MASTER IN COMPUTER SCIENCE

Go Language Support in Hyperledger Fabric Private Chaincode

Master Thesis

Riccardo Zappoli

Université de Fribourg
Faculté des sciences et de médecine

August 2022

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

Smart contracts are a great invention that enables running applications in a distributed way and enables them to take advantage of distributed ledgers' capabilities. Since their inception, the goal of a lot of new systems and frameworks has been to make these smart contracts easier and easier to write and to use, more and more powerful and with ever-increasing features. Hyperledger Fabric made the developing process more straightforward, by allowing smart contracts, here called chaincodes, to be written in well-known programming languages, *i.e.,* Java and Go. Hyperledger Fabric Private Chaincode (FPC) went a step further, by allowing them to treat confidential data within Intel SGX enclaves. Thus, it is in this perspective that this thesis aims to go another step further in that direction, by allowing FPC chaincodes to be written in Go, making them as easy to use as with Fabric, and as secure as with FPC.
Our approach is to recreate a Go Chaincode Enclave and to run it inside an Intel SGX enclave with EGo, a new commercial framework whose purpose is to make Golang applications confidential by allowing them to run with Intel SGX. This is implemented in a new module within FPC, where we have translated the FPC Stub Interface to a Go version and took advantage of an existing mock enclave to make it handle real chaincodes. After evaluating the project, we found a slightly higher end-to-end latency with our solution compared to the classic version of FPC, but with a difference not significative enough to outweigh the benefits of the extension. We show these benefits in a usability evaluation of the extension, where we run existing Golang chaincodes samples with our solution. While remote attestations and not supported due to compatibility issues, making the extension an unsafe proof-of-concept at the moment, we are confident in its ability to ever so simplify Fabric confidential chaincodes in the future.

Christian Cachin, Cryptology and Data Security Research Group, Institut für Informatik, Universität Bern, Supervisor.

Marcus Brandenburger, Zurich Research Laboratory, IBM Research, Co-Supervisor.

Contents

1 Introduction

This chapter introduces the present work, set out some background, explains why it is needed and how this thesis contributes to the particular project that is Fabric Private Chaincode. A detailed structure of this document is present at the end of the chapter.

1.1 Context and Motivation

Secure computation on a blockchain, where both confidentiality and integrity are ensured, seems at first like a far-fetched idea that could not be implemented in practice. In that regard, the work that IBM's and Intel's researchers have pulled together by developing *Fabric Private Chaincode* (FPC) [1] is fairly impressive. How about running smart contracts in a trusted execution environment (TEE)?

Here IBM provides the blockchain environment; Hyperledger Fabric, which uses a form of smart contracts called *chaincodes*, while Intel provides the TEE with its *Software Guard Extensions* (SGX). When we have started this thesis, FPC was already a fairly advanced project. Its *Request for Comments* [2] has enabled it to be accepted for Hyperledger Fabric, and while still in active development, it already works well as a prototype.

It is in this context that we have been put in touch with the developing team, who suggested us this project to extend the language support of FPC. Indeed, initially, FPC was only capable of running non-standard chaincodes written in C++ due to compatibility requirements for SGX at the time it was developed [2]. But perhaps, a new SGX tooling, released in the meantime, would permit Go written chaincodes to be executed and could feasibly be incorporated in the project.

There are several reasons to why we would add support for Golang particularly, and they are being explained in detail later on. It mainly comes down to achieving a native compatibility with most common, already existing, standard Fabric chaincodes, and removing the requirement of rewriting them in C++. Also, it goes without saying that such a transition would massively simplify the existing code base, and make most developers happier!

1.2 Contribution

Problem Statement In this thesis, we are aiming (1) to understand the technologies behind FPC, (2) to add support for Golang chaincodes within FPC, by compiling existing Go written chaincodes and running them with FPC, and (3) to evaluate such integration, by comparing its performance with classic FPC and assessing its usability.

On GitHub, we forked the existing FPC project [3] and added the "golang-support" branch, containing all the contributions of the proposed extension and adding a new module to the project. The new module, "ecc_go", is mostly based on existing code and interfaces, and contains the tools for using native Go written chaincodes mainly with the same commands and interfaces as FPC already does.

A *readme* containing all the specificities of running such chaincodes in FPC is available directly on the GitHub page of the module [4]. We do hope that, once reviewed and after a proper pull request, the proposed extension will be added directly into the main branch of FPC's source code.

1.3 Thesis Structure

In Chapter 2, we present the necessary background to understand the present work. In particular, we introduce the Blockchain and Trusted Executions Environments, which are the two main technologies used in this project. The Chapter 3 dives deeper into Fabric Private Chaincode, we explain its architecture and discuss existing limitations.

Approach and implementation details to enable Golang support for FPC are presented in Chapter 4 and Chapter 5 respectively. The approach section focuses on high-level solutions and new SGX tools resolving the thesis's problem, while the implementation section focuses on how exactly, and by which means, these solutions have been integrated to FPC.

In Chapter 6, an evaluation of the implementation is conducted through the axis of performance, security and usability. It is in this part that the results are presented. Finally, Chapter 7 presents work related to this project, and Chapter 8 concludes the thesis, highlighting where future work on this implementation should be conducted.

2
Background

In this chapter, we discuss the frameworks and technologies that were used in the scope of this project. First, a brief introduction of what a blockchain is, and then more precisely the one used in this project, *Hyperledger Fabric*. Following on, a brief introduction of Trusted Execution Environments, and then more precisely the one used in this project, *Intel SGX*, along with more details on the attestation mechanism and the available development environments.

2.1 Distributed Ledgers and Blockchain

In a distributed architecture, a system that maintains a shared list of immutable records implemented in the form of a peer-to-peer network is called a *distributed ledger* [5]. To really be immutable, such system needs to guarantee both integrity and consistency.

In this context, integrity means that any change of the ledger state cannot change a previous state, *i.e.,* that an already written record cannot be deleted or modified. On the other hand, consistency means that any change of the ledger state cannot be in contradiction with a previous state *i.e.,* that all new records must stay in a logical sequence with the previous ones. Because of this, consensus algorithms are an integral part of these systems, ensuring that all records appear in the same order in every copy of the ledger, so participants in the network – called *nodes* or *peers* – can validate transactions [5].

Nowadays, distributed ledgers are most prominently known from their use in cryptocurrencies, but can also be used to store data or to run computer programs. A distributed ledger can be implemented in different ways, but the most widely recognised and used is the blockchain [6]. Its name comes from the consensus protocol used by the nodes; they arrange transactions into blocks, and build a hash chain over the blocks, a *blockchain*. Blockchains first appeared with Bitcoin [7] and are largely viewed as a viable technology for running reliable digital exchanges and powering cryptocurrencies.

Blockchains can be differentiated into public and private ones, or more formally, permissionless and permissioned ones [8]. In a public, or permissionless, blockchain anyone can participate anonymously. They often use a native coin for consensus and frequently rely on a *proof of work* and economic incentives for members to keep it running, *e.g., Bitcoin* [7] or *Ethereum* [9]. Other blockchains prefer a *proof of stake* system, where incentives derives from peers' own interest in a system that runs properly, often by having a share of the crypto-coins, *e.g., Cardano* [10] or *Algorand* [11].

Permissioned blockchains, on the other hand, operate a blockchain among a group of known members who do have a common aim, but do not entirely trust each other. They could be corporations, universities, associations, NGOs or governments that exchange payments, goods or information, and with various use cases. For instance, there is *TradeLens* [12] which provides a platform for tracking shipping containers and reducing paperwork in global supply chains, or the *KSI Blockchain* [13] which has been developed and is now used by the Estonian government for all its daily operations and public services. And besides, in contrast with permissionless blockchains, the permissioned setting here allows the use of traditional Byzantine-Fault Tolerant (BFT) consensus where identities of the participants are known [8].

As stated above, and firstly demonstrated by Ethereum [9], blockchains are also capable of storing data and running arbitrary, transactions-oriented programs, the so-called *smart contracts*. A blockchain will allow for many distributed programs to run concurrently, but the code should always be considered distrusted, potentially malicious, since it can be deployed by basically everyone. Because of this, on permissionless blockchains, a fee is often charged to run smart contracts, in order to prevent abuse. After that, the security of a smart contract execution is derived directly from the blockchain and the underlying consensus algorithms among nodes, ensuring the integrity and consistency of validated transactions. In addition, smart contracts also benefit from blockchain's features; decentralisation, absence of a third party, *etc.*, which truly makes them trustworthy distributed programs.

A protocol for consensus is needed to order the transactions and to propagate them to all the peers, where each of them will sequentially trigger the execution of the transactions in the same order, ensuring consistency. Such a pattern is known as the *order-execute* architecture, found in many existing blockchain systems *e.g., Ethereum* or *Algorand*, and often needs an additional transaction validation phase *e.g.,* in *Ethereum*, that would be when a block has been mined. Although this architecture may seem obvious in all systems due to the need of atomically ordering the transactions, its requirements, that all nodes must be on the same page and that all transactions must be deterministic, can sometimes be perceived as inherent restrictions of the model [8].

2.1.1 Hyperledger Fabric

Hyperledger Fabric is an open-source permissioned blockchain technology that addresses the limitations evoked previously. It is developed as a framework within the Hyperledger project, under the auspices of the Linux Foundation. Designed from the beginning for enterprise use, it is now used in hundreds of prototypes, proofs of concepts, and real distributed ledger systems from various sectors and use cases [8]. It is developed to serve as a foundation for constructing applications and solutions with a modular architecture, while aiming for robustness, scalability, and privacy [8].

Still designed as a general-purpose blockchain, its smart contracts, here called *chaincodes*, are executed consistently across many peers, which much like other blockchains, gives the appearance of an execution on a single distributed computer. However, it is one of the few systems that support the execution of chaincodes that are written in standard programming languages, namely Go [14], Node.js [15] and Java [16]. According to the authors of the 2018 paper presenting the technology, this makes Fabric the first distributed operating system for permissioned blockchains [8].

Unlike the previously mentioned *order-execute* architecture, Fabric's design uses an *execute-order-validate* architecture, where the three steps may run on different entities in the system. Particularly, when transactions trigger the execution of a chaincode, the result of the execution is "endorsed" by other peers, thereby vouching for the correctness of the execution. The conditions for this process, *i.e.,* the required number of endorsements, which peers are required to issue one, *etc.*, are set in an agreed-upon *endorsement policy*, which can be different for every blockchain. This endorsement mechanism corresponds to a "transaction validation" in other blockchains, which would usually be the final step. Then in a second step, a consensus protocol is responsible to order the transactions, usually through an ordering service. Finally, validation occurs when all the peers validate the transactions and append them to their local copy of the ledger, which prevents inconsistency due to concurrency.

More importantly, both the support for these standard programming languages and the use of an execute-order-validate architecture makes Fabric special, in that it can handle non-deterministic transactions. Indeed, in classic *order-execute* blockchains, smart contracts have to be written in a specifically designed programming language, such as *Solidity*, used by Ethereum, which is intended to eliminate all non-deterministic operations, among other things. That being said, while Fabric does support non-deterministic transactions, it really comes down to which endorsement policy has been chosen, as an endorsement policy requiring all peers to endorse a non-deterministic result might never be satisfied. Therefore, different solutions to this problem have already been proposed [17] [18].

Chaincodes Like a smart contract, a chaincode handles business logic agreed upon by network members, for which it then initialises and manages the ledger state through submitted transactions. It must, though, implement a predefined interface, the methods of which are called in response to these transactions [19]. When a chaincode receives an instantiate or upgrade transaction, the `Init()` method is called so that the chaincode can execute any necessary initialisation, including application state initialisation. When a chaincode receives an invoke transaction, the `Invoke()` method is called to handle transaction proposals.

Another interface that the chaincode "Shim" API must implement is the `ChaincodeStubInter-face`, which is used for various chaincode-related calls, such as getting the arguments of an `Invoke()` transaction with `GetArgs()`, accessing and updating the ledger with `GetState()` and `PutState()` respectively, but also submitting chaincode-to-chaincode invocations.

World State The aforementioned `GetState()` and `PutState()` operations use key-value pairs, the former reading a value from a key and the latter writing a key-value pair to the ledger. The ledger being immutable, these operations are actually just appended to the ledger, and instead of requiring chaincodes to retrace all the modifications applied on a key each time they use one, Fabric introduces the concept of *world state* [20]. The world state is a representation of the ledger, it is a key-value store maintaining the aggregated state of all valid transactions applied on the ledger.

Within the world state, each chaincode is associated with a unique namespace, separate from other chaincodes. Therefore, during an execution, a chaincode can only access the key-value pairs of its own namespace. It is still possible, however, for a chaincode to access another chaincode's namespace if it performs a chaincode-to-chaincode invocation [21].

Transaction Flow Figure 2.1 shows a typical invocation of a transaction on a Fabric chaincode, with all the phases of the *execute-order-validate* scheme. The transaction flow starts with a request, where the client sends a transaction proposal to the endorsing peers, specifying the target chaincode and the input arguments (1). Then, the peer executes the function and creates an endorsement by cryptographically signing the result of the execution, and sends back to the client the endorsed result as a transaction proposal response (2). The client then collects enough endorsements to satisfy the endorsement policy, assembles a transaction and submits it for ordering (3). Afterwards, the ordering service checks if the transaction was submitted by a client that is allowed to submit transactions to the network. Then, it collects a number of transactions, assign them to a block and broadcasts the block to all the peers (4). Finally, each peer in the network validates the execution result and endorsement for itself, marks the transaction as valid or invalid, and appends it to its copy of the ledger.

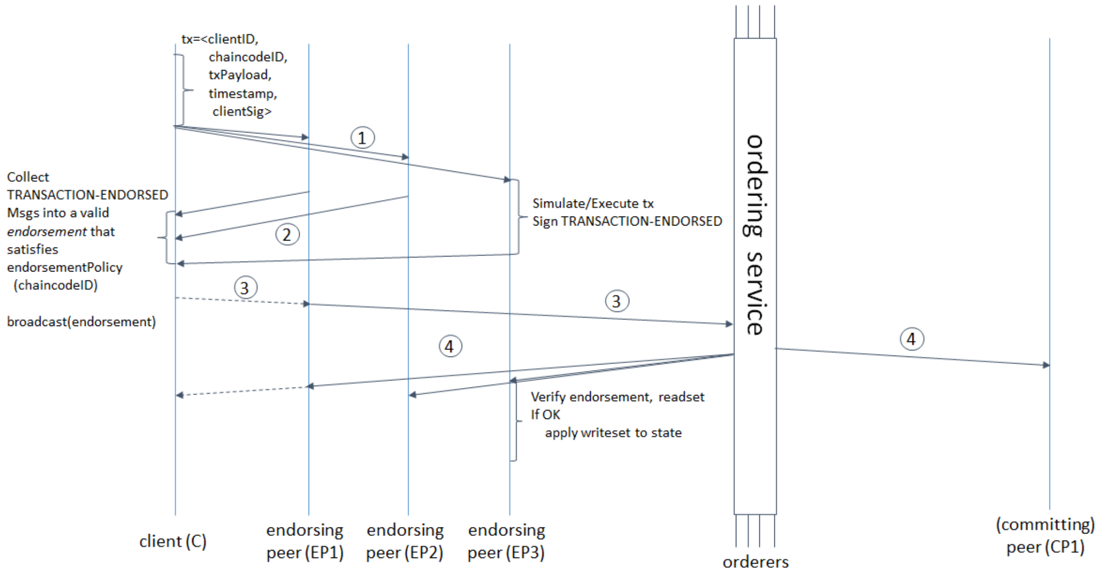
tx=<clientID, chaincodeID, txPayload, timestamp, clientSig>
1
Collect TRANSACTION-ENDORSED Msgs into a valid *endorsement* that satisfies endorsementPolicy (chaincodeID)
2
Simulate/Execute tx
Sign TRANSACTION-ENDORSED
3
broadcast(endorsement)
3
ordering service
4
4
Verify endorsement, readset
If OK
apply writeset to state
client (C)
endorsing peer (EP1)
endorsing peer (EP2)
endorsing peer (EP3)
orderers
(committing) peer (CP1)

Figure 2.1: Hyperledger Fabric Transaction Flow [22]

2.2 Trusted Execution Environment

Trusted computing is a technology for code execution relying on cryptographic signatures and specialised hardware. The original idea was to achieve privacy and data protection by verifying the authenticity and integrity of platforms and programs running on secure and temper-proof hardware [23]. To implement this concept, major IT companies, *i.e.,* Intel, AMD, Microsoft, HP and IBM, formed the *Trusted Computing Group* consortium. While its role was to endorse and promoting trusted computing as ways to increase security, by fighting against malware and “unwanted” hardware modifications, it has also been criticised for helping enforcing DRMs and making use of free software more difficult [24].

