Hub message. Before storing a new coin to the blockchain state, the algorithm verifies the validity of the signature of the REC using the public key of the service company.

Algorithm 4—the first exposed to the client application algorithm—can be used to list a REC for sale. It is responsible to correctly update the user object with the reference to the new listing.

End-users use Algorithm 5 to submit a bid for an active listing. Before storing the bid on the blockchain, the balance of the user is verified to be sufficient for the bid. For a successful bid, the user balance is decremented using the bid value; that is, the user effectively is staking the value for the bid.

[4] ajv schemas

Algorithm 3 A chaincode algorithm creating a new REC

```
1: function newREC(REC)
2:     pubKey ← get the Azure Function public key
3:     producingUnit ← get the producing unit object referenced in the REC
4:     if the signature of REC cannot be verified using pubKey then
5:         Throw an error
6:     Create a Coin using REC
7:     Store the Coin and update producingUnit
```

Algorithm 4 A chaincode algorithm creating a new coin listing

```
1: function newCoinListing(listingInfo)
2:     listing ← create a listing using listingInfo
3:     REC ← get the REC referenced by listing
4:     producingUnit ← get the producing unit referenced by REC
5:     user ← get the user referenced by producingUnit
6:     Store listing and update user and REC
```

Algorithm 5 A chaincode algorithm creating a bid

```
1: function newBid(bidInfo)
2:     bid ← create a bid using bidInfo
3:     user ← get the user referenced in the bidInfo
4:     listing ← get the listing referenced in the bidInfo
5:     if user.balance - bid.value < 0 then
6:         Throw an error
7:     user.balance ← user.balance - bid.value
8:     Store bid and update listing and user
```

Algorithm 6 is used by end-users to end a listing. First, the highest bid in the listing is determined. Then, all users with unsuccessful bids are updated by returning their stake, and the balance of the old REC owner is increased by the value of the successful updates. Lastly, the objects of the user with the highest bid and its producing unit are updated to maintain reference the newly transferred REC, while the old REC is updated to reflect that it has been bought. The listing is updated to disallow any additional bids.

Algorithm 6 A chaincode algorithm ending a coin listing

```
1:  function endCoinListing(listingInfo)
2:      listing ← get the listing object referenced in listingInfo
3:      succBid ← determine the highest bid listing
4:      for each bid in listing do
5:          if bid != succBid then
6:              user ← get user object referenced by bid
7:              Update bid and return bid.value to the user
8:      succBidUser ← get user object referenced by succBid
9:      producingUnitSuccUser ← get the producing unit object referenced by succBidUser
10:     REC ← get the REC object referenced by the listing
11:     originalOwner ← get the user object referenced by the REC
12:     Update originalOwner, succBidUser, producingUnitSuccUser, bid, REC, and listing
```

The rest of the algorithms in the chaincode do not change the blockchain state and are not described in this thesis since they are not important to the implementation. Those algorithms facilitate the interaction with the client application. Some examples include fetching listings from a specific user and fetching all coins owned by a user.

5.2.2 Implementation using Algorand

We now describe a second implementation of the trading platform and client application that is based on Algorand.

There are three significant differences between the two implementations. The burden of bootstrapping the system is removed from energy companies since Algorand is already bootstrapped. Another difference is that we do not rely on Azure Cloud to verify the proof of authenticity and healthiness. Lastly, the trading logic in this implementation is implemented in the client application, instead of handled by a smart contract; that is,

Algorand is only used as a platform to record exchanged messages and provide tamper-proof history. However, recently, Algorand 2.0 has been released, and it can be used to implement an Algorand smart contract that is responsible for the trading logic.

The rest of this section is structured as follows. First, we describe the smart meter metadata update. Following that, we show how RECs are created, and then we conclude with a description of end-user trading.

In this implementation, end-users trust REC aggregators only for availability. REC aggregators are expected to maintain middleware APIs that allow new smart meters to register. Without being able to register with an aggregator, a smart meter cannot produce RECs. Additionally, end-users rely on aggregators to store the client certificates of the smart meters; if a client certificate is not available, an end-user can choose not to trust a REC.

Metadata Update

Figure 5.4 shows the metadata update process between Azure Sphere and middleware API. The exchange corresponds to the design algorithms SM.Reg() and TP.Reg().

Every day the application in Azure Sphere creates and submits a metadata message to the middleware API. However, before creating the message and when Azure Sphere executes the application for the first time, the application generates a SECP256K1 ECC key pair using wolfSSL library and Azure Sphere's HRNG. The key pair corresponds to the smart meter key pair in the design, and it is a workaround for the lack of ability to sign a message with a key pair in the hardware root of trust. After a successful key generation, the application creates a metadata message with properties that specify information such as GPS location, which facilitates the REC origin tracing. Lastly, Azure Sphere signs and sends the metadata update message to a middleware API via HTTPS POST request along with the client certificate issued by Azure Sphere Security Service.

The middleware API first verifies the metadata, before it is added to Algorand. The verification process includes checking the validity of the X.509 certificate and its chain of certificates. Although not included in this implementation, the client certificate, which has remote attestation data, can be used to validate additional information about Azure Sphere; for example, the information can be used to determine the kernel version. The verification process also includes checking the signature of the metadata using the public key of the smart meter. We used the key pair in the registration message for verification, but we envision a real-world implementation to use a key that is included in the client

certificate instead. Lastly, the middleware API checks if the key is present in a list of identities, which are ensured to be smart meter attached to solar panels.

