Making Renewable Energy Certificates Efficient, Trustworthy, and Private

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

Abstract

Although renewable energy costs are declining rapidly, producers still rely on additional incentives, such as Renewable Energy Certificates (RECs), when making an investment decision. An REC is a proof that a certain amount of energy was generated from a renewable resource. It can be traded for cash in an REC market. Unfortunately, existing mechanisms to ensure that RECs are trustworthy—not fraudulently generated and from a universally-agreed renewable energy source—require periodic physical audits of the generation plant, which adds costly administrative overheads and locks out small producers. Although prior work has attempted to address these issues, existing solutions lack privacy and are vulnerable to tampering. In this work, we design, implement, and evaluate a system that is efficient, trustworthy, and anonymous, thus opening the REC market to small-scale energy producers. We describe two implementations based on commercially-available Azure Sphere micro-controller unit combined with a permissioned Blockchain, Hyperledger Fabric, and a permissionless Blockchain, Algorand.

Acknowledgements

TODO:

Dedication

TODO:

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In an effort to combat global warming, many countries provide incentives in the form of policies for the reduction of greenhouse emissions [28]. Although those incentives are diverse, one of the main focus is to promote renewable energy production. The addition of green energy to the energy mix is important since the electricity and heat production sector contributes about a quarter of the global greenhouse emissions [57]. Additionally, unlike sectors, electricity and heat production is amenable to emission reduction because renewable energy sources have become a viable alternative to fossil fuels [58, 59].

In this work, we focus on *quantity-forcing*, a policy that incentivizes deployment of renewable energy by requiring either a specific amount or a portion of the total power to be produced by renewable energy sources [37]. The policy in the USA and Canada is implemented under the name Renewable Portfolio Standard (RPS), and it is a generation-based standard [37, 11]. 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 [36].

To comply with quantity-forcing, electricity supply companies can purchase Renewable Energy Certificates (RECs). RECs are tradeable certificates that are issued to a generator when they produce a certain amount of renewable energy, usually 1 MWh. After their creation, RECs can be exchanged for cash in an REC market. An REC purchaser, which includes companies such as Microsoft, Goldman Sachs, GM, and eBay, can use it to prove that their energy use has been offset by an equivalent amount of 'green' generation or claim to be '100% green' independent of their actual energy mix [30, 10]. An REC is *retired* when it has been used to match energy consumption; retired RECs exit the system. RECs solve the information asymmetry problem in that most electricity consumers cannot be sure that their consumed energy is renewable, even if they are willing to pay more [44]. For these reasons, in Europe alone, over 700 million certificates were issued in 2019 and this number continues to increase every year [1].

Although RECs are an attractive mechanism for incentivizing renewable energy generation, their current implementation leaves much to be desired [44]. One problem is that there are many REC systems that are different based on jurisdictions and as a result cause substantial inconsistencies [55, 44]. For example, should electricity generation from wood pellets derived from Amazonian rainforests be considered renewable? Recently AIB—the organization that develops and promotes a standardized REC system in Europe— discovered that Italy has been issuing RECs for a source that is considered renewable in Italy but not in the rest of Europe [16]. A second problem is the double-spending and fraudulent issue of certificates, especially in the voluntary market [55]. To address these two problems, organizations such as Green-e [17] set uniform criteria for renewable generation and physically audit the energy source. However, this is onerous and expensive: a cost of at least $4,500 for each annual audit. It also excludes small-scale energy producers, who cannot afford these costs, from the market. Finally, the information in the REC leaks privacy, especially for small-scale producers and traders who don't want to reveal their trades and level of market exposure.

1.2 Contributions and System Overview

In this thesis, we investigate how to improve the REC quantity-forcing policy. To address the aforementioned issues, we show how to leverage secure, low-cost hardware, blockchains, ring signatures, and stealth addresses to generate anonymous, trustworthy RECs. At a high level, we rely on low-cost hardware that satisfies several requirements, such as hardware root of trust, to create trustworthy RECs. These RECs are sent to a blockchain so that their trustworthiness is maintained when traded by end-users. Additionally, a purchaser can use proofs based on remote attestation to ensure that the smart meters are not compromised. To anonymize RECs, ring signatures hide the smart meter that created the REC, and stealth address hide the blockchain owner of the REC.

Our contributions include:

- The design of a pragmatic blockchain-based system for low-overhead, trustworthy, anonymous REC generation and trading.
- A detailed evaluation of our design to prove its security properties.
- A prototype implementation using commercially-available hardware and software.

The thesis is structured as follows. First, we discuss all the necessary background and related work (Chapter 2). Then, we describe all the primitives used in our design (Chapter 3). With this sufficient information, we clearly state our design goals and describe our design (Chapter 4). Lastly, we describe our implementation (Chapter 5), evaluate our work (Chapter 6), and conclude (Chapter 7).

Chapter 2

Background and Related Work

2.1 Background

In this chapter, we overview the technologies used in our implementation. First, we introduce Hyperledger Fabric and Algorand blockchains with a focus on transaction processing and parties involved. Although both blockchains are used as a black-box, a description of how transactions are processed enhances the understanding of our security assumptions and the required trust in the involved parties. Second, we summarize the existing literature related to trusted smart meters designs and conclude that there are no practical proposals for real-world resource-constrained devices. Since we address that by choosing a commercially-available hardware Azure Sphere as a microcontroller for our smart meter, we summarize the goals set by Microsoft when designing the smart meter and 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. [35], 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 [35] 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 [43], a throughput permitting a wide variety of applications. Before describing the processing of transactions, we describe the participants in the system.

There are three types of participants 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. Peers are separated into groups 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 blockchains before Fabric. 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 [15] 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 to 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 [42] is a permissionless blockchain that provides security, scalability, and decentralization. The system scales to hundreds of thousands of users, while transactions are

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

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 of stake [22].

Algorand makes several assumptions to achieve its goal [42]. First, at least two-thirds of the cryptocurrency is assumed to be owned by honest users. Second, the protocol requires strong synchrony for liveliness; that is, communication should be bounded by a specific delay so that new transactions can be added [42]. A violation of the strong synchrony assumption does not allow an adversary to change already added transactions as long as the strong synchrony is violated only for a bounded amount of time. 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; the adversary can, at most, add an empty block to the chain [42]. 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 the cryptocurrency in Algorand can be seen as a sub-user with an equal probability to be selected. In other words, the more cryptocurrency users have, the bigger their voting power during the voting phase.

2.1.2 Trusted Smart Meters Design

We now described work on designing trusted smart meters.

Trusted Smart Meters Based on TPM

Lemay at al. [49] were perhaps the first to introduce the idea of combining smart meters with trusted computing technology. They envisioned multiple vendors sharing the same smart meter by deploying various applications; these applications were separated using virtualization. Privacy concerns were addressed with network access control policies restricting 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. [62] 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 the attribute 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. [60] also used trusted computing to verify smart meter's trustworthiness; their proposal was based on *attribute-based* attestation—a method that convinces a verifier that the meter satisfies a specific property. Unlike Zhao et al. [62], 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. [60], 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 [47] [50]. 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. [47] used TEE to provide secure cryptographic key storage and ensure software integrity using remote attestation for a smart meter. 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. [54] 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. [54] 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 restricting key access to the specific TPM and application. Similarly, the PAL implementing the TLS handshake has access to the device keys only if the measurements in the registers 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.

2.1.3 Azure Sphere

Now that we have shown that we have not found a practical trusted smart meter, we describe the hardware we chose to implement a trusted smart meter.

Azure Sphere [27] 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 the seven properties required by Microsoft from each secure resource-constrained device. Then, we explain how Azure Sphere achieves those properties. Lastly, we discuss terminology used throughout this thesis related to software updates.

Hunt et al. [45] 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. [45] using an immune system analogy; it allows system operators to identify device probing by attackers and potential vulnerabilities such as software bugs.

Azure Sphere satisfies all properties set by Microsoft. First, the hardware root of trust is satisfied by a subsystem called *Pluton security subsystem*. Its features [20] include a dedicated Arm M4F Cortex, tightly coupled memory (TCM), read-only memory (ROM), e-fuses, and Pluton engine; the engine includes hardware random number generator (HRNG), among other features integral to the security [56]. Second, Azure Sphere's TCB [27] is limited to the Pluton security subsystem, its runtime, and software referred to as Security Monitor (SM), where SM runs in the secure world of the *application processor subsystem*, which, among other features, has an ARM A7 processor that supports Arm TrustZone. Third, 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 [27]. 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 [27]. Hunt et al. [45] explain how renewable security, certificate-based authentication, and failure reporting properties are satisfied.

Now that we have described Azure Sphere's security properties, we shift focus on introducing the terminology used later in the thesis related to Azure Sphere application updates.

Microsoft splits Azure Spheres into groups to ease the management of application updates. At the highest level, Azure Sphere Tenant—a cloud entity representing and controlled by a user or company—logically separates devices by owners. Lower-level groups are referred to as products [13]; for example, a specific model of solar panels can be a product. At the lowest level, devices are split into groups such as development, testing, and production.

