IoTLogBlock: Recording Off-line Transactions of Low-Power IoT Devices Using a Blockchain

Christos Profentzas
Chalmers University of Technology
Gothenburg, Sweden
chrpro@chalmers.se

Magnus Almgren
Chalmers University of Technology
Gothenburg, Sweden
magnus.almgren@chalmers.se

Olaf Landsiedel
Kiel University, Germany
Chalmers University of Technology, Sweden
ol@informatik.uni-kiel.de

***Abstract*—[1]For any distributed system, and especially for the Internet of Things, recording interactions between devices is essential. At first glance, blockchains seem to be suitable for storing these interactions, as they allow multiple parties to share a distributed ledger. However, at a closer look, blockchains require heavy computations, large memory capacity, and always-on communication to the cloud; these are three properties that are challenging for IoT devices with limited resources.**

In this paper, we present IoTLogBlock to address these challenges. IoTLogBlock connects resource-constrained IoT devices to the blockchain, and it consists of three building blocks jointly enabling recording transactions: a lightweight contract signing protocol, a blockchain network, and a smart contract. The contract signing protocol allows devices to interact locally to perform transactions, even if no communication to the cloud and the blockchain exists at that moment. At a later time, devices forward the stored transactions to the blockchain, where a smart contract ultimately verifies the transactions.

We evaluate our design on low-power devices and quantify the performance in terms of memory, computation, and energy consumption. Our results show that a constrained device can create and sign a transaction within 3 s on average. Finally, we expose the devices to network scenarios with edge connections ranging from 10 s to over 2 h.

***Index Terms*—Internet of Things, Blockchain, Smart contracts**

I. Introduction

Blockchains store immutable records of transactions in a so-called distributed ledger. Transactions are generated by parties which do not necessarily trust each other. As records in a blockchain are immutable and can be verified by all parties involved, a blockchain creates the required trust between the involved parties whether a particular transaction is part of the blockchain or not. This makes the blockchain an ideal candidate for storing records of transactions produced by the plethora of connected devices in the Internet of Things (IoT). These devices interact frequently and produce records of interactions which their applications want to store, while two IoT devices commonly do not trust each other, as they are, for example, owned or operated by different entities.

978-1-5386-5541-2/18/$31.00 ©2018 European Union

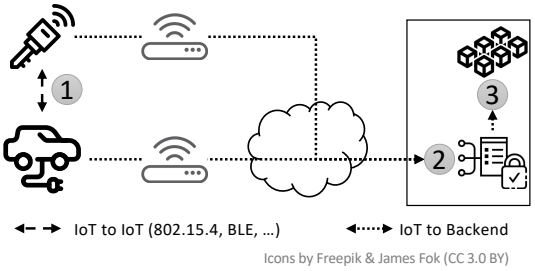


Fig. 1: Motivating scenario: car rentals where (1) a smart-key and a smart-car perform off-line transactions using a contract signing protocol over low-power wireless technologies. Later, the IoT devices independently forward their transactions via an edge device to the blockchain where (2) a smart contract validates and then (3) stores the transactions immutably.

We argue that the following key challenges are overlooked in the context of IoT and blockchains. To perform transactions, blockchain clients need to be (a) connected to the blockchain, (b) often store large parts of the blockchain, and (c) perform heavy cryptographic operations. These are three requirements that today's resource-constrained IoT devices can hardly fulfill.

Recent works demonstrate the use of blockchains for data provenance [12] [16], but there are two significant issues that existing architectures cannot address. First, the integration of resource-constrained devices is limited due to the communication requirements and fixed connection points. For example, in many designs [18] [2] devices need to seek active communication with the network for each transaction, which demands considerable energy consumption. Moreover, in the case of mobile IoT devices such a design demands for complete network coverage through, for example, cellular technologies such as 4G, which in turn further increases cost and energy consumption. Similarly, LPWAN technologies such as LoRa and SigFox do not provide the required bandwidth required by the cryptographic operations of a blockchain. Second, the proposed architectures cannot ensure non-repudiation of the transactions given that stakeholders often have conflicting interests. We argue that it is essential that when two IoT devices create an off-line transaction, the stakeholders cannot later deny their participation therein. Data tampering and device impersonation attacks [1] raise concerns regarding the

secure collection of sensor data.

With this in mind, we propose IoTLogBlock with the following properties. IoTLogBlock combines an (1) **optimistic contract signing protocol** with a (2) **smart contract**, (3) deployed on a **blockchain**, as shown in the motivating car rental scenario in Figure 1. The contract signing protocol allows two or more nodes to sign an off-line transaction mutually and achieve non-repudiation [4]. We assume only intermittent network access (due to power conservation or infrastructure issues) and the IoT devices can store the off-line transactions in local memory while waiting for an edge connection. An edge device forwards the transactions to the blockchain network for validation using a smart contract. The smart contract is similar to a stored procedure of regular databases, and it is triggered upon events sent by the network.

Overall, the combination of a contract signing protocol with the smart contract achieves non-repudiation and enables the integration of resource-constrained IoT devices to a cloud-based blockchain. As a result, IoTLogBlock allows IoT devices, which are connected to the Internet infrequently, to employ blockchains. We make the following contributions.