Indeed, in the late 2000s this consortium developed the Trusted Platform Module (TPM), a secure crypto-processor that brought the concept of trusted computing to most PCs by providing them with a secure environment. More precisely, the TPM is a cryptographic standard for a tamper-evident hardware chip with persistent and volatile storage [23]. It enables its system, among other things, to prove both its hardware and software integrity, to safeguard cryptographic keys within its memory, and to provide a set of cryptographic utilities *e.g.,* as a random number generator, *etc.* [25]. Yet ultimately, these modules only offer some storage and a limited set of APIs for specific trusted computing needs, but they do not provide any isolated execution environment for applications to run within, which is a major drawback in the regard of secure computing [26].

Consequently, a new approach to trusted computing has been imagined, allowing arbitrary code to be executed within a constrained environment that guarantees confidentiality and tamper-resistant execution of its applications [26]. Combined with a distributed architecture, such method would even enable secure remote computing; executing software on a distant computer owned and maintained by a distrusted entity, while still ensuring both confidentiality and integrity of the processed data [23].

This new specialised hardware became the Trusted Execution Environment (TEE), and was first developed by ARM Ltd. as TrustZone [27], a security extension of its own ARM processors family specialised for mobile platforms. Since then, the term quickly took place in marketing materials for chip suppliers and platform providers. In a 2015 paper, wishing to develop a framework for evaluating and comparing TEE solutions, Mohamed Sabt *et al.* [26] offered a clear definition for the concept:

> *"A Trusted Execution Environment (TEE) is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states, and the confidentiality of its code, data and runtime states stored on a persistent memory. In addition, it shall be able to provide remote attestation that proves its trustworthiness for third parties. The content of TEE is not static; it can be securely updated. The TEE resists against all software attacks as well as the physical attacks performed on the main memory of the system. Attacks performed by exploiting backdoor security flaws are not possible."*

The separation kernel is a fundamental part of a TEE, it is the component that ensures the isolated execution and it is intended to simulate a distributed system [26]. It allows multiple systems with varying levels of security to coexist on the same platform. Essentially, it separates the system into partitions and ensures tight isolation between them, with the notable exception of a carefully managed interface, for communication between partitions [26].

Another important aspect raised by this definition that is fundamental for TEEs to work is the remote attestation requirement. Because the TEE would be accessed from a distrusted code base, notably by an operating system, manufacturers have to provide a way to prove that a specific execution is taking place within the isolated execution environment, and not in an eventual fake environment simulated by an intermediate layer. As a result, a unique and concealed private key is permanently stored in the chip, capable of signing a given input when challenged, and producing an attestation that can be verified with the manufacturer's corresponding public key. Remote attestations will be covered more in detail in the following Section 2.2.2.

In the past years, quite a few TEE technologies have been developed by different chips manufacturers. As we have already mentioned, ARM developed TrustZone [27], but there is also AMD that developed a few of them, notably Secure Encrypted Virtualisation (SEV) [28], and Intel developed Software Guard Extensions (SGX) [29]. These various trusted computing mechanisms work differently from one another, and one important aspect on which they might differ is the Trusted Computing Base (TCB) they require.

The TCB denotes the minimal set of hardware and software components absolutely critical to ensure a system's security [30], which in the context of TEEs, is defining which and how many of the hardware and software components will be protected. Because of this, as we will see in the next section covering SGX, the choice of which specific TEE to use for a given trusted application depends on, among other things, how the developer wants to structure the protected part of its application [31].

2.2.1 Intel SGX

As mentioned above, Intel's own TEE is known as Intel Software Guard Extensions (SGX), and was firstly introduced in 2015 with the sixth generation of its Pentium CPUs. The technology is directly implemented in Intel's new CPUs, which therefore enables secure remote computation by exploiting this trusted hardware in a remote machine. More precisely, it is implemented as a set of instruction codes for the CPU, allowing applications to be executed in a protected region of the memory [32].

Intel SGX differs from previously evoked approaches by the amount of code protected in the TCB; here it only contains the private data and operating code strictly necessary for a given computation [33], while other approaches such as TrustZone and SEV will protect the entire system [31]. This way, whatever is processed inside the TCB remains isolated from the outside environment, including any operating system, hypervisor, and hardware devices on the system, while still preserving the integrity and confidentiality of the computation. This makes SGX more targeted at small but highly security-sensitive applications, while TrustZone and SEV would more be suitable for complex or legacy applications and services [34].

The protected memory regions that SGX uses are called enclaves, and they enable to load specific computer programs in which calculations are being conducted while preserving both confidentiality and integrity. Confidentially is preserved by using encryption, the enclave being stored in an encrypted part of the memory and the user being expected to communicate with the enclave in an encrypted way. Likewise, integrity is preserved by using cryptographic authentication, but much like other TEEs, SGX also provides an attestation mechanism ensuring that the software is indeed operating inside an enclave, and that this software remains unchanged. The proof is provided as a cryptographic signature, certifying the hash of everything residing in the enclave: both the application code and any data loaded during its creation [33].

This is useful in the context of secure remote computation, since a distrusted owner of a remote machine could load any software of its choice into the enclave, the user of the remote service can check the returned hash against the expected value, thus detecting and preventing execution on any modified software. We cover more in detail how the remote attestation mechanism operates in Section 2.2.2, however one factor we can mention right now is that the cryptographic signature used by an enclave to produce an attestation is certified by Intel as existing only inside this enclave. The proof is provided as a cryptographic certificate, which implies that for SGX to enable a user to use the machine of a distrusted owner, the user now has to trust Intel.

As a result, an SGX application that requires confidentiality will extendedly make use of encryption, since transfer of data between the enclave and the outside environment will inevitably happen. While, on the other hand, an SGX application that requires integrity will extendedly make use of attestation and signature verifications, in order to make sure that the computation results were indeed produced inside the enclave. At the same time, the SGX approach is still compatible with Intel's traditional software layering, in which the OS kernel and hypervisor manage the computer's resources [33].

2.2.2 Intel SGX Remote Attestation

As described above, remote attestation is a mechanism needed to ensure that a secure remote computation took place inside an enclave. A unique cryptographic key stored in an SGX enabled CPU, which signs attestations transmitted to any challenger asking for a proof that whatever has been signed with this key was indeed signed inside the enclave. Then, the user can then compare the generated attestation with the endorsement certificate provided by Intel, supporting that the attestation key should only be known by the enclave and is only used for attestations [35].

Currently, Intel SGX offers two remote attestation schemes. The first is an algorithm that Intel offered when starting developing SGX, it is called Enhanced Privacy Identity (EPID) and is based on the previous version of Direct Anonymous Attestation (DAA) developed for TPMs [36]. For EPID to work, in addition to the application enclave and a challenger, it needs a quoting enclave and an attestation verification service, provided by Intel by the aptly named Intel Attestation Service (IAS) [37].

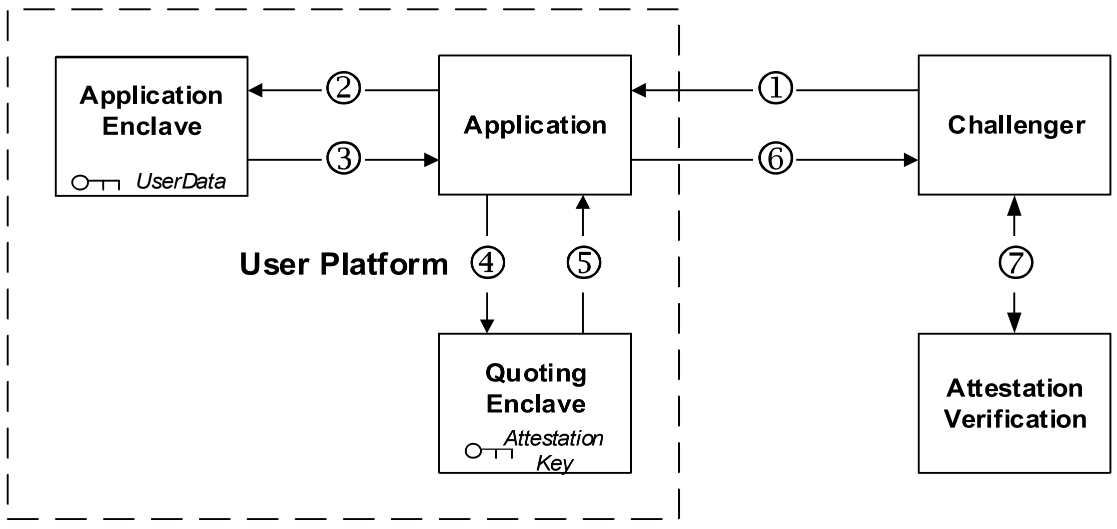

Figure 2.2: EPID Attestation Flow [38]

The EPID attestation mechanism, illustrated in Figure 2.2, works as follows: when a challenger asks the application enclave for an attestation quote, the application returns an attestation then signed by the quoting enclave, and returned outside the enclave. The challenger then checks the quote with IAS, not only for the quote being valid but also for the signing key not being revoked. Finally, the IAS returns a signed attestation result to the challenger, stating if the quote is valid or not.

A second scheme, called Data Centre Attestation Primitives (DCAP), and based on the Elliptic Curve Digital Signature Algorithm (ECDSA), was later developed by Intel to offer some flexibility [38]. The purpose of this new certification infrastructure is to allow for non-Intel parties to author their own attestation without having to rely on the IAS. As a result, entities not connected to the Internet, not willingly to risk outsourcing trust or to have a single point of verification, or just entities concerned with privacy, could still use Intel SGX.

For DCAP to work, one must install an Intel provided Provisioning Certification Enclave (PCE), which acts like a local certification authority for a local quoting enclave. As a result, this quoting enclave can now generate its own attestation keys using whatever algorithm the user prefers, as long as it provides its public key to the PCE. This latter can then authenticate the request and issue a certificate identifying both the quoting enclave and its attestation keys. Because the PCE public key is verified by an Intel certificate, one just has to know Intel's public key without being connected to the Internet to verify the full chain of trust [38].

Hence, when comparing Figure 2.2 and Figure 2.3, we can now see this new PCE component interacting in the attestation scheme. Instead of having the Quoting Enclave holding the Attestation Key, it now gets authenticated by the PCE to provide the certificate authenticating the Enclave. Furthermore, we see the Intel Attestation Service being replaced by the Data Centre Caching Service, which now acts as the "local" IAS verifying the PCE certificates using its own signatures authenticated by Intel.

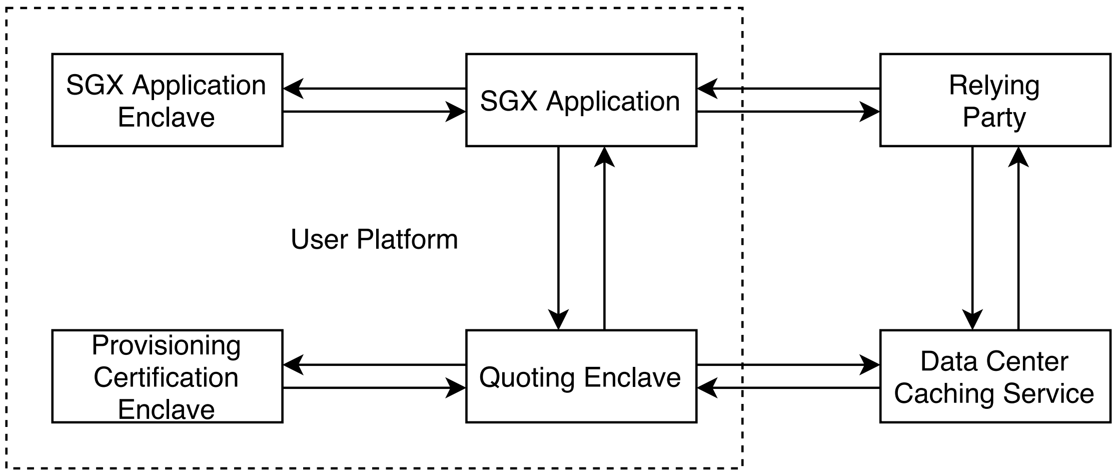

Figure 2.3: DCAP Attestation Flow [39]

2.2.3 Developing with Intel SGX

When developing an SGX-enabled application, a programmer has to deal with a number of restrictions due to the isolated nature of the enclave. Notably, there cannot be any access to system calls, and there is a limited access to CPU operations. On top of that, there cannot be any interaction with any network whatsoever, local or not. Therefore, the developer needs to define an interface between the enclave code and the outside world, taking over every use case where these restrictions have to be circumvented that way.

To facilitate this, Intel introduced Enclave Definition Language (EDL) files, whose purpose is to define the interface between the trusted and distrusted parts of an application [40]. While the distrusted functions must be implemented in the application, the trusted functions are written to the C++ file that will be used for the enclave [40]. Then, after building an SGX application, it is necessary to sign it. This makes SGX able to detect eventual modifications made to the enclave file after signing, and therefore block the application before loading it [40].

To achieve this, Intel provides an open-source SDK for both Linux [41] and Windows [40]. It is a collection of APIs, libraries, documentation, sample applications, and tools allowing to build and debug such SGX-enabled applications, in C and C++. Alongside, they also provide a Platform SoftWare (PSW) [29], containing the drivers, some services, such as the remote attestation handling, and is basically the platform enabling loading and initialisation of enclaves images.

When finishing to build an enclave application, SGX generates a cryptographic hash of the enclave called *MRENCLAVE*. This value identifies the enclave uniquely, as it hashes all and every step of the build process [42]. One can rely on the MRENCLAVE to be exactly the same as long as the enclave application is exactly the same. This is particularly useful when it is needed to make sure an SGX application has not been tampered with, or for verifying its signature at runtime: the SDK will automatically remake every step of the build process, and compare both MRENCLAVEs.

Since applications developed with the Intel SGX SDK require additional efforts to port to other TEE platforms, a community of open-source developers began working on the Open Enclave SDK [43]. It is a hardware-independent library, with the goal of generalising the development of enclave applications across TEEs from different hardware vendors. Currently, the project, which is partially maintained by Microsoft, supports Intel SGX as well as a preview support for ARM TrustZone.

3
Fabric Private Chaincode

This thesis contributes to *Fabric Private Chaincode* (FPC) [44], a framework built on Hyperledger Fabric that improves data confidentiality and integrity for chaincodes by running them in Intel SGX enclaves. Essentially, FPC secures transactional data while it is being used by chaincodes, in transit to and from a client and while stored on the ledger. As a result, unlike traditional chaincode applications, malicious Fabric peers can only see encrypted data associated with FPC chaincodes, which relaxes the trust assumption towards the endorsing peer in terms of confidentiality.

