A novel approach for providing client-verifiable and efficient access to private smart contracts

Alexander Köberl[1,2], Holger Bock[2], Christian Steger[1]
alexander.koeberl@infineon.com, holger.bock@infineon.com, steger@tugraz.at
[1] *Institute of Technical Informatics, Graz University of Technology, Graz, Austria*
[2] *Development Center Graz, Infineon Technologies AG, Graz, Austria*

***Abstract*—Distributed Ledger Technology is a powerful tool to support direct collaboration between organisations, without requiring full trust into a centralised infrastructure. By defining a program logic and access policies with smart contracts, all interactions are verified in the distributed network and the history of the data is recorded on the ledger. Blockchain implementations targeting enterprise use cases also provide means for private transactions, where the content of the transaction is only readable by authorized participants. Direct access to the ledger requires a node with reliable connection to the network and sufficient computational resources, which usually cannot be fulfilled with lightweight Internet of Things devices and mobile applications.**

We present an advanced system for accessing an enterprise Blockchain through dedicated gateway nodes, while preserving the functionality of private transactions. A hybrid approach is used to allow computation- and storage restricted clients to send private transactions through a central gateway, and use Light Ethereum Subprotocol to verify the data integrity based on proofs from distributed nodes.

To increase the client-side security level, we introduce a dedicated Hardware Security Module for key management and efficient execution of the cryptographic primitives. A proof-of-concept implementation, using the *Quorum* Blockchain client and an extension for the *Tessera* transaction manager, validates the feasibility of the approach and can be used for further research in this field.

***Index Terms*—Blockchain, Smart Contract, Privacy, Quorum, Light Ethereum Subprotocol, Blockchain Gateway**

I. Introduction

When an application is jointly developed by a consortium of companies, one partner or a trusted external provider is in charge of hosting the infrastructure and providing access to clients in a fair way. All other partners monitor the conformable operation with recurrent audits and consequences, in case of inconsistencies. For many applications, those strong trust requirements and precautions outweigh the expected benefit of a collaborative platform. Such circumstances may delay the technological innovation within an industry and promotes the growth of data silos. In the area of Internet of Things (IoT) and sensor networks, this prevents easy sharing and commercialisation of sensor measurements in open data markets. End-users face issues with vendor lock-in, where devices from multiple manufacturers are not compatible and open management platforms cannot be used.

This paper has been partially funded by the European Union's Horizon 2020 research project *DataVaults* (Grand Agreement no. 871755).

The rapid emergence of Distributed Ledger Technologies (DLTs) and the resulting acceleration of research and development, especially by individuals and small organisations, produced new concepts for the deployment of distributed Applications (dApps). The distributed execution of program logic, combined with the protected storage of a transaction history and predefined access policies, solves many problems of consortium projects. For this application domain, some Blockchain implementations offer confidential data sharing using off-chain channels, while broadcasting a distinct identifier (e.g. cryptographic hash of the data) to the network as efficient record of the transaction. The network is composed of independent nodes, which store a copy of the protected ledger and process new transactions. Hosting such a server typically requires hundreds of gigabytes for storage, several gigabytes of memory and a permanent network connection [1]. Resource restricted IoT devices and mobile clients often do not meet the system requirements and, in that case, cannot host a so-called *full node* locally. They are dependent on *Blockchain Gateways* to provide them with access to the network. An additional access control is required if the gateway also handles private transactions on behalf of multiple clients.

This paper presents a novel system architecture for lightweight clients accessing a consortium Blockchain. It is a hybrid approach with dedicated gateway nodes for private transactions on the *Quorum* Blockchain, and optional support of the Light Ethereum Subprotocol (LES) to enable distributed transmission and client-side verification of the block headers.

The main contributions of this paper are:

- Design and implementation of a Blockchain Gateway to provide permissioned access to private smart contracts.
- Modifications of *Quorum* and *Tessera* to allow independent verification of the data integrity by the client.
- Implementation of a client application to validate the proposed transaction protocol.

An IoT use case is presented as example application to explained the proposed concepts and highlight the challenges. The system design is based on the requirements of a distributed sensor network: Smart sensors are installed to measure environmental parameters (e.g. weather, pollution, traffic) and periodically publish them to a gateway node. The Blockchain is used for protected storage of the measurements and to transfer them to interested partners.

This is the accepted version of a conference paper which appears in 5th IEEE Conference on Dependable and Secure Computing (DSC 2022).
The final published version can be found in the conference proceedings:
https://doi.org/10.1109/DSC54232.2022.9888820

II. State of the art

A. Distributed ledgers and smart contracts

Expanding the basic concept of digital currency promoted by *Bitcoin* [2], the *Ethereum* project enabled the distributed and traceable execution of program code, so called *smart contracts* [3]. This new concept marks the beginning of the next generation in Blockchain technology, where complex *distributed Applications (dApps)* are deployed on the network, without the need of centralized infrastructure and authorities.

