Execution of Smart Contracts with ARM TrustZone

Designing and Implementing a Prototype for Hyperledger Fabric Chaincode Execution with OP-TEE

Master Thesis

Christina Müller

Faculty of Science, University of Bern

August 2019

Prof Dr Pascal Felber
Prof Dr Christian Cachin
Dr Valerio Schiavoni
Marcus Brandenburger

Abstract

Internet of Things (IoT) and smart contracts (traditional contracts translated into program code) are two current technologies with new opportunities: For example, by using IoT and smart contracts, we can track assets along the supply chain in a verifiable and efficient way.

Smart contracts are integrated in a blockchain network and hence distributed on many nodes. Currently, there is no confidentiality guarantee for smart contracts (their logic) and the processed data. The usage of a Trusted Execution Environment (TEE) - an isolated processing environment - is one approach to protect sensitive smart contract execution.

We design and implement a prototype called Fabric OP-TEE Chaincode (FOC) for Hyperledger Fabric (a platform for permissioned blockchain) chaincode (smart contract) execution with ARM TrustZone (ARM TZ). We use OP-TEE as firmware and software on top of ARM TZ. Our design is based on Fabric Private Chaincode (FPC) which uses Intel SGX as underlying TEE technology. ARM TZ and Intel SGX are two well known TEEs.

To demonstrate chaincode execution with FOC, we implement a coffee tracking chaincode which registers, updates and queries the coffee consumption of different people. By only running the coffee tracking chaincode inside the secure world (isolated and therefore trusted part of OP-TEE) and not the whole peer (Hyperledger Fabric node which executes and validates transactions), we successfully minimize the trusted computing base.

Since there is no native Docker support in the normal world (untrusted part) of OP-TEE, the peer is decoupled from the ARM TZ node. Communication between a *chaincode_wrapper* at the peer (FOC specific wrapper around the actual chaincode) and a *chaincode* inside the secure world is enabled via gRPC (framework for remote procedure calls) and through the API provided by OP-TEE between the normal and secure world.

In contrast to Intel SGX, ARM TZ and OP-TEE do not support remote attestation. Furthermore, hardware support for some security features of OP-TEE is missing. Therefore, FOC does not have the same guarantees as FPC. To fully protect sensitive smart contracts (their logic) and the processed data, future work would be necessary.

Performance measurements of the implemented FOC prototype highlight the overhead (decrease in throughput by factor > 27 for ARM TZ with OP-TEE running on the Raspberry Pi) by executing the smart contracts in the secure world of OP-TEE compared to the execution in the normal world.

Supervisors

Prof Dr Pascal Felber, Complex Systems, Computer Science Department (IIUN), University of Neuchâtel

Prof Dr Christian Cachin, Cryptology and Data Security Group, Institute of Computer Science, University of Bern

Advisors

Dr Valerio Schiavoni, Complex Systems, Computer Science Department (IIUN), University of Neuchâtel

Marcus Brandenburger, IBM Research Zurich

Acknowledgement

I am grateful to...

- Prof Dr Pascal Felber, University of Neuchâtel for initiating this master thesis and for his technical support as supervisor.
- Prof Dr Christian Cachin, University of Bern for his technical support as supervisor.
- Dr Valerio Schiavoni, University of Neuchâtel and Marcus Brandenburg, IBM Research Zurich for their technical support as advisors.
- PhD student Christian Göttel, University of Neuchâtel for his advice regarding OP-TEE and ARM TZ.
- Luca Liechti for proofreading this master thesis.
- My family and my boyfriend Pirmin Herger for their mental support.

Thank you!

Contents

1 Introduction

Internet of Things (IoT) and smart contracts are two current technologies with new opportunities: IoT enables objects to connect and to exchange collected data, smart contracts can control the behaviour of these objects [Tit18]. For example, by using IoT and smart contracts, we can track assets along the supply chain in a verifiable and efficient way [CD16a].

Smart contracts are integrated in a blockchain network [XWS+17]. A blockchain stores blocks of data in a decentralized way and connects them ("chain") via hashes. Due to the blockchain characteristics, smart contract applications guarantee the second and the third concept of the CIA (Confidentiality, Integrity, Availability) information security triad [And11]. What we are concerned with is the C: confidentiality of the smart contract (their logic) and the processed data.

The usage of a Trusted Execution Environment (TEE) [SAB15] is one approach to protect the smart contract (its logic) and the processed data. A TEE isolates the execution of the smart contract from the surrounding system.

One project that makes use of a TEE for smart contract execution is Hyperledger Fabric Private Chaincode (FPC) [BCKS18, fpca]. The underlying TEE technology used in FPC is Intel SGX [CD16b]; smart contract execution is therefore bound to nodes with Intel processors. Since IoT devices are mostly small, battery-powered and equipped with low-power processors [Gar14, Dub19, CD16b], FPC cannot be applied here. In order to guarantee confidentiality of smart contract execution in IoT networks, this thesis explores the execution of smart contracts with ARM TZ [Lim09] - a TEE solution for embedded systems [aD].

Contribution. We design and implement a prototype for Hyperledger Fabric [Hyp] chaincode execution with ARM TZ and OP-TEE [Lin] called Fabric OP-TEE Chaincode. The design is based on FPC.

Organization. In Chapter 2, we introduce the concepts and technologies used in this thesis. The goals are given in Chapter 3. We present the design and implementation of FOC in Chapter 4, the decisions are justified in Chapter 5. We evaluate security of FOC and performance and energy usage of the implemented prototype in Chapter 6. In Chapter 7, we conclude this thesis and describe possible future work.

2 Background

In this Chapter, background knowledge about the principles underlying smart contract execution in TEEs is given. We cannot completely describe all technologies and concepts but we will focus on the definitions relevant for the thesis and used in the following Chapters of report.

We also briefly describe some related work.

2.1 Blockchain

A blockchain is a type of distributed ledger [XWS+17]. It records data (e.g. transactions) in a decentralized way. The data is appended in blocks and connected (”chained”) via hashes. Each transaction is signed by the party it was invoked. Before appending transactions to a blockchain, nodes of the blockchain network must validate and agree on a unique order of these transactions. The latter can be achieved via a consensus mechanism. Due to its characteristics, a blockchain guarantees availability, transparency, immutability and integrity of the stored data. Data privacy and scalability are rather limited in blockchains.

We distinguish between the following two types of blockchains: permissionless (also called public) and permissioned [CV17]. In a public blockchain (e.g. Bitcoin [bit] and Ethereum [eth]), anyone can participate in the network (read data from the blockchain, invoke transactions, validate transactions etc.), whereas in a permissioned blockchain (e.g. Ripple [rip] and Hyperledger Fabric [ABB+18]), the access of the network is restricted and entities are known.

2.2 Smart Contract

The concept of smart contracts was described by Nick Szabo already in the nineties [Sza94]:

”A smart contract is a computerized transaction protocol that executes the terms of a contract.”

With the emergence of blockchain technology, this idea could be put into practice: A smart contract is an agreement translated into program code and stored in a blockchain network [RMC+18, Ose18] . It gets automatically executed when the defined conditions are met. Due to their integration into a blockchain network, smart contracts guarantee availability, transparency, immutability and integrity [XWS+17]. Furthermore, they are efficient, reduce cost and save time because they do not involve a third party system.

However, smart contracts can also be challenging, e.g. how to guarantee data privacy, how to avoid bugs in the contract code or how to prevent attacks (see for example [ABC17] for attacks on Ethereum smart contracts).

2.3 Hyperledger Fabric

Hyperledger Fabric [ABB+18, Hyp] is a permissioned blockchain supporting smart contracts. It belongs to the Hyperledger [Fou] - an open source, Linux Foundation project which includes different frameworks and tools related to blockchain technologies.

In Hyperledger Fabric, a smart contract is called chaincode. Currently, three general-purpose programming languages (Go, Java and Node.js) are supported for writing chaincodes.

There exist three types of nodes in a Hyperledger Fabric network: clients, peers and orderers, see Figure 2.1

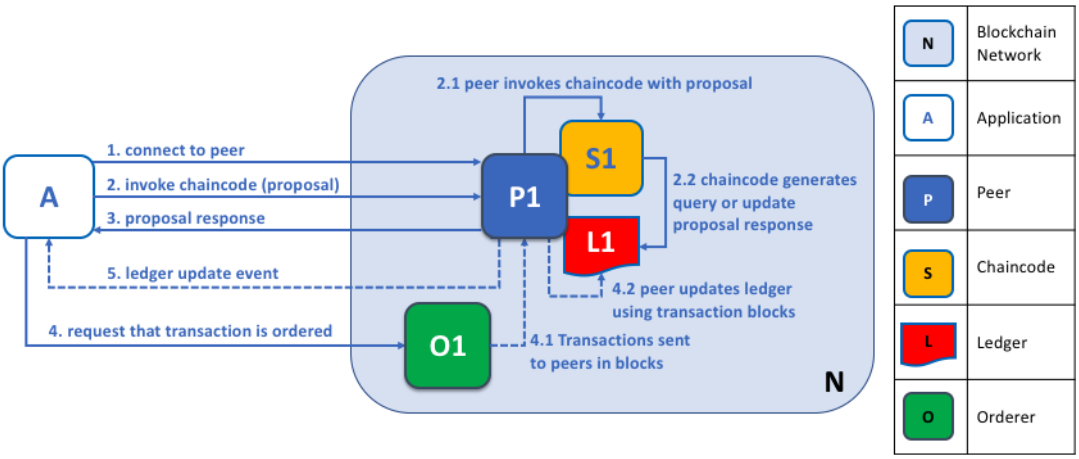

Figure 2.1: Architecture of Hyperledger Fabric.
Source: Figure copied from [Hyp]

Before a chaincode function can get called (invoked), it must be installed (put on the file system) and instantiated (started in a Docker container) at the peer. Now, a client (application) can send a request (transaction proposal) to the peers for invoking a chaincode function (see 1. and 2. in Figure 2.1). In a first phase, called execution or endorsement, the peer executes the called chaincode function (2.1, 2.2) and sends a response (transaction response / (transaction) proposal response) back to the client (3.). The transaction response is signed by the peer and contains the execution response message and the readset and writeset. The readset represents all values of the keys a peer has queried from the ledger via `GetState` during the execution. The writeset contains all key-value pair updates a peer has generated via `PutState`. When the client has collected enough responses as defined by the so called endorsement policy it sends them to the orderer (4.). The orderer puts the transaction into blocks and sends the block to the peers (4.1), this phase is called ordering phase. In the third phase (validation phase), the peers check if the endorsement policy is satisfied and if there is no read-write conflict between the different transactions. Finally, they put the transaction on the ledger (4.2). The ledger has two components: a blockchain and a world state. The world state is a pluggable database. It stores the current values of the keys contained in the blockchain and therefore enables efficient retrieval of the latest states.

2.4 Trusted Execution Environment (TEE)

A Trusted Execution Environment (TEE) is a concept for tamper-resistant and confidential execution of applications [SAB15]. It can be defined as an isolated processing environment on a system. The isolation from the rest of the system is enabled through hardware, firmware and software mechanisms.

When the TEE gets booted/loaded, a trusted, tamper-resistant hardware module integrity checks the TEE code and stops the booting/loading in case the TEE has been modified. At runtime, the TEE ensures integrity and confidentiality of code and data (runtime states). Communication between the TEE and the rest of the system is enabled through a secure interface (memory isolation etc.).

A TEE may offer secure storage and remote attestation. Secure storage means that data produced and used by a TEE can be persistently stored with confidentiality, integrity and freshness guarantee. Remote attestation is used to authenticate a TEE towards a third party.

Intel SGX and ARM TZ are two examples for TEE [SAG+16].

2.4.1 Intel's Software Guard Extensions (Intel SGX)

Intel's Software Guard Extensions (SGX) [CD16b] adds necessary hardware and software components to Intel CPUs (Skylake and onwards[1]) for enabling TEE [SAG+16]. With Intel SGX, multiple TEEs - so called enclaves - can exist per system.

There is a hash check during the initialization of an enclave to ensure integrity of the enclave code [Sel16]. Special SGX enclave instructions like `EENTER` and `EEXIT` are used to enter and leave an enclave. Code inside the enclave as well as code running outside the enclave has access to all system resources with exception of the memory [NMB+16]. Each enclave has assigned a memory region - so called Enclave Page Cache (EPC) pages - to store the enclave code and data. The CPU avoids that any non-enclave code accesses the EPC. To further avoid data leaking, system calls and service of interrupts or faults are not possible from inside an enclave since they would require a call to untrusted code [iDZ16]. Therefore, the enclave must first be exited before system calls can be executed and interrupts or faults can be served.

2.4.2 ARM TrustZone (ARM TZ) and OP-TEE

ARM TrustZone (ARM TZ) [Lim09] provides the hardware components for enabling TEEs on ARM processors [SAG+16, aD]. There are different TEEs which add firmware and software on top of ARM TZ; one open source solution is OP-TEE [Lin], currently owned and maintained by Linaro. OP-TEE follows the TEE architecture and API standardized by GlobalPlatform (GP) [gp-].

ARM TZ. ARM TZ enables a single TEE - called secure world - per system [NMB+16]. The other part of the system is called normal world. The processor can be in one of two security states: secure (for the secure world) and non-secure (for the normal world); switching is possible through so called secure monitor calls. The system resources are strictly separated: The normal world cannot access the resources (memory, peripherals etc.) reserved for the secure world. During the booting of the secure world, a chain of trust is established and there is an integrity check of the secure world software images. This process is called secure boot.

OP-TEE. OP-TEE contains the following components: OP-TEE Client, OP-TEE Linux driver and OP-TEE OS, see Figure 2.2.

[1] `https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/606636`, last accessed on 17.08.2019

The OS of the normal world is also called Rich Execution Environment (REE). The OP-TEE Linux driver provides the driver for the normal world. An application running inside the normal world is referred to as host application. The TEE Client API and the TEE Internal API enable the communication between a host application and an application of the secure world - called trusted application (TA). Both APIs are defined by GP [Glo10, Glo14], the TEE Client API is implemented by the OP-TEE Client component, the TEE Internal API is implemented by the OP-TEE OS.

Before communicating, the host application must establish a connection towards the secure world with the TEE Client APIs `TEEC_InitializeContext` and open a session towards the TA by calling the TEE Client APIs `TEEC_OpenSession` with the unique identifier of the TA (`UUID`) as parameter. Then, the host application can call functions of the TA with the TEE Client APIs `TEEC_InvokeCommand`. `TEEC_InvokeCommand` provides the option to pass data between the host application and the TA via shared memory reference or by value.

Once the host application has finished communication with the TA, it needs to close the session (`TEEC_CloseSession`) and finalize the context (`TEEC_FinalizeContext`) to release resources.