After successful verification of the metadata, the middleware API sends the message to Algorand. The middleware API hashes the X.509 client certificate and adds the hash to the original message. Ideally, the X.509 certificate would be directly included in the transaction, but that is not possible since the maximum supported payload size in an Algorand transaction is one kilobyte, which is less than the size of the certificate. Following that, the middleware API signs the modified message using a key pair that represents the a REC aggregator. The signature is seen as an approval that a particular Azure Sphere can generate RECs. Lastly, the middleware API constructs and submits an Algorand transaction that includes the modified metadata message. A response that includes the matadata transaction identifier and device owner Algorand address is sent back to Azure Sphere; the address would be provided during the deployment of Azure Sphere as described in Section 4.4.

Creating RECs

In our implementation, Azure Sphere creates RECs in 15-minute interval. First, the application creates a message that includes properties such as the amount of produced green energy, Algorand address of the smart meter owner, and transaction identifier of the device registration transaction. Then, the message is signed with the device private key and sent along with the client certificate to the middleware API via HTTPS POST. Similar to the registration process, before accepting the request, the client certificate and REC signature are verified. Additionally, the metadata transaction is fetched and ensured that is signed by a middleware API, which allows the smart meter to create RECs. Lastly, the middleware API creates and sends a transaction to Algorand. The transaction carries the REC as a payload in the note field.

Trading RECs

End-users interact with two Node.JS sub-applications to trade RECs. We refer to the first one as a seller app, and it can be used by end-users to sell RECs. The second sub-application is a buyer app that allows end-users to buy REC. Unless specified otherwise, both sub-applications locally create Algorand transactions that have payload in the note field shown in Figure 5.6. Once, a REC is sold, it is also considered retired under the buyer address.

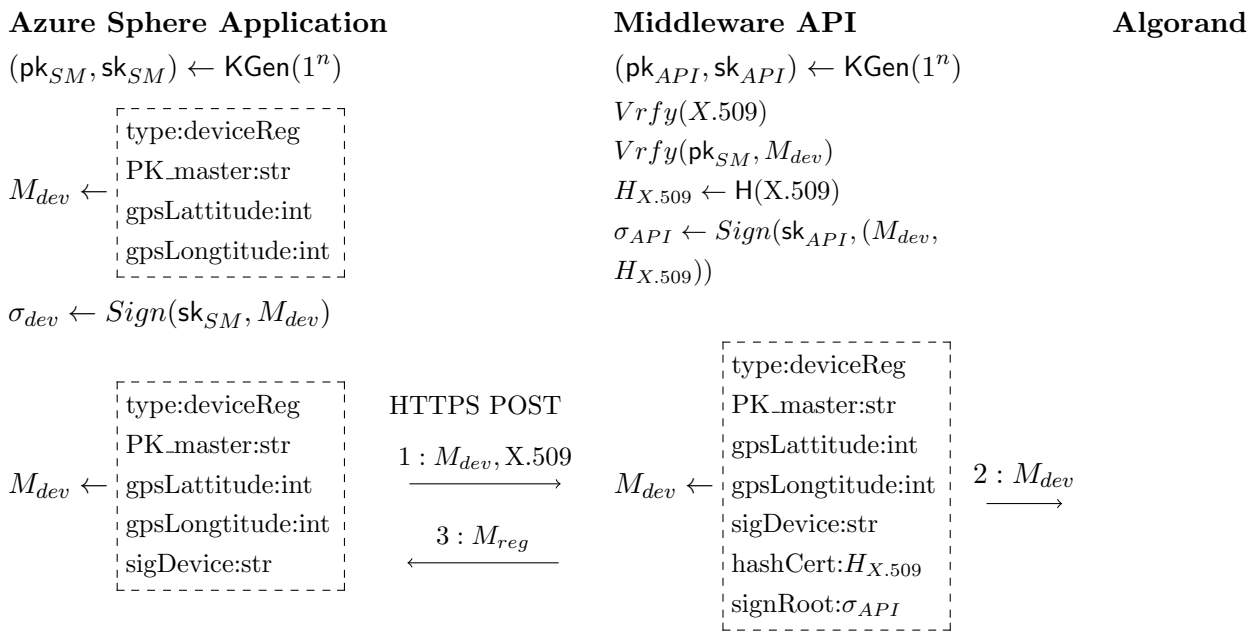

Figure 5.4: The figure shows the metada update process for Azure Sphere. The steps in the figure are in chronological order from top to bottom, and the sequence of exchanged massages is numbered.

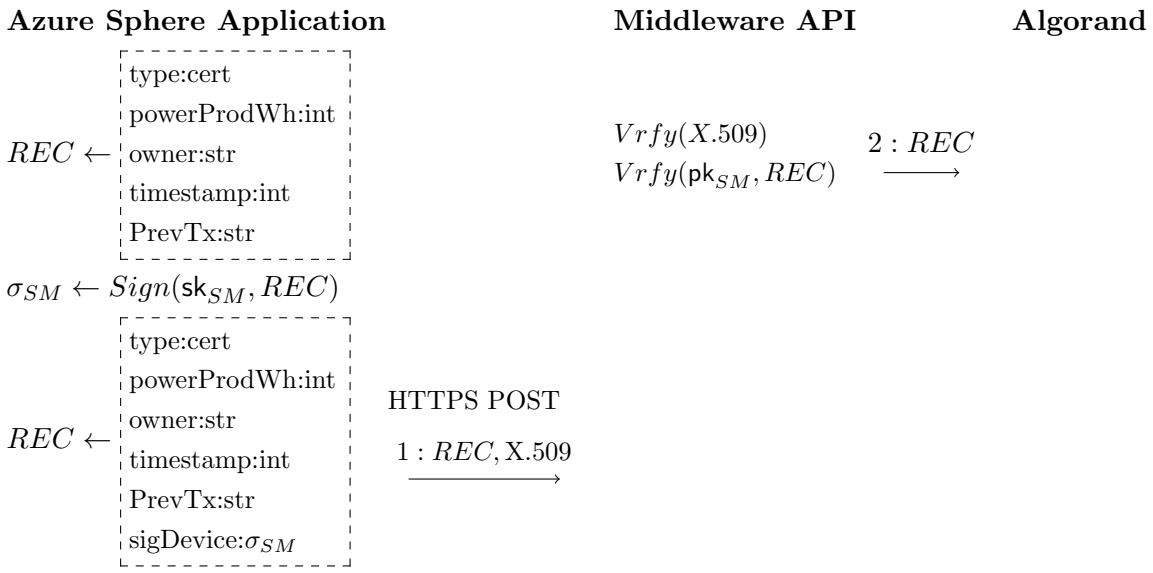

Figure 5.5: The figure shows the process that creates new RECs. The steps in the figure are in chronological order from top to bottom, and the sequence of exchanged massages is numbered.