Transactions holding the bytecode of the *smart contract* targeting the Ethereum Virtual Machine (EVM) are distributed in the network. Subsequent transactions can be used to call functions defined in the Application Binary Interface (ABI) of the contract instance. Both operations are executed by all verifying nodes in the network, which store the code and variables in a data structure called *World-State*.

Blockchains for enterprise use cases have additional requirements. Instead of public accessibility and user-provided pseudonyms, permissioned systems with configurable user roles are employed. This allows for the application of consensus schemes offering higher throughput compared to the Proof of Work (PoW) algorithms usually used by public Blockchains.

Another factor to be considered in this domain is *transaction confidentiality*. The contents of transactions and details of smart contracts should not necessarily be broadcasted to all participants. It should be possible to transmit some data off-chain, while still maintaining the availability and integrity properties offered by the ledger.

B. Quorum Blockchain

The proposed architecture and the implementation use the *Quorum* Blockchain [4] as central component. It is a fork of the *Go Ethereum* client [5] to support enterprise applications with the following extensions:

- **Selectable consensus:** The throughput, latency and efficiency of the network is improved by replacing the PoW consensus. By default, *RAFT* [6] is used for electing a leader, who is in charge of proposing new blocks. The second officially supported option is the Istanbul Byzantine Fault Tolerant (IBFT) [7] consensus algorithm.
- **Private transactions:** By integrating the *Tessera* transaction manager, transactions can optionally be sent privately to a subset of nodes. The payload is encrypted and sent to the selected recipients off-chain, who store it in their local database. Afterwards, the sender creates a Blockchain-transaction containing the hash of the encrypted data as payload. This procedure implicitly signs the encrypted data and commits a protected trace of it on the ledger, without disclosing the raw data. This sequence is described in more detail in Section V.

C. Light Ethereum Subprotocol

The requirement of Blockchain access from resource constrained- or not fully synchronized devices is present in many applications. The Light Ethereum Subprotocol (LES) [8] is an extension to the Ethereum protocol to allow on-demand access to the distributed network while implementing countermeasures to detect manipulated block information. When the client comes online, it queries the latest block headers from multiple nodes on the network and verifies them. However, it does not receive the individual transactions and cannot determine the world state (e.g. values in smart contracts, account balances) on its own. This information is still requested from full nodes on demand, but the client is able to verify the authenticity with a so-called *proof of inclusion*, which matches the transaction data to the root-hash located in the block headers.

III. Related Work

Public Blockchains allow the free participation of users in the network. Distributed nodes support the network by verifying transactions and *mining* new blocks to append them to the ledger. However, operating a so-called *full node* requires a continuously running computer with high system requirements compared to IoT platforms (e.g. *Ethereum*: 500GB memory, 16GB RAM, 25 MBit/s network bandwidth [1]) and advanced knowledge of system and network administration. For occasional interactions with dApps and for simple transfers of cryptocurrency this may pose a large obstacle and prevents access from mobile devices entirely. Potential use cases for IoT devices can also not be realised with those requirements.

In practice, a hybrid approach is used to provide simple access for end-users, while keeping the network distributed: The backbone of the network is defined by the already mentioned *full nodes*. They are managed by organisations and enthusiast users for direct ledger access, transaction verification and optional block *mining* (by attempting to solve the PoW puzzle). End-users, mostly unconsciously, use *Blockchain Gateways* to access the current state and send transaction requests to the network.

One of the most popular gateway providers for the *Ethereum* Blockchain is *Infura* [9]. After registering to the service, users can send JavaScript Object Notation Remote Procedure Calls (JSON-RPCs) to query the world state, send transactions and deploy smart contracts. User authentication is used to enforce a transaction quota and implement a tiered pricing model. It can be used to access the official *Ethereum* chains, but does not support private and custom Blockchain networks.

The goal to develop a light client for Blockchain access was also investigated by [10] for a Smart City IoT application. It was solved by removing unused functions from the *Ethereum* client and preventing the local storage of chain data. The result is a minimal client installation which is able to read data from the public Ethereum network and send new transaction. In contrast to our design, this solution does not enable private transactions and fully trusts the gateway nodes for the operations. A malicious gateway is able to suppress transactions from the light client and respond with forged state information. Our design improves this idea by validating the received block headers and preserve the option of private transactions.

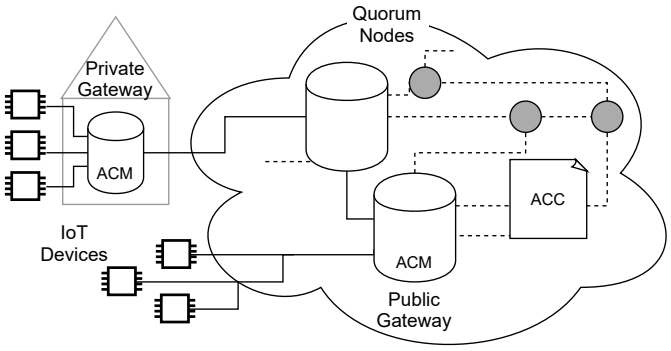

Fig. 1. Example network of Quorum nodes, Gateways and IoT devices.