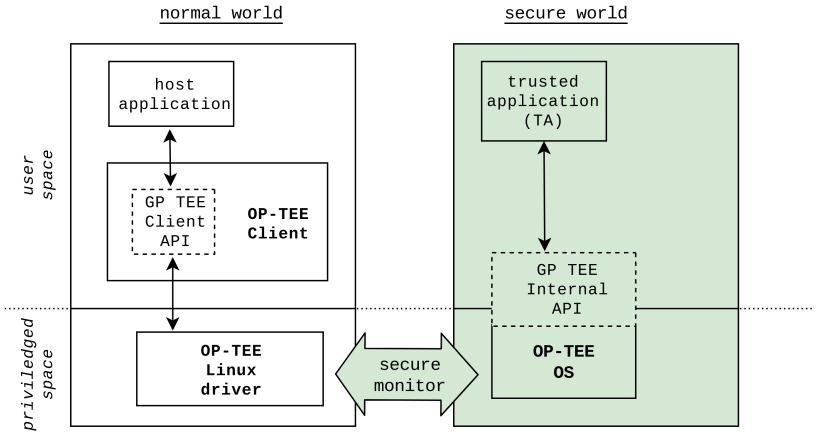

Figure 2.2: Architecture of OP-TEE. Trusted parts are colored in green.
Source: figure drawn by C.M. based on the figures in [Bec14, Lin16]

For this thesis, the availability of remote attestation is one of the most important difference between Intel SGX and ARM TZ with OP-TEE: ARM TZ and OP-TEE do not provide any remote attestation whereas Intel SGX does [Sta18, ogi19c, CD16b].

2.5 Hyperledger Fabric Private Chaincode (FPC)

Hyperledger Fabric Private Chaincode (FPC) [BCKS18, fpca] is a technology based on Hyperledger Fabric which aims to isolate the chaincode execution from potentially untrusted peers by using the Intel SGX technology. FPC adds two SGX enclaves to the architecture of Hyperledger Fabric:

- **Chaincode enclave.** Used to isolate the chaincode execution from the peer.
- **Ledger enclave.** Used to ensure state continuity, i.e. that the state retrieved by the chaincode enclave is correct (integrity-protected and consistent).[2]

[2]FPC guarantees security up to resets, see [BCKS18, Def. 4.1, Sec. 6]. Rollback attacks in the sense that an attacker sets the ledger state to any past state and executes one transaction on top of that stale state are still possible [BCKS18, Sec. 4.2]. There is a concept called barriers, see [BCKS18, Sec. 4.4], to avoid such speculative transaction execution on top of any valid (stale) state.

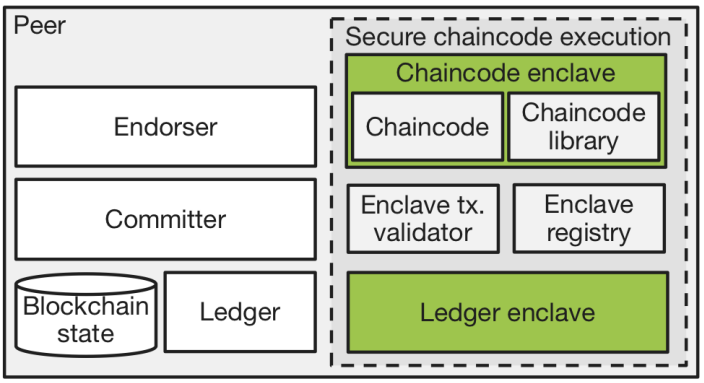

Figure 2.3: Architecture of FPC. The dashed box contains the components added by FPC to the Hyperledger Fabric peer. The SGX enclaves (i.e. chaincode enclave and ledger enclave) are colored in green.
Source: Figure copied from [fpca]

During execution phase, the chaincode enclave is used. This guarantees integrity and confidentiality at runtime so the peer cannot see nor modify the computations. The ledger enclave stores the hashes of the latest key-value pairs. Whenever the chaincode enclave queries the ledger state (for example in case of a `GetState`), it also fetches the according hashes from the ledger enclave. By comparing these two information, the chaincode enclave can ensure the correctness of the state.

During validation phase, in addition to the checks of Hyperledger Fabric (endorsement policy and read-write conflict check), the chaincode enclave signatures are verified by a component called enclave transaction validator. Furthermore, the ledger enclave cross-checks all decisions and stores a hash of the key-value pairs.

Additionally, FPC has the following security features:

- **Integrity check of the chaincode enclave code at its load time**. To ensure that the chaincode has not been manipulated.
- **Remote attestation and signature of the transaction response**[3] **by the chaincode enclave**. So the client can ensure that the transaction proposal gets executed by an authorized chaincode enclave and the committing peer can verify that the transaction response actually originates from an authorized unmodified chaincode enclave.
- **Encryption of the operation** (function and arguments [fpcb]). By the client with the chaincode enclave's public key to guarantee confidentiality.
- **Optional encryption of the passed states, the execution result and the chaincode.** To guarantee full confidentiality of the chaincode (its logic) and the processed data.
- **Trusted state transfer.** To securely update the local ledger state of a peer who has been cut off the blockchain for a while or joins the blockchain for the first time.

[3] In FPC, the response of the chaincode enclave after transaction execution contains the operation (function and arguments [fpcb]), the readset and writeset and the execution result [BCKS18, fpcc]. In order to be consistent with the Hyperledger Fabric documentation [Hyp], we will use the term execution response message instead of execution result.

2.6 Related Work

In this section, we point to some works which share technologies and concepts we have used.

Smart contract execution with a TEE. Apart from FPC, there are some other works about confidential smart contract execution with a TEE: Confidential Consortium Framework (CCF) [Res], Ekiden [CZK+18], ShadowEth [YXC+18] and Private Data Objects (PDOs) [BMSV18], just to mention a few recent once. In contrast to FOC, they all use Intel SGX as underlying TEE technology for a (prototype) implementation. The authors of Ekiden state that their technology may use any TEE which is similar to Intel SGX and supports attestation.

Confidentiality in context of blockchain and IoT. In FOC, we are concerned with confidentiality of smart contracts (their logic) and data in context of IoT networks. There are some works which are concerned with confidentiality of data produced and processed by IoT devices and stored on the blockchain.

- *Trust for data generated by IoT devices.* AnyLedger [DP] is a platform for connecting physical devices to the blockchain. The key feature is an ARM TZ based wallet for IoT devices. The key generation, the private key storage and the process of signing (smart contract) transactions are all placed inside the secure world of ARM TZ. Hence, the AnyLedger wallet guarantees that the IoT data hash/address (linking to the Interplanetary File System IPFS) placed on the blockchain is integrity protected and authenticated. Furthermore, the data stored on the IPFS is encrypted. AnyLedger is pluggable to any blockchain technology (for example Ethereum or Bitcoin). [SDF+19] is another work which equips IoT devices with a TEE to guarantee integrity and confidentiality of the IoT data.
- *Confidential computation.* BeeKeeper 2.0 [ZWAS18] is a blockchain network for IoT systems; it consists of IoT devices, servers and validator nodes. It enables IoT devices to share data with each other. Furthermore, the devices can use the servers for performing homomorphic computations on encrypted data; the computation result is verified by the validator nodes and recorded on the blockchain after successful verification. Since homomorphic encryption is used, the confidentiality of the data sent to and processed by the servers is guaranteed. BeeKeeper 2.0 can be added on top of any blockchain technology (Hyperledger Fabric, Ethereum etc.); in the paper, the authors use Hyperledger Fabric for deployment.

ARM TZ and blockchain technology. There are some works which use ARM TZ not for smart contracts directly but in context of the blockchain technology (which is the underlying technology of smart contracts).

SBLWT (Secure Blockchain Lightweight Wallet) [DDW+18] uses ARM TZ to guarantee confidentiality and integrity for the information generated and stored in the Bitcoin wallet (wallet's private key, wallet addresses, block headers used for Simplified Payment Verification). The synchronization of the block headers and the verification process of the transactions is executed in the secure world to avoid any manipulation by an attacker. The SBLWT is safer than the often used software-wallets but still more portable than hardware-wallets. An implementation of SBLWT using ARM TZ with OP-TEE has been deployed on Raspberry Pi 3 Model B. In future work, the authors of SBLWT want to extend their approach to mobile devices with hardware-based isolation mechanisms other than ARM TZ.

The TrustZone-backed Bitcoin Wallet [GMS17] also uses the ARM TZ technology for protecting sensitive Bitcoin wallet information.

Other use cases of TEEs. There are other use cases of TEEs in context of blockchain technology not mentioned yet. We briefly give two examples. First, Teechain [LNE+18] and Airtnt [ASKP18] use Intel SGX for the execution of off-chain transactions. Second, Hybster [BDK17] makes usage of Intel SGX technology for the implementation of their hybrid state-machine replication protocol.

3
Goals and Motivations

Hyperledger Fabric chaincode execution is integrated in a permissioned blockchain [Hyp]. The blockchain network guarantees the second and the third component of the CIA (Confidentiality, Integrity, Availability) triad [XWS+17, And11]. Furthermore, Hyperledger Fabric offers some privacy features (channels, private data etc.). Privacy can be seen as part of all three CIA components [Bra02]: controlling who can access and modify reliable data in what ways. Nevertheless, since the chaincodes and the data are distributed on the nodes of the network, there is no confidentiality of the chaincode (its logic) and of the processed data.

Executing the chaincode inside a TEE [SAB15] guarantees confidentiality and integrity at runtime. To fully protect sensitive chaincode and data, we must additionally use encryption techniques (for the chaincode, the operation, the passed states, the execution response message) at non runtime [BCKS18]. Furthermore, using a TEE for chaincode execution requires the guarantee of state continuity: We must prevent an attacker from gaining information by passing any ledger state to the chaincode inside the TEE.

FPC demonstrates how Intel SGX can be used in Hyperledger Fabric to guarantee confidentiality of the chaincode and the processed data [BCKS18, fpca]. Since chaincodes bring new opportunities in the context of IoT [Tit18, CD16a], this thesis wants to explore the usage of ARM TZ with OP-TEE for Hyperledger Fabric chaincode execution.

The main goal is to design and implement a prototype for Hyperledger Fabric chaincode execution with OP-TEE. To achieve this goal, we want to transform FPC from Intel SGX to ARM TZ with OP-TEE. The task of this transformation will confront us with multiple challenges: With ARM TZ and OP-TEE we lose support for remote attestation [Sta18, ogi19c, CD16b] and hence, the verification of chaincode execution by an authorized TEE. Furthermore, not all security features of OP-TEE are complete, they need hardware support and developer effort to be fully implemented. Two examples: a. Secure boot which would be used to verify the integrity of OP-TEE is not enabled for the Raspberry Pi [Lin, Sec. 3.2.9]. b. The secure storage key is derived from a stubbed Hardware Unique Key [Lin, Sec. 2.7.2, 2.9]. Hence, encrypted chaincode storage and the storage of any key for the encryption of the operation, the passed states and the execution result is not secure. Facing these challenges, we want to implement a chaincode example running inside the secure world - some confidential guarantees may be left as future work.

4 Design and Implementation

In this Chapter, we first define an architecture and API for Hyperledger Fabric chaincode execution with OP-TEE. The design is called Fabric OP-TEE Chaincode (FOC). Then, we present the implementation of FOC.

4.1 Architecture

The design is based on the transformation of FPC from Intel SGX to ARM TZ with OP-TEE. The intuitive approach would be to run the peer in the normal world of OP-TEE and have a chaincode TA and a ledger TA in the secure world (analogous to the chaincode enclave and the ledger enclave in FPC). Due to missing Docker support in the normal word of OP-TEE, we place the peer on a node decoupled from the ARM TZ / OP-TEE node (see Section 5.1 for a detailed justification). Furthermore, we have faced some limitations regarding the security features and guarantees of ARM TZ and OP-TEE. Therefore, not all security guarantees of FPC could be considered in FOC (see Section 6.2 for the differences in security between FPC and FOC and Section 6.1 for a detailed evaluation of the security in FOC).

Figure 4.1 shows the components of FOC. There are the same type of nodes as Hyperledger Fabric has: a client, an orderer and a peer. Additionally, there is a node (see right side of the Figure) which supports ARM TZ with OP-TEE. The chaincode gets executed inside the secure world of OP-TEE.

In FOC, a chaincode needs to be written in C and gets deployed as an application for the secure world. It is called either `chaincode` or `chaincode TA` in the remaining part of the report. For each `chaincode`, we have a so called `chaincode_wrapper` running at the peer. This `chaincode_wrapper` is installed and instantiated as Hyperledger Fabric chaincode implemented in the Go programming language and is used as an interface towards the peer and the ledger. It forwards incoming chaincode invocations to the `chaincode` in the secure world, handles the communication towards the ledger and sends transaction responses back to the peer. After having instantiated the `chaincode_wrapper`, a transaction proposal with the `UUID` of the `chaincode` must be passed to the `chaincode_wrapper` so that the `chaincode_wrapper` can communicate with the correct `chaincode` inside the secure world. A C++ application in the normal world called `chaincode_proxy` acts as intermediary and forwards the calls between the the `chaincode_wrapper` and the `chaincode`.

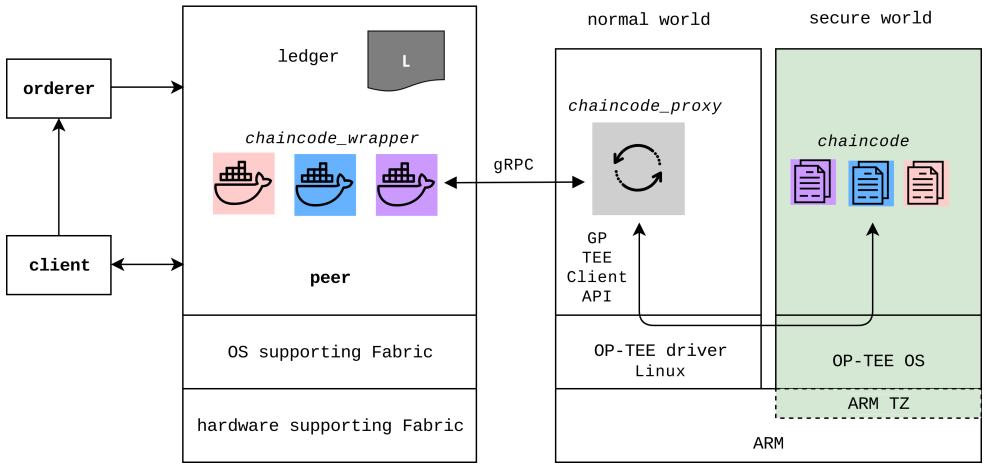

Figure 4.1: Architecture of FOC. Trusted parts are colored in green.

Source: Figure drawn by C.M., icons copied from [Hyp, ico]

For the communication between the `chaincode_wrapper` and the `chaincode_proxy`, the gRPC framework is used [gRPb]. The `chaincode_proxy` and the `chaincode` communicate via the GP TEE Client API [Glo10] supported by OP-TEE [Lin].

4.2 API

The FOC specific messages used for the communication between the `chaincode_wrapper` and the `chaincode` are displayed in Figure 4.2. Each chaincode transaction invoked by a client is forwarded by the `chaincode_wrapper` with an `InvocationRequest` and terminated by the `chaincode` with an `InvocationResponse`. In between these two messages, the `GetStates` and `PutStates` required by the transaction are handled.

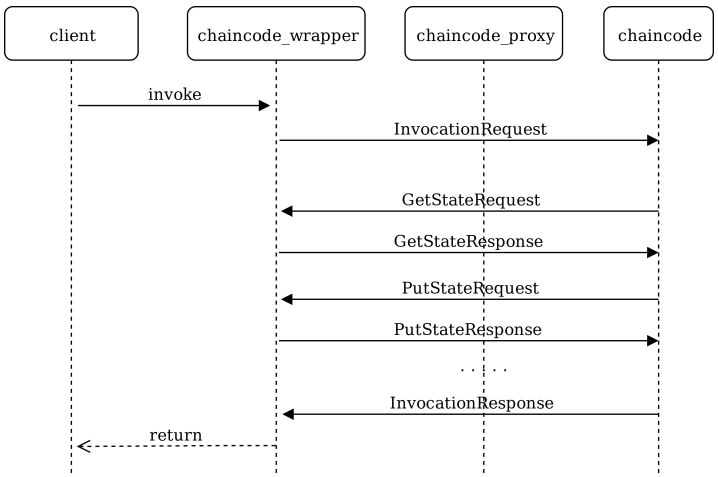

Figure 4.2: API of FOC.

