Edge2BC: a Practical Approach for Edge-to-Blockchain IoT Transactions

Eder J. Scheid, Andreas Knecht, Tim Strasser, Christian Killer, Muriel Franco, Bruno Rodrigues, Burkhard Stiller

Communication Systems Group CSG, Department of Informatics IfI, University of Zürich UZH

Binzmühlestrasse 14, CH—8050 Zürich, Switzerland

E-mail: [scheid,killer,franco,rodrigues,stiller]@ifi.uzh.ch [andreas.knecht2,tim.strasser]@uzh.ch

Abstract—The integration of Blockchains (BC) and Internet-of-Things (IoT) brings several benefits, such as data immutability and traceability, to a myriad of use-cases. The use of intermediary nodes, *e.g.*, an edge node, provides benefits, but opens the possibility for attacks, especially when considering nodes handling the signing of transactions for devices. If the private key of a device is leaked, the data coming from that device cannot be trusted anymore, thus, rendering transactions signed with that private key unreliable.

To tackle this concern, this paper presents Edge2BC, a practical approach to sign automatically BC transactions in an IoT device without revealing its private key to intermediary nodes, thus, increasing trust in the edge-to-BC data transit. The device relies on an edge node to communicate with a BC interoperability API to retrieve dynamic information (*e.g.*, transaction nonce and gas price) and to send the signed raw transaction to the correct BC node. Power measurements (*i.e.*, required current for different method steps) were performed in the prototype implemented to verify Edge2BC's feasibility. Such evaluations indicate that it is possible to have such a method in place, but the high power consumption required during the signing phase should be taken into consideration when deploying Edge2BC in a dedicated scenario.

I. INTRODUCTION

Blockchains (BC) are applied in several novel Internet-of-Things (IoT) applications [8] due to their immutability, high availability, and secure database in distributed communication systems. One field that rationally combines BC and IoT is the supply-chain tracking field of medicines (*i.e.*, cold chain), in which the BC provides a trusted information sharing environment (*e.g.*, for temperature measurements) for (a) unknown stakeholders [37] and (b) product traceability for the end consumer [3], [36]. Tracking the temperature of drugs is crucial, since, as can be seen in the case of Pfizer's Covid-19 vaccine doses, they have to be maintained at -70 °C to arrive at hospitals and clinics in ideal conditions [29].

One crucial aspect in the integration of BC and IoT is the management of private keys and the signing of transactions. This is due to the fact that once a BC transaction is signed using the private key of the BC address it cannot be altered without the possession of the corresponding private key. Thus, by relying on a BC, the signature provides (i) non-repudiation, (ii) authentication, and (iii) integrity to the transaction originating from the device that holds the private key. However, if

the device's private key is not secured or it is leaked [9], a malicious actor can generate or modify transactions, rendering the cold chain measurements untrusted.

This problem of private key management becomes evident as well, when considering highly-constrained IoT devices (*i.e.*, Class 0 IoT devices [4]) that cannot directly interact with a BC node using standard Internet protocols (*e.g.*, HTTP) or have power limitations (*e.g.*, batteries). Thus, relying on low power communication (*e.g.*, Bluetooth Low Energy, BLE) between edge nodes signing transactions and handling key management are possible in a hybrid approach [33]. Such an edge node is a single point of failure and, as it stores all the devices' private keys, if compromised, all devices communicating with that edge nodes are compromised as well. It is crucial that all devices' private keys are secured, meaning that they cannot leave the device for transaction signing, and devices are to be physically secured against tampering.

Thus, this paper proposes the approach Edge2BC, which directly signs Elliptic Curve Digital Signature Algorithm (ECDSA)-based BC transactions in constrained Class 0 IoT devices. To sign the transaction with the device's private key, a JavaCard *applet* was implemented in the device's Hardware Security Module (HSM) exposing an interface for interaction commands. Edge2BC relies on an edge node (*i.e.*, mobile device) to (a) retrieve BC information, such as transaction fees or transaction nonce, and (b) interact with a BC interoperability Application Programming Interface (API) [34], which sends the signed raw transactions to Ethereum [6], Hyper-Ledger [27], or any Remote Protocol Enabled (RPC)-enabled BC. A Proof-of-Concept (PoC) of Edge2BC was implemented (code available at [20]) and a power measurement of the signing phase was performed. Thus, showing the feasibility of Edge2BC and that by relying on Edge2BC, even if the edge node is compromised, the private key of the device is not leaked, since the key never leaves the device.

This paper is structured as follows. Section II provides relevant background information. While Section III describes the design and implementation of Edge2BC, Section IV presents power measurements and discusses challenges. Section V discusses related work compared to Edge2BC. Lastly, Section VI summarizes the paper and foresees future work.

II. BACKGROUND

The basics of devices and protocols employed in Edge2BC are described concerning the IoT device itself, key details of BLE communication, and the ISO7816 standard used in the communication with these device.

A. Temperature Logger

The solution for temperature monitoring considered in this paper is the MODSense T Temperature logger, a device designed for sensitive shipments (*e.g.*, medicines) in the cold chain context [22]. It collects and evaluates temperature data against the alarm criteria defined for each shipment and can be configured over the air using BLE (*cf.* Section II-B).