The security properties of LES have been investigated in detail in [11]. Because the client receives only block announcements and headers, it cannot verify the state transitions caused by the included transactions. This limitation can be used by an adversary to deceive the client with seemingly correct blocks of a forked chain. Because the adversary is in control of the consensus for this unofficial chain, it can alter the state arbitrarily (e.g. change smart contract code and account balances). The authors identified detailed attack scenarios and proposed mitigation strategies, e.g. by communicating with a trusted third party.

A more advanced solution to protect against malicious forks was proposed in [12]. The client is no longer limited to request proofs only from trusted nodes, instead it can identify when the received information comes from a forked chain. The authors of [12] propose the concept of so-called *Beacon Transactions* to confirm to the client, that it is indeed receiving block information for the official chain. Those transactions are periodically added to the Blockchain by trusted nodes to verify the chain up to a certain point in the past. The forged chain of the adversary does not contain the beacons and it is also not able to produce valid copies without owning the private signing key of the trusted node.

A detailed survey about Blockchain integration for IoT is presented in [13]. One of the selected use cases is a data market for distributed sensor networks, similar to our example application. In addition to the improved authenticity and integrity properties of the Blockchain solution, the authors also discuss future applications of *Machine Economy* (e.g. devices can autonomously trade and pay for consumed services).

IV. Contribution and novel architecture

This section describes the contribution and main components of the proposed architecture, which are illustrated in Fig. 1. The example application uses IoT devices to build a network of distributed environmental sensors. They can either be connected to privately owned Gateways, or use a public Gateway Service to access the Blockchain.

A. Contribution

The related work examples show the significance of *Ethereum* and smart contracts for IoT networks. *Infura* and the proposal in [10] also demonstrate the practical implementation of Blockchain gateways and light clients. Our implementation uses the *Quorum* Blockchain which supports private transactions and efficient consensus algorithms. This novel gateway design for the *Quorum* Blockchain is needed for the practical application in consortium networks. It allows efficient operation of the backbone infrastructure with high transaction throughput, while preserving the support for private smart contracts and data validation by the client.

The main contributions of this paper are:

- The high-level architecture of a proposed *Blockchain Gateway* in Section IV, describing the main components and their interactions. This novel design enables lightweight devices to interact with a dApp deployed on the *Blockchain*.
- The design of the client features a hybrid approach by using the dedicated gateway for private transactions and LES to validate the public data of the Blockchain. The implementation also demonstrates the secured key storage and transaction signing with a Hardware Security Module (HSM) embedded in the client platform.
- A detailed description of the application protocol, illustrating the required transaction sequence for deployment of private *smart contracts*, as well as read/write interactions is given in Section V.
- Validation of the proposed scheme is done with a proof of concept implementation. The results in Section VII provide references for future adaptations and deployment in real-world applications.

B. Client Application

The client application offers all features required for the specific application domain, e.g. Graphical User Interface (GUI), interaction to sensors or control systems. In this document only the Blockchain related features are described. A new transaction might be triggered by an external event (e.g. user request, sensor limit reached) or created periodically. The main responsibilities of the client application are authentication to the gateway, encryption and transferring of the raw data to the server, as well as cryptographic signing of the Blockchain transactions. A HSM is used for key management and correct execution of the cryptographic operations for authentication and transaction signing.

In the simplest form, the client exclusively communicates with a statically configured gateway in a *client/server* model. However, platforms with adequate computational resources can optionally use the LES to confirm the validity of transactions by requesting proofs from other nodes in the network.

C. Access Control Manager

All interactions with *Quorum* and *Tessera* are filtered by the Access Control Manager (ACM). Firstly, it authorizes clients to access the restricted Application Programming Interface

(API) by verifying the JSON Web Token (JWT) attached to the request. This prevents unauthorized clients from using the special functions for private smart contracts.

The second step of the access control restricts read requests for private contract data. Because the *Tessera* transaction manager of the gateway handles private transactions on behalf of many clients, it is trusted with the unencrypted data of all those users (authenticity is still preserved with the cryptographic signature of the client). For this reason, the ACM must only allow access to contracts, which belong to the client. This is done by registering the owner in the Access Control Contract (ACC) when a contract is deployed (see next section for details), and verifying each request. Unprivileged clients only see the publicly available data (e.g. encrypted payload hashes) on the ledger.

D. Access Control Contract

For every private contract managed by the gateway, an entry in the ACC is created. It holds mappings of the contract address to the authorized users. For new contract deployments, the ACM creates this entry and registers the owner. An optional API subsequently enables the authorization for additional users, if concurrent access to a smart contract is required by the use case (e.g. to join a private sensor network or to temporarily allow a technician to access service records).