- We design and implement IoTLogBlock, an open-source architecture[2] for recording IoT transactions using blockchain. IoTLogBlock achieves non-repudiation, avoids dependence on fixed network infrastructures, and ensures a low-power radio duty-cycle.
- We propose a practical solution to implement a contract signing protocol on IoT devices.
- We design and implement a smart contract to act as the online validator for the off-line transactions.
- We quantify the performance of IoTLogBlock in terms of computation, delay, memory, and energy consumption. Notably, we find that low-power devices (TI-CC2538) can create a transaction in 3 s. We further calculate the latency of IoTLogBlock in different scenarios, with an edge connection from 10 s to over 2 h.
- We finally provide a discussion of implementation challenges and trade-offs regarding the integration of resource-constrained devices with a cloud-based blockchain.

Organization. The paper is organized as follows. In Section II, we provide the necessary background regarding our approach. In Section III, we describe a motivating example and our adversary model. In Section IV, we provide the system design and highlight security aspects. We evaluate our results in Section V, which is followed by a discussion and a description of related work before concluding the paper.

II. Overview and Background

We provide the background on the building blocks of IoTLogBlock: contract signing protocols, smart contracts, and blockchain.

[2]https://github.com/iot-chalmers/IoTLogBlock.git

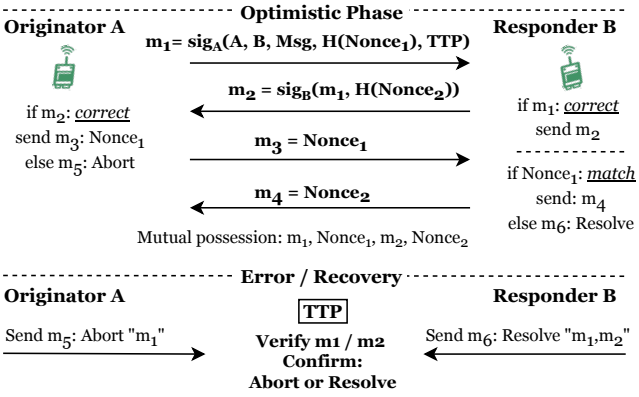


Fig. 2: Signature exchange in the Asokan-Shoup-Waidner protocol. In the optimistic phase, two nodes agree to sign a contract. At the end of the protocol, both participants have a copy of the mutually signed agreement. In the error/recovery phase, a node tries to abort or resolve a previous round of the protocol using a Trusted Third Party (TTP).

A. Contract Signing Protocols

Contract signing protocols [19] allow two or more parties that do not trust each other to sign a pre-agreed text. These protocols provide fairness, timeliness, and sometimes also an abuse-free property [21]. Two parties (Alice and Bob) exchange their signatures in a fair-way [3], where Alice can obtain Bob's signature only if Bob can obtain Alice's signature and vice-versa. Timeliness [21] means that a participant is not able to make the other wait for an indefinite amount of time. Finally, with the abuse-free property [21], a participant cannot determine the outcome of the protocol.

We can group the contract signing protocols in three categories: protocols with an online Trusted Third Party (TTP) [10], protocols without TTP (e.g., time commitments [5]), and protocols with an off-line TTP (e.g., optimistic [3]). As the first two are computationally demanding, they are not suited for resource-constrained IoT devices. Hence, we focus on the third group of optimistic contract signing protocols. Specifically, we build on the **Asokan-Shoup-Waidner (ASW)** [3] protocol, which provides fairness and timeliness, but it is not abuse-free [21]. The ASW protocol allows two parties to exchange signatures as shown in Figure 2, without actively invoking the trusted third party. However, an online TTP is available to resolve issues if one of the parties does not follow the protocol (crashes or behaves maliciously). There are three sub-protocols involved in the process. The first is to complete an optimistic exchange of signatures (without the involvement of the TTP). The second is to allow a participant to abort a signature exchange (abort sub-protocol), and the third is to ask the TTP to resolve an incomplete signature exchange (resolve sub-protocol).

B. Smart Contracts

A smart contract is a digital representation of an agreement between two or more parties, written as an event-based pro-

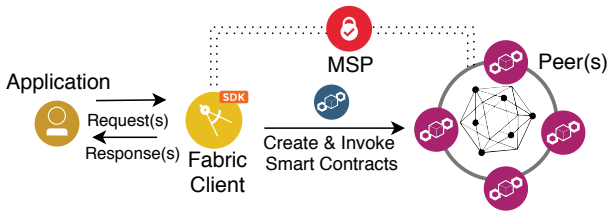


Fig. 3: The Hyperledger Fabric consists of: 1) the Membership Service Provider (MSP), which issues the access list of participants, 2) the private network of peer(s) that maintain the blockchain, and 3) the Fabric client which provides the application interface.

gram and stored immutably on the blockchain. The blockchain assigns a public key to each smart contract, and applications can use these keys to send a transaction to the blockchain, which triggers the execution of a particular smart contract. Each smart contract has its internal state to keep events, and the execution may add a new record to the blockchain. Finally, as described in [9], a smart contract can act as an escrow to resolve (dis)agreements between two or even multiple parties. In IoTLogBlock, we utilize this capability and employ the smart contract as off-line TTP to resolve issues of the contract signing protocol. As the code and transactions of the smart contract can be verified by everyone involved, it inherently gains trust as a third party.