We use several terms associated with deploying a new application. First, Azure Spheres are *claimed* by a user or company. The procedure associates a device with a specific Azure Sphere Tenant [25], 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, a user or company creates an *image package*—a compiled binary and metadata. Then, the package is distributed to a specific product and group via Azure Security Service.

2.2 Related Work

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

The goal of the first line of work, also in which our work fits in, is to design and implement a REC system using blockchain. For example, Knirsch et al. [48] 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 [46], 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 [48], 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. Additionally, we enhance RECs with anonymity.

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. [61] 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. [39], and it attempts to determine the effectiveness of different price strategies on RECs.

Outside of academia, several industrial projects have proposed placing RECs on a blockchain. Perhaps the work closest to our own is the collaboration between Nasdaq stock exchange and Filament Corp. Filament has developed secure hardware that serves as a source of trust [31] and creates RECs that are then traded on Linq—Nasdaq's blockchain [51]. Although this design considers privacy issues and shares our goals, there is very little known publicly about Filament's hardware or Linq, so we are unable to contrast the two approaches from a technical perspective.

Another similar work is the project between the Energy Origin (TEO) [34], a subsidiary of the French multinational utility provider ENGIE, and Ledger. Ledger Origin [33] cryptographically attests its security state to provide guarantees for meter readings. Then, the meter readings are sent to Energy Web Chain [32], an open-source blockchain. Their work initially announced in 2017 is concurrent with ours. An official working solution was announced in 2020, but, as with Filament, no details are publicly known, making it impossible to compare this work with ours.

Several other industry projects[2] for efficiently generating RECs and placing them on a blockchain, including Powerledger [21], automatically trust meters as a source of information and do not consider privacy issues. Thus, our work goes beyond.

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

Chapter 3

Preliminaries

To design anonymous, end-to-end trusted RECs we draw upon techniques in the areas of secure hardware, blockchains, and some specific security primitives. Broadly, we establish trust in an REC by creating it on secure hardware and trading it on a blockchain. We protect the anonymity of the REC creator using ring signatures and stealth addresses. In this chapter, we informally define the techniques used in our system. We assume the algorithm inputs are defined over finite space.

3.1 Secure Hardware

RECs are created by smart meters that measure the energy produced by a renewable energy source. To ensure that these meters are not tampered with, we need to ensure that the hardware itself can be trusted and that the software executing on the hardware has been verified by a trusted third party. As a result, we informally define minimal secure hardware requirements. 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*. Meter managers can securely deploy software, patch security vulnerabilities, or introduce new functionality.

2. **Trusted execution** SH.Exec(APP, *in*): is used to execute an instantiated application APP using an input *in*. The output of a software execution should include not only an execution result but also a proof ϕ—created using *remote attestation*—that guarantees the security of the device; that is, the execution of a program inside the hardware is *virtually black-box* and cannot be observed or modified by the adversary.

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

Property 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.

3.2 Remote Attestation

With remote attestation, a trusted verifier periodically checks the authenticity and security state of a remote device [41]. Francillon et al. [41] 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. [41] 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.

Security: Similar to Definition 2 [41], 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.

3.3 Blockchain

A blockchain is a publicly shared and highly duplicated ledger with an immutable history [52]. Users of a blockchain are represented by their public key. Their interaction with the blockchain can be described using four abstract APIs, as discussed next.

1. **Address generation** B.GenAddr($amount$): is an algorithm that creates a blockchain address (pk, sk) that is later associated with a certain amount of trading currency.
2. **Transaction creation** B.CreateTX(pk_{rec}, sk_{send}, add_info): is an algorithm that creates a transaction for injection into the blockchain. A transaction contains information such as the sender sk_{send} of asset, the receiver pk_{rec}, or input add_info to a *smart contract*.

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

4. **Transaction fetching** B.FetchTX($txid$): is used to fetch a transaction $txid$ from the blockchain along with a proof that the transaction has not been double-spent.

In our implementation, we interact with two different blockchains, Hyperledger Fabric [35] and Algorand [42] using this abstract API.

3.4 Signatures

For completeness, 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.

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.

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.

3.5 Ring Signatures

A ring signature allows a user to generate a signature for a ring of users. Given the signature and the public keys of the users in the ring, anyone can verify that one of the users in the ring indeed signed the message without learning who it was [38]. We reiterate the ring signature definition by Bender et al. [38], 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, set R of k public keys, and 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.

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.

Security: Following Definition 7 [38], 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.

Anonymity: Following Definition 4 [38], 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.

3.6 Stealth Addresses

A stealth address for a signature scheme can be used to generate multiple *stealth* public keys that are indistinguishable from fresh keys. It can be used to protect the anonymity of the blockchain owner of newly created REC. That is, given the smart meter owner's public key for the blockchain, a smart meter can generate a one-time stealth address for each transaction [40]. The owner of the secret key can link the transaction to her original public key and spend it. For everyone else, the one-time address looks random. We define stealth address as 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 stealth public key pk_{new} and an auxiliary information γ.

3. **Address verification** SA.Vrf(pk_{new}, γ, sk): is an algorithm that verifies inputs: stealth public key pk_{new}, auxiliary information γ, and **receiver's secret** sk. The output is a stealth secret key sk_{new}, such that stealth secret key and the stealth public key form a valid key pair.

Correctness: The scheme is correct if any stealth public key generated using a public key of a user can always be used in the address verification with the corresponding secret key of the same user to determine the stealth secret key. The statement is only true for key pairs generated by the key generation algorithm.

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

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

Chapter 4

Design

In this chapter, we begin with a description of the goals and the threat model of our system. We conclude the chapter with a description of our design.

4.1 Desired System Properties

In this section, we specify the goals of our system, sketch its design, state its desired security and anonymity properties, and describe the threat model. A detailed design of the system is presented in Section 4.2.

4.1.1 Goals

Our high-level goal is to design a system that allows RECs to be generated efficiently, to be trusted by purchasers and regulators, and, when purchased, to hide the identity of the REC creator. We would like these goals to be met despite the presence of an adversary that may observe or modify messages.

4.1.2 System sketch

Our system comprises four main blocks (see Figure 4.1):

1. *Smart meters* measure renewable energy generation and create RECs. Each smart meter has a small trusted computing base (TCB) where the hardware root of trust resides. Software outside this TCB is not trusted. Smart meters are connected to a renewable generator using a trusted link.

2. A *blockchain* stores RECs. We trust the blockchain to prevent tampering of RECs stored in it, and to prevent double-selling of an REC.

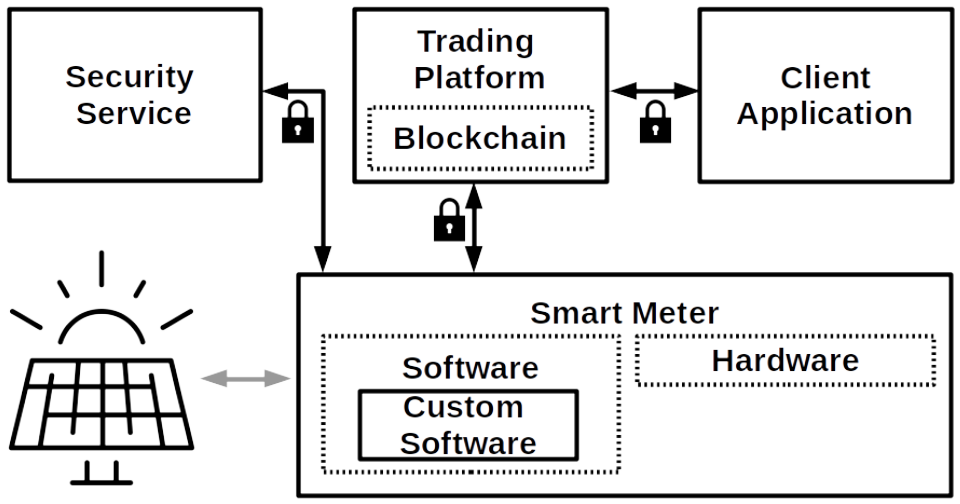


Figure 4.1: A system architecture diagram for trading RECs. The green energy produced by the solar panel is measured by the smart meter, while the security service verifies the internal state and authenticity of the hardware. The microcontroller periodically sends RECs to the trading platform, which at its core, is using a blockchain. End-users can trade RECs by interacting with the client application functionality.

3. A *trading platform* receives RECs from smart meters and stores them on the blockchain to be bought and sold.

4. *Client applications* interact with the trading platform to sell, purchase, and retire RECs.

In addition an external trusted *security service* periodically validates the correctness of the application software running on the smart meter using remote attestation.

We envision that owners of renewable generators are customers of an *aggregator*. The aggregator is responsible for installing authorized smart meters on customer premises, linking smart meter identity to customer accounts. They also collect RECs from the smart meter and use the trading platform to place these RECs on a blockchain, with the ownership of the REC retained by the customer. When an REC is bought, the proceeds are paid directly to the customer, through their blockchain account. Purchasers of RECs record the retirement of an REC on the blockchain (for instance, by sending them to a null key).