Source: figure drawn by C.M.

4.3 Implementation

We have implemented a prototype of FOC with the architecture and API described in Sections 4.1 and 4.2. To demonstrate the Hyperledger Fabric chaincode execution with OP-TEE, we have programmed a `chaincode` which tracks the number of consumed coffees for different people.

The prototype can be found in the `master` branch of the private github repository `fabric-optee-chaincode`.[1] The structure of the repository and the implemented prototype are described in the following Sections.

4.3.1 Repository Structure

The code of the FOC prototype can be found in the following subdirectories of the repository:

- **`chaincode_wrapper`.** Contains the code needed to deploy the `chaincode_wrappers`. The file `chaincode.go` contains the chaincode for the `chaincode_wrappers`.
- **`chaincode_proxy`.** Contains the code needed to deploy the `chaincode_proxy`. The `chaincode_proxy.cpp` contains the implementation of the `chaincode_proxy` in C++.
- **`chaincode`** Contains the code to deploy `chaincode TAs` running in the secure world.
  To demonstrate chaincode execution with OP-TEE, we have implemented an example `chaincode TA` (`coffee_tracking_chaincode.c`) which tracks the consumed coffees of different people. It contains three types of transaction: the `create` transaction registers a new person on the ledger with an initial number of consumed coffees. Once a person is registered, the number of consumed coffees can be queried via the `query` transaction and updated via the `add` transaction. The example is called coffee tracking chaincode.
  The generic methods for writing and reading data from and to the shared memory by the `chaincode TA` are implemented in a separate file called `chaincode_library.c` which is included via an header file in the `coffee_tracking_chaincode.c`. These methods can therefore be reused when new functions are added to the coffee tracking chaincode or when a completely new `chaincode TA` gets implemented.

4.3.2 Used Frameworks and Message Flow

In the following, we will give a detailed description of the communication between the `chaincode_wrapper` and the `chaincode`.

Frameworks and data structures used for communication. As stated in Section 4.1, gRPC is used for the communication between the `chaincode_wrapper` and the `chaincode_proxy`. For structuring and serializing the data, we use Protocol Buffer [Dev] with the proto3 syntax. The structure of the Protocol Buffer data passed between the `chaincode_wrapper` in the `chaincode_proxy` is defined in the file `invocation.proto`.[2] To enable message flow in both direction, we use bidirectional gRPC streaming with the oneOf Protocol Buffer feature.

The `chaincode_proxy` and the `chaincode` communicate via the GP TEE Client API supported by OP-TEE, see Section 4.1. To pass data between the `chaincode_proxy` and `chaincode`, we use three parameters.[3] The first parameter is a shared memory reference used to pass the function name to

[1] `https://github.com/piachristel/fabric-optee-chaincode`, last accessed on 14.08.2019

[2] `https://github.com/piachristel/fabric-optee-chaincode/blob/master/chaincode_wrapper/proto/invocation.proto`, last accessed on 14.08.2019

[3] `https://github.com/piachristel/fabric-optee-chaincode/blob/master/chaincode_proxy/host/chaincode_proxy.cpp#L234-L276`, last accessed on 14.08.2019

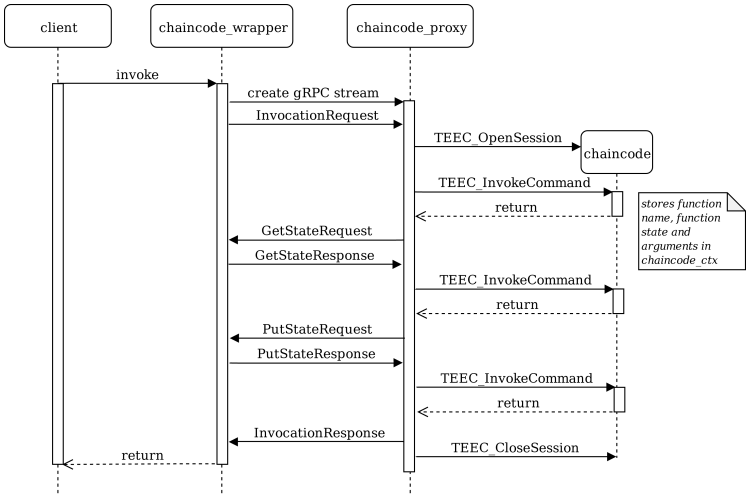

Figure 4.3: API of the implemented FOC prototype.

Source: figure drawn by C.M.

the `chaincode`, the second parameter is a value used to pass the type of message (execution response, `GetState` or `PutState`) from the `chaincode` to the `chaincode_proxy` and the third parameter is a shared memory reference used to transfer the remaining data (arguments, key and value for `GetState/PutState`, acknowledgement of `PutState`, execution response, see[4]) between the two components.

Message flow. The API is shown in Figure 4.3. Whenever a transaction proposal arrives at the `chaincode_wrapper`, the `chaincode_wrapper` (gRPC client) sets up a synchronous bidirectional stream towards the `chaincode_proxy` (gRPC server). The IP address of the `chaincode_proxy` is hardcoded in the `chaincode_wrapper`.[5] The `chaincode_proxy` is a gRPC server listening to all IP addresses at port 50051. Once the stream is setup, the `chaincode_wrapper` sends an `InvocationRequest` message with the `chaincode UUID`, the function name and the arguments to the `chaincode_proxy`.

When the `chaincode_proxy` has received the message, it establishes a context towards the secure world (`TEEC_InitializeContext`) and opens a session towards the `chaincode` by using `TEEC_OpenSession` with the `chaincode UUID` as second parameter. This second parameter is used to identify the `chaincode TA`. Then, the `chaincode_proxy` puts the function name and the arguments into the shared memory and calls the `chaincode` by using the `TEEC_InvokeCommand` of the TEE Client API with `TA_CHAINCODE_CMD_INIT_INVOKE` (defined as 0) as second parameter. Inside the `chaincode`, the `TA_CHAINCODE_CMD_INIT_INVOKE` signals that it is an initial call of the function. Therefore the `chaincode` resets the `chaincode_ctx` - a structure used to store the context of a session - and writes the function name, the arguments and the function state (initially 0) into the `chaincode_ctx`. The functions inside the `chaincode` are divided in subparts, so that the `chaincode` can continue with the correct part after a ledger access.

The `chaincode` then executes the transaction by calling the function just added to the `chaincode_ctx`. Whenever the function needs to do a `GetState` or a `PutState`, the `chaincode` has to return. Before

[4] `https://github.com/piachristel/fabric-optee-chaincode/blob/master/chaincode_proxy/host/chaincode_tee_ree_communication.h#L24-#L40`, last accessed on 14.08.2019

[5] In future work and for a real world application, the IP address should not be hardcoded since this approach is inflexible. The IP address could for example be passed as parameter in the initial transaction proposal which sets the `UUID` of the `chaincode TA`, see Section 4.1

returning, the chaincode does two things: it stores the key (and value in case of a PutState) in the shared memory, and the type of request (GET_STATE_REQUEST or PUT_STATE_REQUEST) in the value parameter and increases the function state of the chaincode_ctx by 1. The chaincode_proxy then forwards the request to the chaincode_wrapper via a GetStateRequest/PutStateRequest gRPC message and waits for the answer. Once the chaincode_proxy receives the answer, it will place the value in case of a GetState or the acknowledgment (in case of a PutState) into the shared memory and call the chaincode by using the TEEC_InvokeCommand with the value TA_CHAINCODE_CMD_RESUME_INVOKE (defined a 1) as second parameter. Therefore, the chaincode will not reset its chaincode_ctx but call the correct subpart of the function state stored in the chaincode_ctx.

Once the chaincode has finished the execution of the function, it stores the execution response in the shared memory and passes the request type INVOCATION_RESPONSE by parameter. The chaincode_proxy forwards the execution response via the InvocationResponse gRPC message to the chaincode_wrapper and then closes the session (TEEC_CloseSession) and finalizes the context (TEEC_InitializeContext).

Coffee tracking chaincode. The implemented create transaction in the coffee tracking chaincode uses one GetState and one PutState, the same holds for the add transaction. So both transactions use the message flow displayed in Figure 4.3. The query transaction only needs one GetState and no PutState; therefore this transaction uses the message flow of Figure 4.3 but without the PutStateRequest, the PutStateResponse and the third TEEC_InvokeCommand and return. With the three implemented transactions (create, add, query), we already have a fully-working chaincode for demonstration and performance measurements. Therefore, no more transactions are implemented.

4.3.3 Development Environment

During the master thesis, we have used the machines provided by the Complex Systems and Big Data Competence Centre of the University of Neuchâtel.[6]

Furthermore, QEMU [QEM] has been used. QEMU is a machine emulator and virtualizer. With its full system emulation mode, QEMU can imitate an entire system with processors and peripherals. Instead of using real ARM TZ, one can use OP-TEE together with the QEMU implementation of ARM TZ [BD14, Bel15]. In order to develop FOC, we have used the qemu_v8.xml[7] manifest, which runs OP-TEE using QEMU ARMv8-A [Lin].

4.3.4 Deploy and Run

To deploy and run FOC, one can consult the Deploy and the Run Sections of the README.md.[8] There are guidelines for deploying the chaincode_proxy and the chaincode either on ARM TZ (ARMv8-A) emulated with QEMU or on the Raspberry Pi 3, Model B.

The used versions for the most important dependencies of the FOC prototype are:

- Hyperledger Fabric v1.4.1
- gRPC v.1.20.0
- OP-TEE v3.5.0

[6] http://ccfs.unine.ch

[7] https://github.com/OP-TEE/manifest/blob/master/qemu_v8.xml, last accessed on 14.08.2019.

[8] https://github.com/piachristel/fabric-optee-chaincode/blob/master/README.md, last accessed on 15.08.2019

5 Rationale

In this Chapter, we justify our decisions for the design of FOC and for the implemented prototype described in Chapter 4.

5.1 Design

This Section justifies the design of FOC described in Section 4.1 and 4.2. Especially the architecture has been shaped to a large degree by some limitations of OP-TEE. The decisions are listed in chronological order, with the oldest decision mentioned first.

- **Decoupled peer.** The normal world of ARM TZ with OP-TEE is no common Linux distribution [Lin]. It is rather limited in functionality, for example programs need to be cross-compiled. In a first approach, we wanted to run the peer in the normal world. Any chaincode invocation would then be forwarded to the secure world by a wrapper chaincode of the peer. Since chaincodes are running in Docker containers [Hyp], Docker needs to be installed in the normal world. Adding the available buildroot Docker packages[1] has not been sufficient. There have been missing dependencies and configurations. We therefore have taken into consideration the following two options to pursue the installation of Hyperledger Fabric in the normal world:
  - Adding the patches of OP-TEE to a common Linux distribution (for example Ubuntu) which supports Hyperledger Fabric.
  - Using the approach of `https://github.com/chainforce/native-fabric` (last accessed on 14.08.2019) which installs the user chaincode as system chaincodes. Since system chaincodes are part of the peer process and do not run in an own Docker container [Hyp, Sec. 4.8.9, 7.9.4, 7.10] we could avoid the dependency on Docker.

  The first approach is not trivial. The second approach is a misuse of system chaincodes: If we run user chaincode as system chaincode, we would have to register and deploy these chaincodes already when the peer starts up and not later when the peer is already running. This makes Hyperledger

[1] `https://git.busybox.net/buildroot/tree/package/`, last accessed on 14.08.2019

Fabric inflexible.
Therefore we have dropped both of the options and decided to go for a decoupled approach where the peer is running on a node which is separated from ARM TZ and which supports Hyperledger Fabric. This approach allows us to create a functioning prototype of FOC during the time available for the master thesis.

- **`chaincode_proxy` in the normal world.** With the decoupled approach described above, the most intuitive design would be a direct communication from a wrapper chaincode at the peer to the chaincode in the secure world. But such a design would face the following limitations:
  - The provided Socket API for network calls towards OP-TEE only supports socket clients, but no socket servers [Lin]. This means that the chaincode (client) inside the secure world needs to establish a connection towards a server (a wrapper chaincode or another component) at the peer before any chaincode invocation can take place. This connection would then need to be maintained all the time (even when no transaction takes place). This approach is disadvantageous: In case of a failure, it may not be easy to reestablish the connection. Furthermore, the memory size of OP-TEE and therefore the number of (chaincode) TAs which can be loaded in parallel is restricted [ogi19d, Lin, Sec. 5.2.4].
  - There is no official support for any message manager (for communication handling and serialization) in OP-TEE [ogi19e]. Since not even all standard C libraries are supported by OP-TEE [ogi16, ogi17], it may not be feasible to add a message manager in the time available for this master thesis.

  Due to these limitations, we have decided to have a proxy in the normal world (called `chaincode_proxy`). This proxy acts as a (gRPC) server for the `chaincode_wrapper` (gRPC client) at the peer and only loads a `chaincode` into the secure world (by opening a session) when a transaction takes place.

- **gRPC for communication.** We need a message manager for communication handling and data serialization. We use gRPC to be consistent with Hyperledger Fabric where the clients, the peers and the orderers communicate via gRPC [ABB+18].
  In OP-TEE, the host examples are written in C code.[2] But gRPC has no official C language support [gRPb], we therefore went for the official C++ support of the gRPC framework. We managed to compile gRPC (C++) as static library, to write and run the `chaincode_proxy` in C++ and to link the gRPC library to the `chaincode_proxy`.

- **`chaincode` in C.** OP-TEE only supports C for writing TAs [Lin]. Therefore, the `chaincode` running inside the secure world needs to be implemented in C.

- **Minimizing the TCB.** The trusted computing base (TCB) is the part of the system relevant for security [Amo11]. The smaller and the less complex a TCB is, the easier we can protect it. In order to minimize the TCB, we only run the `chaincode` inside the secure world. A second reason for minimizing the code running inside the secure world is the limited size of memory available for OP-TEE [ogi19d, Lin, Sec. 5.2.4].

5.2 Implementation

Below, we justify the most important decisions regarding the implementation of the FOC prototype described in Section 4.3.

[2] `https://github.com/linaro-swg/optee_examples/`, last accessed on 14.08.2019

- **Synchronous communication.** We use synchronous gRPC calls [gRPb] from the `chaincode_wrapper` to the `chaincode_proxy` for reasons of simplicity. Synchronous means that the gRPC client (`chaincode_wrapper`) is blocked while waiting for the gRPC message from the server [Pai16]. The gRPC server (`chaincode_proxy`) can handle multiple requests simultaneously by using a gRPC-managed thread-pool.

- **Frameworks used for communication.** For structuring and serializing the data passed between the `chaincode_wrapper` and `chaincode_proxy` we use Protocol Buffer because gRPC uses Protocol Buffer by default [gRPb].