Because the data structure of the ACC is also defined as smart contract, the entries are implicitly synchronized to other *gateway nodes* by the Blockchain network. It enables simple deployment of multiple, synchronized gateways by independent consortium members for increased trust and availability. The client decides during contract deployment, which gateways are allowed to access the private data of the contract, other gateways can only query the associated access rights. The ACC is also a private smart contract to keep the enclosed data confidential between the gateway nodes and prevent other users from getting insights into the authorization structure.

E. Trust and threat model

The gateway requires access to the unencrypted data to process the transactions on behalf of the client and to validate the state transitions caused by incoming transactions. In an example, where two partners cooperatively use a smart contract, our architecture supports three possible configurations with varied trust requirements:

- Both parties own a gateway (e.g. two companies in a consortium): No additional trust requirements, because both sides handle the raw data in any case. Reading from the encrypted state can be done from the local instance without requiring the other gateway.
- One party owns a gateway (e.g. user interacting with a company): No additional trust requirements, if the provided gateway is used exclusively. However, this introduces the risk of a Denial of Service (DoS) by the gateway provider, because the client is fully dependent on this specific node for all operations. A trade-off with reduced confidentiality requirements can be made when additional gateway nodes are included in the private group as backup.
- No party owns a gateway (e.g. two users): The users can agree on a specific gateway node, used to deploy the smart contract and handle subsequent interactions. The same choice from the previous example also applies here, where a higher risk to confidentiality can be accepted to increase the availability.

The trust model regarding confidentiality is similar to a conventional *client/server* architecture, where the user needs to trust the service provider to honour the agreed Terms of service (ToS). If a higher degree of confidentiality is required by the use case, the proposed architecture can still be used when the involved parties deploy a dedicated gateway for handling the private data.

With the additional Blockchain infrastructure, the data is protected from unwanted modifications and enables traceable execution of smart contracts. In a traditional application, a malicious or faulty server might respond with incorrect data and fail to write received values to the database correctly. In our design, in contrast, the data is stored in a distributed manner and the client can always request proofs about the actual Blockchain state by using LES.

Another threat vector is the leakage of user credentials to allow an attacker to impersonate someone else. Our design minimizes the probability of such attacks by employing signature-based authentication with a HSM embedded in the client device. The user signs a fresh access token with limited validity (in our case 15 minutes) for every session. Tokens with longer validity are rejected by the ACM to contain the possible harm of a leaked token. Embedding an HSM into the IoT client device improves the resilience against attacks and enables the allocation of complex cryptography functions to dedicated hardware. This approach also removes the responsibility of managing the keys from the gateway because the client generates them independently.

V. The novel Transaction Sequence

This section gives a detailed description about the proposed transaction sequence. It follows the original protocol of *Quorum* with enhancements introduced by our protocol.

A. Authentication

All requests from the client contain a JWT for authentication to the gateway API. Instead of the usual application, where an authentication server signs this token to vouch for the client, our design uses the Blockchain key to self-sign the token by the client instead. The gateway can recover the public key from this signature and lookup the associated contract permissions from the ACC. The initial goal of JWT to include the permissions in the token is not possible with this design. Instead we mainly use it as standardized format to transmit the self-signed tokens and verify them with available tools. The expiration of the token is set to the duration of a typical session, which can be shorter than one minute for simple interactions.