4.1.3 Properties

We now expand on our goal in Section 4.1.1. More specifically, we explain what it means for the system to be correct, secure, and anonymous.

Correctness: If operating correctly, the system should allow authorized, non-compromised smart meters to create RECs. REC owners should be able to retire RECs or sell the non-retired RECs they own. Finally, any blockchain end-user should be able to purchase any non-retired REC owned by other users.

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.

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* (see Appendix A.1) 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.

Remark. See Appendix A.2 for the security experiment.

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 a set of end-users and smart meters, except at least one other smart meter in the same ring group.

Remark. See Appendix A.3 for anonymity experiment. The idea is similar to a chosen-plaintext attack; an adversary can obtain RECs via smart meter oracle and then try to distinguish the output of smart meters in the same ring.

4.1.4 Threat Model

Threats to the integrity of our system, and hence a violation of the properties, come from five sources. To begin with, the *owner* of a renewable energy source could attempt hardware or software attacks to increase the value of the RECs produced by their smart meter. The owner might also attempt to impersonate other participants to compromise the system. Second, an *external adversary* who does not have access to the hardware could attempt remote attacks on a smart meter or the blockchain. Third, *end-users* might collude on the blockchain to double-spend RECs and increase their wealth. Fourth, the

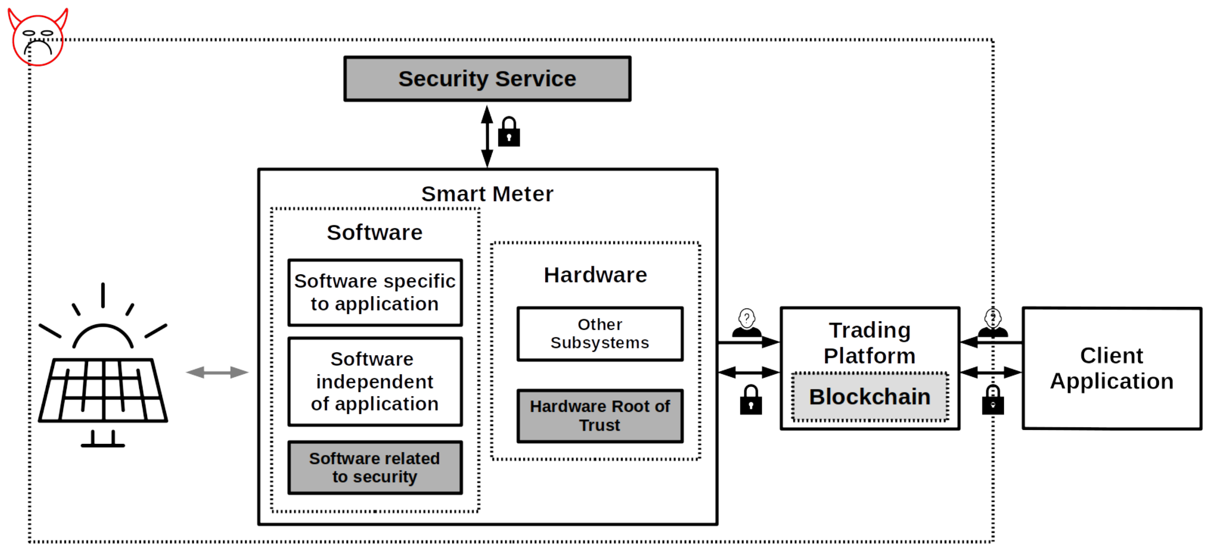


Figure 4.2: The system diagram visualizes our threat model. All trusted elements such as security service and connection between the solar panel and smart meter are in dark grey. No other elements are trusted. There are anonymous links between the smart meter and the trading platform and between the client application and the trading platform. Highlighted in light grey is the blockchain denoting that the trust depends on the choice of blockchain.

trading platform could act as a passive adversary and attempt to identify the smart meter or owner of a REC by observing newly created RECs or REC trades. Finally, an external *passive adversary* could try to learn about the energy usage pattern of an energy producer by observing their REC generation pattern. This energy usage information could reveal the lifestyle of small-scale energy producers such as households with solar panels, depending on the granularity of the RECs.

We model these attacks in the form of an adversary who has complete control over some number of smart meters, so can produce valid RECs from them. Moreover, the adversary can attack any part of the blockchain or smart meters, or passively observe any transaction on the blockchain. However, we assume that the adversary is computationally bounded so that it cannot break the cryptographic primitives we use.

4.2 Design

We now elaborate on our system shown in Fig. 4.1.

4.2.1 Smart Meter

We assume that smart meter hardware satisfies the requirements for secure hardware, as described in Section 3.1. Specifically, a smart meter consists of secure hardware that runs the REC-generator application. This application measures electrical signals (voltages and currents) over a secure link to determine the amount of energy produced over any time period. It then generates RECs and sends these to the trading platform to be placed into a blockchain. The smart meter is also periodically audited using remote attestation. We now describe the details of these processes.

- **Initialization** SM.Init(): The algorithm creates attestation key pair (pk_{Att}, sk_{Att}) and device key pair (pk_{SM}, sk_{SM}).

Cryptographic Component

- **Remote attest** SM.Att(pk_{Att}, sk_{Att}, pk_{SM}, s): The algorithm uses the *attestation key* stored in the root of trust to periodically create attestation tokens and send them to the security service. 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). Upon receiving a message (PROOF, ϕ), the proof ϕ is stored internally and the algorithm outputs the proof ϕ. If a response is not received, the algorithm outputs **fail** .
- **Receive new application** SM.NewApp(pk_{SS}): Received applications are 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*).

Application

The application is responsible for two separate interactions with the trading platform: renewable energy generator information update and REC generation. Algorithms are executed using SH.Exec(APP, in), where in is supplied by the environment, while the output of the execution is sent to the trading platform. The proof *phi* is the latest result from remote attestation. 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.

- **Information Update** APP.Update(pk_{SM}, sk_{SM}, in): Renewable energy generation information update algorithm periodically sends data to the trading platform using the input in to facilitate REC origin tracking, such as GPS coordinates. If a trading platform reply is received, the response, an anonymity set, is stored internally. More formally, the algorithm outputs a message (UPDATE, m, ϕ), 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 the 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 periodically creates and sends RECs to the trading platform using a measurement of the produced energy specified in the input in. Each REC has an owner set to a blockchain address created using stealth addresses and 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 RS.Sign(sk_{SM}, *ring_list*, (in, $pk_{RECOwner}$, γ)). If all steps are successful, the algorithm outputs (REC, *ring_list*), otherwise output **fail** .

Since our design removes the need for expensive physical audits, RECs can be more fine-grained than the typical 1 MWh required by most jurisdictions, which would help small-scale energy producers.

4.2.2 Security Service

The security service is responsible for detecting smart meter tampering without the overhead of physical third-party audits. It is also responsible for authorizing the users who deploy new smart meter software where each software is signed using the secret key of the security service. Examples of such service are Azure Sphere Security Service [27] and Intel Attestation Service [24].

- **Initialization** SS.Init(): The algorithm creates a security service key pair (pk_{SS}, sk_{SS}).

- **Remote Attestation** SS.Att(pk_{SS}, sk_{SS}): The service periodically checks the security state and authenticity of smart meters using standard remote attestation token verification. If verification is successful, the token is signed using a secret key that represents the security service. The signed token represents proof that the smart meter can be trusted, and it has not been compromised. More formally, the 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}) and the signature σ_{ϕ} is the output of S.Sign(sk_{SS}, (pk_{SM}, s, α)).

- **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*, σ_{new_app}) and σ_{new_app} = S.Sign(sk_{SS}, *new_app*). We consider user authorization and software management to be orthogonal issues to our work.

4.2.3 Trading Platform

The trading platform, operated collectively by the aggregators and coordinated using a blockchain, collects RECs from smart meters, places them on the blockchain, and allows them to be traded. The trading platform also collects renewable energy generator information used to determine ring groups, a set of geographically close smart meters. All smart meters that have been able to successfully update their information 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 4.3). This is discussed in greater detail below.

- **Smart Meter Update** TP.Update(pk_{SS}, *auth_keys*): On receiving a renewable generator information update from a smart meter, the trading platform verifies the message by checking for a valid signature signed using an identity that is among a list of authorized device identities. It also ensures that the smart meter itself can be trusted by verifying the remote attestation proof in the message. If successful, to protect the anonymity of energy producers, the trading platform selects a group of smart meter that are geographically closely located[1]. Then it returns the same ring group to every smart meter in the group when updating the information for the renewable energy generator. 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 (m, 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($meter\_list$): On receiving a REC, the trading platform verifies each received REC to have a valid ring signature using its ring group. Additionally, each smart meter in the ring group is ensured to have valid information update and attestation proof; otherwise, the REC may be created by a compromised smart meter. If successful, the trading platform creates and publishes a blockchain transaction that includes the REC and stores it in buffer rec_list. 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

- **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(): On receiving a request with a transaction identifier, the algorithm fetches the corresponding blockchain transaction. Then, it creates a list of

[1]For simplicity, we assume that each end-user will have only one smart meter

smart meter identities and their attestation 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** .