In that regard, FPC adds another privacy mechanism to Fabric, allowing the implementation of use cases with strong privacy requirements. The project, maintained by a community of open-source developers mostly from IBM and Intel, is accessible as an extension of Hyperledger Fabric v2.3 on GitHub [1]. Initially a Hyperledger Labs project, it has now been accepted as an official Hyperledger Fabric feature [2] and is currently under active development.

FPC works by allowing a chaincode to handle encrypted transaction arguments and states, thereby concealing sensitive data, even to endorsing peers. Moreover, the framework enables both the client and other peers to build confidence in a given FPC chaincode, through the successful use of remote attestations. As a result, clients establish a secure channel with chaincodes, rather than with the peer hosting them, preserving privacy on both transaction requests and responses. The enclave on the hosting peer ensures the confidentiality of the data while the chaincode executes it, and encryption is used to protect any data stored publicly on the ledger.

The purpose of FPC is to address the various situations in which it is desired to execute an application in a distributed architecture such as a blockchain, but when that application also requires secrecy on top of the existing integrity measures. Theses use cases may include privacy-preserving analytics on sensitive data *e.g.,* medical data, exchanges requiring contract confidentiality *e.g.,* intellectual property, secret ballot voting, or sealed bid auctions [44]. These examples would not be suitable for normal Fabric's existing confidentiality features, since it will always require full trust from the endorsing peers. For instance, it would be unacceptable for a voting system to be called confidential if the government is running an endorsing peer [44].

Another incentive for FPC is its integrity model, which is based on hardware-enabled remote cryptographic attestation. In regular Fabric, integrity of chaincodes is protected by the endorsing mechanism, which is specified by an endorsement policy as we have seen in Section 2.1.1. In FPC, due to the strong integrity enforcement of the attestation mechanism, the number of replicated executions can be reduced which gives more confidence in integrity as the regular Fabric model, with less processing. TEE-based endorsement and remote attestation provide a new set of endorsement policies, which can minimise the number of needed, and potentially costly, chaincode executions while still providing adequate guarantees of integrity for many workloads. There are trade-offs, as is typical for performance, so measurements are required to determine the best strategy.

3.1 Architecture

FPC is designed in a way that it does not require changes or modifications on Fabric's core, maintaining a user experience as close as it could be to normal Fabric. On top of that, a roadmap beyond the first design aims to enable more uses cases, to obtain some performance benefits and provide a more extensive programming experience [2]. In fact, this is exactly the goal of this thesis; getting closer to a regular Fabric chaincode development by adding support for Go written chaincodes, since, for different reasons that will be explained later on in Section 3.3, the current implementation of FPC requires C++ written chaincodes.

In a nutshell, FPC works just like Fabric, but with an extra FPC layer on top in order to guarantee confidentiality and integrity. As these adjustments are not necessarily compatible or supported by normal Fabric, they are encapsulated in "normal" requests and responses messages in a way that they can be handled natively by peers and clients. Then, whether from the client side or the chaincode side, these messages are recovered and authenticated by these extra FPC layers on both sides.

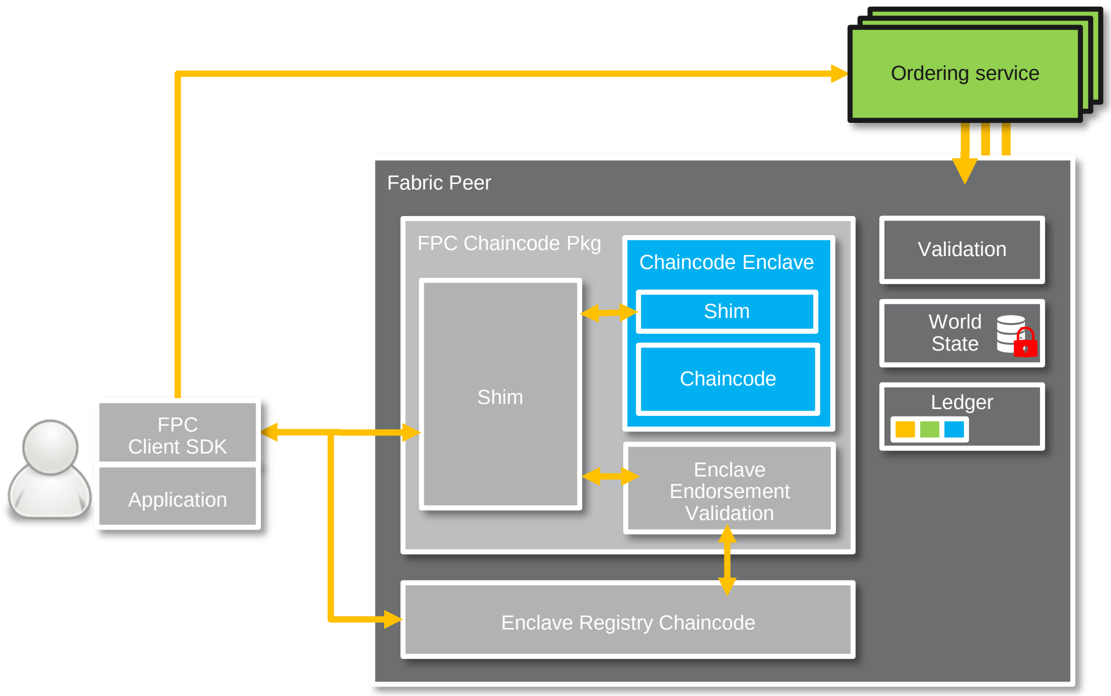

Figure 3.1: FPC Architecture Diagram [2]

In Figure 3.1, we can see how each component interact. On the left, we see that FPC introduces an *FPC Client SDK*, built on top of the regular Fabric Client SDK, enabling application developers to interact with an FPC chaincode. In more details, it encrypts and authenticates the transactions completely, protecting data from any distrusted peer it may pass through. Then, running inside the peer, we see the *FPC Chaincode Package* which is a regular Fabric chaincode containing multiple FPC components.

The main one is the *Chaincode Enclave*, which resides inside an SGX enclave – represented in the figure by a blue box – and contains the actual chaincode. The other component is the *Enclave Endorsement Validation* (EEV), whose purpose is to complement the peer's endorsement mechanism by checking that transactions are signed by registered chaincode enclaves. Finally, there is the *Shim*, which provides an interface for the Chaincode Enclave, and will be covered in detail in the following Section 3.1.1.

Alongside the FPC Chaincode Package, there is an additional Fabric chaincode, the *Enclave Registry Chaincode* (ERCC) whose role is to store a list of all the existing Chaincode Enclaves on the ledger, identifying them by their MRENCLAVE. Furthermore, when an enclave is initialised by a user, the ERCC receives its remote attestation result via a `registerEnclave()` transaction. Thereby, after verifying the result, the enclave registry maintains the results on the blockchain in its list as well, in order to make the attestations results available to all users in the system. This allows peers and clients to check a Chaincode Enclave's attestation before performing chaincode transactions or committing state changes.

3.1.1 FPC Shim

The FPC Shim is the interface than handles FPC-specific operations inside the enclave and the FPC Chaincode Package. The shim implements the normal Fabric chaincode interface and intercepts calls made by either the client and the FPC chaincode. Its role is, for the client, to be the entry point of the FPC chaincode while responding like a normal Fabric chaincode, and for the FPC chaincode, to handle all FPC specific operations in such a way that the chaincode can be written as any other chaincode would be, without any FPC specificities in mind.

A typical example of the Shim usage would be the handling of a request's arguments. When a request is sent to the Chaincode Enclave, the FPC Client encrypts the arguments and encapsulates them into another request, which is first sent to the FPC chaincode in between. Then, in the FPC chaincode, the Shim outside the enclave treats the encapsulated request as a data blob, passing it to the Shim inside the enclave, which then decrypts the data blob and provides the arguments to the Chaincode Enclave.

The final goal is that when a chaincode developer calls the `GetArgs()` method, the FPC Shim will serve as the chaincode's default interface, replacing the arguments of the encapsulating request with the decrypted arguments intended for the Chaincode Enclave, enabling the developer to write its chaincode as if it were regular Fabric. The same applies to other functions, such as `GetState` or `PutState`; the shim wraps these methods, respectively decrypt or encrypt the data, then performs the actual operation.

However, since FPC is still in development, it currently only supports a subset of the original API offered by Fabric, mostly providing the functionalities that are strictly essential to implement a chaincode. The following methods have been transposed by the FPC shim:

- `GetStringArgs(): []string`
- `GetFunctionAndParameters(): string, []string`
- `GetState(string): []byte`
- `PutState(string, string): nil`
- `GetStateByPartialCompositeKey(string): iterator`
- `DelState(string): nil`

Furthermore, because the functions interacting with the world state now handle encrypted values by default, the following "Public" methods have been added in case a user would specifically want to store or retrieve unencrypted values:

- `GetPublicState(string): []Byte`
- `PutPublicState(string, string): nil`
- `GetPublicStateByPartialCompositeKey(string): iterator`

3.1.2 Cryptographic Keys

In FPC, a collection of symmetric and asymmetric cryptographic keys are used to protect the chaincode state and the messages exchanged with the clients. These keys, listed in Table 3.1, serve different purposes, mostly depending on which particular component stores them, and to which other components they can be shared. In this section we elaborate on these keys, grouped by their scope, explaining in detail what they are for and how they are used.

Table 3.1: Cryptographic Keys

Scope	Public	Private
Chaincode	Chaincode_ek	Chaincode_dk
Enclave	Enclave_ek	Enclave_dk
Signing	Enclave_vk	Enclave_sk
Requests	Request_ek	
Responses	Response_ek	
State	State_ek	

Chaincode This pair of asymmetric keys is used for communicating with the client. When a Chaincode Enclave is initialised and registered in ERCC, it will create this pair and store the private key in its memory while sharing the public key with the enclave registry. They are used for encryption of key transport messages. These messages are sent alongside encrypted requests and responses, and contains the corresponding symmetric request or response key (see below).

Enclave This pair of asymmetric keys is not used at the moment. Ultimately, its goal will be to communicate with other enclaves when FPC would be supporting multiple Chaincode Enclaves running in parallel.

Signing This pair of asymmetric keys is used for signing a Chaincode Enclave response message. At initialisation, a Chaincode Enclave will create this pair and store the signing key in its memory while sharing the verification key with the enclave registry. The response message consists of the input arguments, the read/write set, the encrypted result itself and the signature, which is verified later on by the enclave endorsement validation (EEV).

Requests This symmetric key is used for encrypting request payloads, the client generating a fresh one with each request. When communicating with an FPC chaincode, the client will use the key to encrypt the request and embeds the cipher text in a transaction proposal. Then, the request key is encrypted using the chaincode public key and sent alongside the transaction proposal in a key transport message (see above). That way, the FPC chaincode can decrypt the request key and use it to decrypt the request.

Responses This symmetric key is used for encrypting response payloads, the client generating a fresh one with each request. The key is embedded in the key transport message alongside the previously mentioned request key, and with it encrypted with the chaincode public key (see above). When the Chaincode Enclave finishes processing the request, it uses this key to encrypt the response and send it back to the client. That way, the client can decrypt use it to decrypt the response.

State This symmetric key is used for encrypting all data that an FPC chaincode writes to the world state via `PutState()` operations. At initialisation, a Chaincode Enclave will create this key and store it in its memory, without sharing it. This way, the decrypted data from the world state can only be accessed from inside the enclave.

3.1.3 Enclave Endorsement Validation

As we have seen previously, the FPC Chaincode Package is composed of the actual FPC chaincode containing the secure chaincode logic, and the Enclave Endorsement Validation (EEV) chaincode, charged of interacting with the former and with the client for endorsement validation purposes. This is because interactions with the world state are made encrypted; therefore it was necessary to provide other peers on the network a way to endorse the executing peer's result while at the same time preventing them from revealing them the encrypted data.

For this reason, as an additional layer to a normal Fabric execution, a *read/write-set* is built and maintained by the FPC Shim. This set contains the keys of all `GetState()` and `PutState()` operations that have been made during a given execution, in order to enable other peers to apply these changes to their local copy of the world state. For `GetState()` operations, the key and value's hash are stored in the set, and for `PutState()` operations, both the key and the encrypted value are stored in the set.

After the executing peer returns its signed response message, the client asks for other peers to endorse it. The EEV in these peers is able to authenticate the read/write-set thanks to the verification key they fetch from the enclave registry (ERCC). Once sure about the provenance of the read/write-set, they can execute the operations based on the keys and compare the results to the ones they get directly from the world state.

More precisely, if it is a `GetState()` operation, they compare the hash of the encrypted value. If it is a `PutState()` operation, they look that an entry with the corresponding key exists in the world state. If the results match, the peer thus endorses the result (which is still encrypted) and once enough peers have endorsed the result, the original FPC client will package the responses in a transaction for the ordering service.

3.2 Typical Execution

This section follows the typical execution flow of an `Invoke()` transaction. Here, in the situation presented by Figure 3.2 situation, the enclave and the chaincode have already been initialised, and the chaincode is waiting for incoming transactions.

1. The user triggers the transaction when asking for an `Invoke()` operation on the application of its choice. In this example, the client wants to call the function `f(args)`, so the actual call might look like `invoke f arg0 arg1` *etc.*

2. The Enclave Registry (ERCC) looks which peer is hosting the corresponding chaincode in one of its enclaves.

3. Once ERCC has located the chaincode, it retrieves the chaincode's public key (see Section 3.1.2) for encrypting the transaction request, and pass the key to the Client.