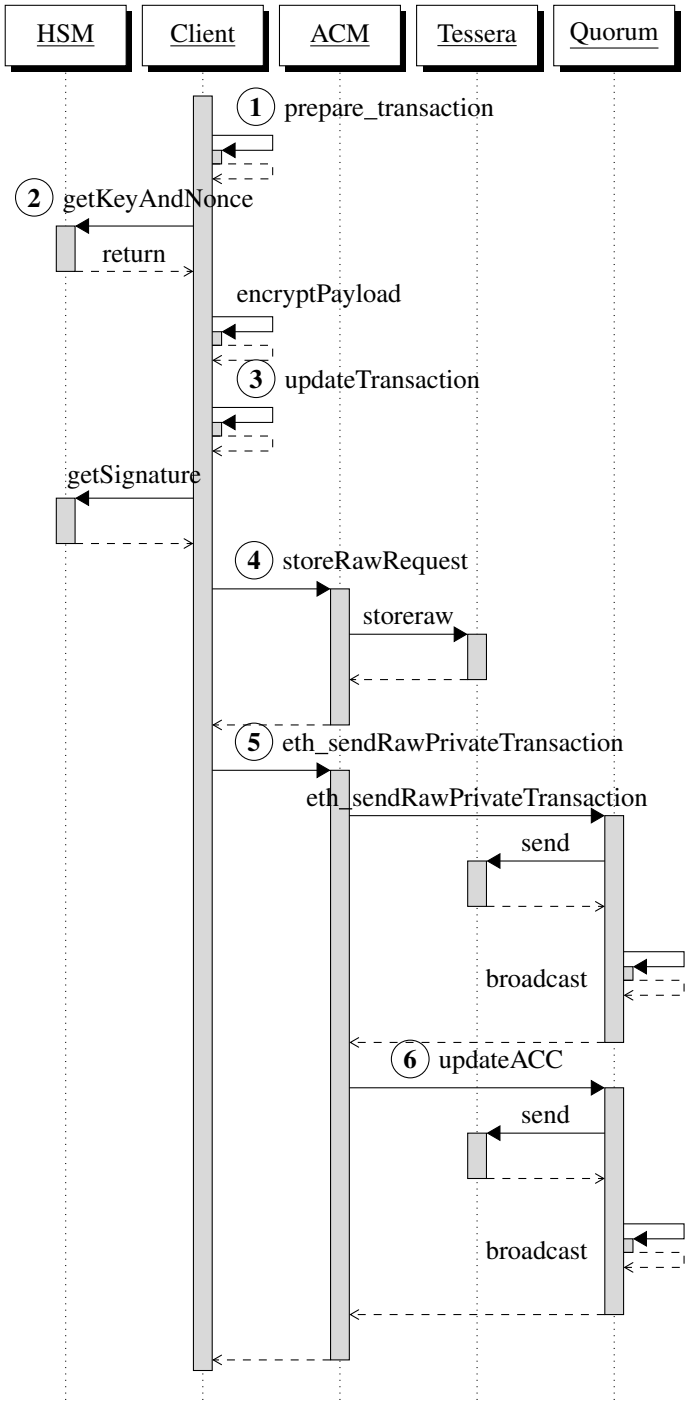

Fig. 2. Sequence for deployment of a private smart contract.

B. Contract deployment

The following steps, as illustrated in Fig. 2, are needed to deploy a private contract with a gateway. The sequence only describes the default case, where an authorized user sends a valid transaction.

1) **Prepare transaction:** The client compiles a smart contract or, as a simpler alternative, retrieves a pre-compiled definition stored in the application. The transaction must follow the structure defined by *Ethereum* [3], which holds the bytecode of the contract in the *'data'* field.
2) **Encrypt transaction payload:** The byte code contained in the *'data'* field is encrypted by the client. For this symmetric encryption with the *XSalsa20* [14] stream cipher, a fresh key and nonce is generated by the client (preferably with the support of the HSM to add high entropy).
3) **Update and sign transaction:** The payload of the original transaction is replaced by the *SHA3-512* [15] digest of the encrypted payload resulting from the previous step. The transaction is then signed with the Elliptic Curve Digital Signature Algorithm (ECDSA) [16] using the *secp256k1* [17] curve parameters.
4) **Store payload:** The encrypted payload, nonce and key from Step 2 are submitted to the gateway through a secured channel. The ACM verifies the user authorization and forwards the data to *Tessera*, if permitted. *Tessera* stores the key and nonce in its key-vault and the encrypted payload in the local database, with the *SHA3-512* digest as key. This identifier is also returned to the client, allowing a comparison to the locally computed value for verifying the integrity of data.
5) **Send transaction:** The signed transaction from Step 3 and the list of target nodes is send by the client. Based on the transaction header, the ACM can classify it as *private contract deployment*. It is forwarded to *Quorum*, where the payload hash and private targets are internally passed to *Tessera*. This request triggers the distribution of the encrypted payload and the symmetric encryption key to the target nodes, who also store it in their private database. Only after *Tessera* confirms the multi-cast, *Quorum* commits the transaction to the Blockchain.
6) **Contract construction:** Once the transaction is included in a block, the contract is fully deployed. With the *transaction receipt* returned by *Quorum*, the ACM can resolve the address of the new contract and inserts a mapping between contract and user address to the ACC. The client can verify the correct deployment by requesting a confirmation from LES.

C. Contract write operation

After deployment, the owner (or any other authorized user) can interact with the contract. A write access requires the sending of a new transaction, similar to the deployment sequence. The identical steps from the previous operation are omitted in Fig. 3.

7) **Prepare transaction:** The transaction payload for writing to a contract follows the specific format of *Ethereum*. It includes part of the function name, as defined in the ABI, and the parameter values.
8) **Encrypt/update/sign transaction, authenticate, store payload:** Steps 2-4 from the *deployment* sequence are identically repeated. *Tessera* stores the encrypted payload, key and nonce locally at this stage. From receiving

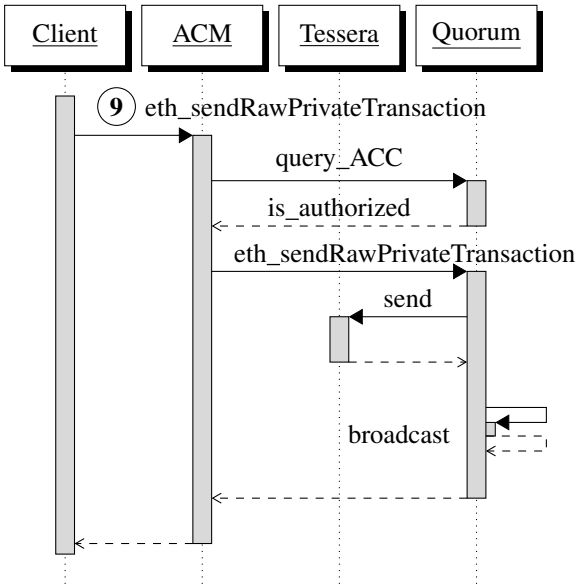

Fig. 3. Sequence for writing to a private smart contract.