4.2.4 Client Application

The client application is the end-user interface to the system, and it allows the end-user to verify, retire, sell, buy RECs. It also has buffers *rec_own* and *rec_bid*—as the name suggests–store the RECs owned by the user and the RECs available for bidding. Our design is blockchain-independent, and, therefore, when the blockchain is permissionless and the end-user has sufficient resources, the client application can directly interact with the blockchain.

- **Initialize** CA.Init(amount): Each client generates (pk_{owner}, sk_{owner}) address shared with the smart meter, and a blockchain identity (pk_{Bid}, sk_{Bid}) specifically to bid for RECs.

Interaction with Trading Platform

- **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 stealth address verification 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 renewable generator information update transaction $txid$ is verified to be valid to ensure that the smart meter is authorized. 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($null$, $sk_{RECOwner}$, 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** .

4.3 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 against 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.

Chapter 5

Implementation

In this section, we present an implementation of the abstract design presented in Chapter 4. Recall that our design left unspecified the hardware used to implement the smart meter as well as the choice of the blockchain. We implement two versions of our design; both use Azure Sphere [6] as the basis for the smart meter, which is integrated with Algorand [42] and Hyperledger Fabric [35] as the choice of the blockchain. In the remainder of this section, we justify our choice of components. Note that our implementation focuses only on REC security, and our design for anonymity based on stealth addresses and ring signatures is left for future work.

5.1 Smart Meter

For REC security, the smart meter needs to build on a platform that supports the requirements outlined in Chapter 3. We find that the commercially available Azure Sphere microprocessor control unit (MCU), unlike other MCUs such as Intel Galileo [18], Raspberry Pi [23], and Arduino [2], satisfies the requirements in our design and, unlike previous proposals described in Section 2.1.2, it is feasible for real-world implementation. This is because the device provides a hardware root of trust via the *Pluton security subsystem*, remote attestation implemented in silicon, and secure software deployment [53]. Additionally, the seven properties defined in Reference [45] are also fully met as described in Section 2.1.3. Thus, this device provides a secure environment that allows for trusted execution.

We do need to assume that the manufacturer correctly manufactures Azure Sphere and loads security information such as software related to secure boot and it's public key at the time of manufacture. We also need to trust Microsoft for remote attestation, maintaining software, and enforcing access control to application deployment. Lastly, the software developer needs to be trusted for correctly implementing and distributing the same Azure Sphere application.

It is important to emphasize that manufacturer is not trusted for generating and protecting the identity of Azure Sphere. The Pluton security engine uses its hardware random number generator to internally generate an attestation key pair that uniquely identifies the device [56]; the process cannot be controlled by the manufacturer.

During the addition of a new producer, as described in Section 4.3, the REC aggregator needs to perform four configurations that require a physical USB connection. The first one is to claim the Azure Sphere (see Section 2.1.3) 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. 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 the configuration is completed, the first interaction between Azure Sphere and Azure Security Service is remote attestation [7]. The cloud 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 [53]. To sign the response, Azure Sphere uses the attestation key pair. This simple challenge-response protocol also prevents replay attacks. If the response is accepted, the cloud issues an X.509 certificate that is valid for a day. Any REC without a valid X.509 certificate is not trusted.[1]

After a successful remote attestation, each Azure Sphere continuously executes the application distributed by the cloud. The application in our implementation handles the renewable energy information update and REC production and 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

[1]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.

based on Microsoft's ADC and intercore communication[2] examples with tweaks to fit our use case.

The **High Level Sub-Application** is written in C language, and it is deployed on the 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; that is, the proof of authenticity and security state was not directly available. As a result, this application does not implement the renewable energy information update algorithm described in our design. Instead, aggregators are trusted to add the appropriate device identities to Hyperledger Fabric and continuously verify the security state of the devices using Azure Cloud.

We detail the second implementation of the application along with the description of the trading platform based on Algorand in Section 5.2.2.

5.2 Trading Platform and Client Application

We now describe two implementations of the trading platform and client application components. Although the one based on Hyperledger Fabric provides REC security, bootstrapping a permissioned blockchain could be troublesome for REC aggregators. As a result, we describe a second implementation based on already bootstrapped permissionless blockchain, Algorand; the second implementation also decreases the trust in the aggregators.

5.2.1 Implementation using Hyperledger Fabric

As shown in Figure 5.1, each Azure Sphere publishes RECs to Hyperledger Fabric via Azure Cloud. We envision several aggregators to maintain the blockchain, and the chaincode to be

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

open-source and correctly implement the trading logic. Therefore, end-users trust several aggregators to not collude and alter the chaincode. Additionally, end-users trust Microsoft and the aggregator—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 the 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.

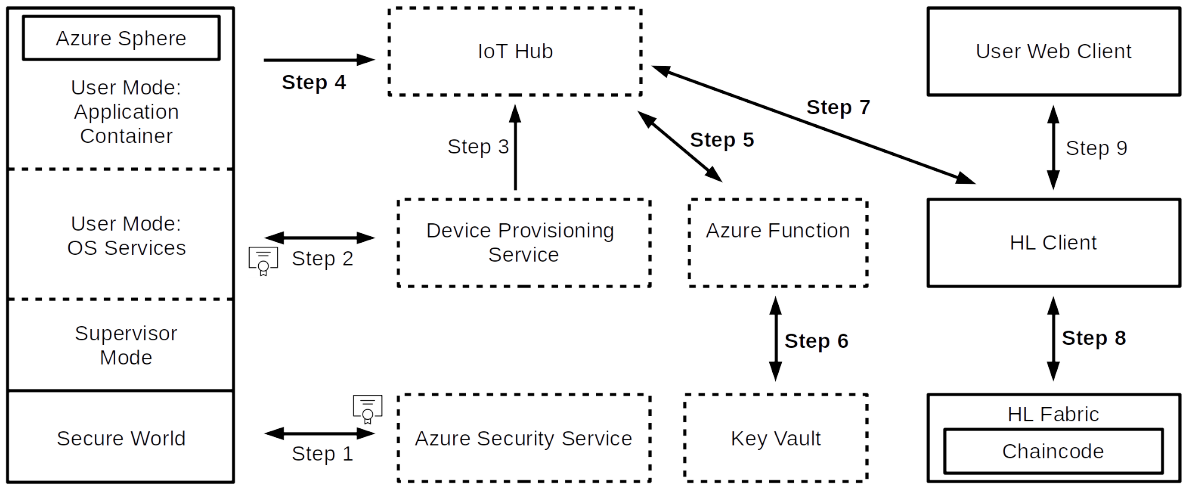


Figure 5.1: The figure visualizes a Hyperledger Fabric implementation of the high-level design. Steps 1 through 9 chronologically show the information flow. The first three steps establish the authenticity and security of Azure Sphere, while the highlighted in bold steps 4 through 8 show the continuous REC production. Lastly, in dashed squares we differentiate Azure Cloud services.

Azure Cloud

Device Provisioning Service

The device provisioning service (DPS) [4] 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 5.1, this process corresponds to step 2 and 3. The verification of X.509 certificate is the main reason this service was used, and it is the workaround to not having direct access to the X.509 certificate on Azure Sphere.

IoT Hub

IoT Hub is a service that provides low-latency and highly reliable communication between IoT devices and cloud [5].

We make several IoT Hub configurations, which are important to the security. Although several underlying protocols can be chosen in the service, we use MQTT [19]—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 given 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 aggregators. Lastly, IoT Hub is also configured to allow HL Client application connections. More specifically, the application is issued a SaS token, and it only receives IoT Hub messages.

Azure Function

Azure Function [3] 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 aggregator. The signature is ensured to pass malleability checks, a measurement that prevents the ability to modify a REC without invalidating its signature [9]. 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 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 [26], 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 aggregators, and two peers per organization, which operate as docker containers. The aggregator 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 and can invoke and query the chaincode algorithms. 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 5.2 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).