- **Data size.** Since in OP-TEE, the size of the shared memory and the secure memory (needed to store the loaded TA) is restricted [ogi19d, Lin, Sec. 5.2.4], the length of the data passed between the `chaincode_proxy` and the `chaincode` is limited, see `chaincode_tee_ree_communication.h`.[3] The restrictions are chosen so that the data for the coffee tracking example fits into the shared memory. Both the `chaincode_proxy` and the `chaincode` ensure that the size of data does not exceed the defined limits before writing to the shared memory. A transaction failure is returned in case the limits are exceeded.

- **Data types.** For consistency reasons and in order to do less conversions, we use the same data type for passing the key (of `GetState/PutState`) between the different components of FOC as defined by the Hyperledger Fabric chaincode API for ledger access (see `shim` package of go[4]): Between the `chaincode_wrapper` and the `chaincode_proxy`, we use the `string .proto` type [Dev] and in the shared memory of the `chaincode_proxy` and the `chaincode`, we use a `char array`.

  For the value (of `GetState/PutState`) as well as for the execution response the type defined by the `shim` package would be `[]byte`. We do not go with this type but use the `string .proto` type and a `char array` for the following reasons: a. There is no need to pass the size of the value and the size of the execution response in an additional field because the `string .proto` type is dynamically sized and the `char array` in C/C++ is terminated by the null character. b. For the conversion of the value to a numeric type inside the `chaincode`, we can use the `strotul` function supported by OP-TEE.[5]

  Also for reason a., we use the `string .proto` type and `char arrays` for passing the function name and the arguments between the different components.

  In case an entity does not exist on the ledger, the `GetState` method of the `shim` package will return nil. Since we cannot pass nil for the field value of the `GetStateResponse` message (which is of type `string .proto`), we convert nil to an empty string. Empty strings are the default values for the `string .proto` type. This means that the `chaincode` will follow from an empty string in field value of a `GetStateResponse` message that the entity with the given key in the foregoing `GetStateRequest` does not exist. Therefore storing empty strings as values on the ledger will mistakenly be interpreted as "entity does not exist" and is not allowed by design.

- **Number of coffees.** The type for the number of coffees received by an `add` transaction and by a `GetStateResponse` is converted to a unsigned long by `strotul` inside the secure world. The maximum value is 18446744073709551615. We do not handle any overflow that might happen in case the value of consumed coffees exceeds 18446744073709551615 $(2^{64} - 1)$ because this is not realistic.

[3] `https://github.com/piachristel/fabric-optee-chaincode/blob/master/chaincode_proxy/host/chaincode_tee_ree_communication.h`, last accessed on 14.08.2019

[4] `https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim`, last accessed on 23.07.2019

[5] `https://github.com/OP-TEE/optee_os/blob/master/lib/libutils/isoc/include/stdlib.h`, last accessed on 23.07.2019

- **Context and session handling.** For each request a separate context and session with the according `chaincode` gets initialized and opened by the `chaincode_proxy`. We do not use the same session for multiple requests since the gRPC calls from the `chaincode_wrapper` (client) to the `chaincode_proxy` (server) are synchronous, which means that the `chaincode_wrapper` gets blocked till the communication has been finished [gRPb, Pai16]. Using the same session for multiple clients may increase the blocking time.
- **Error handling.** The `chaincode_wrapper` catches different errors which all close the connection towards the `chaincode_proxy` and return a response of status `ERROR`[6] to the peer. The following errors are caught by the `chaincode_wrapper`: a. failure to open a stream towards the `chaincode_proxy`, b. timeout if the gRPC call takes longer than 10 minutes, this value can be increased for more complex `chaincodes` which need longer execution time, c. failure to send or receive messages to/from the `chaincode_proxy` and d. arrival of a gRPC error status sent by the `chaincode_proxy`. Due to security reasons the type of error in case d. is not passed to the `chaincode_wrapper`.

[6] `https://github.com/hyperledger/fabric/blob/release-1.4/core/chaincode/shim/response.go#L41-L44`, last accessed on 23.07.2019

6 Evaluation

In this Chapter, different aspects of FOC (design and implementation) are evaluated. We describe the security guarantees and limitations of FOC in Section 6.1 and sketch how the missing features could be added in order to achieve the initial goal of smart contract (its logic) and data confidentiality in future work. A comparison of FPC and FOC is given in Section 6.2. In Section 6.3, we describe how FPC and FOC peers could interoperate in a common network. Section 6.4 summarizes the performance and power measurements done with the implemented FOC prototype.

6.1 Security Evaluation

The thesis is about untrusted peers. We assume that the other components are trusted. This Section therefore analyses the security guarantees and lacks of FOC when it comes to untrusted peers. Sections 6.1.1 and 6.1.3 hold for both - the FOC design and the implemented prototype. Section 6.1.2 about TCB size refers to the implementation only.

6.1.1 Security Guarantees and Limitations

Since in ARM TZ, the hardware and software resources are partitioned between the secure and the normal world [Lim09], the `chaincode TA` execution is isolated from the (untrusted) peer and we have the following security guarantees:

- **Confidentiality of chaincode execution.** The (untrusted) peer cannot see what happens at runtime, especially sensitive chaincode logic and processed data is protected.
- **Integrity of chaincode execution.** The chaincode cannot be accessed and modified by the (untrusted) peer and therefore correct execution and correct output is ensured.[1]

Due to the lack of secure boot, secure integrity check, remote attestation and transaction response signature in the current implementation of FOC, there is no mechanism for a third party (e.g. the client / committing

[1] The integrity check of the `chaincode TA` at load time is not secure since the private key is part of the OP-TEE source code [Lin, Sec. 2.7.8], see Section 6.1.3 for more information.

peer) to verify the confidentiality and integrity of the chaincode execution. For example, the client cannot be sure that the transaction actually gets executed by the authorized `chaincode TA` and the verifying peer cannot check if the transaction response is actually generated by the supposed `chaincode TA` inside the secure world since signatures are missing. In Section 6.1.3.1, we describe what would be necessary to overcome this lack.

6.1.2 TCB Size

The trusted computing base (TCB) of FOC consists of the `chaincode` executed inside the TEE (secure world). The other FOC specific components (`chaincode_wrapper` and `chaincode_proxy`) as well as the components inherited from Hyperledger Fabric (client, orderer, peer etc.) are untrusted.

To count the lines of code of the implemented FOC prototype and of the Hyperledger Fabric peer, we use the CLOC tool.[2]

The code of the FOC prototype contains 516 lines of untrusted code[3] and 356 lines of trusted code in total, see Table 6.1 for more detailed information.

	Go	**C** (with headers)	**C++** (with headers)	**Protocol Buffer**	**total**
`chaincode_wrapper`	142	0	0	40	182
`chaincode_proxy`	0	0	294	40	334
`chaincode`[3]	0	356	0	0	356

Table 6.1: Number of lines of code for the FOC components, `https://github.com/piachristel/fabric-optee-chaincode/`,`master` branch, commit 21531102d8c1e3438c9f75a8691ef85d6b0271e0, last accessed on 14.08.2019.

The results must be considered carefully since not all necessary security features to fully guarantee confidentiality of the chaincode (its logic) and the processed data have been implemented, see Section 6.1.3.

The Hyperledger Fabric peer contains between 20'000 and 100'000 lines of code (v1.4.1). By only executing the chaincode itself inside the TEE and not the entire peer, we can drastically minimize the TCB.

6.1.3 Implementation of Missing Security Features

This Section describes how one could implement the missing security features so that FOC has similar security guarantees as FPC (see Section 2.5). The implementation of these features is not part of the thesis since it would require device-specific information and properties which are often kept secret by the vendors and which are mostly only stubbed by OP-TEE [Lin, Sec. 2.7, 2.9.7].

6.1.3.1 Confidentiality and Integrity of Chaincode Execution

As stated in Section 6.1.1, confidentiality and integrity of the chaincode execution can not be fully guaranteed and verified. In the following, we describe what would be necessary to overcome this lack.

- **Secure boot.** Secure boot verifies the integrity of OP-TEE [Lim09, Sec. 5.2.2], [Lin, Sec. 2.7.4]. For the deployment of FOC, we have used QEMU for Armv8-A and the Raspberry Pi, see Section 4.3.4. In both cases, there is no secure boot enabled [Bec16, Lin, Sec. 3.2.9]. For a productive usage, a device which supports secure boot must be used. One should consult the manufacturer of the device to check if and how secure boot can be enabled [Lin, Sec. 2.8.1].

[2] `https://github.com/AlDanial/cloc/`, v1.74, last accessed on 14.08.2019

[3] Without the untrusted dummy host part `https://github.com/piachristel/fabric-optee-chaincode/blob/master/chaincode/dummy_main.cpp`, last accessed on 14.08.2019, which contains 3 lines of C++ code.

- **Integrity of the chaincode.** In FOC, the `chaincode TA` is of type REE filesystem TA and therefore will be loaded from the normal world and integrity checked in the secure world [Lin, Sec. 2.10.3, 5.14.8]. This integrity check is possible since all TAs are signed with a private key [Lin, Sec. 2.7.8]. The private key is part of the OP-TEE source code. If FOC will be used as real product, the private key must not be stored within the source code but only be accessible by the trusted chaincode developer (for example by storing the private key on a Hardware Security Module).
  An alternative to the integrity check via the development private/public key pair would be to use so called Early TAs [ogi19b, Lin, Sec. 5.14.8]. In that case, the `chaincode TA` binary is part of the OP-TEE firmware binary and already gets verified at the boot process together with the whole OP-TEE firmware image.

- **Remote attestation and signature.** Since OP-TEE does not support remote attestation [ogi19c], the described communication mechanism currently implemented in FOC (see Section 4.3.2) does not ensure that the `chaincode TA` is actually executed inside the secure world and that the code has not been tampered with. In future work, remote attestation could be implemented as follows: the `chaincode TA` generates a hash of itself and signs it with the HUK[4] which uniquely authenticates the device itself [ogi18]. Then, we would also need a trusted third-party to verify the hash and to authenticate the HUK.
  Similarly as in FPC [BCKS18, Sec. 5.3], the `chaincode TA` could generate a private/public key pair and pass the public key together with the hash to the party requesting remote attestation. Like in FPC [BCKS18, Sec. 3.2, 5.3, 5.4, 5.6], the public key can be used by the the client to encrypt the operation and therefore to ensure that the transaction proposal gets executed by an authorized `chaincode TA`. Furthermore it can be used by the committing peers to verify that the transaction response originates from an authorized unmodified `chaincode TA`. In order to avoid that each client and each committing peer has to redo attestation before invoking or committing, the attestation result and the public key may be stored on the ledger. FPC has a chaincode called enclave registry for this purpose [BCKS18, Sec. 5.1, 5.3, 5.4]. FOC may use a similar concept in the future.
  Furthermore, the SSL/TLS support of gRPC should be enabled in the future, so that the `chaincode_wrapper` can authenticate the `chaincode_proxy` [gRPa].

- **Recovery at reboot.** In the current implementation of FOC, the `chaincode TA` is stateless (e.g. no private key used), therefore no recovery support is necessary when rebooting OP-TEE. Once there is a key added to the `chaincode TA` for signature, we need to securely store this key so that it is still available after rebooting OP-TEE. One could use the secure storage of OP-TEE [Lin, Sec. 2.9] for that. In the current implementation of OP-TEE, the key to encrypt/decrypt the secure storage is derived from a stubbed HUK and therefore not secure[4] [Lin, Sec. 2.7.2, 2.9].

6.1.3.2 Protection of the Ledger State

Furthermore, to guarantee state continuity and to avoid information leakage by rollback attacks and speculative execution, we would need to add the following guarantees:

- **Guarantee state continuity.** In the current design, it is possible to pass any world state to the `chaincode TA`. In future work, a mechanism must be added so that the `chaincode TA` can check the correctness of the data retrieved from the ledger. In FPC, this issue is solved by the ledger enclave [BCKS18, Sec. 5.1, 5.5]. In the current implementation of FOC, the peer is running on hardware supporting plain Hyperledger Fabric (e.g. Intel node), therefore one could reuse the SGX ledger enclave of FPC to guarantee state continuity. Of course this would need some adaptations as

[4] In OP-TEE, the Hardware Unique Key (HUK) is stored in the software as zeros [Lin, Sec. 2.7.2]. For a productive usage of FOC, this HUK must be replaced and must not be stored in software but in hardware.

for example the local attestation between the ledger enclave and the chaincode enclave [BCKS18, Sec. 5.3] must be replaced by an authentication mechanism between the ledger enclave and the `chaincode TA`. Another approach to guarantee state continuity would be the implementation of a TA - we call it ledger TA - with similar functionality as the ledger enclave. The `chaincode TA` could communicate directly with the ledger TA by using the Internal Client API implemented in OP-TEE [Glo14, Sec. 4.9] and could store its data encrypted in the secure storage of OP-TEE [Lin, Sec. 2.9]. Here again, we would need to have a secure HUK implementation to derive a key for encryption/decryption of the secure storage[4].

- **Avoid information leakage by speculative execution if needed.** Due to the execute-order-validate paradigm used in Hyperledger Fabric, transaction execution by the peer is speculative [BCKS18, Sec. 4.4]. In future work, we must add a mechanism to avoid information leakage by speculative execution if required. In FPC, this issue is solved by the adding the concept of barriers. The same concept could be used in FOC, this would also facilitate the interoperability of FPC and FOC peers.

6.1.3.3 Optional Confidentiality for Chaincode, State and Execution Response Message

The following security features are not part of the current FOC implementation but could be added as optional security features in the future:

- **Chaincode confidentiality.** In the current implementation of FOC, the compiled `chaincode TAs` are stored unencrypted in the REE filesystem (in the directory `$OPTEE_SRC/out-br/target/lib/optee_armtz/` [Lin, Sec. 5.14.3]). If chaincode confidentiality needs to be guaranteed, one could use the secure storage TA type to store the `chaincode TA` encrypted in the REE filesystem [Lin, Sec. 2.10.3]. In the current implementation of OP-TEE, the key to encrypt/decrypt the secure storage TA is derived from a stubbed HUK and therefore not secure[4] [Lin, Sec. 2.7.2, 2.9].

- **State confidentiality.** Similar to the design in FPC [BCKS18, Sec. 5.5], we could add encryption of the state stored on the ledger and passed by `GetState` or `PutState` operation by using either client-based encryption or encryption per chaincode. Both modes would require a third-party mechanism to exchange the keys (between the client and the invoked chaincode in the first case, and between the same chaincodes running on different ARM TZ nodes in the second case).

- **Encryption of the execution response message.** In the future, FOC could support the encryption of the execution response message with a key provided by the client. This optional security feature is already part of FPC [BCKS18, Sec. 5.5].

6.2 Comparison with FPC

The comparison between FPC and FOC given in this Section holds for both - the FOC design and the implemented prototype. A summary of the differences between FPC and FOC can be found in Table 6.2.

FPC and FOC both aim at isolating the execution of the chaincode from a potentially untrusted peer. To achieve this goal, FPC makes use of Intel SGX [BCKS18] whereas FOC makes use of ARM TZ with OP-TEE.