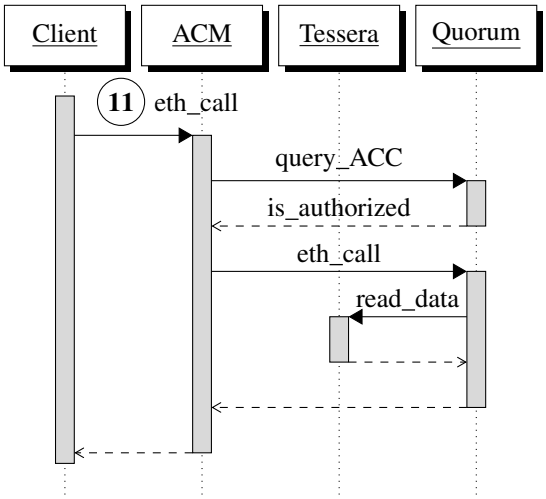

Fig. 4. Sequence for reading from a private smart contract.

the payload alone, the gateway cannot identify the target contract of the transaction and perform any access control. This fact does not pose a significant security risk, because the global Blockchain state is not modified until the remaining information is received in the next step.

9) **Send transaction:** The signed Blockchain transaction is intercepted by the ACM, where the transaction header provides the required information for the permission check executed by the ACC. If the user is authorized, the signed transaction is forwarded to the *Quorum* instance and *Tessera* distributes the private payload. After the transaction is included in a block on the ledger, the state of all nodes is updated with the new value. The client can again verify the correct distribution of the transaction by requesting a confirmation from LES.

D. Contract read operation

Reading from the Blockchain, i.e. calling a contract function which does not modify the ledger state and only returns values, does not broadcast the transaction to the network. The request is executed locally in the *Quorum* instance of the gateway node to retrieve the required data from the state storage. Our scheme must additionally prevent unauthorized users from reading private contract values.

10) **Prepare transaction:** A transaction to call a function is created in the same way as before, where the function identifier and the parameters are encoded in the payload data. However, the payload is not encrypted and this transaction is not signed by the user.
11) **Process transaction:** The ACM retrieves the target address from the transaction and the user identification from the JWT to query the ACC. The need for an additional authentication scheme becomes apparent in this sequence. Whereas deployment and write transactions require cryptographically signed messages, read-only calls are anonymous in the default scheme. In our design, the public key of the client can be recovered by the JWT attached to the request. If the user has the necessary access rights, the request is forwarded to *Quorum*. The private state of the contract is retrieved from the encrypted database of *Tessera* to call the requested function and return the result to the client.
12) **Verify return value:** Because the contract state is private, LES cannot directly be used to confirm the authenticity of the return value. The proposed solution requires a special function to be implemented in the smart contract: The previously received value (or the hash of multiple values) is again passed to this function, which compares it to the actual state. An error is returned if the comparison fails, reverting the transaction. Otherwise the function finishes successfully and a receipt is broadcasted. The client can check a previously received value by calling this function and verifying the resulting transaction with LES.

VI. Implementation

The proposed design was verified by a proof-of-concept implementation of the required components. The client application, ACM and ACC were developed from scratch; we modified *Tessera* to support our API extensions and a modified version of *Quorum* was compiled to allow it to launch as light client. A local network with three validating nodes was deployed, whereof one instance hosted the modified transaction manager and the ACM service.

A. Client implementation

The client application is implemented as *Python* program, offering a GUI to simplify the interaction with the test network and perform test requests. Key management and signing of the authentication and transactions was solved by interacting

with the *"Blockchain Security 2 Go Starter Kit"* [18], a smartcard developed by *Infineon Technologies* offering ECDSA with the *secp256k1* curve parameters. This was done to prove the feasibility of using an external HSM for increasing the security properties. It could also simplify the upgrade of a legacy system, where it is not trivial to add the ECDSA functions and secured key management in software.

B. Gateway implementation

The ACM includes a web server, responsible for handling the requests from the clients and forwarding them to *Quorum* and *Tessera*. It mirrors the JSON-RPC format defined by *Quorum*, to preserve compatibility with third-party tools and libraries. Requests for the *eth_sendRawPrivateTransaction* and *eth_call* JSON-RPCs are filtered with the help of the ACC. Only the publicly available data is return to the client if the verification fails. The ACC offers functions for adding and verifying contract permissions for the authenticated user. The contract owner can add additional authorized keys to share a private contract with other users. Future implementations can also offer permissions with time or usage limit and read-only access.