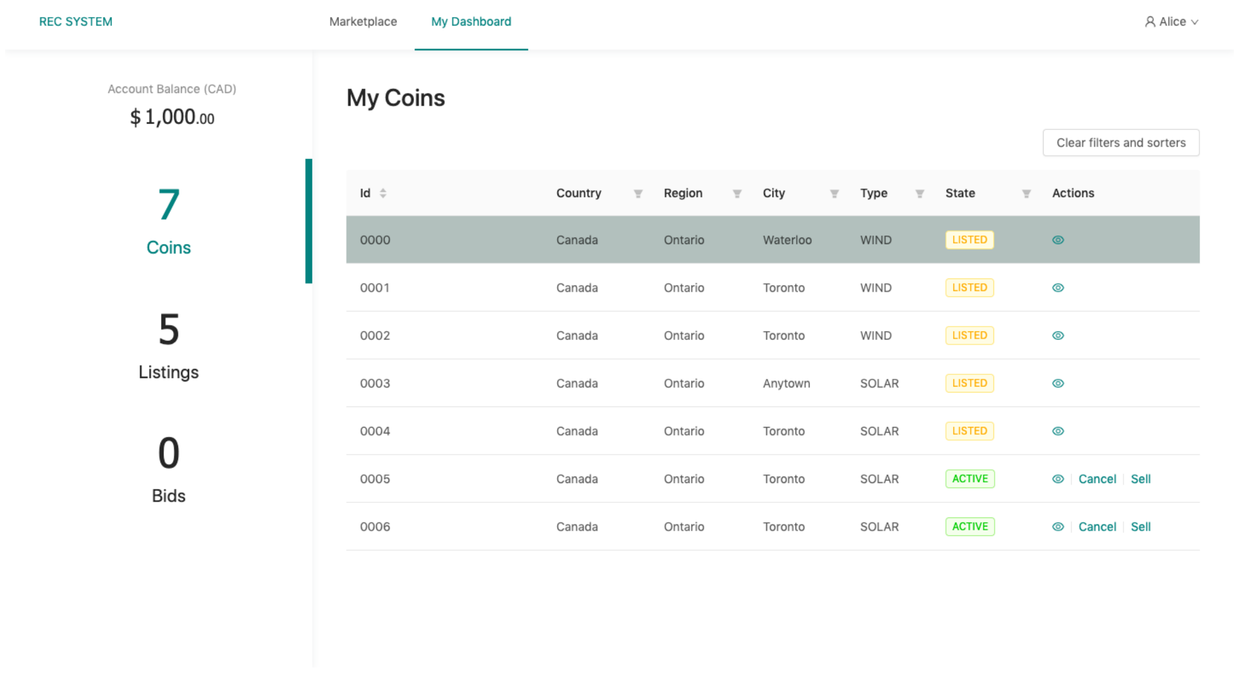
In a real-world deployment, however, HL Client would be removed from the system. Instead, the function in Azure Function would have its own wallet and user credentials that can be used to directly invoke or query Hyperledger Fabric. Each client application would also need to have its own wallet and user credentials as well. Those credentials would be issued by Fabric's membership service maintained by the aggregator using Fabric's Attribute-Based Access Control to enforce permissions.[3]

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.

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.

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



My Coins

Id	Country	Region	City	Type	State	Actions
0000	Canada	Ontario	Waterloo	WIND	LISTED	
0001	Canada	Ontario	Toronto	WIND	LISTED	
0002	Canada	Ontario	Toronto	WIND	LISTED	
0003	Canada	Ontario	Anytown	SOLAR	LISTED	
0004	Canada	Ontario	Toronto	SOLAR	LISTED	
0005	Canada	Ontario	Toronto	SOLAR	ACTIVE	Cancel | Sell
0006	Canada	Ontario	Toronto	SOLAR	ACTIVE	Cancel | Sell

Figure 5.2: Web application used by end-users to trade RECs. The cancel button is equivalent to what we refer to as retiring a REC.

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 [12], 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.[4] 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 aggregator. This is handled by an initialization algorithm, which is always the first algorithm invoked after a chaincode is installed and instantiated. The algorithm submits

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

a TLS request to Azure Function; the response, the public key of the aggregator, 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.[5]

Algorithm 2 is only used by aggregators to add end-users and their smart meters. Before adding a new user and producing unit to the blockchain state, the algorithm verifies that another user does not already own the smart meter; 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 aggregator.

Algorithm 3 A chaincode algorithm creating a new REC

```
1: function newREC(REC)
2:     pubKey ← get the aggregator 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—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 a reference to the new listing.

[5]ajv schemas

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
```

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.

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 with the value of the successful bid. 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.

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.