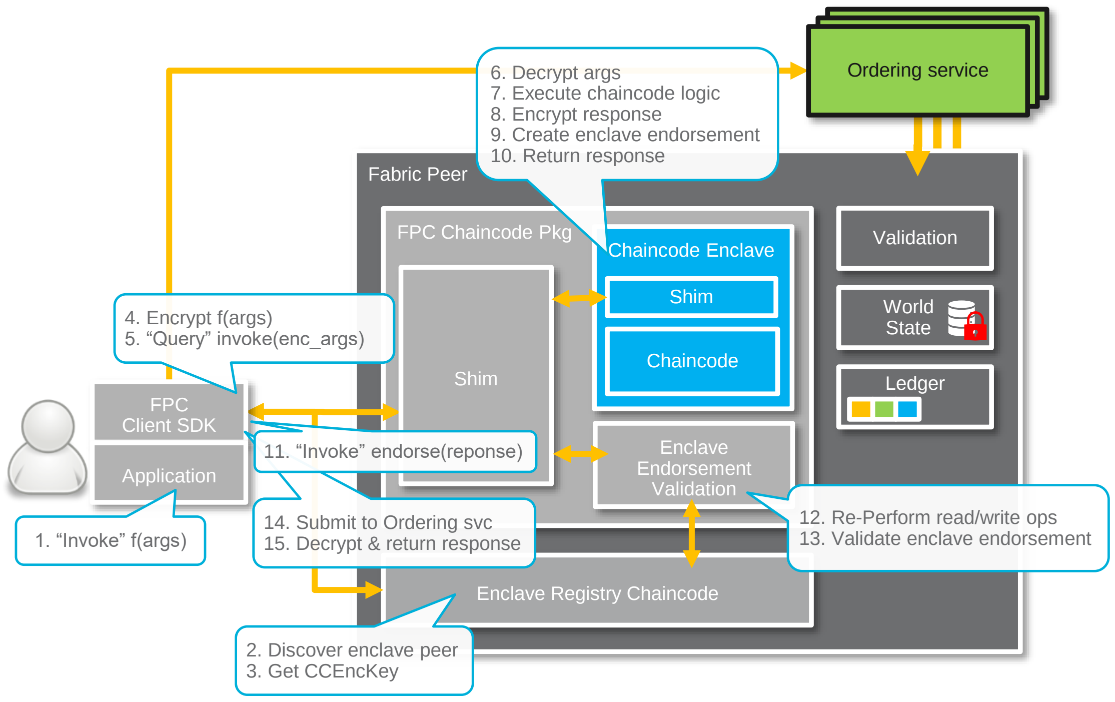
6. Decrypt args
7. Execute chaincode logic
8. Encrypt response
9. Create enclave endorsement
10. Return response
Ordering service
Fabric Peer
FPC Chaincode Pkg
Chaincode Enclave
Shim
Chaincode
Shim
Validation
World State
Ledger
4. Encrypt f(args)
5. "Query" invoke(enc_args)
FPC Client SDK
Application
11. "Invoke" endorse(reponse)
Enclave Endorsement Validation
1. "Invoke" f(args)
14. Submit to Ordering svc
15. Decrypt & return response
12. Re-Perform read/write ops
13. Validate enclave endorsement
Enclave Registry Chaincode
2. Discover enclave peer
3. Get CCEncKey

Figure 3.2: FPC Transaction Flow [2]

4. The Client encrypts the requested function and its arguments with the key provided by ERCC.
5. Once the Client has the encrypted request, it transfers it on to the Chaincode Enclave with a `Query()` operation.
6. The Chaincode Enclave decrypts the final function and arguments with its corresponding private key.
7. Then, the Chaincode Enclave uses the `Invoke()` operation of the actual chaincode, which will then call its own `f(args)` function with the decrypted arguments.
8. Once the chaincode returns its response, the Chaincode Enclave encrypts the response with the private key provided by the Client with the request (see Section 3.1.2).
9. Then, the Chaincode Enclave signs the response proposal with its signing key.
10. Finally, the Chaincode Enclave returns the encrypted and signed response to the Client.
11. The Client starts the endorsement process with the returned response by using the `Invoke()` operation on the FPC Chaincode Package, which will then call its own `endorse(response)` function.
12. The FPC Chaincode Package, via its Enclave Endorsement Validation (EEV) repeats and controls all the operations present in the read/write-set provided with the response proposal.
13. If the operations are valid, the EEV validates the enclave's endorsement and return an "OK" to the Client.

14. The Client then submits the validated transactions to the Ordering Service.
15. Finally, the Client decrypts the response using its private key and returns the decrypted response to the user.

3.3 Limitations

In this section we discuss the current limitations of FPC compared to standard Fabric. We differentiate between conceptional limitations, and those which are just not yet implemented. In the context of this thesis, this list helps to highlight the limitations encountered during development. Also, because these limitations are not covered by FPC, they will not be implemented in this work either.

C/C++ Obviously the first one coming to mind is the requirement to use C/C++ written chaincode. This is due to Intel SGX's SDK especially been produced with low-level language development in mind. This had led to restrictions on functionality simply because a lot of components had to be rewritten, and development had to prioritise certain areas over others.

Init function At the moment, FPC does not give the possibility to use the `Init` function of a chaincode. This is because the `Init` function is used to initialise the encapsulating FPC chaincode, not the actual used chaincode. As a workaround, most chaincodes already use a case statement in the `Invoke` function looking for a function name in the first argument, and thus may still implement an initialisation function.

Empty responses FPC do not use standard `peer.response` objects as invocations responses, but its own format that encapsulates the serialised payload of the peer's response. However, because such response is supposed to be encrypted, it does not support response objects that contains nothing, *i.e.,* `shim.Success()`. As a workaround, if a chaincode needs to return such response, it can simply return a response object containing any string instead, *i.e.,* `shim.Success("OK")`.

Composite keys Composite keys use a non-standard format in FPC, with a simple dot "." as a separator instead of the null character `\x00`. As a workaround, FPC provides a function to convert such keys. Additionally, it appears that when calling `GetStateByCompositeKey`, the read values are not added to the read/write-Set and thus not replayed. This is definitely a security issue and should be corrected in the future.

4 Approach

In this chapter, we describe the approach taken on a high level. We look at the goals of the project, the approach for achieving these goals, as well as the constraints and requirements of the implementation.

In line with what the Intel SGX SDK requires, FPC was designed so that everything interacting with the enclave must be written in C++. However, FPC is based on the regular Fabric implementation which is written in Go, thereby the FPC chaincode package contains an interface that uses Cgo [45] present in both sides of the FPC Shim as we see in Figure 3.1. This interface acts as a bridge between the Chaincode Enclave and the rest of Fabric, and brings undesirable complexity.

Therefore, the plan of this language extension is (1) to replace theses C++ pieces in the FPC Chaincode Package with a new Golang FPC Chaincode Package, and (2) to compile and run the whole package directly in an SGX enclave. At the same time, this extension must allow the use of the same functions and APIs as before, and limit the changes made in other parts of the project.

In order to achieve this second goal, it is necessary to find a method permitting to execute Go code inside an SGX enclave. Thus, in Section 4.1, we introduce the *EGo* [46] SDK, which from the beginning of this project is the technology intended to be used and integrated in FPC. In Section 4.2, we elaborate more about this new Golang FPC Chaincode Package, and how it is designed to allow a Golang chaincode to interface directly with the other modules of FPC without Cgo.

4.1 EGo

Developed in early 2021 by *Edgeless Systems* [47], *EGo* [46] is an open-source SDK that makes the development of confidential Go written applications easy and convenient. It works by compiling any Golang application and, with a few instructions, make it ready to be executed within an Intel SGX enclave.

It is based on a previous work called *Edgeless RT* [48], itself using *Open Enclave* [49] – that we have already mentioned in Section 2.2.3 – as the framework interfacing with the enclave. In a nutshell, Edgeless RT is a runtime for modern programming languages (particularly Go, but also Rust and C++) enabling them to run inside an enclave. It focuses on packaging the binaries into an enclave container, which is then handled by Open Enclave for the low-level operations in SGX [50].

Using Edgeless RT was, according to Edgeless Systems, a tedious and non-user-friendly process [51]. Hence, they developed EGo with the idea of making it as simple as possible for existing Golang applications.

For instance, by changing the way Go handle SYSCALLs to make them compatible with an enclave environment, or by having the compiler seamlessly handling the non-enclave part of the code, unlike other implementations of Open Enclave for programming languages such as C++ or Rust [52].

To achieve this, the SDK provides a modified Go compiler, a Go library [53] and some additional tools. The compiler produces binaries that can run in an SGX enclave when lunched with EGo [51], while the Go library provides methods that may be required for an enclave running software, such as handling attestations, directly as Go code. The additional tools, which are run as command-line argument with EGo, handle various specific tasks described in Table 4.1.

Table 4.1: EGo Tools [54]

Command	Description
`sign`	Sign an executable built with ego-go
`run`	Run a signed executable in standalone mode
`marblerun`	Run a signed executable as a MarbleRun Marble
`bundle`	Bundle a signed executable with the current EGo runtime into a single executable
`signerid`	Print the SignerID of a signed executable
`uniqueid`	Print the UniqueID of a signed executable
`env`	Run a command in the EGo environment
`install`	Install drivers and other components
`help`	Help about any command
`completion`	Generate the autocompletion script for the specified shell

To ease the development process, EGo provides a way to run an enclave application in a simulated environment. It may be used for debugging, as the code and data are not encrypted during execution, or simply in the case that a developer working on a confidential application lacks an SGX-enabled machine. This simulation mode is enabled by using the Open Enclave flag `OE_SIMULATION=1` when running an application with EGo.

Furthermore, developers are able to compile their applications with the EGo compiler and – unless they require access to SGX-specific functions such as the ones provided for remote attestations – run them with the regular Go runtime. This may also be used for debugging on another machine, which does not need to have EGo installed.

For simplicity, EGo works in a way that it takes a program as a whole, and compiles it for an enclave. Therefore, for an application to communicate with the exterior world, it will need additional features, *e.g.,* a TCP connection, preferably using encryption. In one of the few recent works that uses EGo, Porambage *et al.* [55] show that while this approach greatly simplifies porting code in an enclave environment, it may introduce performance issues due to a longer start up time, or during context switches, when the program jumps in and out of the enclave.

4.2 Go Chaincode Enclave

As a reminder, a running FPC chaincode is composed of the FPC Chaincode Package, which itself contains the actual Chaincode Enclave, the Shim, and the EEV. In our approach, a new Go Chaincode Enclave would be created to handle Golang chaincodes, based on the previous FPC Chaincode Enclave. As such, it will include a new FPC Shim translated in Go and an EEV (see Section 3.1.3). Therefore, chaincodes would be able to use the same API as they already do. The functionality provided by the Go Chaincode Enclave interacting with the FPC shim would stay the same, that is:

1. Encrypt and decrypt request and responses.
2. Encrypt and decrypt world state values.
3. Provide attestation capabilities.
4. Host the EEV, this component remaining untouched.
5. Dispatch invocations of the Chaincode Enclave and the EEV.

We have seen that EGo is designed in a way that it takes a whole existing application, and compiles it for a use inside an enclave. Therefore, we will run the whole Go Chaincode Package within EGo, with the bridge between the enclave and the outside world located where the chaincode already establish an encrypted communication with the peer and the client.

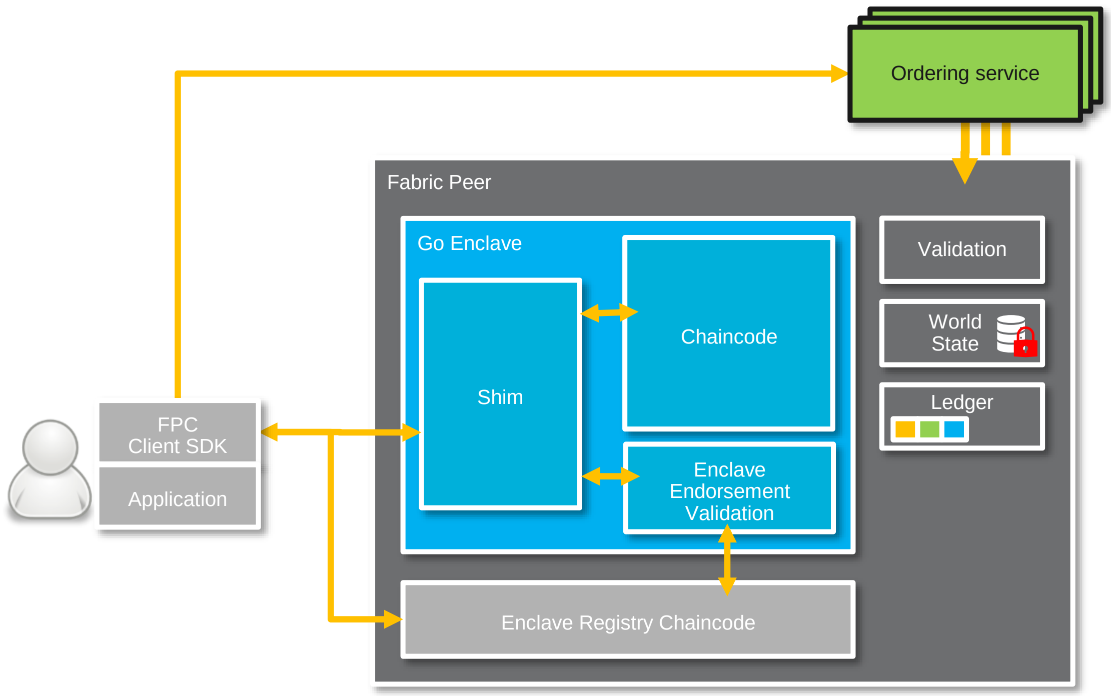

Figure 4.1: FPC Architecture Diagram [2] (Adapted)

In Figure 4.1, we adapted the previous FPC Architecture diagram to show how the new Go Enclave would look like. The blue section of the graphic, representing the Enclave, has been expanded. The Shim, where we would make the most of the modifications, is now entirely simplified and included within the enclave. This way, we can get rid of the Cgo part and lead to a simplification of the FPC Chaincode Package in the future. For the rest, the Client SDK, the ERCC and the EEV are left untouched.

The objective of this approach is to maintain the `main.go` as it is, handling both C++ and Golang implementations, as well accepting normal Fabric chaincodes, without requiring any modification for them to work with FPC. Thus, the changes made by this thesis are made on the components between these two, translating existing parts of the Chaincode Enclave to Go, as well as adapting other existing parts.

4.3 Discussion of the Approach

For this project to still be compatible with C++ chaincodes, we had to keep the modifications and disruptions on FPC to a minimum, in order for it to continue to support current C++ chaincodes. The easiest way to do so, and the path chosen for this project, was to exploit the fact that most components in FPC are already written in Go and would theoretically permit to compile and run any of its components into an enclave with EGo.

Thus, the biggest challenge of this approach was to choose which component to put inside the enclave and which one to leave outside, *i.e.,* to draw the boundary of the trusted computing base. We know it is necessary to protect at the very least both the Chaincode Enclave and the Shim, since non-encrypted data would leave the Chaincode Enclave to be handled by the Shim's wrapper methods. And this is what FPC already do; include only the Chaincode Enclave and the Shim in the TCB, minimising the quantity of code entering the enclave and reducing overhead.

Yet, our approach here is different, because we have to follow the design principles of the EGo compiler. Thus, we place the whole Go Chaincode Package in the TCB, and while this may induce some overhead, it has massively simplified the project. Moreover, since the bridge between the enclave and the outside world is where the communication with the peer and the client is already made, we remove the need to develop a new interface specifically for this purpose, and avoid the performance issues evoked at the end of Section 4.1.

5 Implementation

In this chapter, we describe the implementation details to realise the approach as described in Chapter 4.

We first explain how the approach is implemented, presenting a diagram of the new transaction flow and enumerating the changes required. We then elaborate more on the new modules EEC Go and the FPC Stub Interface, and we end with the limitations of the implementation.

Fortunately for us, when FPC was developed, a mock enclave [56] was added for testing purposes, and for convenience, was written in Go. This mock enclave acts like a usual chaincode by replying to the usual functions calls, but is actually an endpoint returning dummy responses. What this means is that FPC already included some objects, functions, and types implementation already corresponding to the usual interface.

For this reason, this mock enclave has been used as the starting point of the implementation of this thesis's project, by keeping the functions headers intact but modifying greatly how they operate, as well as the initialisation process. Indeed, this mock enclave suddenly became responsible of handling the whole transaction execution process.

Moreover, a new FPC Stub Interface had to be built, based on the previous FPC Shim described in Section 3.1.1. This new interface, described in more detail in Section 5.1.2, acts as an adapter between the chaincode and the Fabric Stub by returning proper arguments, or by encrypting/decrypting world state values. The FPC Stub is a fundamental part in the new Golang implementation, as we will see in Section 5.1.3 where we will show the full sequence of a transaction request invocation.

5.1 Architecture

FPC's architecture is already well compartmented, with low coupling and high cohesion. Of all the modules, the most important for enclave creation and secure chaincode execution are the two modules `ecc` and `ecc_enclave`. The former, written in Go, contains the entry points for the rest of the project to start the enclave creation process. The latter, as we have already mentioned, is written in C/C++ and called through Cgo for the actual enclave creation.

This is because FPC wants to be compatible and built on top of Fabric, minimising the changes required for the implementation of secure computing. By nesting chaincodes, as shown in Figure 3.1, FPC enables compatibility with the normal chaincode processing of Fabric, while implementing its own logic underneath. Thus, the goal of our implementation is to replace this architecture with a new Golang implementation of these nesting chaincodes.