The *Tessera* API was modified to support the storing operation as described in V-B.4. Originally, the raw payload is provided by the client and the hash of the encrypted payload is returned as payload for the Blockchain transaction. This would open an attack vector, because the encryption is done in Tessera and the client cannot verify this hash. It would need to blindly sign the returned hash, allowing a dishonest gateway to arbitrarily change the transaction beforehand. Our modification of Tessera allows the client to supply the symmetric encryption key and nonce, allowing it to contribute entropy from a HSM and independently calculate the hash of the encrypted payload.

VII. Evaluation

The proof-of-concept implementation confirms the feasibility of the proposed scheme. Although the client application was not optimized, it gives a good indication about the additional system requirements when Blockchain interaction is added to an existing IoT application. Because computational- and memory requirements are dependent on the target platform, this evaluation focuses on the security aspects and constant communication overhead of the design.

A. Security aspects

The main motivation to replace a traditional client/server architecture with a Blockchain-based dApp is the increased level of data authenticity. As long as the private key is not leaked, a signed transaction originated from the genuine client with high probability. Our design does not weaken the correctness claims of *Quorum*/*Ethereum*, allowing us to disregard the possibility of successful attacks on the ledger for this evaluation.

The main threat in our design comes from a malicious or faulty gateway, which isolates the client with a forked chain and responds with manipulated data. This attack vector is weakened by providing multiple independent gateways and supporting additional validation by the client with LES.

The secp256k1 ECDSA key, which is used for signing the Blockchain transactions and authentication, requires secured storage (all other keys are only used for single sessions). It should be generated by the end-user with an HSM to have a strong non-repudiation claim. If pre-initialized devices are provided to the user, an attacker (or dishonest issuer) could sign a large quantity of deterministic Blockchain transactions and authentication tokens in advance, and publish them once an attack has the desired effect.

One apparent trade-off caused by the gateway design is the partially compromised confidentiality of private smart contracts. The gateway requires the plain data to distribute it and verify incoming state transitions on behalf of the users. They must trust the gateway to keep their data secret or deploy a dedicated gateway. However, we do not see this as large disadvantage for many applications, as long as multiple certified gateways exist. If clients want to consume a service, they have to reveal the underlying data in any case.

B. Communication overhead

The lightweight design of our proposal makes it a good candidate for mobile- and IoT applications, where connectivity can be limited. This part of the evaluation focuses on the communication overhead of our extended Blockchain protocol for interactions with private smart contracts. We do not consider the overhead of low-level transaction protocols and Transport Layer Security (TLS), because it is also present for traditional applications in a client/server model.

The most commonly used operations are contract deployment, reading and writing of data. The transaction data is consisting of the *Ethereum* header and payload data, encrypted private data and symmetric key, as well as authentication tokens. Figure 5 compares the transaction sizes for the common operations.

The example illustrates the deployment of a smart contract with a raw payload of 658 byte. The deployment operation has a high payload ratio and is usually only performed rarely by the client. Disregarding the authorization token, read operations have the same size for private and public contracts, because the same JSON-RPC from *Ethereum* is used. Writing the same 32 byte data to a private contract nearly triples the transactions size, because this operation requires two transactions when done privately: The encrypted payload is firstly committed to *Tessera* together with the symmetric encryption key and nonce. Afterwards, the 64 byte identifier and the selected recipients are committed by the Blockchain transaction. One-third of the size is the result of adding the authorization token for both operations.

The evaluation shows the high transaction overhead caused by authorizing every private transaction with a JWT. It allows the stateless authorization of each request and removes the need for managing client sessions, at the cost of increasing the size. For applications with short payload sizes and frequent

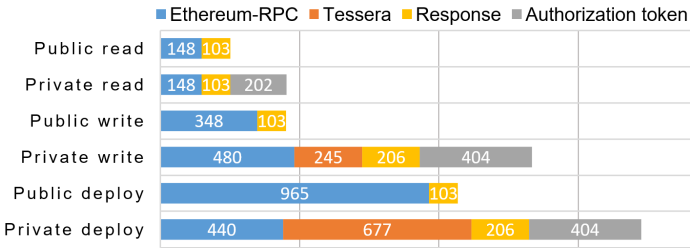

Fig. 5. Total transaction sizes (in byte) when writing a 32 byte payload or deploying a 658 byte contract.

interactions, this might exceed the bandwidth limit and shorter session keys should be used instead.

C. Platform considerations

Other performance parameters are dependent on the specific implementation of the client application. Our test implementation is developed with *Python* and executed on a *Raspberry Pi 3 Model B+*. The client application requires 35 MB of RAM and demonstrated a full contract deployment in under 4 seconds.