C. Blockchains and Hyperledger

A blockchain is essentially a distributed ledger: multiple nodes replicate blocks of records as an append-only data structure. Each block has multiple records which are linked together into a chain using the cryptographic hash value of the previous block. Any change to an older record of a block inherently invalidates all recent blocks and their records. The ledger is replicated over multiple nodes in the network to provide fault tolerance, and the replicas reach a consensus regarding the order in which they add blocks to the chain.

Conceptually, we can categorize blockchains as public or private. The former is the initial design of Bitcoin [15], where any node can join the network and make commits. The latter enforces access control lists to determine who can participate in the consensus process and make commits.

An example of a private blockchain is Hyperledger Fabric [2], which is a collaboration between Linux Foundation and IBM to provide an open-source distributed ledger. The platform has three main components (see Figure 3). First, a *Membership Service Provider (MSP)* manages the identities on the permission-based blockchain [2]. The MSP authenticates and authorizes peers that can make changes to the blockchain. Second, multiple authenticated *peers* participate in the network to maintain the ledger and run the smart contracts. Finally, the *Fabric client(s)* is an authenticated node that allows the blockchain network to interact with the outside world.

III. Application Scenario and Adversary Model

This section discusses a motivating example and outlines our adversary model for IoTLogBlock.

A. Application Scenario

As a motivating scenario, we consider a car rental service with regular customers [23]. The rental company provides cars at a service point or distributed throughout the city's parking lots. The customer signs up and receives a smart token (key) that will open a car at any time. The company wants to ensure that they know what customer used a car during which period, to charge the customer for the use and also be able to charge for potential abuse. Customers, on the other hand, want a well-functioning car, and they are only charged for their specific use based on their agreement with the company.

In the general case, the stakeholders are known, but they have conflicting interests. For example, the driver would like to pay as little as possible while the car rental service would like the highest fee possible. It is also likely that situations (accidents) will occur where it is essential to keep an immutable ledger of the transactions between the stakeholders. The ledger can then be used afterwards to analyze misbehavior or malicious activities. Finally, we argue that even though a modern car can afford LTE or 4G connections for recording its activities, it will still benefit from our design as cellular coverage is not always available, especially in rural areas.

B. Adversary Model

We assume three potential threats as part of the adversary model. First, an adversary may try to exploit insufficient data auditing of a transaction (e.g., an IoT node skips to report a transaction), where the log does not capture enough data to determine the event of a transaction. This is a repudiation threat where, for example, a car or a key may create an off-line transaction in such a way that there is no proof that an online validator can verify the event of the off-line transaction. Second, we consider an adversary that may try to exploit weak audit mechanisms, including attempts to destroy the audit mechanism (e.g., blockchain) of the transactions. This is a scenario where a customer has used a car and tries to remove the events to claim the money back or not pay in the first place. Third, an adversary may try to include data from an unknown source or untrusted device. As a result, the system log may include data from unknown devices. Here the register entry leads to an unknown customer, and the log system is unable to connect a transaction with a real customer.

IV. System Design

A. Design Overview

There are three key building blocks to IoTLogBlock (Figure 1): (1) the interaction between two IoT devices using a contract signing protocol, (2) the verification of the resulting transaction between the devices using a smart contract, and (3) the final immutable storage of the result on the blockchain for all participants (auditing, finding misbehaving nodes).

To prototype IoTLogBlock, we also used several existing technologies to tie it together. Below we describe the full system and mark where our contribution lies with a (*) and the corresponding number in Figure 1. These steps will then be further elaborated below.

1) Node discovery and node interaction (1):* When two low-power wireless devices shall interact, for example, when the smart key attempts to open the car via wireless communication, the two devices first have to discover each other. In our prototype, the nodes are synchronized with an established local and autonomous communication protocol, TSCH [7], and they do not depend on any central entity. Next, they have to perform a wireless transaction, where at the end the car grants access (or not) to the smart key. Here our contribution lies in the combination of established low-power wireless protocols, namely TSCH and 802.15.4 [14], with an optimistic two-party contract signing protocol, which is further described in Section IV-C.

2) Interaction between nodes and the cloud: Sporadically, an IoT node will come into range of an edge device with cloud connectivity. In this case, the IoT node utilizes the edge device as a relay to transfer the off-line transactions into the cloud for verification and storage. We assume that the edge device has robust capabilities, and it can establish secure connections (e.g., SSL) with a node of the blockchain network (see Figure 1). This step builds on established mechanisms and is not further described in the design of IoTLogBlock.

3) Verification and storage in the cloud (2, 3*):* Through the edge device, all transactions reach the cloud, where an authenticated node/peer runs a smart contract to verify each transaction, detecting misbehaving nodes, and finally to store each transaction. The smart contract validation is described in Section IV-D and the final step, permanent storage, is described in Section IV-E. We conclude in Section IV-F with the security analysis of potentially misbehaving nodes.