In FPC, the enclaves are located on the peer, whereas in FOC, OP-TEE and the `chaincode TAs` run on a node separated from the peer. This decoupled approach is justified in Section 5.1. Another architectural difference is the number of TEEs: In FPC, we have one enclave per chaincode; in FOC, each `chaincode`

feature	**FPC**	**FOC**
technology	Intel SGX	ARM TZ with OP-TEE
architecture	• enclaves are on the same node as the peer • one enclave per chaincode	• OP-TEE runs on a node decoupled from the peer • each `chaincode TA` gets loaded and executed inside the same TEE
language	go/C/C++	go/C/C++
isolation of chaincode execution	isolation of chaincode execution from the untrusted peer is guaranteed and can be verified	isolation of chaincode execution from untrusted peer is implemented, but cannot be verified
state continuity	guaranteed	not guaranteed
operation confidentiality	guaranteed	not implemented, but possible w/ extra hardware/firmware support
state, execution response message and chaincode confidentiality	optionally	not implemented, but possible w/ extra hardware/firmware support

Table 6.2: Comparison of the used technologies, architectures, languages and the security features in FPC [BCKS18] and in FOC.

gets loaded and executed in the same TEE (i.e. secure world).

Both FPC and FOC use go, C and C++ programming languages for implementing and adding their components to Hyperledger Fabric.

Due to its security features (see Section 2.5), FPC guarantees confidentiality for the chaincode (its logic) and the processed data. Furthermore, it provides mechanisms to verify that guarantee. In FOC, we make usage of ARM TZ with OP-TEE to isolate the execution of the chaincode but there is no mechanism for verifying that this is actually the case and that no manipulation has happened. Furthermore, state continuity and encryption of the operation, the passed state, the execution response message and the chaincode itself is not enabled in FOC. In Section 6.1.3, we have discussed which security features must be added so that the chaincode and data confidentiality could be fully guaranteed and verified.

6.3 Interoperability of FPC and FOC peers

With the emergence of IoT, more and more peers of a Hyperledger Fabric channel[5] might use a low-power processor [Dub19, Gar14] and therefore not be able to support Intel SGX and FPC but rather ARM TZ and FOC. Imagine such a channel where some of the peers run FPC and others FOC. Clients can invoke a chaincode either on a FPC or on a FOC peer. The validation then takes place on all peers. To make such an interoperability of FPC and FOC peers work correctly, the following points must be considered:

[5] Group of peers which share the same chaincodes and ledger [Hyp].

6.3.1 Chaincode Language

In FOC, the chaincode running inside the secure world must be written in C (OP-TEE only supports C for writing TAs [Lin]). Hence, if we want to use the same programming language for the chaincode in FOC and in FPC, we have to use C.

6.3.2 Chaincode Logic

In FOC, the `chaincode TA` interrupts its transaction execution for each required `GetState` and `PutState` with a return to the `chaincode_proxy`. Therefore, a transaction may be split in different parts. In FPC, there is no need to return in case of a `GetState/PutState` [fpca]. Despite this slight change in code structure between FOC and FPC, one must ensure that the logic for both chaincodes is identical.

6.3.3 Confidentiality Guarantees

FPC has multiple features (see Section 2.5) to guarantee confidentiality of the chaincode (its logic) and data. The current design and implementation of FOC does not consider all of these features (see Section 6.2). Interoperability will make sense as soon as a future design of FOC considers these missing features. In Section 6.1.3, we have sketched how this could be achieved. Once this is done, an interoperability of FPC and FOC peers must consider the following points (we will not go into detail since the exact design and implementation of such an interoperability is left for future work):

(a) **Attestation result.** The attestation result of the chaincode enclave must be verifiable by the `chaincode TA` and vica-versa. Also the client must be able to verify those attestation results to invoke chaincodes on authorized chaincode TEEs (chaincode enclaves and `chaincode TAs`).

(b) **Validation.** In order to commit transactions, the FPC peers must be able to verify the transaction response (its signature) produced by an FOC peer and vica-versa. The attestation result (which contains a hash of the public key) and the public key of the chaincode TEEs could be stored on the ledger (as in FPC [BCKS18, Sec. 5.1, 5.3, 5.4]). Hence, both chaincode TEEs would have access to it and could verify that the transaction response originates from an authorized chaincode TEE in case (a) is satisfied.

(c) **Communication with the client.** The client must have access to the public key of both chaincode TEEs, so that it can encrypt the operation either with the public key of the chaincode enclave or the `chaincode TA`, depending on whom it passes the transaction proposal to.

(d) **State encryption.** If we choose a state encryption per chaincode (as described in [BCKS18, Sec. 5.1, 5.3, 5.4]), one must ensure that the chaincode enclave and the `chaincode TA` have access to the private key, so that the state can be encrypted/decrypted by both chaincode TEEs.

6.4 Performance and Power Measurements

We first describe the setup and methodology used in experiment 1, 2 and 3. Then, we present the experiments. Experiment 1 and 2 are related to performance. Experiment 3 is about power. Finally, we describe what could be measured as future work.

6.4.1 General Setup

Code base. Experiment 1 and 2 use the implemented FOC prototype.[6] Experiment 2 uses some adaption of the `chaincode_wrapper`, `chaincode_proxy` and `chaincode` in order to measure the latency breakdown.[7]

Versions. The used versions for the most important dependencies of FOC are:

- Hyperledger Fabric v1.4.1 with default settings
- gRPC v.1.20.0
- OP-TEE v3.5.0

Hardware and network properties. For the experiments, we use the machines provided by the Complex Systems and Big Data Competence Centre of the University of Neuchâtel.[8] The `chaincode_wrapper` runs on yahoocluster-4.maas.[9] The `chaincode_proxy` and the `chaincode` run either on a machine (eiger-8.maas or cervino-4.maas) emulating ARM TZ (ARMv8-A) with QEMU (using the `qemu_v8.xml`[10]) or on the Raspberry Pi. The properties of the used devices can be found in Table 6.3. For the eiger-8.maas, we disable Hyper-Threading in order to minimize the risk of attacks such as the Spectre vulnerability [Jac19].

Machine (name)	**CPU model**	**# cores**	**OS**	**RAM**	**storage**	**network bandwidth**
yahoocluster-4.maas	Intel(R) Xeon(R) CPU L5420	8@2.5GHz	18.04 LTS	8 GiB	500 GB	1 Gbit/s
cervino-4.maas	Intel(R) Xeon(R) CPU E5-2683 v4	32@2.1GHz	18.04 LTS	128 GiB	480 GB	1 Gbit/s
eiger-8.maas	Intel(R) Xeon(R) CPU E3-1270 v6	4@3.8GHz	18.04 LTS	64 GiB	480 GB	1 Gbit/s
Raspberry Pi 3 Model B	ARM Cortex A53 64bit CPU	4@1.2GHz	OP-TEE	1 GB	-	100 Mbit/s

Table 6.3: Properties of the machines used for performance and power measurements. Information retrieved from the machines themselves and from `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`, `https://en.wikipedia.org/wiki/Raspberry_Pi`, `https://clusterinfo.unineuchatel.ch/MAAS/#/machines`, last accessed on 11.08.2019.

CPU frequency. For all used machines (cervino-4.maas, eiger-8.maas, yahoocluster-4.maas, Raspberry Pi) and also for the machine (nuc-1.maas) used to interact with the Raspberry Pi, we set the CPUfreq governor to performance mode. With this mode, the CPU frequency is statically set to the highest frequency [Bro]:

[6] `https://github.com/piachristel/fabric-optee-chaincode`, `master` branch, commit f41d2f262e013d824b9bcb1c630fed1cc00d4d63, last accessed on 14.08.2019

[7] `https://github.com/piachristel/fabric-optee-chaincode`, `performance-latency-breakdown` branch, commit 8f50dd321b560f54341343df3bfcd98cb84ca236, last accessed on 14.08.2019

[8] `http://ccfs.unine.ch`, last accessed on 16.08.2019

[9] The machine names are only relevant for reference within this Chapter.

[10] `https://github.com/OP-TEE/manifest/blob/master/qemu_v8.xml`, last accessed on 14.08.2019

- yahoocluster-4.maas: 2.499 GHz
- cervino-4.maas: 3 GHz
- eiger-8.maas: 4.2 GHz
- Raspberry Pi 3 Model B: 1.2 GHz
- nuc-1.maas: 4 GHz

QEMU configurations. For the emulation with QEMU we use the following configurations:

- `SMP` (Symmetric MultiProcessing): set to 4 or 8[11]
- `CFG_NUM_THREADS` (number of trusted threads [Lin]): set to 4, 8 or 16[12]
- CPU: cortex-a53[11]

6.4.2 Methodology

Each of the three experiments is executed with OP-TEE on a machine using QEMU for ARM TZ emulation and with OP-TEE running on the Raspberry Pi.

For each of the experiments, we use a Go program to start one or more clients. This Go program runs on the same machine as the `chaincode_wrapper` (yahoocluster-4.maas). Each client is implemented as a goroutine and repeatedly invokes the `query` transaction of the implemented coffee tracking chaincode (see Section 4.3.2) during a fixed amount of time. The timespans are adjusted for each experiment in order to have a reasonable amount of values per client for calculating the results. The invocation is implemented with the `shim.MockStub` so the clients directly call (init and invoke) the `chaincode_wrapper`. There is no usage of any peer.

In our experiments, a transaction starts when the client calls it and is finished when the client has received the response of the `chaincode_wrapper`. We call that timespan execution time (of the `query` transaction) in the following three experiments.

A detailed description on how to run the experiments can be found in the private github repository `fabric-optee-chaincode`.[13]

To measure the times, we have used the following libraries and functions:

- `chaincode_wrapper`: Go `time` package[14], in particular the functions `Now` and `Since`.
- `chaincode_proxy`: cpp `std::chrono` library[15], in particular the function `high_resolution_clock::now()`.
- `chaincode`: The functions `TEE_GetSystemTime` and `TEE_TIME_SUB` implemented by OP-TEE.[16]

[11] This value can be changed in the file `${OPTEE_SRC}/build/qemu_v8.mk`.

[12] This value can be changed in the file `${OPTEE_SRC}/optee_os/mk/config.mk`.

[13] `https://github.com/piachristel/fabric-optee-chaincode/blob/master/performance_power_measurements/README_performance_power_measurements.md`, last accessed on 14.08.2019

[14] `https://golang.org/pkg/time/`, last accessed on 28.06.2019

[15] `https://en.cppreference.com/w/cpp/header/chrono`, last accessed on 28.06.2019

[16] `https://github.com/OP-TEE/optee_os/blob/master/lib/libutee/include/tee_api.h`, `https://github.com/OP-TEE/optee_os/blob/master/lib/libutee/include/utee_defines.h`, last accessed on 28.06.2019. According to the GP TEE Internal Core API [Glo14], this function returns the current system time of a TA instance. The counter is monotonic between the return of one `TEEC_InvokeCommand` and the start of the next `TEEC_InvokeCommand`. We have checked and confirmed this behavior in `https://github.com/piachristel/fabric-optee-chaincode`, `performance-latency-breakdown` branch, commit 6df7a8c251118c60d62d6c0eecbfe469df1e42e8, last accessed on 14.08.2019.

To measure the power (experiment 3) on the machine emulating ARM TZ with QEMU, we use a Lindy IPower Control 2x6 XM and an adaption of the `pdu-power-parser.py`[17] script. For the Raspberry Pi, we use the Alciom PowerSpy2 device together with the `powerspy.py`[18] script.

6.4.3 Experiment 1: Throughput and Latency

Experiment. We measure the latency and throughput for the execution of the `query` transaction of the coffee tracking chaincode in FOC. The measurements are done for different numbers of clients (1, 2, 4, 8) repeatedly invoking the transaction in parallel during (30, 30, 30, 60) seconds.

Expectation. Throughput doubles (or nearly doubles) when doubling the number of clients while latency stays stable up to the point where the number of clients is equal to the minimum of the number of cores of all machines.

Pre Result. A first evaluation of this experiment with QEMU has shown that the throughput only slightly increases (with factor < 1.2) from 1 to 2 clients and then more or less stagnates for 4 and 8 clients. This does not confirm our expectation. We therefore have to refine the experiment.

Refined Experiment and Result. To investigate where the bottleneck comes from, we implement different modifications of FOC:

(a) Baseline: An adaption of FOC which executes the coffee tracking chaincode in the normal and not in the secure world - this implementation is our baseline. The measurements are done for (1, 2, 4, 8) clients repeatedly invoking transactions in parallel during 10 seconds.

(b) Without gRPC: An adaption of FOC, where the clients are placed in the normal world and run as threads there. Each client thread repeatedly performs the following steps: a. initializing a context, b. opening a session, c. invoking the coffee tracking chaincode in the secure world, d. waiting for the transaction to finish, e. closing the session, f. finalizing the context. In this case, we have no gRPC calls. The measurements are done for (1, 2, 4, 8) clients repeatedly invoking transactions in parallel during 30 seconds.

(c) Normal world threading: A program where we have (1, 2, 4, 8) threads doing dummy work in the normal world. This implementation has nothing to do with FOC but is used to test the threading capacity of the normal world.

(d) Plain Hyperledger Fabric: We have implemented the coffee tracking chaincode as Hyperledger Fabric chaincode running at the same node as the clients (yahoocluster-4.maas). The measurements are done for (1, 2, 4, 8, 16) clients repeatedly invoking transactions in parallel during 10 seconds.

The code for the three cases can be found in the folder `FOC-modifications` of the github repository `fabric-optee-chaincode`.[19] For the experiments with modifications (a)-(c) and also for the experiments with the original FOC prototype, ARM TZ emulated with QEMU is used (no Raspberry Pi). We set the number of processors for QEMU to the maximal possible value (`SMP` = 8) and use a host machine with 32 cores (cervino-4.maas) to eliminate any bottleneck from the number of CPUs of the (QEMU) host. The `CFG_NUM_THREADS` is set to 8 in a first investigation.

[17] `https://gist.github.com/gfieni/910d075b7e53a607271c18923047675e`, last accessed on 12.08.2019
[18] `https://github.com/patrickmarlier/powerspy.py`, last accessed on 12.08.2019
[19] `https://github.com/piachristel/fabric-optee-chaincode/tree/master/performance_power_measurements/FOC_modifications`, last accessed on 14.08.2019

For each setup, we run the experiment 3 times and take the results of the run for which the maximum throughput over all clients is the highest. The observations related to the throughput and latency measurements are summarized in the following. Detailed results can be found in the appendix, Table B.1 and in Figure 6.1.

- The results for the coffee tracking chaincode running as Hyperledger Fabric chaincode (case (d)) are plotted in Figure 6.1. The Figure shows that in this case, throughput and latency behave as expected: throughput increases (by factor > 1.5) with each doubling of the number of clients and latency stays stable up to the point where the number of clients is equal to 8 (= number of cores at yahoocluster-4.maas). Therefore the Go program which starts the clients cannot be the explanation for the bottleneck.

- Regarding the baseline, we have changed the `NUM_CQS`, `MIN_POLLERS` and `MAX_POLLERS` options of the gRPC server (at the `chaincode_proxy`).[20] Best results are achieved with `NUM_CQS` equal to the number of cores/2 (= 4) and `MIN_POLLERS`, `MAX_POLLERS` with the default values: throughput is increased by factor 1.9 for 2 clients, by 2.5 for 4 clients and 2.8 for 8 clients (all factors in relation to the throughput for 1 client). These factors are far above the factors for the original FOC implementation (< 1.2). Furthermore, the factors are even better than the ones achieved by the normal world threading application (case (c)). We can therefore rule out the gRPC settings as main reason for the bottleneck.