For using the optional transaction validation with LES, the full *Geth* binary is currently deployed to the client. Due to the large amount of included features (e.g. transaction verification, block mining, management of the state database) the binary has a size of 47 MB when compiled for the ARMv8-A architecture. It is expected that this size can be reduced similar to the results of [10], where the required code size for a minimal deployment of the *Geth Client* was decreased by more than 80%.

Signature based authentication and validation with LES are optional features to reduce the trust requirements to the gateway. They can be removed if a trusted gateway is available in the local network to further reduce the system requirements of the client platform.

VIII. Conclusion and Future Work

This paper presents a lightweight architecture to host *gateway nodes* for interactions with the *Quorum* Blockchain, and describes the necessary transaction sequence in detail. The proposed client implementation allows mobile and resource-constrained IoT-devices to be used as interface for dApps built from *private smart contracts*. The gateway is fully compatible to the default implementation of *Quorum* and does not change the internal structure of the Blockchain network, allowing the installation on already deployed networks. Data authenticity is preserved with signatures and the client can optionally use LES to request proofs for the information received from the gateway. This can even be used to confirm the internal state of private smart contracts with publicly available information.

To the best of our knowledge, the presented architecture is a novel approach to enhance the flexibility of the *Quorum* Blockchain. The evaluation confirmed that the transaction format only adds a constant size overhead to the requests and the security level of the client is improved by introducing a dedicated HSM for key management and efficient execution of the cryptographic primitives. A traditional IoT application could therefore be ported to a dApp without increasing the performance requirements of the system. The flexible design allows the client, at the time of contract deployment, to select the security parameters based on the capabilities of the platform, tolerable communication overhead and security requirements of the application.

Future research will focus on the further application of HSMs for secured key exchange and privacy-preserving authentication schemes. Additionally, attestation of the gateway software and *trusted computing* techniques will be evaluated to reduce the trust requirements and prevent the leakage of confidential data by malicious or faulty gateways.

References

[1] (2021) Ethereum developer resources: Nodes and clients. [Online]. Available: https://ethereum.org/en/developers/docs/nodes-and-clients/
[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 03 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf
[3] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf
[4] "Quorum Whitepaper," 2018. [Online]. Available: https://github.com/ConsenSys/quorum/raw/360d7a8ad8ef51a89d4d4af44aee333ad4cc0e6a/docs/Quorum\%20Whitepaper\%20v0.2.pdf
[5] go-ethereum Authors. (2021) Go ethereum project website. [Online]. Available: https://geth.ethereum.org/
[6] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 305–319.
[7] H. Moniz, "The Istanbul BFT Consensus Algorithm," 2020, _eprint: 2002.03613. [Online]. Available: https://arxiv.org/abs/2002.03613
[8] (2021) Light ethereum subprotocol (les). [Online]. Available: https://github.com/ethereum/devp2p/blob/master/caps/les.md
[9] Infura Inc. (2021) Infura project website. [Online]. Available: https://infura.io/product/ethereum
[10] E. Reilly, M. Maloney, M. Siegel, and G. Falco, "A smart city iot integrity-first communication protocol via an ethereum blockchain light client," in *Proceedings of the International Workshop on Software Engineering Research and Practices for the Internet of Things (SERP4IoT 2019), Marrakech, Morocco*, 2019, pp. 15–19.
[11] S. Paavolainen and C. Carr, "Security Properties of Light Clients on the Ethereum Blockchain," *IEEE Access*, vol. 8, pp. 124 339–124 358, 2020.
[12] S. Paavolainen and P. Nikander, "Decentralized Beacons: Attesting the Ground Truth of Blockchain State for Constrained IoT Devices," in *2019 Global IoT Summit (GIoTS)*, 2019, pp. 1–6.
[13] A. Panarello, N. Tapas, G. Merlino, F. Longo, and A. Puliafito, "Blockchain and IoT Integration: A Systematic Survey," *Sensors (Basel, Switzerland)*, vol. 18, no. 8, p. 2575, 2018, place: Switzerland Publisher: MDPI AG.
[14] D. J. Bernstein, "Extending the salsa20 nonce," in *Workshop record of Symmetric Key Encryption Workshop*, vol. 2011. Citeseer, 2011.
[15] M. Dworkin, "Sha-3 standard: Permutation-based hash and extendable-output functions," 2015-08-04 2015.
[16] E. Barker, "Digital signature standard (dss)," 2013-07-19 2013.
[17] S. E. C. Certicom, "Sec 2: Recommended elliptic curve domain parameters," *Proceeding of Standards for Efficient Cryptography*, 2010. [Online]. Available: http://www.secg.org/sec2-v2.pdf
[18] Infineon Technologies AG, "Blockchain Security 2Go Starter Kit - User Manual," 2019. [Online]. Available: https://github.com/Infineon/Blockchain/blob/master/doc/BlockchainSecurity2Go_UserManual.pdf