Fig. 1: The MODSense T Temperature Bluetooth Logger

The device features a tamper-evident casing, a maximum storage capacity of 45 thousand temperature measurements, Near Field Communication (NFC) support, and traceability via the QR code [23]. One crucial component for Edge2BC is the built-in on-board Hardware Security Module (HSM) that stores a, entirely inaccessible, private key initialized in the factory at manufacturing time. The private key enables the logger to sign any data digitally.

Further, the device's firmware runs on the MicroController Unit (MCU) which contains a Central Processing Unit (CPU), volatile and non-volatile memory, hardware peripherals for bus communication and related tasks, as well as all necessary hardware for a BLE connection except the antenna. Even though the HSM contains its processor with limited performance, it can run code independent of the MCU. Thus, isolating security-related code from generic code.

B. Bluetooth Low Energy (BLE)

BLE is a wireless technology standard that enables the connection of devices in a range of about ten meters, but with significantly less power consumption and costs compared to the classical Bluetooth standard [17].

Fig. 2: BLE Packet Structure. Adapted from [32]

Figure 2 depicts the basic structure of a BLE packet and the underlying structure of the “Data” and “Attribute Protocol (ATT) Data” fields. The “Data” field has a variable size dependent on the Bluetooth specification [2]. Contained in this data field the core protocol responsible for multiplexing and segmentation of packets, the is a Logical Link Control and Adaptation Protocol (L2CAP) header [2].

Contained in the “ATT Data” field of the link-layer “Data” field is an ATT packet. The “Op-code” field is necessary for the protocol to determine the desired operation in this packet, *e.g.*, writing data, reading data, or registering notifications. For specific commands, the “Data” field in the ATT packet is dependent on the ATT Maximum Transmission Unit (MTU), and an additional “Attribute Handle” field is required. During MTU negotiation, both devices exchange the maximum MTU size they can support, and the lower one of those values is assumed as the new MTU for the rest of the connection.

C. ISO 7816

The HSM on the modum.io logger is connected using a bus conforming to the ISO 7816 standard [19]. This ISO specifies a full communication stack from the physical to the application layer; thus, it governs the physical encoding of bits as well as the binary encoding of commands. Commands sent to the HSM are encoded in Application Protocol Data Units (APDUs), in which each APDU contains a class Byte (CLA), an instruction Byte (INS) and two parameter Bytes (P1 and P2). The ISO 7816 standard governs which CLA and INS codes can be used by an application, which ranges of values are reserved for future use and which bits in the CLA and INS codes have a special meaning. APDU commands (as identified by the CLA and INS Byte) are classified in four categories, named cases:

- **Case 1 commands:** do not require a command payload and do not expect a response payload, *e.g.*, cycle the HSM into a different internal state.
- **Case 2 commands:** require a command payload and do not expect a response payload, *e.g.*, store a given certificate on the HSM.
- **Case 3 commands:** do not require a command payload and expect a response, *e.g.*, a read out a public key from the HSM.
- **Case 4 commands:** require a command payload and expect a response, *e.g.*, calculate the hash of an input.

Fig. 3: APDU Command Format

Figure 3 depicts the format of the APDU commands in four cases. Depending on the command case, the L_c field

encodes the length of the command payload, and the L_e field encodes the length of the expected response payload. Figure 4 presents the format of APDU responses. Such a response message contains the optional response payload and finally two mandatory status code Bytes (SW1 and SW2). The response code for “success” is represented by “0x90 0x00”.

Fig. 4: APDU Command Response Format

D. JavaCard

JavaCard [1], [28] is a standard that defines an operating environment, a subset of the Java programming language, and limited Java libraries that allow the development, deployment, and execution of applications on a JavaCard-enabled HSM. It provides abstraction and sandboxing mechanisms, via a Java virtual machine on the HSM, where multiple applications (named *applets*) independently of each other on the HSM.

When the HSM is booted (*i.e.*, initialized), APDU commands (*cf.* Section II-C) are processed by the JavaCard platform. A SELECT command allows selection of an *applet*, after which most APDU commands are forwarded to the *applet* and processed. GlobalPlatform [14] is a standard that specifies the management of a HSM platform and its *applets*. Thus, it specifies commands to (i) load, (ii) configure, and (iii) delete applets. Further, it standardizes the life cycle states of the HSM, and multiple protocols for secure communication with the HSM or its *applets*.

III. DESIGN OF EDGE-TO-BLOCKCHAIN TRANSACTIONS

The design and implementation of Edge2BC includes the overview of those interactions between the Edge2BC’s components (*i.e.*, modum.io logger, edge node, and server) and the respective transactions preparations. Additionally, technical details of the edge node, *i.e.*, mobile application, are presented to provide the main aspects to reach an implementation of the solution. The implementation of the HSM applet is detailed, while Edge2BC’s code is available at [20].