B. Setup: Deploying new Devices

When we add a new device to the system, i.e., we deploy a new smart key, this device creates a private/public key pair. Each device will use its private key to sign its transaction. The public key of each device is stored in the blockchain and is used to authenticate clients later and verify their transactions. Moreover, we also employ a smart contract to manage the list of registered devices.

C. Creating and Signing Transactions

Each device maintains a sequence number that uniquely identifies each of its transactions, by simply incrementing a counter for each new transaction. The sequence number is later used for verification by the smart contract. In IoTLogBlock, a transaction is created as follows: Once two devices have discovered each other, they exchange essential information such as their IDs and the transaction upon which they want to agree. Next, they sign and exchange a message containing their IDs, current local sequence numbers, and the transaction itself, following the optimistic two-party contract signing protocol (see Section II). If the transaction completes, each device has a transaction signed with the private key of both parties that states the IDs of the participants, i.e., their public keys, their respective sequence numbers, and the transaction content itself. Both IoT nodes individually upload this information into the smart contract via an edge device once they come within range. As transactions are signed, manipulations of these, for example, at the edge device are not possible.

Algorithm 1: Smart Contract - Validator

```
/* Error checking, such as checking for key
   revocation, is omitted for brevity.    */
Data: Transaction
validTransaction = True;
forall NodeID in Transaction do
    Pubkey = RegisteredDevices(NodeID);
    Status = ECDSA.Verify(NodeID, PubKey, Transaction);
    if Status is not valid then
        report NodeID;
        validTransaction = False ;
    end
end
if validTransaction is True then
    if Pending_resolve_request then
        run Resolve-SubProtocol(Transaction);
    else if Pending_abort_request then
        run Abort-SubProtocol(Transaction) ;
    else
        record Transaction;
end
```

D. Validation with the Smart Contract

IoTLogBlock deploys a smart contract in the blockchain to act as a validator for the received transactions from IoT devices. Algorithm 1 shows an abstract pseudo-code version of the validator. The algorithm starts by checking the validity of the devices' signatures and reports any misbehavior. As a result, transactions will only be committed when signed by registered devices. Finally, the smart contract resolves any conflict by using the sub-protocols of the ASW contract signing protocol [3], as we discuss next.

Abort sub-protocol: If a node crashes, disconnects or misbehaves during the execution of the first half of the contract signing protocol, the abort sub-protocol is invoked. Thus, if there is a timeout or the signature verification fails, the participant sends an abort request to the smart contract by signing a new message which includes the original message_1 of the protocol (see Figure 2). If the smart contract has not received a request to resolve an uncommitted transaction (see the resolve sub-protocol below), it confirms the request and registers the abort message to the blockchain.

Resolve sub-protocol: If the two nodes have already exchanged the first two messages of the protocol, before one of the nodes crashes or disconnects, the resolve sub-protocol is invoked. As the nodes have made some progress, the signature exchange can be resolved. The node sends a resolve request containing the first two messages (m_1,m_2) of the protocol (see

Figure 2) to the smart contract. The resolve sub-protocol is also invoked when a node tries to abort even if it has already sent message_2.

E. Register Transactions in the Blockchain

After the transactions are validated by the smart contract, they are appended to the blockchain. By its very nature, this leads to a distributed ledger open to all stakeholders. Transactions stored in the blockchain are immutable and cannot be changed without invalidating previous transactions, as explained in Section II.

F. Security Analysis

In IoTLogBlock, we focus on the following misbehaviors:

1) Bypass sequence numbers: A node could skip or reuse sequence numbers to hide its misbehavior. However, when a node uploads its transactions, the smart contract detects any duplication or gaps in the sequence numbers. Since the ledger in IoTLogBlock is available to all stakeholders, it is trivial to detect this behavior.

2) Hide transactions: A node could execute a transaction with another node and not report it or even try to delete it. By using the contract signing protocol, both parties have a copy of a transaction. If the other node behaves correctly, it will upload all the transactions, and the smart contract detects the misbehaving node. Thus, a transaction will only go unnoticed if both involved parties are misbehaving or both fail to establish an edge connection. We argue that it is very uncommon for two parties to collude in this way, as they commonly would not share the same connection or the same goals. For example, while the smart key, or more precisely, the corresponding users might be interested in driving a car for free, the smart car has precisely the opposite interest. Finally, the blockchain ensures the immutability of its records, and a node cannot delete already stored transactions.

3) Unregistered device: An untrusted device can try to overcome the contract signing protocol, and send unsigned or invalid transactions to the blockchain. However, this is taken care of by the protocol itself and the trusted party (here the smart contract), which accepts transactions only by registered devices. In case of a transaction violation, a node can then invoke the sub-protocols to abort or resolve a transaction. Since a node reports all the abort attempts to the blockchain, it is trivial to detect a node that abuses the protocol and block it in the future.