- Regarding the setting (b), we can increase throughput only by factor 1.3 for 2 clients, with factor 1.4 for 4 clients and with factor 1.5 for 8 clients (all factors in relation to the throughput for 1 client). These factors are are a bit above the ones from the experiment with the original FOC implementation, but still far below the factors of the baseline case. We therefore conclude that the bottleneck mainly originates from the secure world calls. To exactly locate the bottleneck, we have done a latency breakdown in experiment 2. This experiment shows, that the `TEEC_OpenSession` makes up 65-75% of the whole latency time. An investigation of the code shows that at least part of this API is serialized[21] which may explain the observed bottleneck. This serialization is not documented, but we have become aware to it due to the github issue [ogi19a]. In Section 6.4.6, we describe which code change may resolve the observed performance bottleneck.

- To explore if performance can be increased by a smaller or larger value for `CFG_NUM_THREADS`, we execute case (b) with a value of 4 and 16, see second and third Table of B.1. But since no improvement of throughput and latency could be achieved by this configuration change, we keep `CFG_NUM_THREADS` equal to the value of `SMP` for further experiments.

- For the original FOC prototype, the throughput and latency for the default `NUM_CQS` value and for `NUM_CQS` equal to `SMP` are similar. But since we have a higher failure rate with `NUM_CQS` set equal to the number of `SMP` than with the default value, we do further measurements with the default value.

Final Experiment and Result. With the insights from the refined experiment (`CFG_NUM_THREADS` equal to number of cores; default value for `NUM_CQS, MIN_POLLERS, MAX_POLLERS` in case of the original FOC implementation), the final throughput and latency measurements are done, once running the

[20] `NUM_CQS` is the number of queues at the gRPC server which will listen to incoming RPCs. `MIN_POLLERS` and `MAX_POLLERS` define the minimum and maximum number of threads per queue. The default values are: `NUM_CQS` = 1, `MIN_POLLERS` = 1, `MAX_POLLERS` = 2. See `https://github.com/grpc/grpc/blob/v1.20.x/include/grpcpp/server_builder.h`, last accessed on 11.08.2019.

[21] The loading of the TA into the secure world is serialized due to a global mutex hold in OP-TEE, see `https://github.com/OP-TEE/optee_os/blob/6ff2e3f10cab2a086ebb159bdab6ccc497aa50ab/core/kernel/tee_ta_manager.c#L630-L661`, last accessed on 13.08.2019.

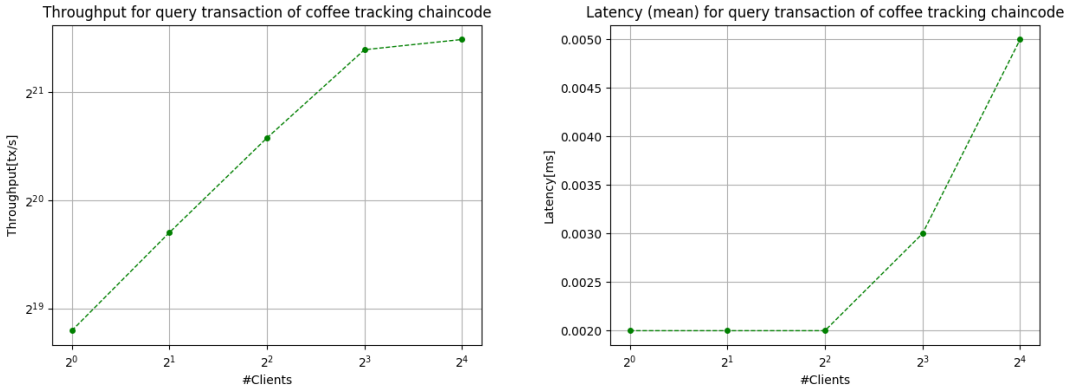

Figure 6.1: Throughput and latency for the execution of the coffee tracking chaincode `query` transaction invoked by different numbers of clients in parallel. Measurements are done with the coffee tracking chaincode running as plain Hyperledger Fabric chaincode.

`chaincode_proxy` and the `chaincode` on a machine emulating ARM TZ with QEMU and once running them on the Raspberry Pi. For the emulation with QEMU we chose a host machine with 4 cores (eiger-8.maas) and set `SMP` to 4 to simulate the number of cores in the Raspberry Pi.

Again we run each experiment 3 times and take the results of the run for which the maximum throughput over all clients is the highest.

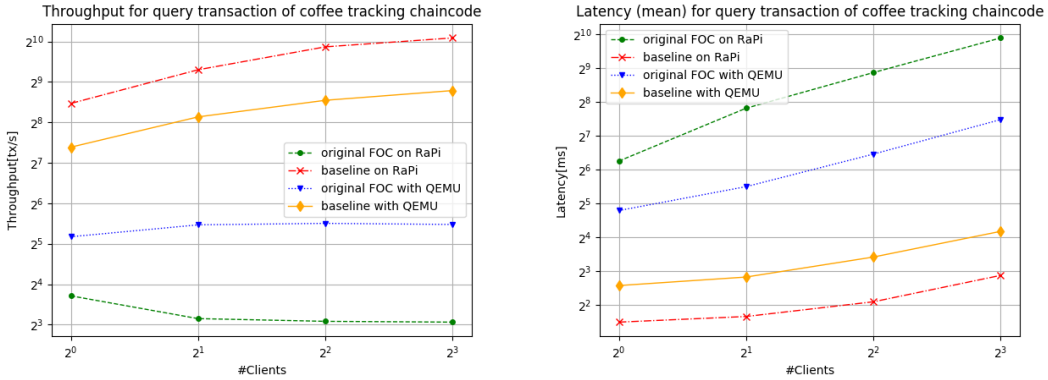

Figure 6.2: Throughput and latency for the execution of the coffee tracking chaincode `query` transaction invoked by different numbers of clients in parallel. Measurements are done for the original FOC implementation and for the baseline once on the Raspberry Pi (`CFG_NUM_THREADS` =4) and once with QEMU (`SMP`=4, `CFG_NUM_THREADS`=4, machine eiger-8.maas). The latency is the mean over the latency of all single transactions in ms.

The detailed results are displayed in Table B.2 in the appendix. We plot the throughput and latency for the baseline (with the following gRPC configurations: `NUM_CQS`=2, default values for `MIN_POLLERS, MAX_POLLERS`[20]) and for the original FOC implementation in Figure 6.2. Our evaluation:

- Considering the original FOC implementation, we can observe that throughput does not increase with an increasing number of clients in case of the Raspberry Pi. In case of QEMU, there is a slight

increase with about factor 1.2 from 1 to 2 clients. For more clients, throughput stagnates. This is the bottleneck we have already observed in the refined experiments before.

- In case of the baseline, throughput increases up to 8 clients. With the `chaincode_proxy` and the `chaincode_wrapper` running on the Raspberry Pi, we can increase throughput by factor 3.1 for 8 clients compared to 1 client. For QEMU, it is an increase by factor 2.6 from 1 to 8 clients.
- Considering the experiment on the Raspberry Pi for 1 client, the throughput of the baseline is about 27 times higher than with the original FOC implementation. In case of 8 clients, we even have an increase with about factor 130. For QEMU, the difference is less extreme: throughput is between 5 to 10 times higher (depending on the number of clients) for the baseline compared to the original FOC implementation.

The observations show, that the execution of the `chaincode` inside the secure world comes with its cost in terms of throughput.

6.4.4 Experiment 2: Latency Breakdown

Experiment. We measure the duration of the different subparts of the coffee tracking chaincode `query` transaction executed with the original FOC implementation. The measured timespans are graphically displayed in Figure 6.3.

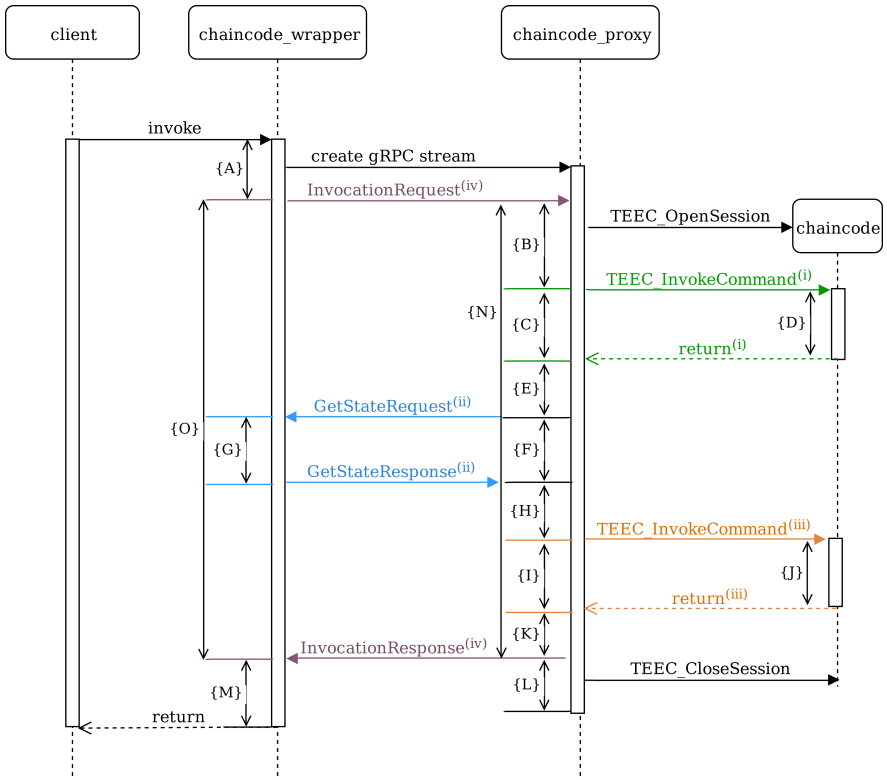

Figure 6.3: Timespans measured for the `query` transaction of the implemented coffee tracking chaincode in FOC are indicated with capital letters. The round-trip times for the colored arrows labelled with roman numerals will be calculated as differences of the measured timespans, for example round-trip time for (iv) = O - N.

Source: figure drawn by C.M.

A is the initialization phase at the chaincode_wrapper. B is the initialization at the chaincode_proxy. It includes initializing the context (TEEC_InitializeContext), opening the session (TEEC_OpenSession), allocating the shared memory and passing all message parameters to this shared memory. D is the preparation of the GetState in the chaincode. E is the time used by the chaincode_proxy for forwarding this GetState to the chaincode_wrapper via GetStateRequest. G is the handling of the GetStateRequest at the chaincode_wrapper (i.e. call of the ledger). H is the postprocessing of the GetStateResponse in the chaincode_proxy. J is the preparation of the execution response message in the chaincode. K is the forwarding of this message at the chaincode_proxy via a InvocationResponse. L is the time used for releasing the resources, for closing the session (TEEC_CloseSession) and for finalizing the context (TEEC_FinalizeContext) at the chaincode_proxy. M is the finalization at the chaincode_wrapper. The timespans C, I, N and O are measured for calculating the round-trip times (arrows i - iv).

The measurements are done once for 1 client repeatedly invoking the query transaction during 30 seconds because with 1 client we have aimed best results in terms of throughput and latency in experiment 1. And once for 2 clients repeatedly invoking the query transactions in parallel during 30 seconds because we want to investigate the bottleneck observed in experiment 1.

The experiment is executed twice: once running the chaincode_proxy and the chaincode on a machine emulating ARM TZ with QEMU (SMP = 4) and once running them on the Raspberry Pi; with CFG_NUM_THREADS = 4 in both cases. Regarding the gRPC configuration, we set the default values for NUM_CQS, MIN_POLLERS, MAX_POLLERS[20].

Result. For each setup we have run the experiment 3 times and taken the results for the run with the highest throughput. The results are summarized in Table 6.4. An evaluation of the results is given in the following:

- If we sum up the means of the different parts for one transaction (A, B, D, E, G, H, J, K, M, (i), (ii), (iii), (iv) = sum without L), we should achieve the mean latency measured in experiment 1. This is actually the case.
- For both, the setup with QEMU and the Raspberry Pi, phase B makes up 65-75% of the total time (= sum without L). We can observe that the vast majority of phase B is needed for TEEC_OpenSession. This knowledge has given us some hints where to locate the throughput bottleneck experienced in experiment 1.
- Each of the average times for the parts D, E, G, H, J, K, M is smaller than 1 ms and is less then 1 % of the total latency (sum without L).
- There are some observations related to the timespans in the Raspberry Pi which would require more research in the future: With the Raspberry Pi, timespan A doubles from 1 to 2 clients whereas in the setup with QEMU, timespan A stays stable from 1 to 2 clients. Furthermore, on the Raspberry Pi, the round-trip times for TEEC_InvokeCommand (arrows (i) and (ii)) is less then 1 ms in case of 1 client, but more than 8 ms in case of 2 clients.

eiger-8.maas, QEMU (SMP = 4, CFG_NUM_THREADS = 4)										
	1 client					2 clients				
	median	**mean**	**std**	**min**	**max**	**median**	**mean**	**std**	**min**	**max**
A	0.888	0.929	0.169	0.637	1.828	1.006	1.067	0.315	0.590	4.896
B	17.732	17.788	0.516	16.551	21.822	32.834	32.944	2.539	18.687	86.876
thereof										
TEEC_OpenSession	16.815	16.865	0.514	15.662	20.956	31.856	31.930	2.492	17.285	84.353
D	0.0	0.234	0.424	0.0	1.0	0.0	0.186	0.389	0.0s	1.0
E	0.016	0.018	0.011	0.014	0.238	0.017	0.020	0.016	0.011	0.293
G	0.002	0.003	0.002	0.002	0.020	0.003	0.004	0.003	0.001	0.040
H	0.014	0.017	0.025	0.012	0.505	0.011	0.014	0.011	0.008	0.191
J	0.0	0.134	0.341	0.0	1.0	0.0	0.141	0.351	0.0	2.0
K	0.018	0.020	0.018	0.017	0.337	0.015	0.019	0.037	0.012	0.835
L	8.476	8.682	0.530	7.915	14.001	9.431	9.928	3.025	8.071	54.247
M	0.015	0.016	0.009	0.006	0.054	0.015	0.017	0.010	0.006	0.108
arrows (i)	-	1.410	-	-	-	-	1.662	-	-	-
arrows (ii)	-	1.508	-	-	-	-	1.515	-	-	-
arrows (iii)	-	1.468	-	-	-	-	1.658	-	-	-
arrows (iv)	-	4.227	-	-	-	-	7.057	-	-	-
sum without L	-	27.772	-	-	-	-	46.303	-	-	-

Raspberry Pi (CFG_NUM_THREADS = 4)										
	1 client					2 clients				
	median	**mean**	**std**	**min**	**max**	**median**	**mean**	**std**	**min**	**max**
A	5.345	6.163	1.915	4.788	10.646	10.498	12.009	4.202	4.750	31.487
B	67.245	61.142	6.376	54.582	67.980	167.477	165.862	20.218	81.082	224.546
thereof										
TEEC_OpenSession	67.088	60.995	6.372	54.445	67.826	167.334	165.719	20.223	80.922	224.381
D	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
E	0.008	0.009	0.003	0.006	0.026	0.008	0.008	0.004	0.002	0.022
G	0.002	0.003	0.002	0.002	0.014	0.002	0.003	0.001	0.001	0.011
H	0.005	0.006	0.005	0.003	0.037	0.004	0.005	0.005	0.001	0.053
J	0.0	0.0	0.0	0.0	0.0	0.0	0.011	0.104	0.0	1.0
K	0.005	0.006	0.002	0.004	0.029	0.006	0.006	0.003	0.002	0.032
L	24.096	19.677	4.592	14.860	24.414	56.086	103.365	69.695	23.505	248.155
M	0.008	0.013	0.008	0.006	0.046	0.013	0.014	0.008	0.006	0.076
arrows (i)	-	0.153	-	-	-	-	16.482	-	-	-
arrows (ii)	-	0.964	-	-	-	-	2.536	-	-	-
arrows (iii)	-	0.225	-	-	-	-	8.166	-	-	-
arrows (iv)	-	6.985	-	-	-	-	11.248	-	-	-
sum without L	-	75.667	-	-	-	-	216.350	-	-	-

Table 6.4: Latency breakdown in ms for the `query` transaction of the coffee tracking chaincode. The experiment was executed emulated with QEMU and on the Raspberry Pi with different numbers of clients (1, 2) invoking the transaction in parallel for 30 seconds. Milliseconds are the most accurate time unit we can measure in the secure world, therefore measured timespan duration in the secure world may be 0.