A. Workflow

The workflow between Edge2BC’s components is different depending on the BC in use. Figure 5 depicts an overview of the information flow of Edge2BC when creating Ethereum transactions. An Ethereum transaction include fields, such as “to” and “from” addresses, a nonce (specific to the sender’s address), the gas price, and the gas limit. As the modum.io logger cannot communicate with the Ethereum RPC to retrieve dynamic information (*e.g.*, gas price and nonce), the mobile application (*cf.* Section III-C) retrieves this information from the server (*i.e.*, Bifröst) using a Representational State Transfer (REST) API. It sends this information to the logger via BLE. Then, the modum.io logger creates an Ethereum transaction and signs it using an ECDSA [31] key using the

Fig. 5: Ethereum Transaction Signing Flow

secp256k1 curve [7]. Lastly, the signed raw transaction is sent back to the mobile application, which forwards it to the Bifröst API.

Fig. 6: Hyperledger Sawtooth Transaction Signing Flow

In contrast, Figure 6 illustrates the flow for the creation of a Hyperledger Sawtooth transaction. In this process, no communication between the mobile application and the server is required, except when sending the signed raw transaction. A Hyperledger transaction contains (a) a “Header” field, which contains a serialized (using Protocol Buffers (Protobuf) [15]) version of the transaction header signed by a private key, (b) a “Payload” field, which contains the data used during transaction execution to apply the state change, and (c) a “Header Signature” field, which contains the the transaction header

signature. Hyperledger Sawtooth transactions are collected in a list, which is included in the structure of a batch, and their IDs are included in the batch header. Both transaction header and batch header are signed directly in the modum.io logger and the edge node (*i.e.*, mobile application) forwards them to server (*i.e.*, Bifröst API).

It can be seen that in both cases (*cf.* Figure 5 and Figure 6), the private key is never exchanged between the logger and edge node or between the logger and the server. By relying on this scheme, the private key is protected against man-in-the-middle attacks and the creation of fake transactions that do not originate from the logger. Thus, allowing the secure integration of solutions enabling the selection of the most suitable BC [35] based on high-level user requirements as proposed in [38].

B. Transaction Preparation

For Ethereum, transaction information (*e.g.*, nonce, gas price, gas limit, receiver address, value, data, *recovery ID*, and signature fields) is serialized using Ethereum's serialization format Recursive Length Prefix (RLP) [12]. The mobile application prepares an RLP encoded transaction and the modum.io logger includes the shipment information hash in the prepared transaction and then signs the transaction. Thus, Edge2BC avoids de- and re-serializing the nonce, gas price, gas limit, receiver address and value fields of the transaction structure. In summary, the serialization format of the transaction data sent to the modum.io logger is the same as is used to send the transaction to the Ethereum network. Thus, not requiring any posterior modification.

Due to size restrictions of the data that can be exchanged in one message between the modum.io logger and mobile application (*cf.* Section IV-B) only the raw signature and shipment information hash are returned from the modum.io logger. The transaction signature and shipment information hash are incorporated in the prepared transaction on the mobile application. The outcome of this approach does not differ from returning the full encoded final transaction and has no security implications because any modification of the transaction in a way that does not reflect those performed on the modum.io logger for signing would result in an invalid signature.

For Hyperledger, the same approach was followed. As described in Section III-A Hyperledger Sawtooth transactions are a Protobuffer [15] encoded structure of Concise Binary Object Representation (CBOR) [5] encoded payload. Thus, all the transaction information that do not rely on the private key are prepared in the edge node, while the modum.io logger only injects the shipment information hash into the transaction and signs the transaction with its private key.

To process transactions, the modum.io logger counts on different levels of responsibility within Edge2BC, they are defined as:

- **minimal**: The modum.io logger injects the shipment information hash into the transaction at a hard-coded offset. The full encoded transaction is signed and only the signature is returned. The mobile application integrates

the signature into the final transaction and forwards to the BC interoperability API.

- **intermediate-1**: The modum.io logger injects the shipment information hash into the transaction at an offset that is passed on to it from the mobile application. This allows for flexibly to, *e.g.*, modify the API of the Smart Contract (SC) that receives a transaction without requiring the modification of the modum.io logger's firmware. This is required, if serialized transactions contain variable-length values before the shipment information hash so that a hard-coded offset does not work.
- **intermediate-2**: The modum.io logger parses the serialization format of the transaction and injects the shipment information hash at the correct location without receiving an offset from the mobile application. No further processing of the other fields of the transaction is performed.
- **maximal**: The modum.io logger understands the full semantics of the transaction and is able to parse the serialization format of the transaction. This allows the modum.io logger to perform checks on the values of the transaction and reject signing it in certain cases.

The Edge2BC prototype was implemented as an **intermediate-2** level for Ethereum and a **minimal** level for Hyperledger. The discussion of benefits of having the modum.io logger firmware understand the structure of the prepared transactions and applying checks on the values before signing the transaction is presented in Section IV-B.

C. Mobile Application

The mobile application acting as an edge node was implemented using the React Native framework [13]. Figure 7 depicts a modum.io logger device connected to this mobile application, its MAC address, and its Ethereum address.

Once the "Send Ethereum Transaction" or the "Send Hyperledger Transaction" button is pressed, the mobile application begins to construct a transaction according to Section III-B. The prepared transaction for Ethereum is a template for an encoded method call for a simple `store()` function in a SC. The transaction prepared for Hyperledger Sawtooth is a template for a `set` operation creating a new key-value pair with the first 20 characters of the shipment ID, being the key and the first 4 Bytes of the temperature data hash being the value. Storing the entire temperature data hash is not possible since the *IntegerKey* transaction family limits valid values to integers in the range of 0 to 2^{32} , *i.e.*, 4 Byte integers, while keys are limited to 20 characters [18].