V. Experimental Evaluation

IoTLogBlock includes low-power IoT devices, edge computing, and a cloud service (Hyperledger). In our evaluation, we focus on the IoT devices; the edge device plays a secondary role in our architecture, and Hyperledger Fabric has previously been evaluated in detail [2].

Goals. With this section, we answer the following questions: a) is IoTLogBlock technically feasible on low-power IoT devices? b) what is the overhead in terms of computation, memory, and energy consumption of the contract signing protocol? c) what is the overall performance of IoTLogBlock?

Outline. We divide the evaluation into three parts. First, we discuss our implementation of IoTLogBlock and our choices in terms of IoT hardware, software stack, edge computing, and blockchain. Second, we evaluate our implementation of IoTLogBlock in terms of run-time performance, energy consumption, and memory consumption. Third, we present our overall system performance including 1) the time it takes for a sensor node to register a record on the blockchain, and 2) the reliability and limitations of IoTLogBlock when creating periodic off-line transactions.

A. Implementation and Experimental Setup

IoT Software Stack. We implement the contract signing protocol in C as an application for Contiki-NG [11]. Our implementation employs the Elliptic Curve Digital Signature Algorithm (ECDSA) using the recommended NIST-P 256-bit curve. All the ECDSA operations are performed by the cryptographic hardware support of our target platform. For communication between IoT nodes and to edge devices, we use the TSCH protocol [7] readily available in Contiki-NG. TSCH employs radio duty cycling and provides us with a robust and energy efficient communication substrate, tailored to resource-constrained and potentially battery-driven IoT devices. We would like to note, that the design of IoTLogBlock is not bound to a particular low-power wireless protocol and would also support, for example, BLE.

IoT Hardware and Platform. We target sensor nodes equipped with cryptographic hardware support such as the TI-CC2538 SoC. The SoC contains a 32-bit ARM Cortex M3 CPU at 32 MHz, 32 KB of RAM, and 512 KB of ROM. Moreover, the SoC features a cryptographic engine at 250 MHz and a 802.15.4 radio transceiver. This SoC is common in the community for resource-constrained IoT devices and readily supported by numerous operating systems. In particular, we use the OpenMote platform [25], which combines this SoC with further sensors and a battery.

Edge Devices and Blockchain. In IoTLogBlock, the edge devices receive transactions from the IoT device and upload them to the smart contract of the Hyperledger instance via the Fabric API. For these connections, we use Python and Node.js scripts while we implement our smart contract in GO. Our edge devices run a standard Linux. For simplicity, we deploy the edge devices and the Hyperledger Fabric network using Docker containers on a desktop machine, which features an Intel Core i5 at 2.3 GHz and 16 GB of RAM.

Source Code. We provide our implementation of IoTLogBlock as open-source code in a public repository.[3]

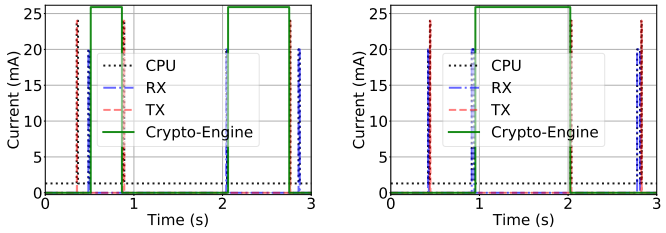
B. Evaluation of IoTLogBlock on the IoT Node

As the first step, we evaluate our contract signing protocol in terms of memory, computation, and energy consumption. For the memory footprint we use the tools **arm-none-eabi-readelf** and **arm-none-eabi-size** on the binary files. For the

[3]https://github.com/iot-chalmers/IoTLogBlock.git

Component	RAM Bytes	RAM Percent	ROM Bytes	ROM Percent
Contiki-NG OS	11,394	37%	40,527	10%
Cryptographic library	1,520	4%	10,775	1%
Application	4,201	13%	7,993	1%
Total footprint	17,115	54%	59,295	12%
Available memory	14,885	46%	452,705	88%

TABLE I: Memory Footprint of the IoT application (considering max size) on CC2538, which facilitates 32KB of RAM and 512 KB of ROM.



(a) Originator: This node starts the contract signing protocol.

(b) Responder: This node responds accordingly.

Fig. 4: The electric current drawn by a complete round of the contract signing protocol.

computation and energy performance, we rely on **Contiki's Energest** module with a 30 µs resolution timer to measure the duration of each task, log the power modes of the system, and derive the energy consumption. For the values of the electric current (see Table III) we rely on the CC2538 data sheet [22] and a previous evaluation [20].

Memory Footprint. Table I presents the static memory allocation of our implementation, divided into three parts: (a) the operating system (Contiki-NG) with the network stack, (b) the cryptographic library for CC2538, and (c) the implementation of the contract signing protocol (Application). The operating system has a significant impact and consumes 37% of the available RAM. The application combined with the cryptographic library consumes 17% of the available RAM. In total, we consume 54% of the available memory. This leaves 46% of the available RAM to be dynamically used at run-time by the stack and for the temporary storage of off-line transactions, which we evaluate later. Finally, the whole program consumes only 12% of the ROM.