REC owners only interact with the seller app. When the app is opened, it immediately sends a middleware API query for all RECs owned by the user. The response is displayed as a list of tuples that have RECs and their corresponding transaction identifiers. End-users are prompted to choose between listing a REC, viewing all bids for a listing, or retiring a REC.

When an end-user chooses to list a REC, the sell app prompts for two inputs. The first one is the transaction identifier of the REC to be sold, and the second one is a minimal price for the REC. Then, using the input, the sub-application creates and sends a listing transaction to Algorand.

When an end-user chooses to view all bids for a listing, the sell app prompts for the transaction identifier of the listing. Then, a request is sent to the middleware API for all relevant bids; that is, in this implementation, the end-users rely on the middleware API for being able to view all bids. Lastly, the end-user selects a bid, which is used to create and send an Algorand transaction; we refer to the transaction as a propose settle transaction, and it has information such as final price and transaction identifier of the user-chosen bid.‘

The last option in the seller app prompts the user for a transaction identifier of REC to retire. Then, the app creates and sends a retire transaction to the middleware API.

All REC aggregators monitor Algorand and verify and approve retire transactions, so that end-users can use the RECs for claims. When an aggregator detects a retire transaction, it fetches the REC and metadata update transactions, which are then used for several verifications. First, the owner of the REC—specified as a property in the REC—is checked to match the sender of the retire transaction; that is, end-user can only retire their RECs. Then, the aggregator verifies the signature in the REC using the device public key in the metadata transaction. If the check is successful, using signature non-repudiation, it is guaranteed that the REC originates from a specific device. Lastly, the metadata registration is ensured to be signed by an energy company, or otherwise, the device would not be authorized to create RECs. If all verifications are successful, the REC aggregator creates and sends an approve retire transaction to Algorand. The transaction authorizes the end-user to make green energy claims using the REC.

All end-users interested in buying RECs interact with the buyer app. Similar to the seller app, the buyer app queries the middleware API for all available REC listings in Algorand. The response is displayed as a list of tuples that have RECs and their corresponding transaction identifiers. End-users are prompted to choose between bidding for a REC or check if a previous bid has been successful.

If the end-user decides to bid for a REC, the application verifies the validity of the REC before creating a bid transaction. First, the application prompts for a transaction identifier

of a REC listing. Then, it submits a request to the middleware API to retrieve all relevant to the REC listing information; it includes the REC and device metadata transactions referenced in the listing, and Azure Sphere's client certificate. The app verifies that the listing transaction sender matches the REC owner. Then, the REC signature is verified using the device public key, and the device metadata transaction is verified to be signed by a REC aggregator. Lastly, the client certificate is checked to be valid. If all verifications are successful, the end-user can be sure that any bidding would be for a legitimate REC. After the end-user specifies a bidding price for a REC, the app creates a bid transaction.

When the second option in the app is selected, the buyer app checks if a bid is successful and transfers funds to the original REC owner with the approval of the end-user. After the end-user specifies the bid of interest, the buyer app sends a request to the middleware API to scan Algorand for propose settle transactions with the user-specified bid. If such a transaction is found, the middleware API returns the propose settle transaction and the corresponding listing. Then, the application checks if the propose settle transaction sender matches the listing transaction sender and if the propose settle is for the bid specified by the end-user. These verifications prevent tricking buyers into paying for a REC without being selected by the REC owner. If all verifications are successful and the end-user approves the conditions in the propose settle transaction, the application creates a settle transaction that transfers the amount of algos specified in the propose settle transaction from bidder's address to the REC owner's address.

Similarly to retiring RECs, REC aggregators continuously monitor Algorand for settle transactions. If such a transaction is detected, the aggregators repeats the validations that the buyer app does before end-users can bid. Additionally, it verifies that the settle transaction transfer sufficient algos and the settle transaction sender matches the bid transaction sender. If successful, the energy company sends an approve settle transaction, which, similar to the approve retire transaction, indicates that the REC can be used for green energy claims, and it cannot be used for future trading.