Both Ethereum and Hyperledger apply the restriction of EIP-2 [11] on their signatures, limiting the range that the *s* component is allowed to assume. Since the HSM is unaware of these restrictions, it generates a signature that is not in all cases immediately a valid Ethereum or Hyperledger signature. Thus, it was applied the EIP-2 fix to the signature's *s* component in the mobile application and, in the case of Ethereum, compute the *recovery ID* to derive the value *v*.

Fig. 7: Mobile Application Connected to the Logger

Regarding the BLE communication, the minimum of the largest MTU (*cf.* Section II-B) that both devices can support is negotiated as the connection's MTU. The largest MTU on the modum.io logger side is limited to 510 Bytes, where 2 Bytes are deducted for the "Op-code" field and 1 Byte for the "Attribute Handle" field. MTU sizes for Android phones can vary [39], with higher-end phones, maximum MTUs of 256 Bytes are common, but some phones only support the default BLE MTU of 23 Bytes. The Noble JavaScript library [25] supports a maximum MTU of 256 Bytes. For *Edge2BC*, a client (*i.e.*, Android device) must support at least an MTU of 244 Bytes. Thus, the mobile application was deployed on a smartphone supporting a large (*i.e.*, 256 Bytes) MTU.

D. Firmware

The modum.io logger firmware provides an interface of a basic temperature logger. Endpoints are provided to start and stop a recording, and reading out the recorded data. One endpoint implements preparing and signing of an Ethereum transaction and four endpoints implement the preparing and signing of a Hyperledger transaction. Table I describes the endpoints of the modum.io logger interface.

The common application layer protocol for BLE is Generic Attribute (GATT). Endpoints are realized in "characteristics", which are grouped into "services". Characteristics can be read-only (ro) or read-write (rw). Reads and writes can be rejected by the server based on the circumstances and can have side-effects, such as starting a recording on the modum.io logger upon write of a defined characteristic. Due to memory limitations, the number of characteristics on a Bluetooth device is limited. Further, a single read or write of a characteristic has a maximum size, preventing the exchange

of larger data structures. In the *Edge2BC* prototype, a single rw characteristic provides a serial data exchange channel. No fragmentation is implemented in a dedicated protocol layer but is implemented at the application layer if needed (*e.g.*, by splitting the Hyperledger transaction header signature into endpoints 0x10 and 0x11).

The ISO7816 bus driver for the modum.io logger has an important limitation, which is that no concurrent connection with a Bluetooth client and the HSM is possible. However, this limitation was addressed by disconnecting the Bluetooth client in the modum.io logger firmware as soon as a signature is requested (*i.e.*, endpoints 0x04, 0x11 and 0x13). After the signature has been computed, the modum.io logger restarts its Bluetooth advertising. The mobile application scans for Bluetooth devices until it rediscovers the modum.io logger. Then, it can read the result from the BLE protocol endpoint command by reading the BLE characteristic.

As Ethereum transactions are hashed using Keccak-SHA3, and after finding that implementing such a function on the HSM is not feasible (see Section IV-B), it was decided to implement hashing of the transaction in the modum.io logger firmware. Thus, only a pre-computed hash is sent to the HSM, which signs it with its private key and returns the signature. There is no security difference between sending an arbitrary hash to the HSM to be signed and sending arbitrary data to the HSM to be hashed and signed. The only security improvement to be gained from sending data instead of a hash could be if the HSM understands the semantics of the data. For example, if the HSM understands the *nonce* part of the transaction, it can enforce that the nonce is strictly incrementing and refuses to sign transactions for apparently invalid nonces. A discussion of the security implications of this decision can be found in Section IV-B. It was followed the same approach for Hyperledger transactions and pre-compute the hash to be signed in the modum.io logger firmware.

E. Applet

The *applet* receives a pre-computed hash of a transaction and signs it with the ECDSA key using the *secp256k1* curve. The private and public key pair is generated once during installation of the applet and stored in the HSM Read Only Memory (ROM). Table II describes the HSM interface endpoints. Retrieving the public key is a *Case 2* command, signing a hash is a *Case 4* command (*cf.* Section II-C).

To develop the *applet* and debug the code, a special smart card fitted with the same HSM that is programmed with a debug interface was utilized. The smart card can be inserted into any USB smart card reader and connected to a computer. The integrated development environment allows *applet* installation on the smart card and stepping through the code as with any other debugger. The HSM's implementation of the ECDSA algorithm encodes signatures in the Distinguished Encoding Rules (DER) format. Thus, the applet also contains code to extract the raw *r* and *s* values from the DER encoded binary data to return a 64 Byte raw signature.