6.4.5 Experiment 3: Power

Experiment. We measure the power at the machine running OP-TEE (i.e. the `chaincode_proxy` and the `chaincode TA`) during the execution of the coffee tracking chaincode `query` transaction in FOC. The measurements are done once with 1 client repeatedly invoking the `query` transaction during 30 seconds and once with 2 clients repeatedly invoking the transaction in parallel during 30 seconds.

The experiment is executed twice: once running the `chaincode_proxy` and the `chaincode` on a machine emulating ARM TZ with QEMU (`SMP` = 4) and once running them on the Raspberry Pi; with `CFG_NUM_THREADS` = 4 in both cases. Regarding the gRPC configuration, we set default values for `NUM_CQS`, `MIN_POLLERS`, `MAX_POLLERS`[20].

From the power measurements, we calculate the average consumed energy per transaction.

Expectation. We expect that the execution is slower but needs less energy per transaction when running OP-TEE on the Raspberry Pi compared to running OP-TEE on a machine (eiger-8.maas) emulating ARM TZ with QEMU.

Result. For each setup, we run the experiment 3 times and take the results of the run with the lowest energy consumption per transaction. The results are displayed in Table 6.5. The expectation is confirmed. The throughput achieved with the Raspberry Pi is about 3 to 5 times lower than the one with QEMU but the execution of one `query` transaction on the Raspberry Pi needs only about 1/10 of the energy used by QEMU.

	1 client			2 clients		
	[J]	**# tx**	**[J/tx]**	**[J]**	**# tx**	**[J/tx]**
eiger-8.maas, QEMU (`SMP` = 4, `CFG_NUM_THREADS` = 4)	1918.0	1115	1.72	2181.5	1302	1.68
Raspberry Pi (`CFG_NUM_THREADS` = 4)	61.66	393	0.16	62.32	268	0.23

Table 6.5: The total energy consumption in Joules of the machine running OP-TEE for the execution of the coffee tracking `query` transaction during 30 seconds is displayed in column **[J]**. The total number of successful transactions is displayed in column **# tx**. The column **[J/tx]** shows the used Joules per transactions in average.

6.4.6 Future Work

We have located the throughput bottleneck faced in experiment 1 and found a potential explanation: `TEEC_OpenSession` is at least partly serialized [ogi19a]. To confirm this explanation, we would need to modify our FOC prototype code and re-execute the performance measurements. Instead of having a separate session for each `InvocationRequest`, we could open a fixed number of sessions (all with their own context) at the startup of the `chaincode_proxy` and store them in a global pool. The number of sessions should be equal to the value of `CFG_NUM_THREADS` for maximal concurrency. For each incoming `InvocationRequest` the `chaincode_proxy` will take one session from the pool (if there is any or else wait for one) and put it back to the pool once the `InvocationRequest` has been sent back to the `chaincode_wrapper`. With this change in the FOC implementation, we could avoid the serialization due to the global mutex in the `TEEC_OpenSession` calls.

The microbenchmarks we have done could be refined and extended. For example, one could investigate how a change in the transaction size influences the throughput and latency.

In the future, one could measure the throughput and latency for all phases (execution, ordering and validation) of Hyperledger Fabric and not only for the execution of the transaction itself. In our experiments,

we have directly invoked the `chaincode_wrapper` without using any peer and orderer (`shim.MockStub`). For future macrobenchmarks, one could use Hyperledger Fabric client SDK[22] and have a complete Hyperledger Fabric network with peers and orderers.

One could also compare the performance of FOC and FPC in future work, although such results must be considered with caution since Intel SGX and ARM TZ with OP-TEE are two different technologies and do not provide the same features and guarantees.

We have focused on the `query` transaction of the implemented coffee tracking chaincode in FOC. One could compare the transaction and latency of the `query` (only has one `GetState`) and `create` (needs one `GetState` and one `PutState`) transaction in future work.

We have mainly varied the number of clients invoking transactions in parallel. There would be more parameters as for example the network characteristic one could consider in future measurements. Furthermore, one could investigate if and how asynchronous gRPC communication[23] could improve performance.

[22] `https://hyperledger-fabric.readthedocs.io/en/release-1.4/glossary.html#software-development-kit-sdk`, last accessed on 17.08.2019

[23] `https://grpc.io/docs/tutorials/async/helloasync-cpp/`, last accessed on 17.08.2019

7 Conclusion and Future Work

The first contribution is the design and implementation of a prototype called FOC for Hyperledger Fabric chaincode execution with ARM TZ and OP-TEE. With the coffee tracking chaincode, we have demonstrated a fully working smart contract running inside the secure world. The contribution of the thesis may be useful in the context of IoT. It ensures confidentiality of the smart contract (its logic) and the processed data. In contrast to the two other properties of the CIA triad [And11, XWS+17], confidentiality is not inherently guaranteed by the blockchain network.

The second contribution is the exploration of the limits with ARM TZ and OP-TEE. During the transfer of FPC from Intel SGX to ARM TZ with OP-TEE, we have faced some challenges: missing support for remote attestation in ARM TZ and OP-TEE [Sta18, ogi19c, CD16b] and missing hardware support for some security features of OP-TEE [Lin]. This leads to the following consequences for the designed and implemented FOC prototype:

- The intuitive approach of running the peer in the normal world of OP-TEE could not be realised due to missing Docker support. Therefore in FOC, the peer is decoupled from the ARM TZ node.
- FOC lacks features to fully guarantee confidentiality of smart contracts (their logic) and the processed data: a. Remote attestation of the `chaincode TA`, b. signature of the transaction response by the `chaincode TA`, c. fully secure integrity check of the `chaincode TA` at its load time, d. guarantee of state continuity, e. encrypted `chaincode TA` storage and f. encryption of the operation, the passed states and the execution response message.

Measuring the performance of the implemented FOC prototype (with ARM TZ and OP-TEE running on the Raspberry Pi) has shown that the throughput for the execution of the coffee tracking chaincode `query` transaction is about 27 - 130 times slower (depending on the number of clients invoking transactions in parallel) than the execution of the `query` transaction with our baseline (coffee tracking chaincode placed in the normal world). This shows that the confidentiality guarantee comes with its cost. Furthermore, we have faced a bottleneck in performance measurements with the Raspberry Pi: throughput could not be increased with an increasing number of clients although we had multiple cores and trusted threads available. We have identified a potential reason for this bottleneck and suggested a code change for future work.

7.1 Future Work

In the following, we describe possible future work for the design and the potential improvements related to the implemented prototype.

7.1.1 Design.

- **Architecture.** With the current design, the peer is running decoupled from ARM TZ on a separate node supporting plain Hyperledger Fabric, see Section 4.1. For a productive system it would be practical and more efficient to place the peer in the normal world of ARM TZ.
- **Confidentiality.** In Section 6.1.3 we have described the missing security features of FOC. These features must be considered in the future design and implementation to fully guarantee confidentiality for the chaincode and the processed data.
- **Interoperability of FPC and FOC.** The interoperability of FPC and FOC peers (see Section 6.3) could be considered in a future design and implementation.

7.1.2 Implementation.

- **C, C++ mix.** In OP-TEE, the official host examples are written in C[1]. Since gRPC has no official C language support [gRPb], we use the C++ support of the gRPC framework for writing the gRPC server (`chaincode_proxy`). Therefore, the `chaincode_proxy` is a C++ program but may contain C code as well. This mix of C++ and C code should be cleaned up in the future.
- **GlobalPlatform compatibility.** The `chaincode TA` code (`coffee_tracking_chaincode.c` file) is not fully GlobalPlatform conform. For example the used functions `strlen`, `stroul` and `snprintf` are implemented by OP-TEE[2] but not supported by GlobalPlatform [Glo14]. This is not considered an issue since we just demonstrate the execution of smart contracts with OP-TEE. In the future, fully GlobalPlatform compatible code could be implemented, so that the `chaincode TA` can be transferred to other environments supporting the GlobalPlatform standard.
- **Argument check.** The functions of the coffee tracking chaincode running inside the secure world (`create`, `add`, `query`) should check the number and the type of the received arguments; this is not implemented in the current prototype.
- **Argument encryption.** Currently, the arguments are passed via the Protocol Buffer `bytes .proto` type. When the arguments are encrypted in future work, they should not be passed via this type, but as one unit to avoid the leaking of the number of arguments.
- **Ledger access.** The current implementation of chaincode execution with OP-TEE only provides `GetState` and `PutState` to access the ledger. Support of other APIs of the `ChaincodeStub`[3] is left for future work.
- **Performance.** We have experienced a bottleneck regarding the throughput for the execution of the coffee tracking chaincode `query` transaction in FOC. A potential explanation and the necessary future code changes in order to remove this bottleneck are described in Section 6.4.6. There, we also describes performance measurements left for future work.

[1] `https://github.com/linaro-swg/optee_examples`, last accessed on 14.08.2019

[2] `https://github.com/OP-TEE/optee_os/tree/master/lib/libutils/isoc/include`, last accessed on 24.07.2019

[3] `https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim#ChaincodeStub`, last accessed on 23.07.2019

Bibliography

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: a Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15, 2018.

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 164–186, 2017.

[aD] arm Developer. TrustZone. `https://developer.arm.com/ip-products/security-ip/trustzone`. Last accessed July 9, 2019.

[Amo11] Edward G. Amoroso. *Cyber Attacks*. Butterworth-Heinemann, Burlington USA, 2011.

[And11] Jason Andress. *The Basics of Information Security*. Syngress Press, Waltham USA, 2011.

[ASKP18] Mustafa Al-Bassam, Alberto Sonnino, Michal Król, and Ioannis Psaras. Airtnt: Fair Exchange Payment for Outsourced Secure Enclave Computations. *CoRR*, abs/1805.06411, 2018. URL: `http://arxiv.org/abs/1805.06411`, `arXiv:1805.06411`.

[BCKS18] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric. *CoRR*, abs/1805.08541, 2018. URL: `http://arxiv.org/abs/1805.08541`, `arXiv:1805.08541`.

[BD14] Greg Bellows and Christoffer Dall. Arm TrustZone in QEMU. `https://www.linaro.org/blog/arm-trustzone-qemu/`, September 2014. Last accessed July 22, 2019.

[BDK17] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 222–237, 2017.

[Bec14] Joakim Bech. OP-TEE, open-source security for the mass-market. `https://www.linaro.org/blog/op-tee-open-source-security-mass-market/`, September 2014. Last accessed July 29, 2019.

[Bec16] Joakim Bech. TEE Development With No Hardware - Is That Possible? `https://www.linaro.org/blog/tee-development-with-no-hardware-is-that-possible/`, November 2016. Last accessed June 22, 2019.

[Bel15] Greg Bellows. Testing QEMU Arm TrustZone. https://www.linaro.org/blog/testing-qemu-arm-trustzone/, January 2015. Last accessed July 22, 2019.

[bit] Bitcoin. https://bitcoin.org/en/. Last accessed on August 22, 2019.

[BMSV18] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. Private Data Objects: an Overview. *CoRR*, abs/1807.05686, 2018. URL: http://arxiv.org/abs/1807.05686, arXiv:1807.05686.

[Bra02] Roberta Bragg. *CISSP Certified Information Systems Security Professional - Training Guide*. Pearson IT Certification, Hoboken USA, 2002.

[Bro] Dominik Brodowski. Linux CPUFreq - CPUFreq Governors. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt. Last accessed August 11, 2019.

[CD16a] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access*, 4:2292–2303, 2016.

[CD16b] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016. URL: http://eprint.iacr.org/2016/086.

[CV17] Christian Cachin and Marko Vukolic. Blockchain Consensus Protocols in the Wild (Keynote Talk). In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 1:1–1:16, 2017.

[CZK+18] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *CoRR*, abs/1804.05141, 2018. URL: http://arxiv.org/abs/1804.05141, arXiv:1804.05141.

[DDW+18] Weiqi Dai, Jun Deng, Qinyuan Wang, Changze Cui, Deqing Zou, and Hai Jin. SBLWT: A Secure Blockchain Lightweight Wallet Based on Trustzone. *IEEE Access*, 6:40638–40648, 2018.

[Dev] Google Developers. Protocol Buffers. https://developers.google.com/protocol-buffers/. Last accessed July 22, 2019.

[DP] Bogdan Djukic and Lorenzo Pieri. AnyLedger: Embedded wallet for decentralized IoT. Version 0.8.0. http://www.anyledger.io/whitepaperAnyLedger.pdf. Last accessed on August 19, 2019.

[Dub19] Yetnesh Dubey. CPU Comparison: X86 vs ARM — Will Intel i9 9900K Stay Atop? https://fossbytes.com/cpu-comparison-x86-arm-cpu-benchmark/, April 2019. Last accessed July 9, 2019.

[eth] Ethereum. https://www.ethereum.org/. Last accessed on August 22, 2019.

[Fou] The Linux Foundation. Hyperledger. https://www.hyperledger.org/. Last accessed July 8, 2019.

[fpca] fabric-private-chaincode github repository. https://github.com/hyperledger-labs/fabric-private-chaincode. Last accessed July 9, 2019.

[fpcb] fabric-private-chaincode github repository, enclave.cpp file, encrypted operation. https://github.com/hyperledger-labs/fabric-private-chaincode/blob/master/ecc_enclave/enclave/enclave.cpp#L76-#L130. Last accessed June 20, 2019.

[fpcc] fabric-private-chaincode github repository, enclave.cpp file, signature. https://github.com/hyperledger-labs/fabric-private-chaincode/blob/master/ecc_enclave/enclave/enclave.cpp#L159-#L197. Last accessed June 20, 2019.

[Gar14] Paul Garden. The IoT Requires A New Type Of Low-Power Processor. https://www.electronicdesign.com/communications/iot-requires-new-type-low-power-processor, April 2014. Last accessed July 9, 2019.

[Glo10] Inc. GlobalPlatform. TEE Client API Specification, Version 1.0. https://globalplatform.org/specs-library/, July 2010. Last accessed July 22, 2019.

[Glo14] Inc. GlobalPlatform. TEE Internal Core API Specification, Version 1.1. https://globalplatform.org/specs-library/, June 2014. Last accessed June 25, 2019.

[GMS17] Miraje Gentilal, Paulo Martins, and Leonel Sousa. TrustZone-backed bitcoin wallet. In *Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC 2017, Stockholm, Sweden, January 24, 2017*, pages 25–28, 2017.