Since the project is already mostly written in Go, to extend the support of Golang is actually to massively simplify the whole process. Therefore, we have created a new module named `ecc_go` [4], which processes all the tasks previously handled by `ecc` and `ecc_enclave`. As we have seen in Chapter 4, the main particularity of this modification is that, while in the previous implementation the two modules represented the outside and inside part of the enclave; here the entire module will be built inside an enclave with EGo.

5.1.1 ECC Go

For the same reasons that FPC is built on top of Fabric while minimising modifications, we built the `ecc_go` module on top of the `ecc` module while keeping most of its architecture intact. Thereby, when executing a Golang chaincode in an enclave, the three following code files are used and executed:

`main.go` This is the entry point of the module. Its role is to create the Chaincode Enclave (`ecc`) and to start it. No changes are required.

`ecc/ecc.go` This is the Chaincode Enclave. Its role is to represent the front end of a "normal" chaincode for Fabric, while handling the actual chaincode with FPC's specific interface. Because we are not changing FPC's interface in this project, it is kept intact as well.

Listing 5.1: `interface.go` [1]

```go
type StubInterface interface {

  // Init initializes the chaincode enclave.
  // The input and output parameters are serialized protobufs
  // triggered by an admin
  Init(chaincodeParams, hostParams, attestationParams []byte) (credentials []byte, err
      error)

  // GetEnclaveId returns the EnclaveId hosted by the peer
  GetEnclaveId() (string, error)

  // GenerateCCKeys returns a signed CCKeyRegistration Message including
  // The output parameters is a serialized protobuf
  GenerateCCKeys() (signedCCKeyRegistrationMessage []byte, err error)

  // ExportCCKeys exports chaincode secrets to enclave with provided credentials
  // The input and output parameters are serialized protobufs
  ExportCCKeys(credentials []byte) (signedExportMessage []byte, err error)

  // ImportCCKeys imports chaincode secrets
  // The output parameters is a serialized protobuf
  ImportCCKeys() (signedCCKeyRegistrationMessage []byte, err error)

  // ChaincodeInvoke invokes fpc chaincode inside enclave
  // chaincodeRequestMessage and chaincodeResponseMessage are serialized protobuf
  ChaincodeInvoke(stub shim.ChaincodeStubInterface, chaincodeRequestMessage []byte) (
      chaincodeResponseMessage []byte, err error)

}
```

`Chaincode/enclave/go_enclave.go` This is the actual Go Enclave, and it is where the link between `ecc` and the actual chaincode is made. Its interface is defined by `interface.go`, which as shown in Listing 5.1, is an extension of regular Fabric's interface intended for FPC's use of cryptographic keys. This implementation is based on the previous `mock_enclave.go` of the previous ECC module. Indeed, the mock enclave was created for testing purposes in the previous build, so it already contained the logic and a lot of reusable code for the wrapping of a chaincode. As a result, most of the modifications were made so that instead of handling a fake chaincode, it could be handling a real one, while keeping the interface intact for ECC, notably:

- Generate, store and return the necessary asymmetric and symmetric cryptographic keys (see Section 3.1.2) need to run the enclave during the init sequence.
- Retrieve, unmarshall and decrypt the input parameters such as the request message instead of using a dummy request.
- Store an empty `fpcKvSet`, which encapsulates the read/write-set of the chaincode instead of creating a dummy one.
- Create a wrapper of the chaincode stub interface that will handle some special operations (see Section 5.1.2).
- Invoke the actual chaincode and passing it the wrapper.
- Retrieve, encrypt, marshal and return the chaincode's response instead of returning a dummy one.

5.1.2 FPC Stub Interface

As we have previously discussed in Section 3.1.1, a Fabric chaincode is expecting a regular Fabric `ChaincodeStubInterface`. Therefore, while calling the `Invoke` function, we had to create a wrapper of `ChaincodeStubInterface` in order to handle special FPC operations. The wrapper takes as arguments ECC's stub, the request input arguments, the previously created `fpcKvSet` and the `stateKey` required for encryption on the ledger. For most of its numerous functions, the FPC stub interface wraps a simple call on the same inherited functions of the chaincode stub. But the remaining ones had to be modified, either to handle FPC operations, or because they had to change their output. Here is a list of these modified functions:

- `GetArgs()`: This method overrides the inherited `stub.GetArgs()` by returning the arguments intended for the actual chaincode, because the inherited method would return the arguments intended for the whole Go Chaincode Package.
- `GetStringArgs()`: This method calls the FPC Stub own `GetArgs()`, but returns the arguments as strings and not as bytes.
- `GetFunctionAndParameters()`: This method calls the FPC Stub own `GetArgs()`, but returns the first argument aside as a function name string and the remaining as a string array.
- `GetState()`: This method calls the FPC Stub own `Get`*`Public`*`State()` and decrypts the returned value using the `stateKey` before returning it.
- `PutState()`: This method encrypts the provided value with the `stateKey` and calls the FPC Stub own `Put`*`Public`*`State()` with the provided key and the encrypted value.
- `Get`*`Public`*`State()`: This method wraps the inherited `stub.GetState()` method while adding the requested key and hashed value to the read-set.

- `Put`*`Public`*`State()`: This method wraps the inherited `stub.PutState()` method while adding the returned key and encrypted value to the write-set.
- `GetStateByPartialCompositeKey()`: This method calls the FPC Stub own `Get`*`Public`*`StateByPartialCompositeKey()`, providing it with a composite (partial) key, and decrypts the returned values using the `stateKey` before returning them.
- `Get`*`Public`*`StateByPartialCompositeKey()`: This method wraps the inherited `stub.GetStateByPartialCompositeKey()` method while adding the requested keys – but not the hashed values (see Section 3.3) – to the read-set.
- `CreateCompositeKey()`: This method wraps the inherited `stub.CreateCompositeKey()`, but returns a transformed key to make it compatible with FPC's format (see Section 3.3).

5.1.3 Sequence Diagram

In Figure 5.1, we see a chaincode invocation flow that uses the chaincode API to retrieve and store data by using the functions we explained in the previous section. The diagram shows, in detail, the used functions with their arguments, called by each of the components during an invoke request. The components whose name is in a round box, on a red background, are the one developed for this work, while the components whose name is in a squared box, on a white background, are the ones left untouched.

The transaction starts with the Chaincode Enclave transmitting the encrypted request to the new Go Enclave, which authenticates the request and decrypts it using the keys included in the key transport message and the chaincode decryption key (see Section 3.1.2). Then, it creates a new instance of the FPC Stub Interface with the following parameters: first there is a pointer to the regular Fabric Stub Interface, for redirecting method calls to, with the decrypted arguments of the actual chaincode, specifically for `GetArgs()` (see Section 3.1.1). Then, there is a pointer to the read/write-set, for appending reads and writes keys to it, and the state key for the actual encryption (see Section 3.1.3). Finally, the Go Enclave calls the invoke method of the chaincode passing it a pointer of the newly created FPC Stub Interface.

Afterwards, the diagram shows three examples of calls that the chaincode makes to the Stub Interface. First, it calls `GetArgs()`, so the FPC Stub Interface returns the decrypted arguments it received earlier from the Go Enclave. Then, when the chaincode calls `GetState()` and `PutState()`, we notice that the stub uses two new methods: `GetPublicState()` and `PutPublicState()`. These methods, which can be called by either the stub itself or the chaincode for reading or writing non-encrypted values in the world state, are here called internally. They wrap around the regular Fabric Stub Interface's `GetState()` and `PutState()`, which we received a pointer to earlier. Thus, when the chaincode calls `GetState()` or `PutState()`, the FPC Stub Interface redirect the call to the Chaincode Stub Interface, while using its own wrapping methods for handling decryption and encryption, respectively.

The final steps of this invoke request are reached when the chaincode terminate its execution and returns a chaincode response to the Go Enclave. There, the Go Enclave constructs a response object compatible with FPC's interface, and then encrypts and encapsulates the chaincode's response in it. Finally, it signs the response with its signature key (see Section 3.1.2), serialises it and sends the bytes back to the ECC which ends this execution. It should be noted that the need to serialise requests and responses is a requirement that is a remnant of the C++ implementation. Here we left it as it is, since we are not modifying FPC's operations, but in an exclusive Golang implementation, one could just exchange Golang structures.

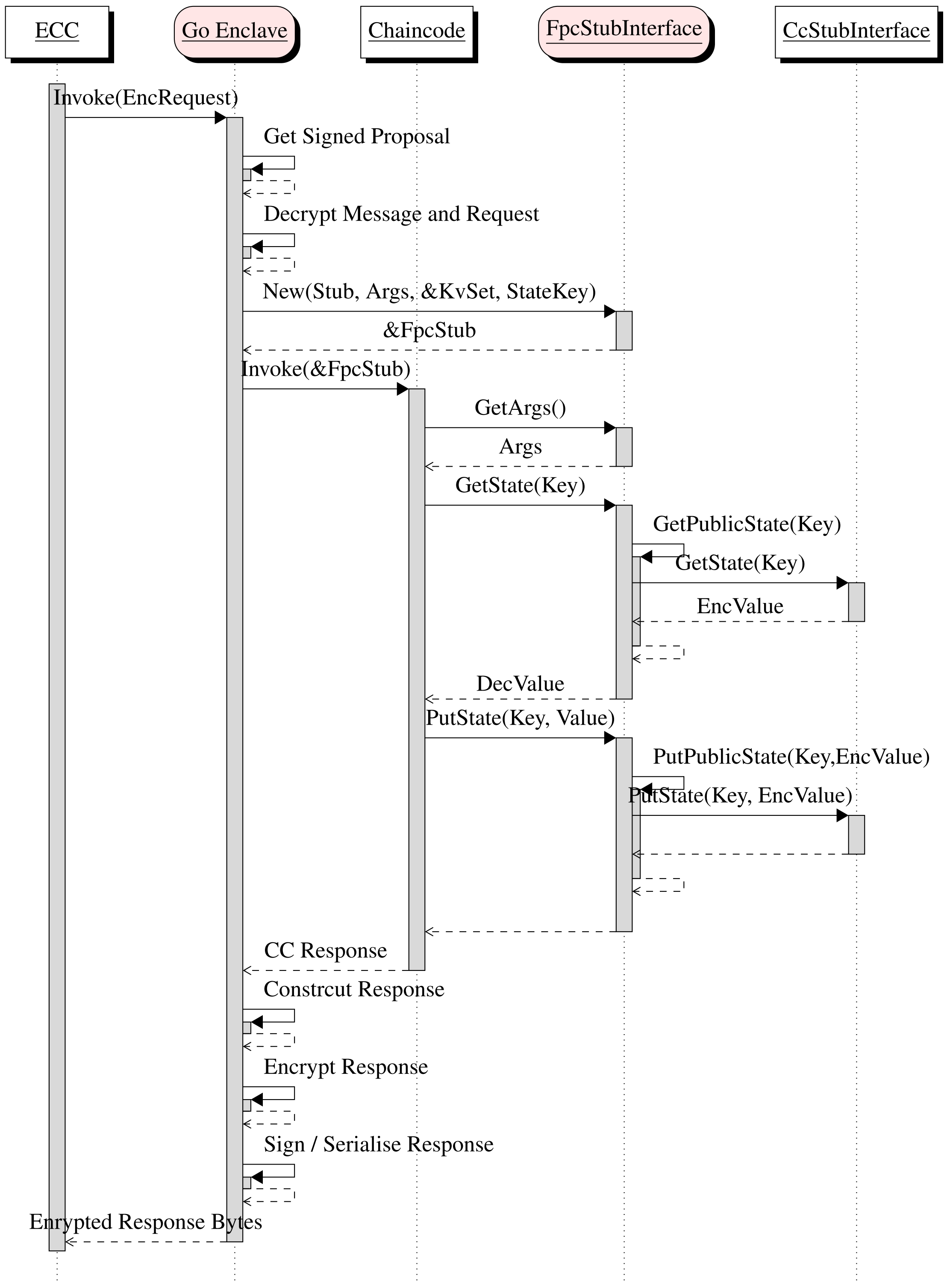

Figure 5.1: Sequence Diagram of an Invoke Request

5.2 Current Limitations

Since this extended language support is built on top of the existing FPC, and because FPC is itself built on top of Fabric, some of its pre-existing limitations will inevitably be inherited by this work.

Attestation This implementation does not support attestations for the moment. At the moment of writing, FPC uses EPID-based attestations but EGo only supports DCAP-based attestations. This is because Open Enclave, on which EGo is based, only supports DCAP [52]. Therefore, supporting attestations would require deeply modifying core parts of FPC, and while it would be essential for FPC to fully support Go written chaincodes, it would require an amount of work which is beyond the scope of this thesis. Thus, while the current state is enough for this work to be valid as a proof of concept, it should never be used in production until DCAP is fully implemented, as verifying attestations is a core part of TEE-based software. Not using them would imply severe security issues.

5.3 Implementation Issues

This section covers the various issues we faced during the implementation of this thesis project.

EGo Version When using Golang chaincodes it is required to use EGo at two distinct moments; firstly, when compiling and signing a chaincode to make it ready for execution, and secondly when actually executing the chaincode. Particularly in this project, the developers need to be extra careful that they use the same EGo version when building FPC the chaincode, and during runtime. Otherwise, this may lead to issues, where an error message will recommend to resign again the binary file, but will not give a specific error of the version mismatch.

Empty Responses Many existing chaincodes sometimes return a simple `Shim.Success()` object without any payload. This is unfortunately not handled by FPC, it outputs an error to the user stating that the decryption failed because the response must be longer than 28 characters, but the response provided is 28 characters long. This is easily tackled by returning any string inside the payload, instead of nothing, which will then be encrypted and decrypted normally as it moves through the response process.

SGX Hardware Mode At one moment during development, the test network `Makefile` was modified to add an option to run the network in the background for benchmarking purposes. Inadvertently, this modification changed how the file handled environment variables, and notably, was not transmitting the content of the variable `SGX_DEVICE_PATH` to the docker containers anymore. This variable contained the path to the SGX driver, so each time one would try to run the project in hardware mode, the driver was not to be found, and made it seem like there was a problem with the drivers installation.

6
Evaluation

When extending the language support of FPC to Go, we wanted to simplify the project by enabling the use of native Fabric compliant chaincodes, but without a significant drop in performance that would render this extension unusable. In that regard, a comparison of the performance has been conducted between our Golang support and existing FPC C++ chaincodes. Furthermore, an evaluation regarding the usability of the language extension has been conducted.

6.1 Performance

This section covers the performance evaluation of the project. For this purpose, we have ported the Auction chaincode [57] from the FPC samples as closely as possible to Golang, and we are using it as our baseline. This Auction Chaincode implements a basic auction system, with an `init()` method to initialise an auction house, with `create()`, `open()`, `close()` and `eval()` methods to manage an auction, and with a `submit()` method to submit a bid.

The evaluation is being conducted on a HP *Elitebook* x360 1040 G6 laptop, running on an Intel Core i7-8565U CPU and 16 GB of memory, operating on Ubuntu 20.04 and the `5.15.0-43-generic` kernel. The project is forked from FPC's latest commit at the moment of writing, `a977029(...)` [58] committed on July 5th, which uses Fabric 2.3.3 [59]. For the hardware mode, we use SGX drivers version 2.11.0 [60] and EGo version v1.0.0 [61]. The test network provided by FPC – consisting of two organisations with one peer each plus an ordering node – will be the base network for the evaluation. No modifications have been made to the test network.

6.1.1 Parameters

This evaluation is being conducted by measuring the end-to-end latency of the whole process executed by the *Simple CLI Client*. The purpose of this macro evaluation is to experience the measurements from a client perspective. Also, it is the easiest to implement since it only implies to build measurements on top of the CLI commands and to run each of the chaincodes and compile the differences.