Performance. Next, we evaluate the performance of the cryptographic functions. All cryptographic operations are performed by the cryptographic engine running at 250 MHz, and in Table II we present the performance of each task. The average time to complete all cryptographic functions of the contract signing protocols in IoTLogBlock is 2.1 s. The most burdensome task – concerning the performance in time – is the signature verification, which takes 715 ms per node. Our protocol uses the SHA256 hash-function multiple times during a single transaction, but the impact is low. The cryptographic operations combined with the latency of the wireless TSCH protocol stack (see Figure 4) gives us a total of 3 s per transaction. We argue that this overhead, while significant, is feasible for many applications. For the application scenario of the car rentals, it means a user will need three seconds to unlock a car, which we consider practical.

Energy Consumption. Focusing on energy efficiency, we assume that nodes have discovered each other, which is a functionality provided by the TSCH protocol known as TSCH synchronization. This discovery happens quickly, and it has been evaluated previously [7]. In Figure 4, we depict the flow of the electric current (in mA) during a complete round of the contract signing protocol, including wireless communication and the use of the cryptographic engine.

In this example, a node (originator) initiates a transaction with a second node (responder). The protocol begins with the originator and responder exchanging a hello message to indicate their availability, visible at time 0.5 s in Figure 4. Next, the originator starts the protocol by signing the first message, which takes 0.3 s (see Table II). The responder is waiting in low-power mode for the signature of the originator, which it receives at time 0.9 s. Immediately, the node starts the verification and signs the message (in case of a correct signature). This process takes 1 s, while the originator waits in low-power mode. At time 2.1 s, the originator receives the signature from the responder and starts the verification, which takes 0.7 s. Next, at 2.8 s the originator (in case of a correct signature) replies with its secret nonce value, followed by a reply of the responder.

In Table III, we report the total energy consumption (in mJ) of the protocol, which also includes the wireless protocol stack. We notice that the cryptographic engine contributes significantly (58%) to the total energy consumption. The wireless communication via the TSCH protocol contributes 35%. In total, a transaction consumes 98.5 mJ. We observe that the cryptographic computations and radio communication are the two main drivers of energy consumption in IoTLogBlock.

By default, OpenMote is powered by two standard AA alkaline battery of 2500 mAh. Assuming a self-discharge of 20%, we can utilize 80% of the capacity, i.e., 2000 mAh [24]. As a result, we can expect 10,000 Joules of energy from the cells, which allows IoTLogBlock to perform roughly in the order of 100,000 transactions. While this is merely an estimate, we argue that this order of magnitude of transactions is practical for a wide range of application scenarios, including ours of a battery-powered smart key.

Reflecting on previous work [20] [13], we conclude that, while the cryptographic module consumes the largest share of the energy, the design of IoTLogBlock would not be feasible without it. Implementing the cryptographic functions on the main CPU and without the module would significantly extend the computation time. As a result, the cryptographic handshake would get impractically long and consume even more energy.

C. System Performance of IoTLogBlock.

We evaluate the overall system in terms of a) local storage capacity b) delay of registering a transaction to the blockchain, and c) the reliability of nodes when creating off-line transactions. In the experiments, we utilize the remaining available RAM for storing off-line transactions while waiting for an

Function type	Time
ECDSA-Sign on originator	350 ms
ECDSA-Verify on responder	715 ms
ECDSA-Sign on responder	350 ms
ECDSA-Verify on originator	715 ms
SHA256-Hash function	1 ms
Total time	2131 ms

TABLE II: Performance of cryptographic functions using the CC2538 cryptographic engine running at 250 MHz.

State	Time [ms]	Current [mA]	Energy [mJ]
Cryptographic Engine	1,065	25.9	57.9
TX	32	24	1.6
RX	836	20	35.1
CPU @ 32 MHz	38	13	1.1
CPU @ LPM2	1,029	1.3	2.8
Total	3,000		98.5

TABLE III: Derived energy consumption of running the contract signing protocol on the CC2538 SoC given a supply voltage of 2.1 V. Note we configure Contiki-NG to use the low-power mode 2 (LPM2) [22].

edge device to come into range. We collect our results after sending at least 200 packets, and we report the standard deviation when it is not negligible.

We provide edge connectivity to the IoT devices within a period of 1, 10, 100, 1,000, and 10,000 s. These experiments expose the devices to different types of scenarios, ranging from a dense topology with access points to a much more sparse network topology with little edge connectivity. As evaluated previously, a transaction takes 3 s. Thus, in our evaluation, two nodes generate transactions every 3, 30, and 300 s (see Figure 5).

Transaction Storage Capacity. We now present how many transactions a device can store until it connects again to an edge device. This is essential, as a device has to drop previous transactions or refuse new ones once its temporary storage is full. Figure 5a shows that with edge connections in the same order of magnitude as the transaction generation rate, not much memory is used. As the edge connections get more sparse compared to the transaction generation rate, the devices start to utilize more of the storage capacity, and they reach full utilization of their memory when we provide edge connections in terms of hours (10,000 s). After this point, we need to start dropping records or refuse new ones, which affects the reliability (or availability) of the system (see Figure 5c and discussion below).