TABLE I: Logger Interface Endpoints

ID *	Name	Access †	Description
0x01	Get Pubkey	ro	Returns the public key of the keypair that was generated on the HSM during provisioning
0x02	Start Recording	rw	Starts a temperature recording
0x03	Stop Recording & Readout	rw	Returns the hardcoded temperature values of the "last shipment"
0x04	Get Ethereum Signature	rw	Receives as payload the raw prepared Ethereum transaction prepares the transaction with the hash of the shipment information and returns the signature concatenated with hash of the transaction
0x09	Put Hyperledger Payload	rw	Receives as payload the raw Hyperledger "intkey" payload and stores it internally
0x10	Get Hyperledger Transaction Header Signature 1	rw	Receives as payload the first 240 Bytes of the raw Hyperledger transaction header and stores them internally
0x11	Get Hyperledger Transaction Header Signature 2	rw	Receives as payload the remainder of the raw Hyperledger transaction header, inserts (part of) the shipment information hash into the previously stored payload, prepares the transaction using the updated payload, signs it and returns the signature concatenated with the updated payload
0x13	Get Hyperledger Batch Header Signature	rw	Receives as payload the raw Hyperledger batch header, signs it and returns the signature

* The ID corresponds to the prefix of the endpoint † Access either ro for read-only or rw for read-write

TABLE II: HSM Applet Endpoints

CLA	INS	P1	P2	Lc	Payload	Le	Response Payload
0x80	0x02	0x00	0x00	N/A	N/A	0x40	Public key of the HSM's keypair as raw x and y value (32 Byte each)
0x80	0x04	0x00	0x00	0x20	32 Byte hash to sign	0x40	Signature as raw r and s value (32 Byte each)

IV. EVALUATION AND DISCUSSION

Energy efficiency is a crucial aspect for restricted IoT devices to be able to perform their actions during the required operation period (*e.g.*, origin and destination length). In this regard, the analysis of impacts in terms of energy consumption of Edge2BC is key, being complemented with a discussion on challenges faced during such an implementation.

A. Power Measurements

Figure 8 depicts a power measurement of the modum.io logger during the process of obtaining a signed Hyperledger Sawtooth transaction structure, where the signatures are computed on the HSM (*cf.* Figure 8a) and in software (*cf.* Figure 8b). The measurements were obtained using the Nordic Semiconductor Power Profiler Kit [26]. The Hyperledger process running on the *Device Under Test* includes signing the transaction header and the batch header.

Further, Figure 8 details Edge2BC's steps from characteristic peaks during each of these phases. First, the modum.io logger advertises (depicted in red) until the client connects (depicted in green). After reading data, such as the public key, the transaction header signature is requested and the modum.io logger disconnects from the client and computes the transaction header hash (depicted in blue). The signature is computed on the HSM (depicted in yellow) or in software (depicted in orange). As soon as the signature is ready, the modum.io logger starts advertising again (depicted in red). During the next connection (depicted in green), the previously calculated signature is read from the modum.io logger. A second cycle of similar operations is performed for the batch header.

For each of these distinct phases Table III details the average and maximum power consumption and references the overlay colors used to mark these phases in the Figure 8. Table IV

TABLE III: Power Consumption of HSM-enabled PoC

Phase	Avg. Current	Max. Current	Color
BLE advertising	1.00 mA	19.92 mA	Red
BLE client connected	1.47 mA	17.94 mA	Green
Hashing on MCU	4.82 mA	13.76 mA	Blue
HSM operation	11.60 mA	11.69 mA	Yellow

TABLE IV: Power Consumption of Software-Crypto PoC

Phase	Avg. Current	Max. Current	Color
BLE advertising	1.00 mA	20.19 mA	N.A
BLE client connected	1.37 mA	18.00 mA	N.A
Hashing on MCU	5.16 mA	13.79 mA	N.A
Signature computation	8.10 mA	8.16 mA	Orange

details the average and maximum power consumption during each of the distinct phases using the signatures in software. Analyzing the measured data, it is evident that using the HSM incurs a higher power consumption (11.60 mA) than running all cryptography in software (8.19 mA), approximately 41% increase in power consumption.

Additionally, the HSM signature requires more time than performing the signature in MCU software. However, the security advantages (*e.g.*, isolating of security-related code and dedicated signature functions) of the HSM justify its use. Further, the power consumption is fairly high in these measurements because the UART peripheral used for transmitting log statements to the development computer has a high power consumption. In this sense, logging via UART must be deactivated for a production-ready of the firmware to not impact in the battery life of the device.

Fig. 8: Power Measurements Results: (a) Represents the Signing Process Using HSM and (b) Represents the Signing in Software

B. Discussion

The Edge2BC logger accepts external values, such as nonce, gas price, gas limit, receiver address, value and chain ID for Ethereum, that are included in the transaction without checking. The data field is not checked since the logger only inserts the shipment data hash into the correct position. For Hyperledger Sawtooth the external values are the transaction family name, family version, inputs, outputs, and dependencies. Malicious or accidental modification of those fields allows construction of (a) a transaction that is invalid (rejected by the network), (b) a transaction that unnecessarily wastes funds or (c) that does not point to the correct method of the correct SC on the correct network. Thus, not leading to an inclusion of the shipment information at the expected location of the BC network. However, none of those modifications threaten the integrity of shipment data because it is directly included into the transaction by the modum.io logger and signed, preventing any external modification thereof.