In this evaluation, we assume the results of the Golang extension to present a similar latency as the baseline. We do not expect to find significantly different results.

The measurement is effectuated with the aforementioned Auction Chaincode. To ensure accuracy in the results, the 9 following commands have been executed 500 times in a freshly restarted test network, successively for both the regular and Golang version of FPC, and in both the simulation and hardware SGX modes.

- `init peer0.org1.example.com`
- `invoke init("House" + i)` (Where i is the execution round number)
- `invoke create("Auction")`
- `invoke submit("Auction", "John", 100)`
- `invoke submit("Auction", "Jane", 200)`
- `query submit("Auction", "John", 400)`
- `query submit("Auction", "Danny", 100)`
- `invoke close("Auction")`
- `invoke eval("Auction")`

As we have seen earlier, FPC does not support the standard `Init()` method inside a secure chaincode. Therefore it is important to understand that the first command only initialise the Go Chaincode Package as a whole, and does not forward the method call to the Chaincode inside the package. On the contrary, the subsequent commands use `Invoke()` on the Go Chaincode Package, which forward the call to the Chaincode Enclave. Hence, the second command ultimately calls `Invoke(init, args[])` instead of `Init(args[])` on the Chaincode Enclave.

6.1.2 Results

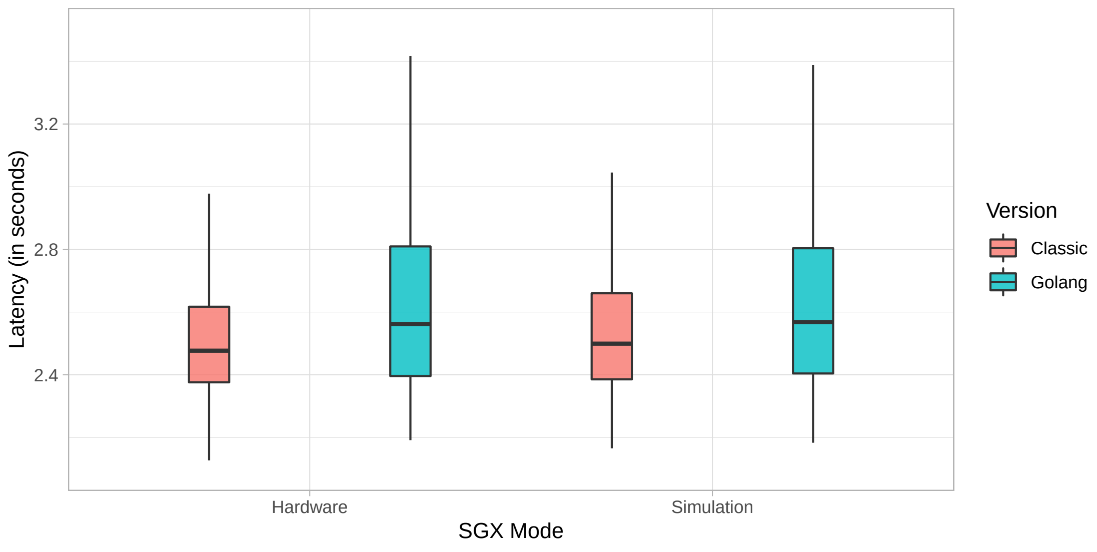

Figure 6.1: Results for `Init()` function

The first immediate observation after running the experiment is that the client always seems to wait for at least two seconds before outputting the results. This delay, however, does not occur if the transaction outputs an error. This is due to the client default configuration for the event handler, that waits for an answer from the ordering service before outputting any result [62], and due to the ordering service default configuration making it wait 2 seconds because of the `BatchTimeout` in the configuration file [63].

In Figure 6.1, we show the results for the `Init()` operation to initialise chaincode. We can see that latency results from the Golang version look more dispersed and with a slightly superior median for the latency than results from the Classic version. This is even more remarkable for hardware mode.

This is an interesting result, since in the Golang version, the enclave creation already happens when starting the docker containers, as opposed to the Classic version where the enclave is created when the `Init()` command is made. This means that, despite lacking the enclave creation, the Golang version still takes more time, which is a surprising result. More precisely, the end-to-end latency of the Golang version is 13.81% (in software mode) to 20.17% (in hardware mode) higher than the classic version, when removing the 2 seconds baseline from the equation.

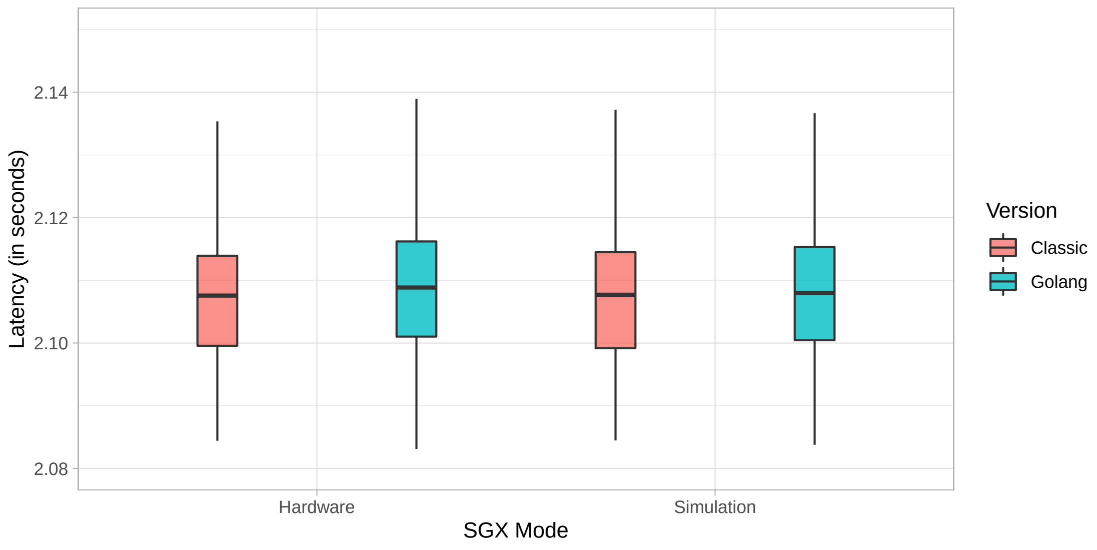

Figure 6.2: Results for `Invoke()` function

In Figure 6.2 we show the results for all the `Invoke()` operations on the chaincode. We notice that the results are almost identical for all versions, with as expected, a slightly higher median for the hardware mode in both versions, which is a good sign for the viability of our project.

However, we do notice than the Golang results present a higher latency that the C++ version, especially in hardware mode. This could be due to the complexity of using EGo, as opposed as the native SGX SDK, closer to the system. It could also be due to the way we implemented our language extension, which when refactored and integrated more deeply into FPC might achieve a better performance, as opposed as the current implementation that acts more like an add-on.

It is important to note, however, that these results only show the big picture, which is all the `Invoke()` operations combined. When plotting graphs for some specific `Invoke()` calls that were made with the Auction chaincode, we discovered some interesting results. Two of them will be discussed below, while all the remaining results graphs are in the Appendix at the end of this document, and all the data files with all the results values are available on GitHub [64].

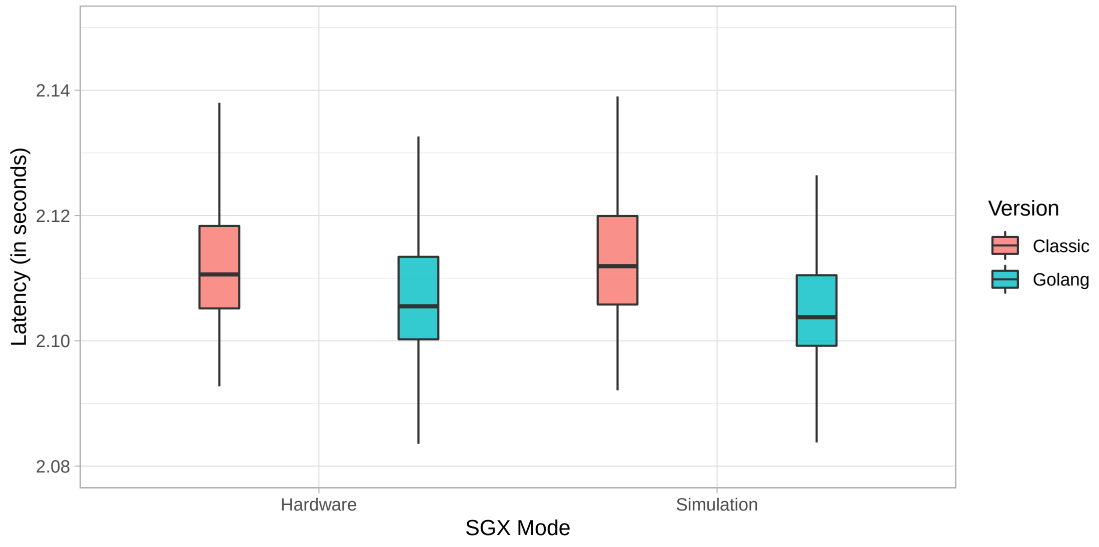

Figure 6.3: Results for `Invoke(init)` function

In Figure 6.3 we show the results for all the `Invoke(init)` operations on the chaincode. It is important to note at this point that the chaincode is already initialised, and this function is just creating the Auction House by doing two basic `PutState()` operations, as we can see in Listing 6.1. Therefore, it is interesting to note that of all the `Invoke()` calls made in the evaluation – where the Golang version performs slightly less than the regular version – this is the only one where it actually performs better.

Listing 6.1: `initAuctionHouse()` in `auction.go` [57]

```go
func (t *Auction) initAuctionHouse(stub shim.ChaincodeStubInterface, auctionHouseName
    string) string {
  stub.PutState(AUCTION_HOUSE_NAME_KEY, []byte(auctionHouseName))
  stub.PutState(INITIALIZED_KEY, []byte{0x1})
  return "OK"
}
```

Interestingly, this could also be due to the proximity to the system that the C++ language has compared to Go. That, the more complicated a function is, the more it tends to perform better on a close-to-system language. Alternatively, it could be that our "translation" of the chaincode may not be as optimised as it should be, or that we may have introduced new code that raises unforeseen performance issues, except on this particular straightforward method. Finally, it could be that the compiler itself is producing less optimised code.

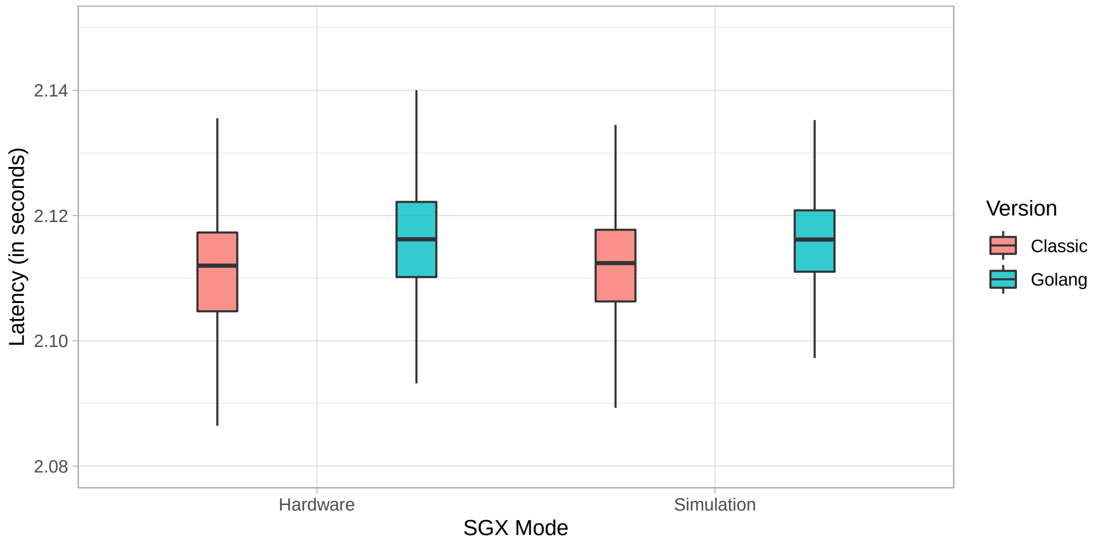

Figure 6.4: Results for `Invoke(eval)` function

On the other hand, in Figure 6.4 we show the results for all the `Invoke(eval)` operations on the chaincode, where it seems that the opposite is happening. The set out some context, the `Invoke(eval)` is the most computationally demanding method of the chaincode, because it is the one that gets all the submissions and computes the winner.

Therefore, it is interesting to note that the difference in latency between these two versions is higher than for all `Invoke()` calls combined in Figure 8.2. This could confirm the assumption described above that, the more complicated a function is, the more it tends to perform better on a close-to-system language.

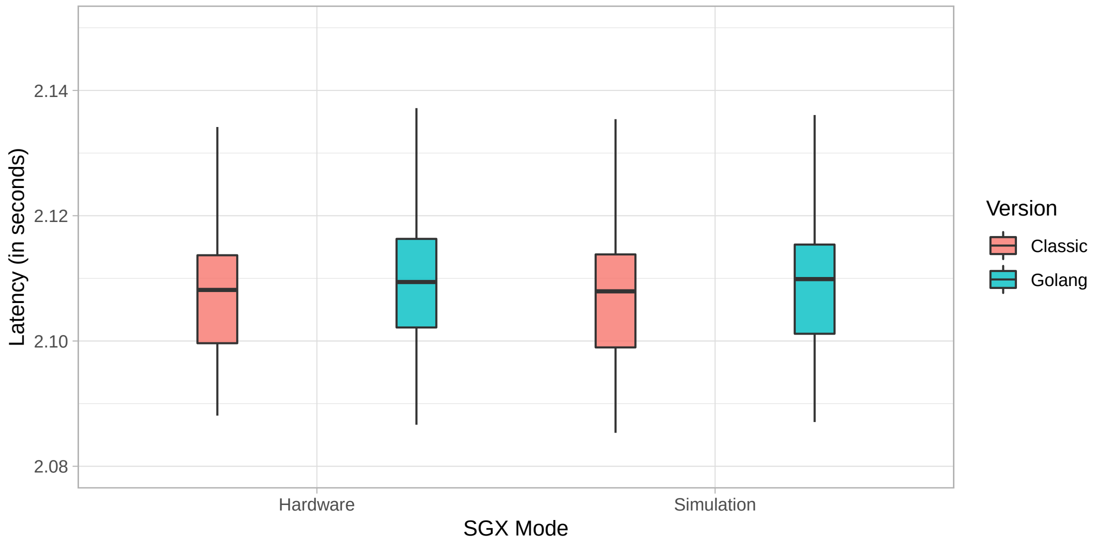

Figure 6.5: Results for `Query()` function

Finally, in Figure 6.5 we show the results for all the `Query("submit")` operations on the chaincode. They are different from `Invoke()` operations in that `Query()` operations are just executing calls on the chaincode without validating them through the ordering service. Nevertheless, we observe that there is no significant latency difference for these two operations.

6.2 Usability

One of the most important aspects of the project is to enable the use of Go written Fabric chaincodes to be used directly in FPC, albeit without attestation at the moment. This section covers the evaluation of the derived usability of the project.

The evaluation consists of executing each Fabric publicly available Golang chaincode samples [65] with our implementation. Due to FPC limitations evoked in (see Section 3.3), notably the inability to use Fabric's standard `Init()` function, they might, however, need some modifications. Therefore we will look if it were possible to adapt them with minimum changes. If not, we will note that they cannot be ported easily, thus reducing the usability of the project.