Register Delay. In Figure 5b, we present the total number of seconds (delay) that it takes to create, sign, and register a transaction to the blockchain. We can see a correlation between the delay and the periodicity of the edge connection.

System Reliability. We have counted the total amount of attempts a device tried to create a transaction. We quantify the reliability IoTLogBlock as the percentage of the successfully stored transactions on the Hypeledger, which is presented in Figure 5c. We notice that the reliability highly depends on two factors: the periodicity of the edge connection, and the transaction generation rate.

VI. Discussion and Limitations

In this section, we discuss the benefits, limitations, and remaining challenges regarding our approach.

Resource Constraints. The performance of the contract signing protocol (a full-round) averages 3 s. We argue that this demonstrates IoTLogBlock to be functional in practice. Moreover, the protocol utilizes radio and CPU duty cycling for low-power consumption (see Table III). However, the energy consumption of the cryptographic engine demonstrates that security comes with a price. We note that the cryptographic operations sign and verify are essential to any transaction flow of the Blockchain. Moreover, they must be performed on the IoT devices themselves and cannot be delegated to the potentially untrusted edge or cloud services. Thus, we argue that this energy cost is fundamental to security and would also be required in any other design integrating IoT devices and blockchain.

System Latency. The experiments confirm the feasibility of connecting resource-constrained IoT devices with blockchain technologies. Under fair conditions of around 5-30 min transient connectivity and a fair number of transactions, we show that the memory of small IoT devices is sufficient to store the transactions. For further robustness, we can store them additionally in flash memory. Our result shows that the main bottleneck in terms of memory consumption and end-to-end latency lies on the periodicity of the edge connection.

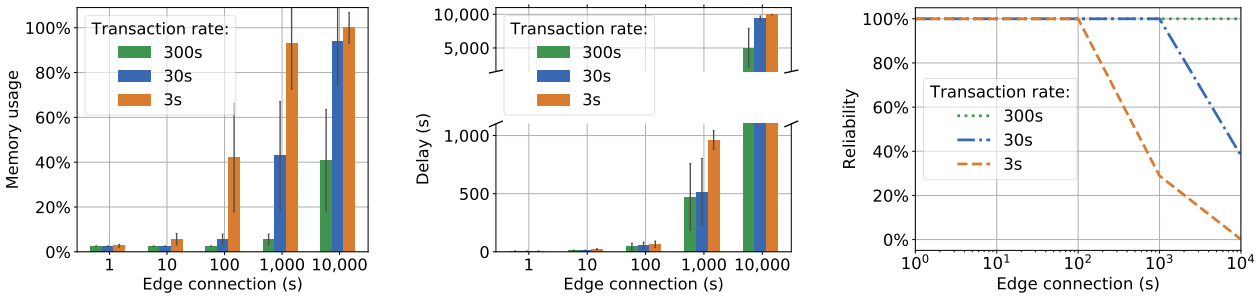
Security Aspects. We use a fair authentication scheme with a smart contract, which validates each device and transaction. However, each device will need to use a white-list of authenticated nodes, which will increase the local storage needs. We recognize a trade-off between the off-line transaction storage and the number of devices stored in the white-list.

Limitations. A general limitation of our study is the privacy of stored data in a cloud-based blockchain. A blockchain is by nature publicly available for all the stakeholders.

VII. Related Work

Fair exchange. Similar to our work, FairSwap [8] also proposes the use of smart-contracts for fair exchange, avoiding costly solutions like zero-knowledge proofs. However, the protocol has not been tested and evaluated on any IoT devices.

Blockchains and IoT. Several architectures [12] [16] have been proposed to achieve data provenance using blockchain technology. However, they do not integrate IoT devices, and they do not apply to resource-constrained environments. Nonetheless, there are many benefits [6] of integrating IoT with blockchains. There are approaches to integrate IoT devices with blockchains such as AGasP [9] and Edgechain [18]. The existing architectures are limited in two ways: a) there is a lack of non-repudiation of the transactions created by the IoT nodes, and b) they do not take into account the constraints faced by low-power IoT devices. In contrast, IoTLogBlock focuses on data provenance, achieving non-repudiation, and using low-power IoT devices.



(a) Memory usage for storing off-line transactions in the local memory of CC2538. The black vertical line shows the standard deviation.

(b) The delay between creating an off-line transaction from IoT devices and registering it to the Hyperledger Fabric. The black vertical line shows the standard deviation.

(c) We count all the transaction attempts, and we quantify the system reliability as the percentage of the successfully created and stored transactions to Hyperledger. When the device memory is full, the devices refuse to create further transactions.

Fig. 5: We create transactions every 3, 30, and 300 s. The IoT device offloads its transactions with periodic edge connection every 1, 10, 100, 1,000, and 10,000 s.

Cryptographic Capabilities on IoT Devices. Previous studies show the affordability of cryptographic operations on small sensor devices [17] [13]. IoTLogBlock differs from previous work in that we use blockchain to record transactions and a contract signing protocol to achieve non-repudiation.