Further, data aggregation could be performed in the edge node (*i.e.*, smartphone). However, the security of the private key employed to sign and send transactions to the BC network relies on the edge node, which, if compromised, compromise all the aggregated data. In this sense, for Edge2BC, the edge node is just a support to fetch information from the BC network that the IoT devices cannot directly fetch due to their limited network communication capabilities.

Moreover, there were challenges encountered during development of Edge2BC's prototype. Firstly, the BLE MTU size was too small to exchange full transaction, which was solved by implementing simple fragmentation on the application layer by splitting the affected endpoints (*cf.* Section III-C and Section III-D). Secondly, the Keccak-SHA3 hashing computation was slow on the HSM; thus, the hashing was implemented in the modum.io logger firmware (*cf.* Section III-D). Thirdly, the HSM's signatures consist of a tuple (r, s) without EIP-

2 restrictions [11], which required the application of EIP-2 restrictions in the mobile app by transforming s if necessary, and recover the v value in the mobile app (*cf.* Section III-C). Fourthly, the ISO 7816 driver does not allow concurrent HSM and BLE connection; therefore, BLE was disconnected before using the HSM and reconnected after the signature computation (*cf.* Section III-D). Lastly, the `nonce` in an Ethereum transaction must exactly match the number of transactions successfully sent by the address. Thus, the `nonce` was retrieved from the server backend before creating the transaction (*cf.* Section III-C).

Besides the power measurement conducted in Section IV-A, to extend the current PoC of Edge2BC presented herein to a full-fledged solution, aspects that were not address yet must be solved. For example, the dependency of the IoT devices in the edge node could be minimized by fixing the gas price and gas limit, and updating the `nonce` directly in the device (in the case of Ethereum). Moreover, the private key is kept secure within the device throughout the process. However, the interaction between the edge node and the server must be analysed in terms of the security of its communication and possible attack vectors. Nevertheless, the implementation of Edge2BC's PoC and the power measurements conducted do show that it is a practical and feasible solution.

V. RELATED WORK

In [24], the performance of selected cryptography functions (*e.g.*, SHA-256, SHA-512, and Ed25519) in different IoT devices, such as TelosB, ATmega 2560, and Raspberry PI is evaluated. The authors claim that submitting a transaction is possible but incurs in a high CPU usage on the device. A major difference between Edge2BC and the work presented in [24] is that the IoT devices only sign the collected data and do not create the raw transactions directly in the IoT node. Instead, IoT devices create and send the transaction in the edge node.

Further, [30] evaluates the impact of the BC integration in IoT devices. The authors implement a cross-platform library to perform several BC-related functions, such as encoding, hashing, and signing in constrained IoT devices. However, IoT devices communicate via LoRaWAN radio technology, whereas the logger employed in Edge2BC utilizes BLE for its communication. The devices are able to perform all the functions in the library directly in the node, which the modum.io logger is not able.

Following a similar approach, [21] studies the feasibility of deploying BC nodes and running cryptographic operations for BCs in IoT devices. Different aspects, such as network latency, CPU usage, and energy consumption are measured for the devices running a private Ethereum BC. Even though the authors claim that the feasibility of utilizing BC-based decentralized security at the IoT edge devices is high, the resource of the devices must be shared between sensing and actuation functions and performing BC-specific functions. Thus, there is a trade-off between BC performance and the accommodation of sensing and actuation functions to be considered.

[10] proposes to minimize the impact of the communication of IoT devices with the BC (*e.g.*, block synchronization and broadcast of transactions) by introducing a proxy service that performs such communication, while the IoT device stills signs the transaction. The authors implement such a proxy for Hyperledger Fabric and perform two evaluations, (*i*) bandwidth and (*ii*) compute power using a RaspberryPi as an IoT device. These evaluations showed a reduction in the data sent and received and on the CPU time spent for an individual transaction.

Within the context of traceability of the food-chain, [16] proposes a solution where IoT devices directly sign transactions without relying in third-parties nodes. In the solution proposed by the authors, IoT devices monitor the temperature of fridges and the position of the vehicle used during the transport and send such measurements to a SC. The communication with the BC is performed using a mobile network connection, which sends data to an Ethereum RPC server. This server acts as a gateway between the IoT device and the BC network.

Signing a BC transaction directly in the IoT device is not a novel approach due to its security benefits. However, such approaches consider devices that communicate with an IP network using special protocols (*e.g.*, Constrained Application Protocol, CoAP), or are capable of implementing full network protocols stacks [4]. In this sense, relying in an limited way, or not relying at all, in a edge or gateway node. In contrast, the modum.io logger is not able to directly communicate with a BC node, relying on a edge node and BLE to perform the communication with the BC network. Therefore, not every function required to interact with a BC was implement but rather selected functions were performed in the edge node that did not compromised the security of the device's private key or the communication.

VI. SUMMARY AND FUTURE WORK

The combination of Blockchains (BC) and Internet-of-Things (IoT) in supply-chains achieves several benefits for different stakeholders, such as data immutability and trust that is one of the most important aspects when using non-native data (*e.g.*, IoT) in the BC. However, it introduces concerns, especially when considering the integrity of the device and the private key security. Thus, the Edge2BC approach practically demonstrates that a direct creation and signing of BC transactions in a highly-constrained IoT device (*i.e.*, not being able to communicate with an IP network) is possible.