6.2.1 Smart Contract API

Due to some limitations on running complex chaincodes using the initial init/invoke method, Fabric developed a *Smart Contract API* for Go, Node.js and Java chaincodes that enables the use and the invocation of custom-defined functions [66]. Unfortunately, FPC does not handle this method at the moment. That means that most chaincodes who use this API could not be compatible with the extension since they will not be defined around a central invoke function.

This greatly diminishes the ability to just get any Golang chaincode to work from scratch with FPC. However, it would not be too complicated to make them compatible with minimal modifications by writing this specific invoke function and pointing to the other with it. While it will change the chaincode and does change the calls being made for the different invocation of this chaincode, it will still be way less tedious as a task than rewriting the entire thing.

For this reason, the following evaluation focuses on chaincodes that were not written using the Smart Contract API, or chaincodes for which an older version has been found on GitHub, before being upgraded to use the Smart Contract API.

6.2.2 Results

For each of all the chaincodes in the Hyperledger Fabric samples [65], we have attempted to answer the following questions:

1. Is it usable without modifications?
2. If not, would it be usable by making superficial changes?
3. If not, would it be usable by making changes less substantial than rewriting everything?

Simple Asset CC This is a basic chaincode that interacts with the ledger using `GetState()` and `PutState()` operations. The latest version of this chaincode does not use the Smart Contract API, making it straight away (1) usable without modifications! It does have, however, an existing `Init()` function, but this function is not required for the proper functioning of the chaincode, since another identical function, `set()`, is accessible directly with the `Invoke()` pivot function.

AB Store This is a basic chaincode that allows to transfer integer data from entity A to entity B. An older version of this chaincode, not using the API, has been found [67]. The chaincode is (1) not usable without modifications; it relies on an `Init()` function and return empty responses, which are both incompatible with FPC. However, it is (2) fully usable by making superficial changes: we had to make the public `Init()` function a private one and include it in the `Invoke()` pivot function, and make the empty returns throughout the chaincode return `[]byte("OK")` instead.

FabCar This slightly more complex chaincode interacts with the ledger to manipulate “Car” objects, stored as byte-encoded JSONs. An older version of this chaincode, not using the API, has been found [68]. The chaincode is (1) not usable without modifications; it does not rely on an `Init()` function, but returns empty responses, which is incompatible with FPC. It is (2) mostly usable by making superficial changes when replacing empty returns by `[]byte("OK")`, except for the `queryAllCars()` function. This one method relies on `GetStateByRange()` of the Shim API, which has not been implemented in FPC at the moment. However, we are not considering the logic of the chaincode as being negatively impacted, as it is still possible to make individual queries with the function `queryCar()`.

Marbles This complex chaincode interacts with the ledger to manipulate “Marble” objects, stored as byte-encoded JSONs, and is designed to interact with state database implementations supporting rich queries. The chaincode is (1) not usable without modifications. It is (2) somewhat usable by making superficial changes when replacing empty returns by `[]byte("OK")`, and fixing the `if` condition in line 193. When creating a marble object, a condition checks if a marble with the same name already exists with a `GetState()` operation. If this `GetState()` returns an error, which it does when trying to decrypt an entry that does not exist in the world state, the whole chaincode then returns an error. The easy fix is to comment line 194, since a failing `GetState()` would return an error anyway with or without this condition. With these fixes, `initMarble()`, `readMarble()`, `delete()`, `transferMarble()` and `transferMarblesBasedOnColor()` are functioning. Unfortunately, the chaincode is (3) not fully usable by making changes less substantial than rewriting everything. The functions () `queryMar-blesByOwner()`, `queryMarbles()`, `getHistoryForMarble()`, `getMarblesByRange()`, `getMarblesByRangeWithPagination()` and `queryMarblesWithPagination()` are relying on API calls not implemented yet by the FPC Shim, and would have to be extensively modified to achieve the same features without those calls.

7 Related Work

In this section, we present works related in one way or another to our thesis project; whether it is a work on the topic of secure blockchain computation, a work built around EGo, or both.

TZ4Fabric Because FPC is built around SGX, it is essentially only compatible with computers having an Intel CPU. For this reason, Müller *et al.* developed *TZ4Fabric* [69], a project inspired by FPC for executing chaincodes in ARM TrustZone. It is aimed at devices such as phones, nano-computers and IoT devices, which typically have an ARM architecture. Its architecture is similar to that of FPC, but uses OpenEnclave's preview for compatibility with TrustZone [69]. Its main limitation compared to FPC is that TrustZone does not support remote attestation natively, and while workarounds are possible, it enables a potential attacker to interfere with the chaincode before an execution [69]. Furthermore, all its chaincodes reside in the same secure space with no isolation guarantees among the chaincodes, which is something instead provided by FPC as a result of the ability to run multiple SGX enclaves on the same processor [69].

Corda SGX Update *Corda* is a permissioned distributed ledger developed by R3 and aimed at businesses, mostly from the financial sector [70], and is capable of executing Kotlin and Java smart contracts [71]. Transactions between members are private in Corda, but their validation, here called ”resolution”, have to be public for validation purposes [72]. Therefore, Corda adopted SGX to protect these resolutions, by encrypting the identity of the entities involved in transactions and revealing them only inside an enclave [72]. Moreover, Corda protects its smart contract executions as well by hosting an entire Java Virtual Machine (JVM) inside an enclave. This definitely induces more overhead and could lead to security issues due to the large TCB; however, R3 argues that by using Java, a managed language that eliminates native code exploits such as those found in C/C++ would alleviate these issues [72].

Rust-SGX While EGo is a framework meant to support Go code in SGX, other similar projects have been developed for supporting other languages, such as *Rust-SGX* [73] by Wang *et al.*. This SDK introduces a Foreign Function Interface (FFI) to Intel SGX SDK for supporting Rust code, and aims at enhancing security by eliminating memory corruption vulnerabilities inherent to traditional programming languages such as C/C++ [73]. Rust-SGX enables developers to use it as a replacement of the Intel SGX SDK to build applications on, making it distinct from EGo which is more a ready-to-use solution for quickly converting entire existing programs to run inside enclaves. The team behind Rust-SGX has also demonstrated a slower end-to-end latency in their benchmark [73], which is in line with our findings.

MarbleRun As a complement to EGo, Edgeless Systems developed *MarbleRun* [74], a framework for creating distributed confidential computing apps. By building confidential applications with EGo and distributing them with Kubernetes on an SGX compatible cluster, MarbleRun is then able to manage the whole distributed architecture [75]. Following instructions from a manifest written by the user, MarbleRun can, replace failing nodes, set up encrypted connection between services or verifying their integrity, *etc.* [75]. It is one of the first works to use EGo in a production setting.

8 Conclusion and Future Work

It is worth noting that, despite initial doubts on the actual feasibility of the extension, notably due to the fact that EGo was in a pre-release form and in active development when starting this thesis, everything seems to be working fine and smoothly. In our evaluation in Chapter 6, we have shown that the extension does not present an end-to-end latency significantly higher than the existing C++ implementation, which is promising for our project.

In addition, we have shown that most of the existing Golang chaincodes are compatible with the extension with minimal modifications. The more complex chaincodes would require additional modifications to FPC on order to be compatible with the API calls they use. Yet, FPC is still undergoing active development and make them compatible with the Golang extension would thus be beyond the scope of this thesis.

The biggest caveat of this implementation would be that attestations were never used, due to the inability of FPC to handle DCAP attestations. However, we do not believe that this fundamentally breaks the trust that we have in the extension to work in a production environment in a near future, as we do know that EGo is more than capable to handle attestations properly, and would be a matter of a few lines of code to add as soon as FPC is capable to handle such attestation. In terms of future work, this is definitely the most important aspect to focus from now on.

Another further development that would be great to implement in order to further integrate FPC with Fabric would be the compatibility of the Smart Contract API and the ability to run all commands through this interface. This requires many more modifications to existing code, as it implies to change the client, the SDK and the interfaces between the Go FPC Chaincode and the SDK.

However, we believe that the support for Golang could soon be seen as a better way to handle secure chaincodes in FPC for the main release. Then, the Smart Contract API could be implemented if the whole project is refactored to handle it, while removing some unnecessary components that were only required to be compatible with C++ chaincodes and thus simplifying the project.

This last point being, in our opinion, what this extension would aim to achieve as its final goal. What is the point of FPC being compatible with C++ chaincodes, if this method, used nowhere else, was developed solely because of compatibility issues with SGX? We could remove this compatibility that would soon turn out to be unnecessary, as well as the support for EPID attestations, and then simplify the API and interfaces for a further integration of EGo and Golang chaincodes.

On step ahead, the next logical step would be to handle Node.js and Java chaincodes, as these two are the other main languages used by Fabric. This would be obviously a bit trickier than Golang, as it will definitely imply new technologies for running inside enclaves as well as some shim or interface between the two worlds.

Also thinking ahead, EGo is now compatible only with Intel SGX, but it is based on OpenEnclave, an SDK whose purpose is to provide a unified interface for multiple TEE independently of the manufacturer. It would be reasonable to think that EGo moves towards achieving such compatibility and, when it does, it would be great to see FPC moving towards a broader range of compatible hardware as well.

Bibliography

[1] Hyperledger Foundation, *Hyperledger Fabric Private Chaincode*, Jul. 5, 2022. [Online]. Available: https://github.com/hyperledger/fabric-private-chaincode.

[2] Hyperledger Foundation, *Hyperledger Fabric RFCs Process*, Dec. 29, 2021. [Online]. Available: https://github.com/hyperledger/fabric-rfcs/blob/main/text/0000-fabric-private-chaincode-1.0.md.

[3] R. Zappoli, *Golang-support · Fabric Private Chaincode*, Oct. 28, 2021. [Online]. Available: https://github.com/ricc-zappoli/fabric-private-chaincode/tree/golang-support.

[4] R. Zappoli, *Ecc_go at golang-support · Fabric Private Chaincode*, Aug. 5, 2022. [Online]. Available: https://github.com/ricc-zappoli/fabric-private-chaincode/tree/golang-support/ecc_go.

[5] L.-D. Ibáñez, E. Simperl, F. Gandon, and H. Story, "Redecentralizing the Web with Distributed Ledgers," *IEEE Intelligent Systems*, vol. 32, no. 1, pp. 92–95, Jan. 2017, ISSN: 1941-1294. DOI: 10.1109/MIS.2017.18.

[6] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, "Blockchain," *Business & Information Systems Engineering*, vol. 59, no. 3, pp. 183–187, Jun. 1, 2017, ISSN: 1867-0202. DOI: 10.1007/s12599-017-0467-3.

[7] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," Mar. 24, 2009. [Online]. Available: http://bitcoin.org/bitcoin.pdf.

[8] E. Androulaki, A. Barger, V. Bortnikov, *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, Association for Computing Machinery, Apr. 23, 2018, pp. 1–15, ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190538.

[9] V. Buterin, "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform," vol. 3, no. 37, pp. 2–1, Dec. 2014. [Online]. Available: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[10] Cardano Foundation, *Documentation for the Cardano ecosystem*, Aug. 5, 2022. [Online]. Available: https://docs.cardano.org/.

[11] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine Agreements for Cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, Association for Computing Machinery, pp. 51–68, ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132757.

[12] T. Jensen, J. Hedman, and S. Henningsson, "How TradeLens Delivers Business Value With Blockchain Technology," *MIS Quarterly Executive*, vol. 18, no. 4, pp. 221–243, Dec. 3, 2019, ISSN: 15401960. DOI: 10.17705/2msqe.00018.

[13] M. Priit, “Estonia – the Digital Republic Secured by Blockchain,” PwC, 2019, p. 12. [Online]. Available: https://www.pwc.com/gx/en/services/legal/tech/assets/estonia-the-digital-republic-secured-by-blockchain.pdf.

[14] Hyperledger Foundation, *Hyperledger Fabric Client SDK for Go*, Jul. 20, 2022. [Online]. Available: https://github.com/hyperledger/fabric-sdk-go.

[15] Hyperledger Foundation, *Hyperledger Fabric Client SDK for Node.js*, Aug. 10, 2022. [Online]. Available: https://github.com/hyperledger/fabric-sdk-node.

[16] Hyperledger Foundation, *Hyperledger Fabric SDK for Java*, Jul. 18, 2022. [Online]. Available: https://github.com/hyperledger/fabric-sdk-java.

[17] C. Cachin, S. Schubert, and M. Vukolic, “Non-Determinism in Byzantine Fault-Tolerant Replication,” in *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, P. Fatourou, E. Jiménez, and F. Pedone, Eds., vol. 70, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dec. 2016, 24:1–24:16, ISBN: 978-3-95977-031-6. DOI: 10.4230/LIPIcs.OPODIS.2016.24.

[18] S. Zhang, E. Zhou, B. Pi, J. Sun, K. Yamashita, and Y. Nomura, “A Solution for the Risk of Non-deterministic Transactions in Hyperledger Fabric,” in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, May 2019, pp. 253–261. DOI: 10.1109/BLOC.2019.8751453.

[19] Hyperledger Foundation, *Chaincode for Developers*, Sep. 18, 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.3/chaincode4ade.html.

[20] Hyperledger Foundation, *Ledger*, Apr. 27, 2018. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.3/ledger/ledger.html.

[21] Hyperledger Foundation, *Chaincode namespace*, Feb. 11, 2021. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/chaincodenamespace.html.

[22] Hyperledger Foundation, *Transaction Flow*, Nov. 4, 2021. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/txflow.html.

[23] C. Mitchell, *Trusted Computing*. IET Digital Library, Dec. 2005, ISBN: 978-0-86341-525-8. DOI: 10.1049/PBPC006E.

[24] R. Anderson, “Cryptography and competition policy: Issues with ’trusted computing’,” in *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, Association for Computing Machinery, Jul. 13, 2003, pp. 3–10, ISBN: 978-1-58113-708-8. DOI: 10.1145/872035.872036.

[25] R. N. Akram, K. Markantonakis, and K. Mayes, “An Introduction to the Trusted Platform Module and Mobile Trusted Module,” in *Secure Smart Embedded Devices, Platforms and Applications*, K. Markantonakis and K. Mayes, Eds., Springer, Sep. 13, 2013, pp. 71–93, ISBN: 978-1-4614-7915-4. DOI: 10.1007/978-1-4614-7915-4_4.

[26] M. Sabt, M. Achemlal, and A. Bouabdallah, “Trusted Execution Environment: What It is, and What It is Not,” in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug. 2015, pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[27] ARM Limited, *ARM Security Technology - Building a Secure System using TrustZone Technology*, Apr. 2009. [Online]. Available: https://documentation-service.arm.com/static/5f212796500e883ab8e74531.

[28] Advanced Micro Devices, Inc., *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*, Jan. 2020. [Online]. Available: `https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf`.

[29] 01.org - Intel Open Source, *Intel Software Guard Extensions SDK for Linux*, Mar. 1, 2017. [Online]. Available: `https://01.org/intel-software-guard-extensions`.

[30] "Trusted Computer System Evaluation Criteria," in *The 'Orange Book' Series*, US Department of Defense, 1985, pp. 1–129, ISBN: 978-1-349-12020-8. DOI: `10.1007/978-1-349-12020-8_1`.

[31] Fortanix, *Why we chose Intel SGX to power Runtime Encryption*, Oct. 25, 2018. [Online]. Available: `https://fortanix.medium.com/why-we-chose-intel-sgx-to-power-runtime-encryption-d0a89522d864`.