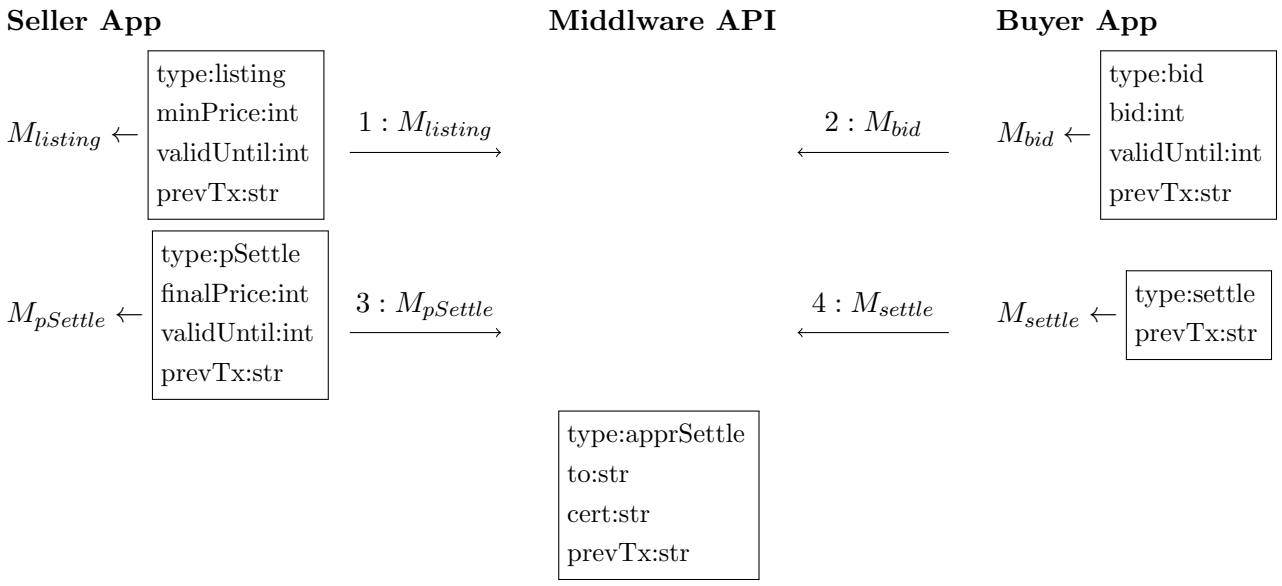

Figure 5.6: The figure shows how RECs are traded in Algorand. The sequence of exchanged massages is numbered.

Chapter 6

Evaluation

We now evaluate the smart meter instantiation and two implementations of the trading platform and client application described in Chapter 5. We argue that the combination of the smart meter with either trading platform implementation is secure, as defined in Section 4.1.2. Furthermore, both implementation incur small cost (see Appendix A), an important factor for real-world adoption.

6.1 Azure Sphere

Theorem 6.1.1. RECs produced by Azure Sphere are guaranteed to originate from a specific device and correctly represent the produced green energy.

To support the theorem, we show that each Azure Sphere has an identity that cannot be separated from the device. We also describe why the environment in which applications run is secure and why the application captures the produced green energy correctly. Lastly, we need to show that end-users can verify the previous statements. This is done using the following assertions:

Assertion 6.1.1. The application in Azure Sphere creates RECs that represent the produced by the solar panel green energy.

As already discussed in Section 4.2, we make the assumption that the link between the sensor that measures the produced green energy and the smart meter is trusted; that is, the application will receive correct sensor readings. Additionally, we envision that the application deployed on Azure Sphere is open-source. We hope the fact that the application in Section 5.1 is open-source and simple would translate into a better trust that the application is secure.

Assertion 6.1.2. The application is running in a secure environment.

Since the application runs in Azure Sphere, the assertion is equivalent to saying that the Azure Sphere is secure.

Azure Sphere begins its operation in a secure state. We rely on secure boot implemented in Azure Sphere to guarantee the secure state; that is, the only software that the device can load is software signed by Microsoft. We include an in-depth discussion of the secure boot in Section 5.1.

After booting up, Azure Sphere continues to maintain a secure state. We describe in Section 2.1.2 and Section 5.1 that Azure Sphere implements in silicon mitigations against attacks such as rollback and physical-side channel attacks, and the device satisfies crucial properties such as hardware root of trust. Additionally, Azure Security Service periodically remotely attests the device to detect any changes in Azure Sphere's state. These properties, mitigations, and functionalities together guarantee that the device continues to behave as expected and provides a secure environment for the application.

Assertion 6.1.3. The REC is created by an Azure Sphere.

Each Azure Sphere has an identity that is not separable from the silicon. The hardware root of trust mandates this inseparable identity, and in Azure Sphere, it is implemented in the Pluton security subsystem. The key pair that represents the identity is stored in e-fuses; the pair is not accessible to any software, and it is protected against physical side-channel attacks. Therefore, no attacker can gain access to the key pair and create fake RECs, as if they are created by Azure Sphere. We sign the REC with such a key pair, which guarantees the REC origin.

Assertion 6.1.4. Azure Security Service or end-users will detect malicious applications or any tampering with Azure Sphere.

One of the most important reasons why we picked Azure Sphere is that the device has all features necessary to securely implement remote attestation, as discussed in Section 5.1. The remote attestation measurements—which capture the device state—are used by Microsoft to detect any tampering. Additionally, the attestation data is included in the X.509 client certificate. So, even if the application is changed maliciously and differentiates from the open-source version, it would be detectable by end-users when verifying the client certificate.

6.2 Hyperledger Fabric

Theorem 6.2.1. The system based on Hyperledger Fabric is secure as defined in Section 4.1.2.

To support the theorem, we make the following assertions:

Assertion 6.2.1. Only authorized and non-tampered smart meters can create RECs

Algorithm 3 in the chaincode is the only algorithm that can create RECs in the system, and it guarantees that the RECs originate from authorized and non-tampered smart meters. The algorithm checks the input for a REC signature that is verifiable using the public key of the service company. If the signature is valid, then the smart meter is authorized and non-tampered.

The service company can only sign RECs that originate from authorized and non-tampered smart meters, assuming the cloud configurations describe in Section 5.1. The code in Azure Function is the only code that can create RECs with valid signatures because the secret key of the service company is stored in Key Vault, and Key Vault is configured to allow only Azure Function to access the key. Moreover, Azure Function is only executed when a new IoT Hub message is published, and only Azure Spheres that are claimed by the service company and have valid X.509 certificates are allowed to send readings in IoT Hub. As discussed in Assertion 6.1.4, each Azure Sphere with a valid X.509 certificate is guaranteed to be non-tampered with. However, the code in Azure Function is trusted, otherwise it could create RECs out of thin air.