Edge2BC was developed to sign Ethereum and Hyper-Ledger Sawtooth transactions containing the temperature of medicines measured by a Bluetooth-enabled device. Such transactions are forwarded by a mobile application, acting as an edge node, to a BC interoperability API, which forwards them to the BC network in use without requiring BC-specific commands from the edge node.

Results from the power consumption evaluation during the different steps of Edge2BC show that it is very well feasible to implement signing functions on the device. Although implementing them in the device's Hardware Security Module (HSM) incurs higher power consumption than implementing them as generic software, the isolation of the code from the general processor to the HSM increases the security of the key management. Dedicated challenges were encountered during Edge2BC's implementation once the device is not able to communicate directly with an IP network. *E.g.*, it must rely on the edge node to retrieve external information (*e.g.*, Ethereum transaction nonce) and concurrent HSM operations and the Bluetooth connection required a closing and opening of the connection before and after the signing is completed.

Based on the prototype implementation and evaluations, it is concluded that Edge2BC is feasible to create and sign BC transactions in a highly-constrained IoT device without the private key being exposed to external components (*e.g.*, edge node). However, specific aspects that cannot be solved without the employment of more resource-capable IoT devices must be taken into consideration for production-ready deployments, where larger data are to be maintained or frequent signing of transactions are required.

Future work includes (*i*) the application of Edge2BC to different BCs (*e.g.*, Stellar or EOS), (*ii*) the evaluation of energy consumption and resource utilization for different IoT devices (*e.g.*, TelosB or Arduino Nano), (*iii*) a standardization of Edge2BC's communication flow to cover a wider range of use cases, (*iv*) an analysis of the Edge2BC's security and scalability, and (*v*) the research on approaches to reduce the dependency of Edge2BC on the edge node.

ACKNOWLEDGMENTS

This work was supported partially by (a) the University of Zürich UZH, Switzerland and (b) the European Union's Horizon 2020 Research and Innovation Program under grant agreement No. 830927, the Concordia Project.