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
```

5.2.2 Implementation using Algorand

We build a second implementation of the trading platform and client application on top of Algorand, which we choose due to its high throughput of 1,000 transactions per second [42].

In this second implementation, we make several improvements to the implementation based on Hyperledger Fabric. The burden of bootstrapping the system is removed from REC aggregators since Algorand is already bootstrapped. End-users—that is, sellers, buyers, or even auditors—can directly access the proof of authenticity and security state to decide if a device can be trusted. Additionally, REC aggregators are no longer trusted to maintain cloud services.

An important difference in Algorand implementation is that the trading logic 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.

End-users trust REC aggregators only for availability. REC aggregators are expected to maintain middleware that allows new smart meters to update the renewable energy information. Without being able to update the information 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.

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

Renewable Energy Information Update

Figure 5.3 shows the renewable energy information update process between Azure Sphere and middleware API, which is the first interaction between the smart meter and the trading platform.

Every day the application in Azure Sphere creates and submits an information update message to the middleware API. The interaction facilitates REC origin tracing since the information in the message includes GPS location. Additionally, the message is signed with a SECP256K1 ECC device key pair that is generated by the application we developed using wolfSSL library [29] and Azure Sphere's HRNG. The smart meter sends the message to the middleware API via HTTPS POST request along with the client certificate issued by Azure Cloud.

The middleware API verifies the information update request before it is added to Algorand. The X.509 certificate and its chain of certificates are validated to ensure that the smart meter is not compromised. Additionally, the signature of the information update is verified using the device public key of the smart meter, which is included in the update message. Lastly, the middleware API checks if the device key is present in a list of authorized identities, which are ensured to be smart meter attached to solar panels during registration described in Section 4.3.

After successful verification of the information, 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 a REC aggregator. The signature is seen as approval that a particular Azure Sphere can generate RECs. Lastly, the middleware API constructs and submits an Algorand transaction that includes the modified information update message. A response that includes the update message transaction identifier and device owner Algorand address is sent back to Azure Sphere; the owner address is mapped to the smart meter by the REC aggregator during the deployment of Azure Sphere as described in Section 4.3.

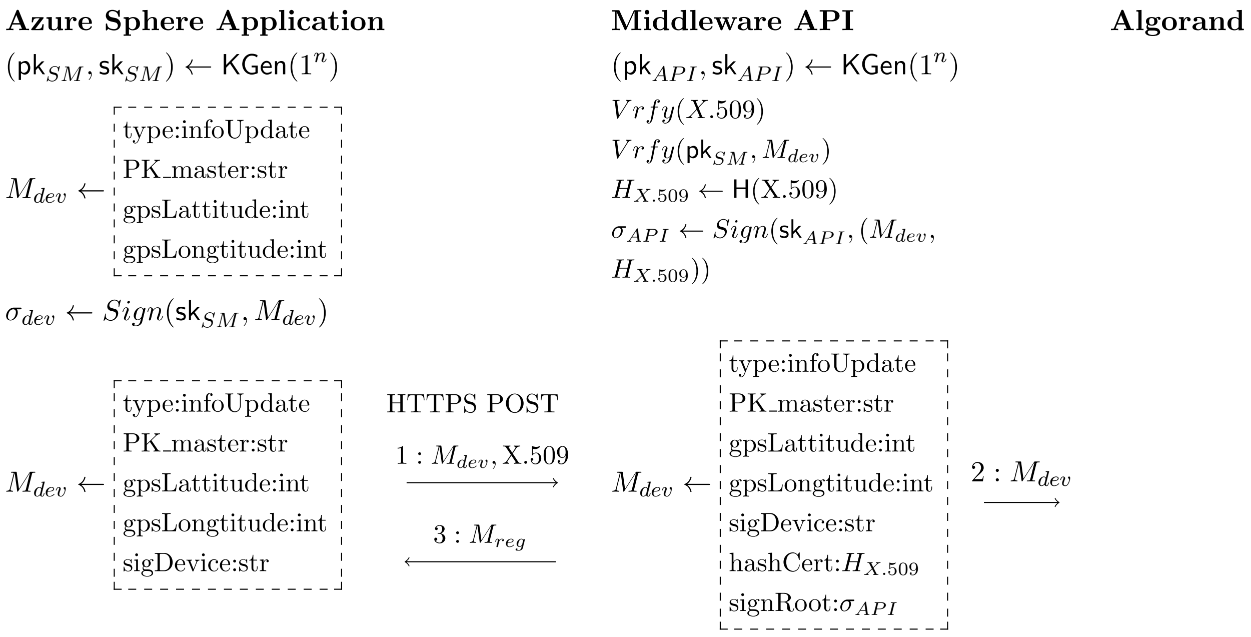


Figure 5.3: The figure shows the renewable energy information update process. The steps in the figure are in chronological order from top to bottom, and the sequence of exchanged massages is numbered.

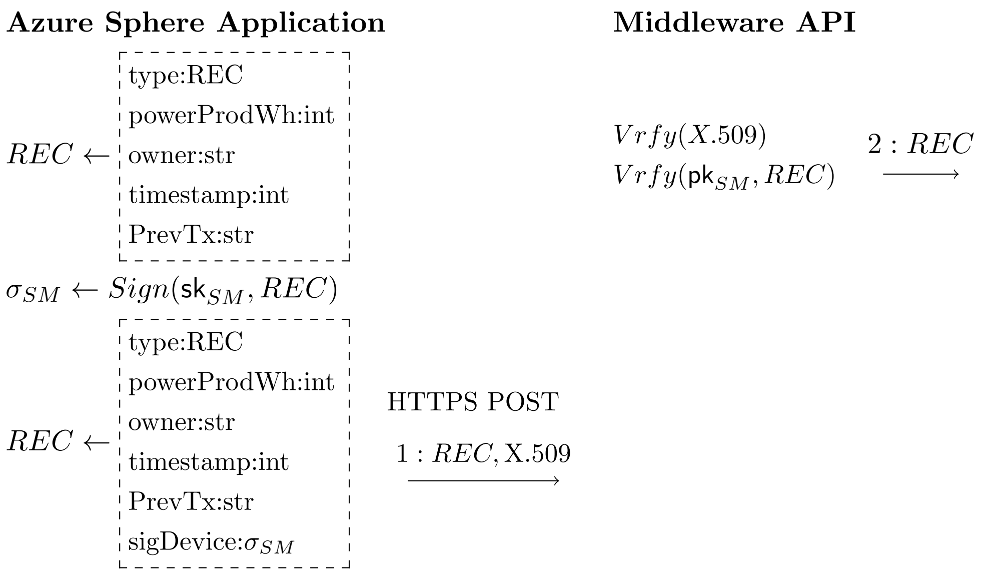


Figure 5.4: The figure shows the process that creates new RECs. The steps in the figure are in chronological order from top to bottom, and the sequence of exchanged massages is numbered.

Creating RECs

In our implementation, Azure Sphere creates RECs in intervals that can be as fine-grained as, for example, 15-minutes. The REC includes the amount of produced green energy, Algorand address of the smart meter owner, and transaction identifier of the latest device information update transaction. Additionally, the REC is signed with the device private key and sent along with the client certificate to the middleware API via HTTPS POST.

Similar to the information update process, before accepting the REC, the X.509 certificate of the device and REC signature are verified. Additionally, the update information transaction is fetched and ensured that it is signed by an aggregator, otherwise the smart meter is not authorized to create RECs. This is to ensure that invalid RECs are not added to the Algorand in vain. Lastly, the middleware API creates and sends a transaction to Algorand. The transaction carries the REC as a payload in the note field.

It is important to note that the REC does not need to be necessary sent to the middleware API. Before approving the retirement or selling of a REC, the aggregator always verifies that the smart meter is authorized to create the REC.

Trading RECs

End-users interact with two Node.JS sub-applications to trade RECs. Unless specified otherwise, both sub-applications locally create Algorand transactions with payloads in the note field shown in Figure 5.5.

The first application allows REC owners to sell or retire RECs. End-users are presented with the list of RECs they own, which is created based on a query to the middleware API. If the owner wants to retire a REC, the application creates and publishes to Algorand a retire transaction that specifies the transaction identifier of the user-chosen REC. If the owner wants to sell a REC, the application creates and sends to Algorand a transaction that lists the user-chosen REC for sale; the transaction includes input such as minimum price and the length of the listing. The length is specified in the *lastround* field of the Algorand transaction. This would prevent the transaction from being published in Algorand if the block number has passed the number specified in the lastround field.

After waiting sufficient time, the owner can check the available bids for the REC via a query to the middleware API. If satisfied, the end-user selects a bid that is used by the application to create and send a propose settle Algorand transaction; the transaction has information such as final price and transaction identifier of the user-chosen bid. Similar to the listing, it also specifies how long the offer is valid via the lastround field of the Algorand transaction.

REC aggregators monitor Algorand and verify and approve retire transactions so that end-users can use the RECs for claims, such as being '100%' green. When an aggregator detects a retire transaction, it fetches the corresponding REC and information update transaction, 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 the RECs they own. Then, the aggregator verifies the signature in the REC using the device public key in the information update transaction, which guarantees that the REC originates from a specific device. Lastly, the information update transaction 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.

End-users interested in buying RECs interact with the second application. Similar to the first one, the application queries the middleware API for all available REC listings in Algorand.

If the end-user decides to bid for a REC, the application verifies the validity of the

REC before creating a bid transaction. All relevant to the REC listing information is gathered by the application via a request to the middleware API. This includes the REC, the information update transaction referenced in the REC, and the Azure Sphere's client certificate referenced in the information update transaction. The application verifies that the listing transaction sender matches the REC owner. Then, the REC signature is verified using the device public key in the information update transaction, and the information update transaction is verified to be signed by a REC aggregator; otherwise, the smart meter has not created the REC or has not been authorized to create the REC. Lastly, the client certificate and its chain are checked to be valid to ensure that the smart meter is not compromised. 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 the REC, the application creates a bid transaction.

The second application also allows an end-user to check if their bid was successful and transfer funds to the original REC owner. 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 cryptocurrency specified in the propose settle transaction from bidder's address to the REC owner's address. The lastround field of the settle Algorand transaction is set to the last round of the propose settle transaction to prevent publishing the agreement after the propose settle has expired.

Similarly to retiring RECs, REC aggregators continuously monitor Algorand for settle transactions. If such a transaction is detected, the aggregators repeat all the validations that the second application does. Additionally, the middleware verifies that the settle transaction transfers a sufficient amount of cryptocurrency, 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 cannot be used for future trading. Once a REC is sold, it is also considered retired under the buyer's address.

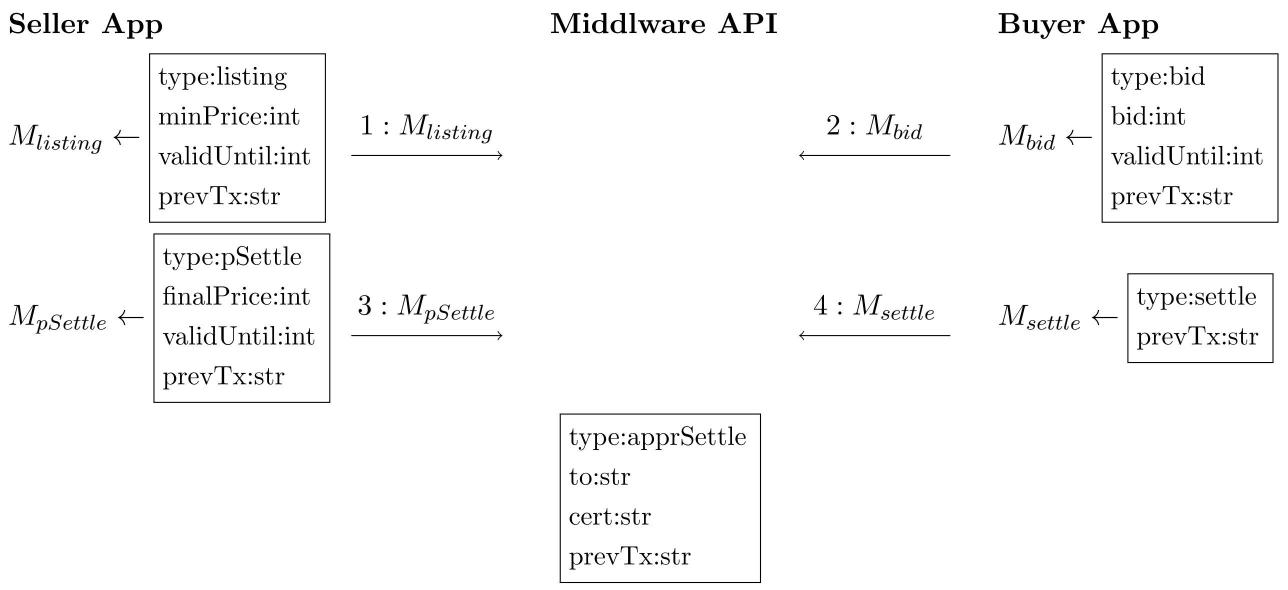


Figure 5.5: The figure shows how RECs are traded in Algorand. The sequence of exchanged massages is numbered.

Chapter 6

Evaluation

In this chapter, we evaluate the security and anonymity of the design presented in Chapter 4 and the implementations in Chapter 5. We make three assertions about security and one about anonymity.

Remark. Both implementation incur small cost (see Appendix B), an important factor for real-world adoption.

6.1 Security

Security Assuming that link between the smart meter and solar panel is secure and remote attestation, secure hardware, ring signatures, and blockchain are correctly implemented and satisfy the defined properties, the system is secure as described in Section 4.1.3.

Assertion 6.1.1. (Security) Only non-compromised smart meters can create RECs.

In our design, an adversary attempting to create a REC using a compromised smart meter needs to forge the attestation proof; otherwise, the trading platform won't accept the REC because each smart meter in the REC ring group is verified to have a valid proof. However, hardware that correctly implements remote attestation always produces a proof that correctly captures the state and software on the smart meter [41]. Additionally, the attestation key used during the interaction is part of the hardware root of trust; that is, it is restricted to be only used by the remote attestation in the hardware, and it cannot leave the device. Therefore, the interaction cannot be forged.

Our chosen commercially available MCU implements several protections to prevent compromise of the smart meter. The secure boot [53]—that is, the only software that the device can load is software signed by Microsoft—implemented in Azure Sphere guarantees that the device begins its operation in a secure state. Additionally, we rely on the silicon mitigations against attacks such as rollback [56]—an attack that restores a previous vulnerable state in the device—and physical-side channel attacks to maintain the secure state.