Assertion 6.2.2. RECs cannot be tampered with and cannot be double-spent.

We depend on the fact that that the logic in the chaincode correctly implements the algorithms described in Section 5.2.1 and that the peers in Hyperledger Fabric do not violate the underlying consensus protocol assumption so that RECs cannot be double-spent. As already mentioned, each REC has a signature signed by the service company; that is, if a REC is tampered with, the signature would be invalidated and the REC cannot be added to the blockchain.

6.3 Algorand

Theorem 6.3.1. The system based on Algorand is secure as defined in Section 4.1.2.

We note that assertions for Algorand implementation differ from the assertions for Hyperledger Fabric implementation because the registration process for smart meter is different. On the one hand, in Hyperledger Fabric, smart meters are automatically provisioned into appropriate IoT Hub when claim by Azure Sphere Tenant; that is, the communication channel to Hyperledger Fabric automatically establishes the security of Azure Sphere. On the other hand, in Algorand, smart meters are not automatically provisioned into a service or component that establishes a communication channel with Algorand.

To support the theorem, we make the following assertions:

Assertion 6.3.1. Only authorized and non-tampered smart meter can register.

Energy companies, which we envision to operate middleware APIs, are expected to have access to the identities of the smart meters in the solar panels—similarly to how Azure Cloud is given identities of Azure Spheres by the manufacturer. During the registration process described in Section 5.2.2, Azure Sphere sends a registration message and X.509 certificate. The public key of the Azure Sphere is checked to be in the list of smart meter identities that are authorized to register. Also, the registration message has a signature that is verified using the public key of the smart meter and wolfSSL; that is, we ensure that the message is not tampered with, and it originates from the smart meter as discussed in Assertion 6.1.3. Lastly, the middleware API verifies the validity of the X.509 certificate, which guarantees that the smart meter is not tampered with, as discussed in Assertion 6.1.4.

Assertion 6.3.2. Only registered smart meters can create valid RECs

Azure Sphere creates and signs RECs using wolfSSL and its public key, as discussed in Section 5.2.2. Since Algorand is permissionless blockchain, anybody can create a REC; however, such REC is not necessarily valid.

In Algorand, to consider a REC valid, it needs to satisfy several requirements. The REC is expected to have a property "PrevTx" that points to the registration transaction, as shown in Figure 5.5. Anybody should be able to use the "PrevTx" property and verify that an energy company has signed the registration message. Also, the public key in the registration message is used to verify the signature of the REC. Lastly, a verifier that does not trust the energy company can verify that a smart meter is not tampered with, or it has the expected Azure Sphere application, as discussed in Assertion 6.1.4. A verifier can request the smart meter X.509 certificate from an energy company that possesses it; then, the verifier needs to ensure that the received X.509 certificate has a hash that matches the registration message, and verify the certificate.

Assertion 6.3.3. RECs cannot be tampered with and cannot be double-spent.

REC owners are able to perform two operations with RECs. Either they can sell a REC or retire a REC. We discuss the possibility of double-spending with those two operations, with the assumption that Algorand's security assumptions are not violated.

REC owners cannot double-spent when retiring a REC. Since each REC needs approval from an energy company, we rely on the energy company to proactively prevent double-spending; that is, energy companies parse all transactions and keep a state of all non-retired RECs. When an owner submits a request for retiring an already retired REC, an energy company is expected to deny the request. However, in this implementation, the energy company could also misbehave and approve a second retiring transaction of a REC. In such a case, we rely on what we call retroactive prevention of double-spending; that is, we envision an external auditor would detect such misbehavior so that the energy company can be penalized. We note that the external auditor could be anyone, including end-users.

REC owners cannot double-spent when selling a REC. Since energy company needs to approve every sell of a REC, the argument for not being able to double-spend when selling a REC is the same as the argument for retiring a REC. However, it is important to mention that a seller can also publish multiple proposals for settling a transaction. Similarly, in this implementation, the buyer depends on the energy company to ensure that the seller has not sent a second proposal for other end-user. However, if the energy company acts maliciously, the blockchain can serve as a public record that can be used to penalize the involved parties.

We emphasize that Algorand 2.0, which was not available during our implementation, can eliminate the issue with a malicious energy company, by implementing the trading logic in smart contracts.

Chapter 7

Future Work and Conclusions

7.1 Future Work

There are several possible directions for future work. First, both implementations do not provide any anonymity, which could be an inhibitor to adoption. Another interesting direction for future work is how to reduce trust in Microsoft. For example, can we remove the need to trust Azure Security Service for remote attestation. Another important issue is reducing the price volatility, a problem common to permissionless blockchains; that is, an increase in the transaction fees might reduce the end-user profit to the point that it might not make sense to participate in the system.

In the future, we could also improve the current functionality in the implementation. One drawback of our implementation is that users have to trade each REC individually. A trading agent could be built to trade on behalf of the user according to some predefined parameters. Additionally, incentives can be added to stimulate certain behavior. For example, geographical regions that do not have enough green energy projects can be stimulated by increasing the producer profits in that region.

7.2 Conclusions

In this work, we designed a high-level system for trading RECs that satisfies two properties. The first property requires secure REC trading; that is, only authorized non-compromised smart meters can produce RECs, and no REC can be double-spent. The system provides that by relying on two building blocks: secure resource-constrained hardware and blockchain. The former guarantees that the smart meter is not tampered with, while the latter addresses concerns with double-spending. The second property mandates REC anonymity; that is, a specific REC cannot be attributed to the smart meter that created the REC nor its owner. To address that, we used ring signatures to hide the smart meter that created the REC, while stealth addresses hide the REC owner.