REFERENCES

- [1] M. Baentsch, P. Buhler, T. Eirich, F. Horing, and M. Oestreicher, "JavaCard - From Hype to Reality," *IEEE Concurrency*, vol. 7, no. 4, pp. 36–43, October 1999.
- [2] Bluetooth SIG Proprietary, "Bluetooth Specification Version 4.2," December 2014, <https://bit.ly/3odo3Me>, Last visit October 22, 2020.
- [3] T. Bocek, B. B. Rodrigues, T. Strasser, and B. Stiller, "Blockchains Everywhere - a Use-Case of Blockchains in the Pharma Supply-Chain," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2017)*, Lisbon, Portugal, May 2017, pp. 772–777.
- [4] C. Bormann, M. Ersue, and A. Keranen, "RFC 7228: Terminology for Constrained-Node Networks," May 2014, <https://tools.ietf.org/html/rfc7228>, Last visit October 21, 2020.
- [5] C. Bormann and P. Hoffman, "RFC 7049: Concise Binary Object Representation (CBOR)," October 2013, <https://tools.ietf.org/html/rfc7049>, Last visit November 4, 2020.
- [6] V. Buterin, "A Next-Generation Smart Contract and Decentralized Application Platform," October 2020, <https://ethereum.org/en/whitepaper/>, Last visit November 3, 2020.
- [7] Certicom Research, "SEC 2: Recommended Elliptic Curve Domain Parameters - Version 2.0," January 2010, <http://www.secg.org/sec2-v2.pdf>, Last visit November 3, 2020.
- [8] H. Dai, Z. Zheng, and Y. Zhang, "Blockchain for Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8076–8094, June 2019.
- [9] A. Davenport and S. Shetty, "Air Gapped Wallet Schemes and Private Key Leakage in Permissioned Blockchain Platforms," in *IEEE International Conference on Blockchain (Blockchain 2019)*, Atlanta, GA, USA, July 2019, pp. 541–545.
- [10] G. Dittmann and J. Jelitto, "A Blockchain Proxy for Lightweight IoT Devices," in *Crypto Valley Conference on Blockchain Technology (CVCBT 2019)*, Rotkreuz, Switzerland, June 2019, pp. 82–85.
- [11] Ethereum Community, "Ethereum EIP-2," <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.md>, Last visit October 27, 2020.
- [12] Ethereum Foundation, "Recursive Length Prefix," June 2020, <https://eth.wiki/en/fundamentals/rlp>, Last visit November 3, 2020.
- [13] Facebook Inc., "React Native - A Framework for Building Native Apps Using React," 2020, <https://reactnative.dev/>, Last visit November 4, 2020.
- [14] GlobalPlatform Technology, "Card Specification," v2.3.1.
- [15] Google, "Protocol Buffers," <https://developers.google.com/protocol-buffers>, Last visit November 3, 2020.
- [16] J. Grecuccio, E. Giusto, F. Fiori, and M. Rebaudengo, "Combining Blockchain and IoT: Food-Chain Traceability and Beyond," *Energies*, vol. 13, no. 15, 2020.
- [17] M. Honkanen, A. Lappetelainen, and K. Kivekas, "Low End Extension for Bluetooth," in *IEEE Radio and Wireless Conference (IEEE Cat. No. 04TH8746 2004)*, Atlanta, GA, USA, September 2004, pp. 199–202.
- [18] Intel Corporation, "IntegerKey Transaction Family," 2017, <https://bit.ly/2TT7RBJ>, Last visit November 4, 2020.
- [19] International Organization for Standardization (ISO), "ISO 7816," June 1987, <https://www.iso.org/standard/14732.html>, Last visit October 22, 2020.
- [20] A. Knecht, T. Strasser, and E. J. Scheid, "Edge2BC Repository," 2020, <https://gitlab.ifi.uzh.ch/scheid/edge2bc>, Last visit October 15, 2020.
- [21] S. Misra, A. Mukherjee, A. Roy, N. Saurabh, Y. Rahulamathavan, and M. Rajarajan, "Blockchain at the Edge: Performance of Resource-Constrained IoT Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 174–183, 2021.
- [22] modum.io AG, "MODsense - Next Generation Tracking and Monitoring," 2020, <https://modum.io/solutions/modsense>, Last visit October 20, 2020.
- [23] —, "MODsense T Gen 1 - Device Specification Sheet Version 1.6," January 2020, https://modum.io/sites/default/files/documents/2020-01/MODsenseOne_Spec_Sheet_1.6_0.pdf, Last visit November 3, 2020.
- [24] S. R. Niya, E. Schiller, I. Cepilov, and B. Stiller, "BIIT: Standardization of Blockchain-based IIoT Systems in the I4 Era," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2020)*, Budapest, Hungary, April 2020, pp. 1–9.
- [25] Noble Community, "Noble - A Node.js BLE (Bluetooth Low Energy) Central Module," 2018, <https://github.com/noble/noble>, Last visit November 4, 2020.
- [26] Nordic Semiconductor, "Power Profiler Kit," <https://www.nordicsemi.com/Software-and-tools/Development-Tools/Power-Profiler-Kit>, Last visit November 4, 2020.
- [27] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery, "Sawtooth: An Introduction (White Paper)," 2018, https://www.hyperledger.org/wp-content/uploads/2018/01/Hyperledger_Sawtooth_WhitePaper.pdf, Last visit November 3, 2020.
- [28] Oracle, "Java Card Platform Specification Version 2.2.2," 2006, <https://www.oracle.com/java/technologies/java-card/platform-specification-v222.html>, Last visit October 22, 2020.
- [29] C. Paris and J. S. Hopkins, "Pfizer Sets Up Its 'Biggest Ever' Vaccination Distribution Campaign," October 2020, <https://www.wsj.com/articles/pfizer-sets-up-its-biggest-ever-vaccination-distribution-campaign-11603272614>, Last visit October 30, 2020.
- [30] M. Pincheira, M. Vecchio, R. Giaffreda, and S. S. Kanhere, "Exploiting Constrained IoT Devices in a Trustless Blockchain-based Water Management System," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2020)*, Toronto, Canada, May 2020, pp. 1–7.
- [31] T. Pornig, "RFC 6979: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," August 2013, <https://tools.ietf.org/html/rfc6979>, Last visit November 3, 2020.
- [32] Punch Through, "Maximizing BLE Throughput Part 2: Use Larger ATT MTU," November 2017, <https://punchthrough.com/maximizing-ble-throughput-part-2-use-larger-att-mtu-2/>, Last visit October 22, 2020.
- [33] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On Blockchain and its Integration with IoT. Challenges and Opportunities," *Future Generation Computer Systems*, vol. 88, pp. 173 – 190, 2018.
- [34] E. J. Scheid, T. Hegnauer, B. Rodrigues, and B. Stiller, "Bifrost: a Modular Blockchain Interoperability API," in *IEEE Conference on Local Computer Networks (LCN 2019)*, Osnabrück, Germany, October 2019, pp. 332–339.
- [35] E. J. Scheid, D. Lakic, B. B. Rodrigues, and B. Stiller, "PleBeuS: a Policy-based Blockchain Selection Framework," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2020)*, Budapest, Hungary, April 2020, pp. 1–8.
- [36] E. J. Scheid, B. Rodrigues, and B. Stiller, "Toward a Policy-based Blockchain Agnostic Framework," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019)*, Washington, D.C., USA, April 2019, pp. 609–613.
- [37] E. J. Scheid, B. B. Rodrigues, L. Z. Granville, and B. Stiller, "Enabling Dynamic SLA Compensation Using Blockchain-based Smart Contracts," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019)*, Washington, D.C., USA, April 2019, pp. 53–61.
- [38] E. J. Scheid, P. Widmer, B. Rodrigues, M. Franco, and B. Stiller, "A Controlled Natural Language to Support Intent-based Blockchain Selection," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2020)*, Toronto, Canada, May 2020, pp. 1–9.
- [39] J. Siva and C. Poellabauer, "Connection-Oriented BLE Traffic Servicing Characteristics on Android Devices," in *IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR 2020)*, Stevenson, WA, USA, May 2020, pp. 1–6.