[gp-] GlobalPlatform. https://globalplatform.org/. Last accessed July 12, 2019.

[gRPa] gRPC. gRPC Authentication. https://grpc.io/docs/guides/auth/. Last accessed July 10, 2019.

[gRPb] gRPC. gRPC framework. https://grpc.io/. Last accessed July 22, 2019.

[Hyp] Hyperledger. Hyperledger Fabric Documentation. https://hyperledger-fabric.readthedocs.io/en/release-1.4/. Last accessed July 10, 2019.

[ico] ICONFINDER. https://www.iconfinder.com. Last accessed July 11, 2019.

[iDZ16] intel Developer Zone. What does it implies to disable syscall in Intel SGX. https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/539803, 2015/2016. Last accessed on August 22, 2019.

[Jac19] Joab Jackson. Fresh Spectre Vulnerabilities May Force Cloud Providers to Disable Intel Hyper-Threading. https://thenewstack.io/fresh-spectre-exploits-may-force-cloud-providers-to-disable-intel-hyper-threading/, May 2019. Last accessed on August 26, 2019.

[Lim09] ARM Limited. ARM Security Technology Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2005-2009. Last accessed July 10, 2019.

[Lin] Linaro. OP-TEE Documentation. https://optee.readthedocs.io/. Last accessed July 10, 2019.

[Lin16] Linaro. BKK16-110 A Gentle Introduction to Trusted Execution and OP-TEE. https://www.slideshare.net/linaroorg/bkk16110-a-gentle-introduction-to-trusted-execution-and-optee, 2016. Last accessed July 29, 2019.

[LNE+18] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Peter R. Pietzuch, and Emin Gün Sirer. Teechain: Reducing Storage Costs on the Blockchain With Offline Payment Channels. In *Proceedings of the 11th ACM International Systems and Storage Conference, SYSTOR 2018, HAIFA, Israel, June 04-07, 2018*, page 125, 2018.

[NMB+16] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. TrustZone Explained: Architectural Features and Use Cases. In *2nd IEEE International Conference on Collaboration and Internet Computing, CIC 2016, Pittsburgh, PA, USA, November 1-3, 2016*, pages 445–451, 2016.

[ogi16] optee_os github issue. Adding #include<math.h> to a TA. https://github.com/OP-TEE/optee_os/issues/1158, 2016. Last accessed July 10, 2019.

[ogi17] optee_os github issue. Include new 3rd party library into TA and call library function in TA code. https://github.com/OP-TEE/optee_os/issues/1255, 2017. Last accessed July 10, 2019.

[ogi18] optee_examples github issue. Remote Attestation process/protocol for OP-TEE. https://github.com/linaro-swg/optee_examples/issues/26, 2018. Last accessed June 20, 2019.

[ogi19a] optee_client github issue. TA calls by multi-threaded host application. https://github.com/OP-TEE/optee_client/issues/168, August 2019. Last accessed August 13, 2019.

[ogi19b] optee_os github issue. Integrity check & encryption of early TA. https://github.com/OP-TEE/optee_os/issues/3095, 2019. Last accessed June 23, 2019.

[ogi19c] optee_os github issue. remote attestation for OP-TEE. https://github.com/OP-TEE/optee_os/issues/3057, 2019. Last accessed June 20, 2019.

[ogi19d] optee_os github issue. (shared) memory: questions. https://github.com/OP-TEE/optee_os/issues/2964, 2019. Last accessed July 23, 2019.

[ogi19e] optee_os github issue. Socket API (TCP): support for serialization and for any messaging library? https://github.com/OP-TEE/optee_os/issues/2981, 2019. Last accessed July 23, 2019.

[Ose18] Victor Osetskyi. What Is Smart Contracts Blockchain and Its Use Cases in Business. https://dzone.com/articles/what-is-smart-contracts-blockchain-and-its-use-cas-1, June 2018. Last accessed July 10, 2019.

[Pai16] Vijay Pai. gRPC Design and Implementation - Stanford Platform Lab Seminar. https://platformlab.stanford.edu/Seminar20Talks/gRPC.pdf, April 2016. Last accessed August 11, 2019.

[QEM] QEMU. QEMU the FAST! processor emulator. https://www.qemu.org/. Last accessed July 22, 2019.

[Res] Microsoft Research. CCF's documentation. `https://github.com/Microsoft/CCF`. Last accessed August 19, 2019.

[rip] Ripple. `https://www.ripple.com/`. Last accessed on August 22, 2019.

[RMC+18] Ana Reyna, Cristian Martín, Jaime Chen, Enrique Soler, and Manuel Díaz. On blockchain and its integration with IoT. Challenges and opportunities. *Future Generation Comp. Syst.*, 88:173–190, 2018.

[SAB15] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 57–64, 2015.

[SAG+16] Carlton Shepherd, Ghada Arfaoui, Iakovos Gurulian, Robert P. Lee, Konstantinos Markantonakis, Raja Naeem Akram, Damien Sauveron, and Emmanuel Conchon. Secure and Trusted Execution: Past, Present, and Future - A Critical Review in the Context of the Internet of Things and Cyber-Physical Systems. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, pages 168–177, 2016.

[SDF+19] Vasilios A. Siris, Dimitrios Dimopoulos, Nikos Fotiou, Spyros Voulgaris, and George C. Polyzos. IoT Resource Access utilizing Blockchains and Trusted Execution Environments. In *2019 Global IoT Summit, GIoTS 2019, Aarhus, Denmark, June 17-21, 2019*, pages 1–6, 2019.

[Sel16] Surenthar Selvaraj. Overview of an Intel Software Guard Extensions Enclave Life Cycle. `https://software.intel.com/en-us/blogs/2016/12/20/overview-of-an-intel-software-guard-extensions-enclave-life-cycle`, December 2016. Last accessed July 9, 2019.

[Sta18] StackExchange. Does the ARM TrustZone technology support sealing a private key under a code hash? `https://security.stackexchange.com/questions/56203/does-the-arm-trustzone-technology-support-sealing-a-private-key-under-a-code-has`, 2014/2018. Last accessed August 15, 2019.

[Sza94] Nick Szabo. Smart Contracts. `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`, 1994. Last accessed 10 July, 2019.

[Tit18] Daniel Tittlus. *Blockchain - Smart Contracts, Smart Cities, Smarte Welt*. epubli, Berlin Germany, 2018.

[XWS+17] Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. A Taxonomy of Blockchain-Based Systems for Architecture Design. In *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, pages 243–252, 2017.

[YXC+18] Rui Yuan, Yubin Xia, Haibo Chen, Binyu Zang, and Jan Xie. ShadowEth: Private Smart Contract on Public Blockchain. *J. Comput. Sci. Technol.*, 33(3):542–556, 2018.

[ZWAS18] Lijing Zhou, Licheng Wang, Tianyi Ai, and Yiru Sun. BeeKeeper 2.0: Confidential Blockchain-Enabled IoT System with Fully Homomorphic Computation. *Sensors*, 18(11):3785, 2018.

A
Pointers to Code

An overview of the code written during the master thesis is given in the following:

- **Coffee tracking chaincode for Hyperledger Fabric.** To get to know Hyperledger Fabric, we have written a chaincode (in Go programming language) which tracks the consumed coffees of registered people. Since this chaincode is also used for performance measurements it can be found in the `performance_and_power_measurements` directory of the private github repository `fabric-optee-chaincode`.[1]
- **Coffee tracking chaincode for FPC.** To get to know FPC, we have written a chaincode in C++ which tracks the coffee consume for registered offices. The chaincode, the README with a detailed description of the example and the script for running the example can be found in the private github repository `fabric-secure-chaincode-my-example`.[2] The repository itself is a copy of the official `fabric-private-chaincode` repository [fpca], only the coffee tracking example is part of the master thesis.
- **Implementation of the FOC prototype.** The `master` branch of the private github repository `fabric-optee-chaincode`[3] contains the implemented FOC prototype.
- **Code for performance and power measurements.** The `performance_power_measurements` directory in the `master` branch of the private github repository `fabric-optee-chaincode`[4] as well as the `performance-latency-breakdown` branch contain the code used for performance and power measurements.

[1] `https://github.com/piachristel/fabric-optee-chaincode/tree/master/performance_power_measurements/plain-fabric/coffee_tracking_chaincode/`, last accessed on 14.08.2019

[2] `https://github.com/piachristel/fabric-secure-chaincode-my-example/tree/master/ecc_enclave/enclave/coffee_tracking/`, last accessed on 14.08.2019,
`https://github.com/piachristel/fabric-secure-chaincode-my-example/blob/master/fabric/sgxconfig/demo/run_sgx_coffee_tracking.sh`, last accessed on 14.08.2019

[3] `https://github.com/piachristel/fabric-optee-chaincode/`, last accessed on 14.08.2019

[4] `https://github.com/piachristel/fabric-optee-chaincode/tree/master/performance_power_measurements`, last accessed on 14.08.2019

B

Performance Measurements

cervino-4.maas, QEMU (SMP = 8, CFG_NUM_THREADS = 8)																			
	1 client			**2 clients**					**4 clients**					**8 clients**					
	L	T	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R
original	44.667	22.267	0	76.547	1.714	26.067	1.171	0	149.537	3.348	26.667	1.198	0	320.051	7.165	24.667	1.108	0.033
original (NUM_CQS=4)	43.135	23.167	0	74.972	1.738	26.667	1.151	0	153.382	3.556	26	1.122	0	328.561	7.617	23.25	1.004	0.143
baseline	8.072	123.7	0	9.786	1.212	204.2	1.651	0	18.391	2.278	217.3	1.757	0	32.738	4.056	243.9	1.972	0
baseline (NUM_CQS=4)	9.64	113.5	0	9.366	0.972	213.4	1.88	0	14.254	1.479	280.3	2.47	0	24.765	2.569	322.5	2.841	0
baseline (NUM_CQS=4, MIN_POLLERS=1, MAX_POLLERS=1)	10.442	95.7	0	9.742	0.933	205.1	2.143	0	14.53	1.392	275.1	2.875	0	29.309	2.807	272.6	2.848	0
baseline (NUM_CQS=8)	11.38	87.8	0	9.55	0.839	209.3	2.384	0	14.552	1.279	274.7	3.129	0	26.748	2.35	298.5	3.4	0
baseline (NUM_CQS=8, MIN_POLLERS=1, MAX_POLLERS=1)	9.046	110.4	0	9.525	1.053	209.9	1.901	0	14.686	1.623	272.2	2.466	0	35.105	3.881	227.7	2.062	0
baseline (NUM_CQS=16)	8.835	113.1	0	9.42	1.066	212.1	1.875	0	14.88	1.684	268.6	2.375	0	28.04	3.174	284.6	2.516	0
without gRPC	34.3	21.467	0	60.507	1.764	27.067	1.261	0	114.761	3.346	30.667	1.429	0	202.467	5.903	31.5	1.467	0.001
normal world threading	-	-	-	-	-	-	1.633	-	-	-	-	1.52	-	-	-	-	1.394	-

cervino-4.maas, QEMU (SMP = 8, CFG_NUM_THREADS = 16)																			
	1 client			**2 clients**					**4 clients**					**8 clients**					
	L	T	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R
without gRPC	34.024	21.533	0	60.506	1.778	26.933	1.251	0	117.129	3.443	30	1.393	0	209.262	6.15	30.5	1.416	0.004

cervino-4.maas, QEMU (SMP = 8, CFG_NUM_THREADS = 4)																			
	1 client			**2 clients**					**4 clients**					**8 clients**					
	L	T	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R
without gRPC	35.437	20.867	0	62.515	1.764	26.2	1.256	0	123.05	3.472	28.267	1.355	0	229.248	6.469	28.033	1.343	0.002

Table B.1: Throughput and latency measurement for the execution of the coffee tracking chaincode `query` transaction. Measurements are done for the original FOC implementation, for the baseline, for FOC without gRPC and for dummy normal world threading. Throughput is measured in [tx/s] and written in column T. The latency in column L is the mean over the latency of all single transactions in ms. The column T-F shows the factor by which the throughput has changed relative to the throughput for 1 client. Same for the latency in column L-F. The failure rate can be found in column F-R.

eiger-8.maas, QEMU (SMP = 4, CFG_NUM_THREADS = 4)																	
	1 client			**2 clients**					**4 clients**					**8 clients**				
	L	T	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R
original	27.715	36.067	0	45.211	1.631	44.2	1.226	0	87.972	3.147	45.367	1.258	0	178.373	6.436	44.5	1.234	0.015
baseline	6.154	162.5	0	7.679	1.248	260.3	1.602	0	13.991	2.273	285.5	1.757	0	22.95	3.729	348.3	2.143	0
baseline (NUM_CQS=2)	5.971	167.3	0	7.11	1.191	281.1	1.68	0	10.704	1.793	373.5	2.233	0	18.104	3.032	441.3	2.638	0
baseline (NUM_CQS=4)	5.797	172.4	0	6.74	1.163	296.6	1.72	0	10.177	1.756	392.7	2.278	0	17.589	3.034	454.3	2.635	0
baseline (NUM_CQS=8)	5.781	172.9	0	6.63	1.147	301.5	1.744	0	10.075	1.743	396.7	2.294	0	18.081	3.127	441.8	2.555	0
without gRPC	19.778	37.267	0	35.159	1.778	46.6	1.25	0	67.738	3.425	51.367	1.378	0	132.661	6.708	50.6	1.358	0.002
normal world threading	-	-	-	-	-	-	1.656	-	-	-	-	1.535	-	-	-	-	1.543	-

Raspberry Pi (CFG_NUM_THREADS = 4)																		
	1 client			**2 clients**					**4 clients**					**8 clients**				
	L	T	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R	L	L-F	T	T-F	F-R
original	76.26	13.1	0	225.013	2.951	8.867	0.677	0	466.741	6.12	8.467	0.646	0	947.936	12.43	8.35	0.637	0
baseline	2.801	356.8	0	3.233	1.154	618.3	1.733	0	4.767	1.702	838.8	2.351	0	8.413	3.003	950.4	2.664	0
baseline (NUM_CQS=2)	2.825	353.8	0	3.171	1.123	630.3	1.782	0	4.284	1.516	933.4	2.638	0	7.355	2.604	1087.1	3.073	0
baseline (NUM_CQS=4)	2.847	351	0	3.204	1.125	623.9	1.777	0	4.159	1.461	961.4	2.739	0	7.428	2.609	1076.4	3.067	0
baseline (NUM_CQS=8)	2.882	346.8	0	3.21	1.114	622.7	1.796	0	4.214	1.462	948.8	2.736	0	7.514	2.607	1064.3	3.069	0
without gRPC	54.571	14.467	0	166.111	3.044	9.7	0.671	0	257.858	4.725	8.5	0.588	0	592.218	10.852	8.6	0.594	0
normal world threading	-	-	-	-	-	-	1.891	-	-	-	-	3.358	-	-	-	-	3.697	-

Table B.2: Throughput and latency measurement for the execution of the coffee tracking chaincode `query` transaction. Measurements are done for the original FOC implementation, for the baseline, for FOC without gRPC and for dummy normal world threading. Throughput is measured in [tx/s] and written in column T. The latency in column L is the mean over the latency of all single transactions in ms. The column T-F shows the factor by which the throughput has changed relative to the throughput for 1 client. Same for the latency in column L-F. The failure rate can be found in column F-R.