To show feasibility, we describe two implementations of the high-level system. In both, the smart meter is instantiated using Azure Sphere, which guarantees the device integrity and correct meter readings. We implement the trading functionality on permissioned blockchain, Hypereledger Fabric, to prevent double-spending. We also provide a second implementation based on permissionless blockchain, Algorand, to ease the burden for energy companies that would need to bootstrap the system. Both implementations provide secure REC trading.

References

[1] Azure Functions documentation. https://docs.microsoft.com/en-us/azure/azure-functions. Last accessed on 2020-04-13.

[2] Azure IoT Hub Device Provisioning Service. https://docs.microsoft.com/en-us/azure/iot-dps/about-iot-dps. Last accessed on 2020-04-13.

[3] Azure IoT Hub documentation. https://docs.microsoft.com/en-us/azure/iot-hub/. Last accessed on 2020-04-13.

[4] Azure Sphere Device Authentication and Attestation Service. https://azure.microsoft.com/en-ca/resources/azure-sphere-device-authentication-and-attestation-service/. Last accessed on 2020-04-12.

[5] Azure Sphere MT3620 development kit-US version. https://www.seeedstudio.com/Azure-Sphere-MT3620-Development-Kit-US-Version.html. Last accessed on 2020-04-13.

[6] BMO Financial to offset 100% of electricity use with RECs.

[7] Canada's renewable power landscape 2016 – energy market analysis. https://www.cer-rec.gc.ca/nrg/sttstc/lctrct/rprt/2016cndrnwblpwr/plcncntv-eng.html. Last accessed on 2020-05-06.

[8] CouchDB as the state database (Hyperledger Fabric documentation). https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_as_state_database.html. Last accessed on 2020-04-13.

[9] Deployment basics (for Azure Sphere Application). https://docs.microsoft.com/en-ca/azure-sphere/deployment/deployment-concepts. Last accessed on 2020-05-15.

[10] Documentation for MT3620 (Microsoft Azure Sphere MCU). https://www.mediatek.com/products/azureSphere/mt3620. Last accessed on 2020-04-13.

[11] Guaranteeing the origin of european energy.

[12] How to join (Green-e). https://www.green-e.org/programs/energy/join. Last accessed on 2020-05-09.

[13] M-RETS. https://www.mrets.org/. Last accessed on 2020-06-02.

[14] MT3620 support status. https://docs.microsoft.com/en-ca/azure-sphere/hardware/mt3620-product-status. Last accessed on 2020-05-15.

[15] NEPOOL Generation Information System. https://www.nepoolgis.com/. Last accessed on 2020-06-02.

[16] Pololu ACS711EX current carrier. https://www.canadarobotix.com/products/1409?variant=14423704272945. Last accessed on 2020-04-13.

[17] Solar investment tax credit (ITC). https://www.seia.org/initiatives/solar-investment-tax-credit-itc. Last accessed on 2020-05-06.

[18] Terminology (Azure Sphere). https://docs.microsoft.com/en-us/azure-sphere/product-overview/terminology. Last accessed on 2020-05-15.

[19] Trusted Platform Module (TPM) summary. https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/. Last accessed on 2020-03-01.

[20] U.S. renewable electricity market. https://www.epa.gov/greenpower/us-renewable-electricity-market. Last accessed on 2020-05-09.

[21] Western Renewable Energy Generation Information System. https://www.wecc.org/WREGIS/Pages/Default.aspx. Last accessed on 2020-06-02.

[22] What is Azure Key Vault? https://docs.microsoft.com/en-ca/azure/key-vault/key-vault-overview#securely-store-secrets-and-keys. Last accessed on 2020-04-13.

[23] What is Azure Sphere? https://docs.microsoft.com/en-ca/azure-sphere/product-overview/what-is-azure-sphere. Last accessed on 2020-05-13.

[24] The world's most influential companies, committed to 100% renewable power. http://there100.org/. Last accessed on 2020-05-07.

[25] Filament unveils industry's first blockchain hardware device in a USB form factor for existing IoT devices plug and play, secure blockchain operations enable significantly accelerated deployments... https://markets.businessinsider.com/news/stocks/

filament-unveils-industry-s-first-blockchain-hardware-device-in-a-usb-form-factor-
2018. Last accessed on 2020-04-24.

[26] The past, present and future of climate change. https://www.economist.com/briefing/2019/09/21/the-past-present-and-future-of-climate-change, September 2019. Last accessed on 2020-06-02.

[27] Energy web foundation onboards engie blockchain startup's dApp. https://www.ledgerinsights.com/energy-web-foundation-engie-blockchain-teo-dapp/, 2020. Last accessed on 2020-04-24.

[28] Ledger Origin. https://www.ledger.com/origin, 2020. Last accessed on 2020-04-24.

[29] TEO (The Energy Origin) proposes green energy transparency with blockchain. https://innovation.engie.com/en/news/medias/new-energies/teo-the-energy-origin-proposes-green-energy-transparency-with-blockchain/13243, 2020. Last accessed on 2020-04-24.

[30] Merlinda Andoni, Valentin Robu, David Flynn, Simone Abram, Dale Geach, David Jenkins, Peter McCallum, and Andrew Peacock. Blockchain technology in the energy sector: A systematic review of challenges and opportunities. *Renewable and Sustainable Energy Reviews*, 100:143 – 174, February 2019.

[31] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[32] Galen L. Barbose. U.S. Renewables Portfolio Standards: 2019 Annual Status Update. Technical report, July 2019.

[33] Fred Beck and Eric Martinot. Renewable energy policies and barriers. 2016.

[34] Adam Bender, Jonathan Katz, and Ruggero Morselli. Ring signatures: Stronger definitions, and constructions without random oracles. In *Theory of Cryptography Conference*, pages 60–79. Springer, 2006.