VIII. Conclusion

In this paper, we propose IoTLogBlock, an open-source architecture allowing low-power devices to use a cloud-based ledger to record transactions between devices. We argue that using a blockchain directly is not an option for hardware-constrained IoT devices, as it implicitly assumes the devices are capable of heavy computations and having large memory capacity and always-on communication to the cloud. IoTLogBlock combines an optimistic contract signing protocol, a private-based blockchain, and a smart contract, to cope with these three challenges. The IoT devices create off-line transactions, which later are verified by the smart contract. The blockchain provides to the stakeholders an immutable ledger of all the collected transactions.

We evaluated IoTLogBlock, especially in regards to the extent it allows IoT devices with transient connectivity to create and register transactions in a cloud-based blockchain. We quantify the cost of IoTLogBlock in terms of computation, delay, memory, and energy consumption. It takes on average 3 s for IoT devices to mutual sign and exchange a transaction. Moreover, IoTLogBlock supports intermittent connectivity ranging from 10 s to over 2 h.

IX. Acknowledgments

This work was supported by the Swedish Research Council (VR) through the project "AgreeOnIT", the Swedish Civil Contingencies Agency (MSB) through the projects "RICS" and "RIOT", and the Vinnova-funded project "KIDSAM".

References

[1] S. F. Aghili, M. Ashouri-Talouki, and H. Mala, "DoS, impersonation and de-synchronization attacks against an ultra-lightweight RFID mutual authentication protocol for IoT," *The Journal of Supercomputing, Springer*, 2018.

[2] E. Androulaki, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *ACM European Conference on Computer Systems (EuroSys)*, 2018.

[3] N. Asokan, V. Shoup, and M. Waidner, "Asynchronous protocols for optimistic fair exchange," in *IEEE Symposium on Security and Privacy*, 1998.

[4] G. Ateniese, "Efficient verifiable encryption (and fair exchange) of digital signatures," in *ACM Conference on Computer and Communications Security (CCS)*, 1999.

[5] D. Boneh and M. Naor, "Timed commitments," in *Advances in Cryptology (CRYPTO)*. Springer, 2000.

[6] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," *IEEE Access*, vol. 4, 2016.

[7] S. Duquennoy, A. Elsts *et al.*, "TSCH and 6tisch for Contiki: Challenges, Design and Evaluation," in *IEEE Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2017.

[8] S. Dziembowski, L. Eckey, and S. Faust, "Fairswap: How to fairly exchange digital goods," in *ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[9] Y. Hanada, L. Hsiao, and P. Levis, "Smart Contracts for Machine-to-Machine Communication," in *IEEE Conference on Internet of Things and Intelligence System (IOTAIS)*, 2018.

[10] Jianying Zhou and D. Gollman, "A fair non-repudiation protocol," in *IEEE Symposium on Security and Privacy*, 1996.

[11] A. Kurniawan, *Practical Contiki-NG Programming for Wireless Sensor Networks*. Apress, 2018.

[12] X. Liang, S. Shetty *et al.*, "ProvChain: A Blockchain-Based Data Provenance Architecture in Cloud Environment with Enhanced Privacy and Availability," in *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[13] A. Liu and P. Ning, "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks," in *IEEE Conference on Information Processing in Sensor Networks (IPSN)*, 2008.

[14] G. Montenegro, N. Kushalnagar *et al.*, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," RFC Editor, Tech. Rep. RFC4944, 2007.

[15] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[16] R. Neisse, G. Steri, and I. Nai-Fovino, "A blockchain-based approach for data accountability and provenance tracking," in *ACM Conference on Availability, Reliability and Security (ARES)*, 2017.

[17] A. L. M. Neto, H. K. Patil *et al.*, "AoT: Authentication and Access Control for the Entire IoT Device Life-Cycle," in *ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2016.

[18] J. Pan, J. Wang *et al.*, "EdgeChain: An Edge-IoT Framework and Prototype Based on Blockchain and Smart Contracts," *arXiv:1806.06185 [cs]*, 2018.

[19] B. Schneier, "Advanced Protocols," in *Applied Cryptography, Second Edition*. John Wiley & Sons, Inc., 2015.

[20] H. Shafagh, A. Hithnawi *et al.*, "Talos: Encrypted query processing for the internet of things," in *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2015.

[21] V. Shmatikov and J. C. Mitchell, "Finite-state analysis of two contract signing protocols," *Theoretical Computer Science*, 2002.
[22] Texas Instruments, "CC2538 System-on-Chip for 2.4-GHz IEEE 802.15.4," www.ti.com.cn/cn/lit/ug/swru319c/swru319c.pdf, 2013.
[23] J. Tomi and W. Kempton, "Using fleets of electric-drive vehicles for grid support," *Journal of Power Sources*, vol. 168, no. 2, 2007.
[24] S. Tozlu and M. Senel, "Battery lifetime performance of wi-fi enabled sensors," in *IEEE Consumer Communications and Networking Conference (CCNC)*, 2012.
[25] X. Vilajosana, P. Tuset *et al.*, "OpenMote: Open-Source Prototyping Platform for the Industrial IoT," in *Ad Hoc Networks*. Springer, 2015.