The periodical remote attestation between Azure Sphere and Azure Cloud guarantees that any tampering with Azure Sphere's state will be detected. As already mentioned in Section 5.1, Microsoft issues X.509 certificate only if Azure Sphere successfully completes the remote attestation interaction.

In our implementations, the verification of the remote attestation proof is done via verification of the X.509 client certificate sent by the smart meter. In the implementation based on Hyperledger Fabric, the verification of the certificate is done by the DPS service in Azure Cloud, as described in Section 5.2.1. If DPS cannot successfully verify the X.509 certificate, Azure Sphere is not allowed to send telemetry on IoT hub, and, therefore, cannot create RECs. In the implementation based on Algorand, the middleware verifies the X.509 certificate's validity during the generator information update. Without an information update signed by an aggregator, the smart meter cannot create RECs that are considered valid. Additionally, an end-user can request the X.509 certificate of the smart meter from an aggregator that possesses it; then, the user needs to ensure that the received X.509 certificate has a hash that matches the information update transaction, and verify the certificate and its chain. Therefore, end-users do not need to trust the aggregators.

Assertion 6.1.2. (Security) Only authorized smart meters can create RECs.

An adversary attempting to create a REC using an unauthorized smart meter needs to compromise the device key pair in the smart meter application. In our design, the key pair would allow the adversary to create a REC with a ring signature that would be accepted by the trading platform since each key in the ring group is verified to be among a list of authorized keys that is created based on every register energy producer with valid update information as discussed in Section 4.3.

In our implementation based on Algorand, the application in Azure Sphere generates its own key pair using wolfSSL [29] and Azure Sphere's hardware random number generator. The generated public key is periodically sent to the trading platform in the Sphere's update messages via HTTPs. As a consequence, we trust that the software developer has correctly implemented the Azure Sphere application. Obtaining the device key pair would require a compromise of the smart meter application, which would be detected by remote attestation. Additionally, the smart meter application state is isolated by Azure Sphere. This is done using hardware since the device satisfies the compartmentalization properties in Azure Sphere described in Section 2.1.3. Thus, the key is secure against a passive observer and active tampering attacks.

Assertion 6.1.3. (Security) RECs cannot be double-spent or tampered with.

In our design, since RECs are signed, any change to the REC will invalidate its signature. Additionally, double-spending is prevented using a blockchain. In both implementations, we need to assume that the underlying consensus protocol assumptions are not violated. However, in the implementation based on Hyperledger Fabric, we also trust that the logic in the chaincode correctly implements the algorithms described in Section 5.2.1, while in implementation based on Algorand, the blockchain is used only for the exchange of messaged and the correctness and security are enforced off-chain.

Since the trading platform logic is enforced off-chain in Algorand, we discuss in more detail the possibility of double-spending while selling or retiring a REC, which needs to be approved by a REC aggregator.

REC aggregators parse all transactions and keep a state of the 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 retirement of a REC. In such case, external auditors are expected to detect the misbehavior and, therefore, the aggregator will be penalized. We note that the external auditor could be anyone, including end-users with sufficient resources.

Similarly to retiring RECs, REC aggregators would be penalized if they misbehave when approving the selling of a REC. However, it is important to mention that a REC owner can also create multiple proposals for settling. In such case, the buyer needs to ensure that the proposal is valid, and the seller has not sent a second proposal for other end-user. A proposal is considered valid if it is active. A proposal becomes active when it is published on the blockchain, and it expires as specified in the lastround field. Only the earliest non-expired proposal is considered valid. If there are two proposals published at the same time, both are considered invalid. It is responsibility of the client application of the end-user to keep track of the validity of the propose settles, or in our implementation, this has been offset to the middleware.

Remark. The settle has the same end time as the proposal, and, therefore, if the transaction is intercepted and attempted to be published at a later time, it will fail.

We reiterate that our implementation could be enhanced with Algorand 2.0—not available during our implementation—by implementing the trading logic in smart contracts.

6.2 Anonymity

Anonymity 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 described in Section 4.1.3.

Assertion 6.2.1. (Anonymity) An REC cannot be attributed to a specific smart meter or 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 in our design.

The first possible source is the REC signature. However, since each REC is signed using a ring signature, no adversary can determine the signer of a REC per definition. The most an adversary can learn is that the smart meter is one of the parties in the ring group, where each smart meter in the group will receive the exact same group of identities that are geographically closely located since we assume that the trading platform is at most a passive adversary.

The second possible source is the information in the REC. In our design, the REC includes the owner blockchain address of the REC that is generated using stealth addresses. Since any derived stealth address using stealth address is indistinguishable from a randomly sampled address, the REC does not reveal any information about the REC 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 only observes the blockchain transactions cannot link two RECs to the same smart meter or end-user address that owns the smart meter.

Chapter 7

Conclusions

In this work, we design a system for trading of anonymous RECs that prevents fraud, helps detect inconsistencies via increased transparency and reduces the burden for the end-user. For security, we rely on secure hardware, while a blockchain prevents double-spending. Additionally, for anonymity, we use ring signatures to hide the smart meter that creates the RECs, and stealth addresses to hide the smart meter owner. Our work, therefore, demonstrates the feasibility of end-to-end trust for RECs, and indeed, for any physical quantity that can be reliably measured by secure hardware.

We note two limitations of our work. First, we currently do not implement ring signatures and stealth addresses. Second, we currently do not protect the anonymity of the REC purchaser. This could be achieved by creating multiple unrelated blockchain accounts. Then, a purchaser could use off-chain zero-knowledge proofs to convince a third party that the RECs retired on these accounts belonged to them. We leave these extensions to future work.

References

[1] Activity statistics | AIB. https://www.aib-net.org/facts/market-information/statistics/activity-statistics-all-aib-members. Last accessed on 2020-06-09.

[2] Arduino. https://www.arduino.cc/. Last accessed on 2020-06-09.

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

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

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

[6] Azure Sphere. https://azure.microsoft.com/en-us/services/azure-sphere/. Last accessed on 2020-06-09.

[7] 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.

[8] 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.

[9] Bip-0062. https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki. Last accessed on 2020-06-09.

[10] BMO Financial to offset 100% of electricity use with RECs. https://www.smartenergydecisions.com/blog/2020/04/20/bmo-financial-to-offset-100-of-electricity-use-with-recs. Last accessed on 2020-06-02.

[11] 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.

[12] 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.

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

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

[15] Empowering App Development for Developers. https://www.docker.com/. Last accessed on 2020-06-09.

[16] Guaranteeing the origin of european energy. https://www.aib-net.org/. Last accessed on 2020-06-09.

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

[18] Intel Galileo Gen2. https://www.arduino.cc/en/ArduinoCertified/IntelGalileoGen2. Last accessed on 2020-06-09.

[19] MQTT. http://mqtt.org/. Last accessed on 2020-06-09.

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

[21] Power Ledger. https://www.powerledger.io/. Last accessed on 2020-06-09.

[22] Proof of Stake. https://en.bitcoin.it/wiki/Proof_of_Stake. Last accessed on 2020-06-09.

[23] Respberry Pi. https://www.raspberrypi.org/. Last accessed on 2020-06-09.

[24] Strengthen Enclave Trust With Attestation. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/attestation-services.htmls. Last accessed on 2020-06-09.

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

[26] 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.

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

[28] What is the Paris Agreement? https://unfccc.int/process-and-meetings/the-paris-agreement/what-is-the-paris-agreement. Last accessed on 2020-06-09.

[29] wolfSSL. https://www.wolfssl.com/. Last accessed on 2020-06-09.

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

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

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

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

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

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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

Definitions

Informally, a function is negligible if there exist a n_0, such that for any real none negative c and every $n > n_0$, as n grows the function goes to zero faster than $\frac{1}{n^c}$, referred to as $\mathsf{negl}(n)$.

A.1 Smart Meter Oracle

The smart meter oracle is modeled as stateful, and, as the name suggests, it simulates the smart meter operation. It consists of two sub-algorithms. The first sub-algorithm initializes the oracle by creating a set of authorized smart meters, where the size of the set is specified in the input.

Algorithm 7 A sub-algorithm capturing the smart meter attestation, registration, and REC production