[35] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, pages 1–6. Association for Computing Machinery, June 2015.

[36] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. Reconciling remote attestation and safety-critical operation on simple IoT devices. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[37] J. A. F. Castellanos, D. Coll-Mayor, and J. A. Notholt. Cryptocurrency as guarantees of origin: Simulating a green certificate market with the ethereum blockchain. In *2017 IEEE International Conference on Smart Energy Grid Engineering (SEGE)*, pages 367–372, August 2017.

[38] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, page 400–409. Association for Computing Machinery, November 2009.

[39] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A protocol for property-based attestation. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing*, page 7–16. Association for Computing Machinery, November 2006.

[40] Nicolas T Courtois and Rebekah Mercer. Stealth address and key management techniques in blockchain systems. *ICISSP*, 2017:559–566, 2017.

[41] David Roberts. RECs, which put the "green" in green electricity, explained. https://www.vox.com/2015/11/9/9696820/renewable-energy-certificates, 2015. Last accessed on 2020-05-09.

[42] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS 2012, 19th Annual Network and Distributed System Security Symposium*, pages 1–15, February 2012.

[43] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6. IEEE, March 2014.

[44] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

[45] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. *arXiv preprint arXiv:1901.00910*, 2019.

[46] Daan Hulshof, Catrinus Jepma, and Machiel Mulder. Performance of markets for european renewable energy certificates. *Energy Policy*, 128:697 – 710, 2019.

[47] Galen Hunt, George Letey, and Ed Nightingale. The seven properties of highly secure devices. Technical Report MSR-TR-2017-16, March 2017.

[48] F Imbault, M Swiatek, R De Beaufort, and R Plana. The green blockchain: Managing decentralized energy production and consumption. In *2017 IEEE International Conference on Environment and Electrical Engineering and 2017 IEEE Industrial and Commercial Power Systems Europe (EEEIC/I&CPS Europe)*, pages 1–5. IEEE, June 2017.

[49] Marek Jawurek, Florian Kerschbaum, and George Danezis. Privacy technologies for smart grids - A survey of options. Technical report, November 2012.

[50] Jillian Ambrose . Hmrc cracks down on gangs over renewable energy vat fraud, 2019.

[51] Jim Chappelow. Market failure. https://www.investopedia.com/terms/m/marketfailure.asp. Last accessed on 2020-05-18.

[52] Georgios Karopoulos, Christos Xenakis, Stefano Tennina, and Stefanos Evangelopoulos. Towards trusted metering in the smart grid. In *2017 IEEE 22nd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 1–5. IEEE, June 2017.

[53] Fabian Knirsch, Clemens Brunner, Andreas Unterweger, and Dominik Engel. Decentralized and permission-less green energy certificates with gecko. *Energy Informatics*, 3(1):1–17, 2020.

[54] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1–14. Association for Computing Machinery, April 2014.

[55] N. Kuntze, C. Rudolph, I. Bente, J. Vieweg, and J. von Helden. Interoperable device identification in smart-grid environments. In *2011 IEEE Power and Energy Society General Meeting*, pages 1–7. IEEE, July 2011.

[56] Michael LeMay, George Gross, Carl A Gunter, and Sanjam Garg. Unified architecture for large-scale attested metering. In *40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, pages 115–115. IEEE, January 2007.

[57] Michael LeMay and Carl A. Gunter. Cumulative attestation kernels for embedded systems. In Michael Backes and Peng Ning, editors, *Computer Security – ESORICS 2009*, pages 655–670. Springer Berlin Heidelberg, May 2009.

[58] Yanlin Li, Jonathan M McCune, and Adrian Perrig. SBAP: Software-based attestation for peripherals. In *International Conference on Trust and Trustworthy Computing*, pages 16–29. Springer, 2010.

[59] A. R. Metke and R. L. Ekl. Security technology for smart grid networks. *IEEE Transactions on Smart Grid*, 1(1):99–107, June 2010.

[60] Michael del Castillo. Nasdaq explores how blockchain could fuel solar energy market. https://www.coindesk.com/nasdaq-blockchain-solar-power-market, 2016. Last accessed on 2020-04-24.

[61] Mihail Mihaylov, Sergio Jurado, Narcís Avellana, Kristof Van Moffaert, Ildefons Magrans de Abril, and Ann Nowé. NRGcoin: Virtual currency for trading of renewable energy in smart grids. In *11th International conference on the European energy market (EEM14)*, pages 1–6. IEEE, May 2014.

[62] A. Nagarajan, V. Varadharajan, M. Hitchens, and E. Gallery. Property based attestation and trusted computing: Analysis and challenges. In *2009 Third International Conference on Network and System Security*, pages 278–285. IEEE, October 2009.

[63] Ed Nightingale. Anatomy of a secured MCU. https://azure.microsoft.com/en-ca/blog/anatomy-of-a-secured-mcu, 2018. Last accessed on 2020-05-15.

[64] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. {VRASED}: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium ({USENIX} Security 19)*, pages 1429–1446, August 2019.

[65] Andrew J. Paverd and Andrew P. Martin. Hardware security for device authentication in the smart grid. In Jorge Cuellar, editor, *Smart Grid Security*, pages 72–84. Springer Berlin Heidelberg, 2013.

[66] Massimiliano Raciti and Simin Nadjm-Tehrani. Embedded cyber-physical anomaly detection in smart meters. In Bernhard M. Hämmerli, Nils Kalstad Svendsen, and Javier Lopez, editors, *Critical Information Infrastructures Security*, pages 34–45. Springer Berlin Heidelberg, January 2013.

[67] Richard Martin. How corporations buy their way to green. https://www.technologyreview.com/s/541701/how-corporations-buy-their-way-to-green/, 2015. Last accessed on 2020-05-07.

[68] Ronald L Rivest, Adi Shamir, and Yael Tauman. How to leak a secret: Theory and applications of ring signatures. In *Theoretical Computer Science*, pages 164–186. Springer, January 2006.

[69] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77. Association for Computing Machinery, September 2004.

[70] Bernardo Sarti. Policies for the deployment of renewable energies: An overview. 2018.

[71] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software attestation for key establishment in sensor networks. In *International Conference on Distributed Computing in Sensor Systems*, pages 372–385. Springer, 2008.

[72] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–16. Association for Computing Machinery, October 2005.

[73] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282. IEEE, May 2004.

[74] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In *European Workshop on Security in Ad-hoc and Sensor Networks*, pages 27–41. Springer, July 2005.

[75] Umesh Shankar, Monica Chew, and J Doug Tygar. Side effects are not sufficient to authenticate software. In *13th USENIX Security Symposium*, August 2004.

[76] D. Stiles. The hardware security behind Azure Sphere. *IEEE Micro*, 39(2):20–28, March 2019.

[77] United States environment protection agency. Greenhouse gas emissions, Global Greenhouse Gas Emissions Data. https://www.epa.gov/ghgemissions/global-greenhouse-gas-emissions-data#Sector, Last accessed on 2020-05-06.

[78] U.S. Department of Energy. New solar opportunities for a new decade. https://www.energy.gov/eere/solar/sunshot-2030. Last accessed on 2020-05-06.

[79] U.S. Department of Energy. Photovoltaic (PV) pricing trends: historical, recent, and near-term projections. https://www.nrel.gov/docs/fy13osti/56776.pdf. Last accessed on 2020-05-06.

[80] Shaomin Zhang, Tengfei Zheng, and Baoyi Wang. A privacy protection scheme for smart meter that can verify terminal's trustworthiness. *International Journal of Electrical Power & Energy Systems*, 108:117 – 124, June 2019.

[81] Fangyuan Zhao, Xin Guo, and Wai Kin Victor Chan. Individual green certificates on blockchain: A simulation approach. *Sustainability*, 12(9):3942, 2020.

[82] J. Zhao, J. Liu, Z. Qin, and K. Ren. Privacy protection scheme based on remote anonymous attestation for trusted smart meters. *IEEE Transactions on Smart Grid*, 9(4):3313–3320, July 2018.

APPENDICES

Appendix A

Cost

A.1 Azure Sphere

We now estimate the cost to add an Azure Sphere to a solar panel. Although in our implementation, we used SEEED Azure Sphere development kit [5] that currently costs about 85 USD, a custom PCB will be designed for real-world deployment, which will significantly reduce the price. Currently, Azure Sphere microcontroller costs about 11 USD, while the current sensor we used costs about 3.8 USD. According to MediaTek [10], Azure Sphere is packaged in 12 mm by 12 mm, which sets the minimum PCB dimensions. Using this information, we believe that the total cost to add Azure Sphere to a solar panel could be reduced to approximately 16 USD.

A.2 Hyperledger Fabric

We now estimate the incurred cost to maintain Hyperledger Fabric. In our prototype, Hyperledger Fabric peers were deployed on a local machine, but we use Azure Cloud to make our estimation since it provides similar service. According to Azure Cloud, each peer that has two vCPUs and 8 GiB of memory cost 105 USD per month. Additionally, since Hyperledger Fabric throughput supports up to 20 000 tx/s [45], we estimate that Fabric will support at most 18 000 000 smart meters when RECs are created in 15 minutes intervals. Therefore, using the default Hyperledger Fabric settings on Azure Cloud of one peer and one ordered, the very **minimal** Hyperledger Fabric monthly cost per solar panel is less than ten thousands of a cent.

Additionally, the Hyperledger Fabric makes use of several cloud services that also incur a cost. IoT Hub, one of the services we use, costs about 2500 USD per month for 300 000 000 messages per day.[1] If each solar panel sends about 96 messages per day, each IoT Hub

[1]The estimations in this section were calculated using: https://azure.microsoft.com/en-ca/pricing/calculator/

would be able to handle about 3 125 000 solar panels. However, we will need at least six IoT Hubs so that all 18 000 000 solar panels supported by Hyperledger Fabric would have available communication channels; this will cost about 15 000 USD per month in total. Furthermore, Azure Function will cost about 53.60 USD per month for a workload of 300 000 000 messages where each execution takes less than 100 ms or 324 USD per month for all six IoT hubs. Lastly, the Key Vault costs 0.03 USD per 10,000 operations. Therefore, the cloud service will cost in total 16224 USD per month or less than a thousand of a cent per solar panel.

A.3 Algorand

We now estimate the monthly cost related to REC production. Since in our implementation RECs are produced in fifteen-minute intervals, each smart meter would send about 2880 Algorand transactions each month. Additionally, each solar panel has to register daily, which adds roughly 30 transactions per month. Therefore, REC production requires approximately 2910 transactions.

REC trading involves several transactions. Every time an end-user want to retire a REC, the client application needs to create a retire transaction, and the energy company needs to approve the retired transaction; that is, retiring a REC involves two transactions. Every time an end-user want to sell a REC, the seller needs to create two transactions, the buyer needs to create two transactions, and the energy company needs to approve the whole process; that is, retiring a REC involves five transactions.

At the time of the writing, the incurred cost in Algorand is very small. We use the minimum transaction fee of 1000 micro Algo to give an approximation, which at the current price of about 0.2 USD per Algo[2] equals to 0.0002 USD. The monthly operation cost of a smart meter to produce RECs is about 0.582 USD, while the cost to retire a REC and to sell a REC is about 0.0004 USD and about 0.001 USD, respectively.

[2]According to https://coinmarketcap.com/currencies/algorand/