```
1: function SmartMeter.Init($m$)
2:     Initialize an empty list $auth\_keys$
3:     for i ∈ [1,..,$m$] do
4:         $[(pk_{SM}, sk_{SM}), (pk_{Att}, sk_{Att})] \leftarrow$ SM.Init()
5:         Append $[(pk_{SM}, sk_{SM}), (pk_{Att}, sk_{Att})]$ to $auth\_keys$
```

The second sub-algorithm can be used to create RECs. It ensures that the smart meter i is authorized, has a valid proof for authenticity and secure state, and, then, creates a REC using the measured green energy in, the public key of the owner, and the specified device state.

Algorithm 8 A sub-algorithm capturing the smart meter attestation, registration, and REC production

```
1: function SmartMeter.REC($in$, $pk_{owner}$, $s$, $i$)
2:     if $auth\_keys$[i] = empty then
3:         Output fail
4:     else
5:         $(pk_{SM}, sk_{SM}), (pk_{Att}, sk_{Att}) = auth\_keys$[i]
6:     if $\phi$ stored by the smart meter is not valid then
7:         $out \leftarrow$ SM.Att($pk_{Att}$, $sk_{Att}$, $s$)
8:         If $out$ = fail , stop and output fail , else $\phi = out$
9:         $out \leftarrow$ SM.Update($pk_{SM}$, $sk_{SM}$, $in$, $\phi$)
10:        If $out$ = fail , stop and output fail , else $ring\_list = out$
11:    $(REC, ring\_list) \leftarrow$ SM.NewREC($pk_{SM}$, $sk_{SM}$, $in$, $ring\_list$, $pk_{owner}$)
12:    If $REC$ is empty, stop and output fail . Otherwise output the $(REC, ring\_list)$.
```

A.2 Security

Definition A.2.1. Security: The system is secure, if for every PPT $\mathcal{A}$, Experiment 9 and Experiment 10 output success with at most negligible probability $\mathsf{negl}(n)$ for any n $\in \mathbb{N}$, set of authorized device *auth_keys*, and set of end users *end_users*.

A.3 Anonymity

Definition A.3.1. Anonymity: The REC production is anonymous, if for every PPT $\mathcal{A}$, Experiment 11 outputs success with at most $\mathsf{negl}(n)$ probability and Experiment 12 outputs success with at most $\frac{1}{k} + \mathsf{negl}(n)$ probability, where k is the size of a ring group. The definition should be valid for any n $\in \mathbb{N}$, set of authorized devices *auth_keys*, and set of end users *end_users*.

Experiment 9 Non-authorized or compromised smart meter creating a REC

```
1: $(pk_{SS},\ sk_{SS}) \leftarrow$ SS.Init() and SmartMeter.Init($m$)
2: Let $auth\_keys$ = L$_{adv}$ ∪ L$_{other}$, where L$_{adv}$ is a set of smart meters controlled by the adversary, while L$_{other}$ is a set of all other meters. Furthermore, let $end\_users$ = U$_{adv}$ ∪ U$_{other}$, where U$_{adv}$ is the set of users controlled by the adversary
3: $\mathcal{A}$ is given access the oracle SmartMeter.REC($\cdot$, $\cdot$, $s_{appr}$, $\cdot$) (Algorithm 8), where $s_{appr}$ is an approved smart meter state. For smart meters in L$_{adv}$, the adversary can specify the input in and $pk_{owner}$.
4: The adversary queries the oracle SmartMeter.REC($\cdot$, $\cdot$, $s_{appr}$, $\cdot$) $q \leq \mathsf{poly}(n)$ times to get a list of RECs [($REC_1$, $ring\_list_1$),...,($REC_q$, $ring\_list_q$)], where each device in L$_{adv}$ is used at least once
5: ($REC_{adv}$, $ring\_list_{adv}$, $txid_{adv}$) $\leftarrow \mathcal{A}(1^n$, $pk_{SS}$, $auth\_keys$), where the $REC_{adv}$ is for a combination of a device state and a smart meter not queried before.
   *Remark.* Adversary can view the RECs in step 4 using CA.Fetch().
6: **Output of the experiment is success**, if CA.Vrfy($txid_{adv}$) is successful
```

Experiment 10 Double-spending RECs

```
1: $(pk_{SS},\ sk_{SS}) \leftarrow$ SS.Init() and SmartMeter.Init($m$)
2: Let $appr\_keys$ = L$_{adv}$ ∪ L$_{other}$, where L$_{adv}$ is the set of smart meters controlled by the adversary, while L$_{other}$ is the set of all other meters. Furthermore, let $end\_users$ = U$_{adv}$ ∪ U$_{other}$, where U$_{adv}$ is the set of users controlled by the adversary
3: $\mathcal{A}$ is given access the oracle SmartMeter.REC($\cdot$, $\cdot$, $s_{appr}$, $\cdot$) (Algorithm 8), where $s_{appr}$ is an approved smart meter state. For smart meters in L$_{adv}$, the adversary can specify the input $in$ and $pk_{owner}$.
4: Every user in $end\_users$ can arbitrary interact with any client application algorithms to trade RECs. During the interaction, any newly produced valid REC is added to $rec\_list$, while any retired REC is moved from $rec\_list$ to $rec\_retired$.
5: The adversary queries the oracleSmartMeter.REC($\cdot$, $\cdot$, $s_{appr}$, $\cdot$) $q \leq \mathsf{poly}(n)$ times to get a list of RECs [($REC_1$, $ring\_list_1$),...,($REC_q$, $ring\_list_q$)], where each device in L$_{adv}$ is used at least once
6: **Output of the experiment is success**, if at any point of time $|rec\_list \cup rec\_retired|$ is bigger then the number of times the oracle SmartMeter.REC is queried
```

Experiment 11 Determining smart meter owner

```
1: Let, $b'$ be a random number between 0 and $q$, where $q \leq \mathsf{poly}(n)$
2: $(pk_{SS}, sk_{SS}) \leftarrow$ SS.Init() and SmartMeter.Init($m$)
3: Let $auth\_keys = \mathrm{L}_{adv} \cup \mathrm{L}_{other}$, where $\mathrm{L}_{adv}$ is the set of smart meters controlled by the
   adversary, while $\mathrm{L}_{other}$ is a set of all other meters. Furthermore, let $end\_users = \mathrm{U}_{adv}$
   $\cup\ \mathrm{U}_{other}$, where $\mathrm{U}_{adv}$ is the set of users controlled by the adversary
4: $\mathcal{A}$ is given access the oracle SmartMeter.REC($\cdot$, $\cdot$, $s_{appr}$, $\cdot$) (Algorithm 8) where $s_{appr}$
   is an approved smart meter state. For smart meters in $\mathrm{L}_{adv}$, the adversary can specify
   the input $in$ and $pk_{owner}$.
5: $(pk_{owner_0}, ..., pk_{owner_q}) \leftarrow \mathcal{A}(pk_{SS}, auth\_keys, \mathrm{U}_{adv})$, where none of the keys is part of
   the set $\mathrm{U}_{adv}$
6: The challenger invokes the oracle SmartMeter.REC($\cdot$, $pk_{owner_{b'}}$, $s_{appr}$, 1)
7: $b \leftarrow \mathcal{A}(pk_{SM}, pk_{SS}, \mathrm{U}_{adv})$, where $\mathrm{pk}_{SM}$ is the public key of the smart meter used to
   create the REC
8: If $b = b'$, then output success
```

Experiment 12 Determining the smart meter

```
1: Let $b'$ be a random number between 0 and $k$
2: $(pk_{SS}, sk_{SS}) \leftarrow SS.Init()$ and $SmartMeter.Init$(m)
3: Let $appr\_dev = \mathrm{L}_{adv} \cup \mathrm{L}_{other}$, where $\mathrm{L}_{adv}$ is the set of smart meters controlled by the
   adversary, while $\mathrm{L}_{other}$ is all other meters. Furthermore, let $end\_users = \mathrm{U}_{adv} \cup \mathrm{U}_{other}$,
   where $\mathrm{U}_{adv}$ is the set of users controlled by the adversary
4: $\mathcal{A}$ is given access the oracle SmartMeter.REC($\cdot$, $\cdot$, $s_{appr}$, $\cdot$) (Algorithm 8) where $s_{appr}$
   is an approved smart meter state. For smart meters in $\mathrm{L}_{adv}$, the adversary can specify
   the input $in$ and $pk_{owner}$.
5: $list\_meters \leftarrow \mathcal{A}(pk_{SS}, auth\_keys, \mathrm{U}_{adv})$, where $list\_meters$ consist of indexes of all
   smart meters in the same ring
6: The challenger invokes the oracle: SmartMeter.REC($\cdot$, $pk_{owner_1}$, $s_{appr}$, $list\_meters[b']$)
7: $b \leftarrow \mathcal{A}(pk_{owner_1}, pk_{SS}, \mathrm{U}_{adv})$
8: If $b = b'$, then output success
```

Appendix B

Cost

B.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 [8] 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 [14], 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.

B.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 [43], 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.

B.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/