[32] S. Cetola, *Trusted Execution Environments: A Technical Overview of Intel SGX, Arm TrustZone, and RISC-V PMP*, Feb. 1, 2021. [Online]. Available: `https://www.youtube.com/watch?v=MREwcSo0uz4`.

[33] V. Costan and S. Devadas, "Intel SGX Explained," 086, Jan. 31, 2016. [Online]. Available: `https://eprint.iacr.org/2016/086`.

[34] S. Mofrad, F. Zhang, S. Lu, and W. Shi, "A comparison study of intel SGX and AMD memory encryption technology," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, Association for Computing Machinery, Jun. 2, 2018, pp. 1–8, ISBN: 978-1-4503-6500-0. DOI: `10.1145/3214292.3214301`.

[35] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, no. 7, Aug. 14, 2013. [Online]. Available: `https://www.intel.co.uk/content/www/uk/en/developer/articles/technical/innovative-technology-for-cpu-based-attestation-and-sealing.html`.

[36] SSLab - Georgia Institute of Technology, *Attestation - SGX 101*, Jul. 13, 2019. [Online]. Available: `https://sgx101.gitbook.io/sgx101/sgx-bootstrap/attestation`.

[37] Intel Corporation, *Attestation Services for Intel Software Guard Extensions*, 2021. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html`.

[38] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, "Supporting Third Party Attestation for Intel® SGX with Intel® Data Center Attestation Primitives," Apr. 19, 2019. [Online]. Available: `https://www.intel.com/content/dam/develop/external/us/en/documents/intel-sgx-support-for-third-party-attestation-801017.pdf`.

[39] M. U. Sardar, R. Faqeh, and C. Fetzer, "Formal Foundations for Intel SGX Data Center Attestation Primitives," in *Formal Methods and Software Engineering*, S.-W. Lin, Z. Hou, and B. Mahony, Eds., Springer International Publishing, Dec. 19, 2020, pp. 268–283, ISBN: 978-3-030-63406-3. DOI: `10.1007/978-3-030-63406-3_16`.

[40] Intel Corporation, *Getting Started with Intel Software Guard Extensions SDK for Microsoft Windows OS*, Jun. 21, 2017. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/articles/guide/getting-started-with-sgx-sdk-for-windows.html`.

[41] Intel Corporation, *Intel Software Guard Extensions for Linux OS*, Aug. 10, 2022. [Online]. Available: `https://github.com/intel/linux-sgx`.

[42] Intel Corporation, *Introduction to Intel SGX Sealing*, Apr. 5, 2016. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-intel-sgx-sealing.html`.

[43] Open Enclave, *Open Enclave SDK - Product Page*, Apr. 2, 2021. [Online]. Available: `https://openenclave.io/sdk/`.

[44] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, “Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric,” May 22, 2018. arXiv: `1805.08541 [cs]`. [Online]. Available: `http://arxiv.org/abs/1805.08541`.

[45] Google, *Cgo command - Go Packages*, Aug. 2, 2022. [Online]. Available: `https://pkg.go.dev/cmd/cgo`.

[46] Edgeless Systems, *EGo*, Aug. 14, 2022. [Online]. Available: `https://github.com/edgelesssys/ego`.

[47] Edgeless Systems, *EGo - Product Page*, Apr. 28, 2022. [Online]. Available: `https://www.edgeless.systems/products/ego/`.

[48] Edgeless Systems, *Edgeless RT*, Jul. 22, 2022. [Online]. Available: `https://github.com/edgelesssys/edgelessrt`.

[49] Open Enclave, *Open Enclave SDK*, Aug. 5, 2022. [Online]. Available: `https://github.com/openenclave/openenclave`.

[50] M. Eckert, *EGo & Marblerun*, Mar. 19, 2021. [Online]. Available: `https://www.youtube.com/watch?v=e_7q1uOpCqw`.

[51] Edgeless Systems, *How we built EGo*, Jun. 25, 2021. [Online]. Available: `https://blog.edgeless.systems/how-we-built-ego-c02220360503`.

[52] F. Schuster. “EGo: Effortlessly build confidential apps in Go.” (Feb. 21, 2021), [Online]. Available: `https://blog.edgeless.systems/ego-effortlessly-build-confidential-apps-in-go-dc2b1460e1bf`.

[53] Edgeless Systems, *Ego module - Go Packages*, Jul. 19, 2022. [Online]. Available: `https://pkg.go.dev/github.com/edgelesssys/ego`.

[54] Edgeless Systems, *EGo - Documentation*, Jul. 19, 2022. [Online]. Available: `https://docs.edgeless.systems/ego`.

[55] P. Porambage, Y. Siriwardana, R. Sedar, *et al.*, “INtelligent Security and PervasIve tRust for 5G and Beyond,” INSPIRE-5Gplus Consortium, WP3, T3.3, Mar. 3, 2022. [Online]. Available: `https://www.inspire-5gplus.eu/wp-content/uploads/2022/03/i5-d3.3_5g_security_new_breed_of_enablers_v1.0.pdf`.

[56] Hyperledger, *Mock_enclave.go at a97702902bb6a97475fe4d2170e34569755f03a5 · Hyperledger Fabric Private Chaincode*, Aug. 4, 2022. [Online]. Available: `https://github.com/hyperledger/fabric-private-chaincode/blob/a97702902bb6a97475fe4d2170e34569755f03a5/ecc/chaincode/enclave/mock_enclave.go`.

[57] R. Zappoli, *Chaincode at golang-support · Fabric Private Chaincode*, Dec. 3, 2021. [Online]. Available: `https://github.com/ricc-zappoli/fabric-private-chaincode/tree/golang-support/samples/chaincode`.

[58] Hyperledger Foundation, *Hyperledger Fabric Private Chaincode at a97702902bb6a97475fe4d2170e34569755f03a5*, Jul. 5, 2022. [Online]. Available: `https://github.com/hyperledger/fabric-private-chaincode/tree/a97702902bb6a97475fe4d2170e34569755f03a5`.

[59] Hyperledger Foundation, *Release v2.3.3 · Hyperledger Fabric*, Sep. 8, 2021. [Online]. Available: https://github.com/hyperledger/fabric/releases/tag/v2.3.3.

[60] Intel Corporation, *Release version 2.11 · Intel SGX driver*, Sep. 2, 2020. [Online]. Available: https://github.com/intel/linux-sgx-driver/releases/tag/sgx_driver_2.11.

[61] Edgeless Systems, *Release v1.0.0 · EGo*, Jul. 19, 2022. [Online]. Available: https://github.com/edgelesssys/ego/releases/tag/v1.0.0.

[62] Riki95. "Why do I take more than 2 seconds to just do a transaction?" Stack Overflow. (Jul. 8, 2019), [Online]. Available: https://stackoverflow.com/q/56936560.

[63] Hyperledger Foundation, *Configtx.yaml at a97702902bb6a97475fe4d2170e34569755f03a5 · Hyperledger Fabric Private Chaincode*, Dec. 28, 2020. [Online]. Available: https://github.com/hyperledger/fabric-private-chaincode/blob/a97702902bb6a97475fe4d2170e34569755f03a5/integration/config/configtx.yaml#L286.

[64] R. Zappoli, *Simple-cli-go at golang-support · Fabric Private Chaincode*, Aug. 5, 2022. [Online]. Available: https://github.com/ricc-zappoli/fabric-private-chaincode/tree/golang-support/samples/application/simple-cli-go.

[65] Hyperledger Foundation, *Hyperledger Fabric Samples*, May 9, 2022. [Online]. Available: https://github.com/hyperledger/fabric-samples/tree/d3a61e1d4f8f685a4bb3ea71259591365f33f51d/chaincode.

[66] Hyperledger Foundation, *Fabric Contract APIs and Application APIs*, Feb. 8, 2022. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/sdk_chaincode.html.

[67] Hyperledger Foundation, *Abstore.go at 5e5d2c8e01728dad15b11fd83126a1d29b961be1 · Hyperledger Fabric Samples*, Aug. 23, 2019. [Online]. Available: https://github.com/hyperledger/fabric-samples/blob/5e5d2c8e01728dad15b11fd83126a1d29b961be1/chaincode/abstore/go/abstore.go.

[68] Hyperledger Foundation, *Fabcar.go at c4d8bb74cfeffe263c772e1a2edba4aea07146f4 · Hyperledger Fabric Samples*, Sep. 24, 2019. [Online]. Available: https://github.com/hyperledger/fabric-samples/blob/c4d8bb74cfeffe263c772e1a2edba4aea07146f4/chaincode/fabcar/go/fabcar.go.

[69] C. Müller, M. Brandenburger, C. Cachin, P. Felber, C. Göttel, and V. Schiavoni, "TZ4Fabric: Executing Smart Contracts with ARM TrustZone," *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pp. 31–40, Sep. 2020. DOI: 10.1109/SRDS51746.2020.00011.

[70] J. Kelly, "Nine of world's biggest banks join to form blockchain partnership," *ReutersBanks*, Sep. 15, 2015. [Online]. Available: https://www.reuters.com/article/us-banks-blockchain-idUSKCN0RF24M20150915.

[71] R3 LLC, *Smart contracts - R3 Documentation*, Apr. 7, 2020. [Online]. Available: https://docs.r3.com/en/platform/corda/4.8/open-source/key-concepts-contracts.html.

[72] R3 (R3CEV LLC), *Corda and SGX: A privacy update*, Jun. 22, 2017. [Online]. Available: https://www.corda.net/blog/corda-and-sgx-a-privacy-update/.

[73] H. Wang, P. Wang, Y. Ding, *et al.*, "Towards Memory Safe Enclave Programming with Rust-SGX," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, Association for Computing Machinery, Nov. 6, 2019, pp. 2333–2350, ISBN: 978-1-4503-6747-9. DOI: `10.1145/3319535.3354241`.

[74] Edgeless Systems, *MarbleRun - Product Page*, Oct. 19, 2021. [Online]. Available: `https://docs.edgeless.systems/marblerun/#/`.

[75] Edgeless Systems, *MarbleRun - Documentation*, Jun. 2, 2022. [Online]. Available: `https://www.edgeless.systems/products/marblerun/`.

Appendix

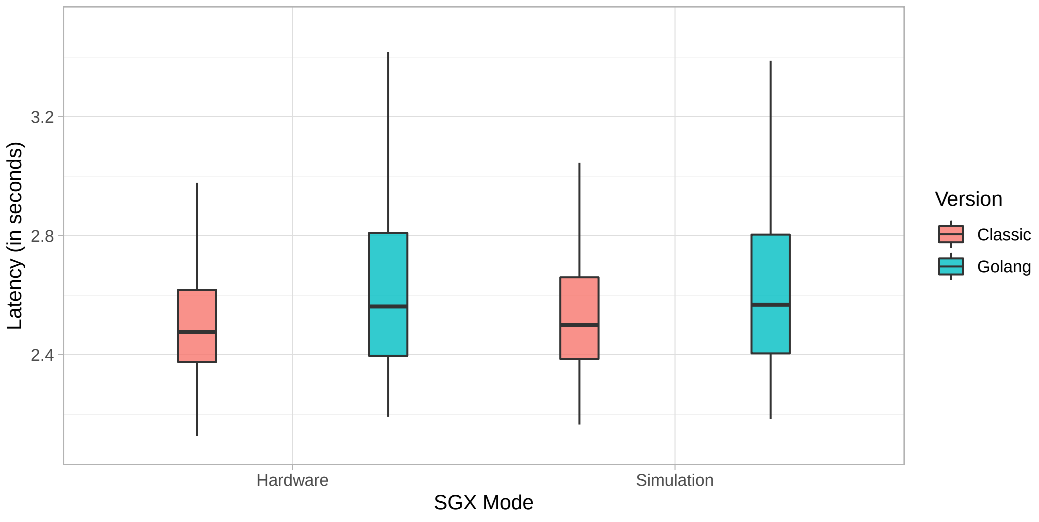

Figure 8.1: Results for all `Init()` calls

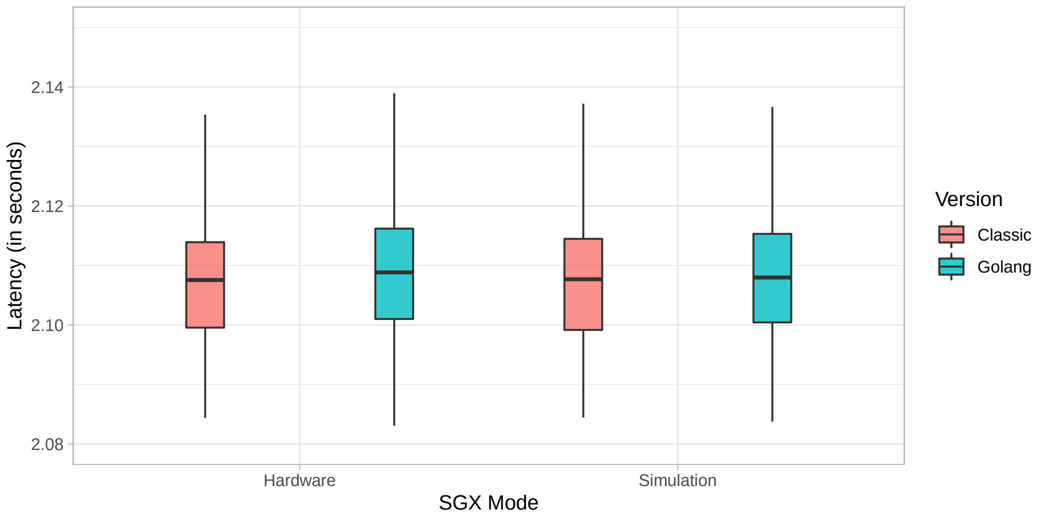

Figure 8.2: Results for all `Invoke()` calls

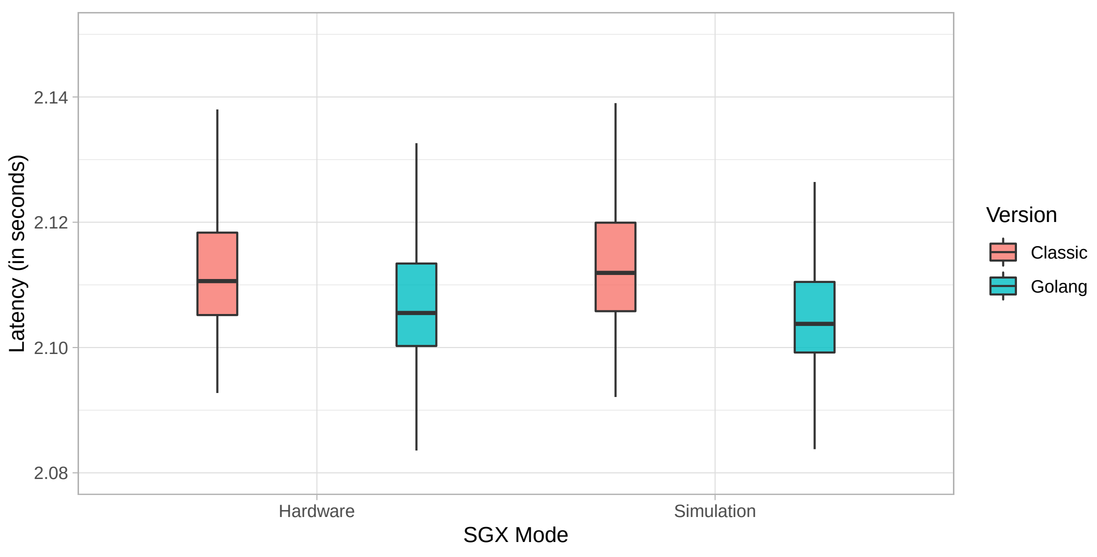

Figure 8.3: Results for `Invoke("init")` calls

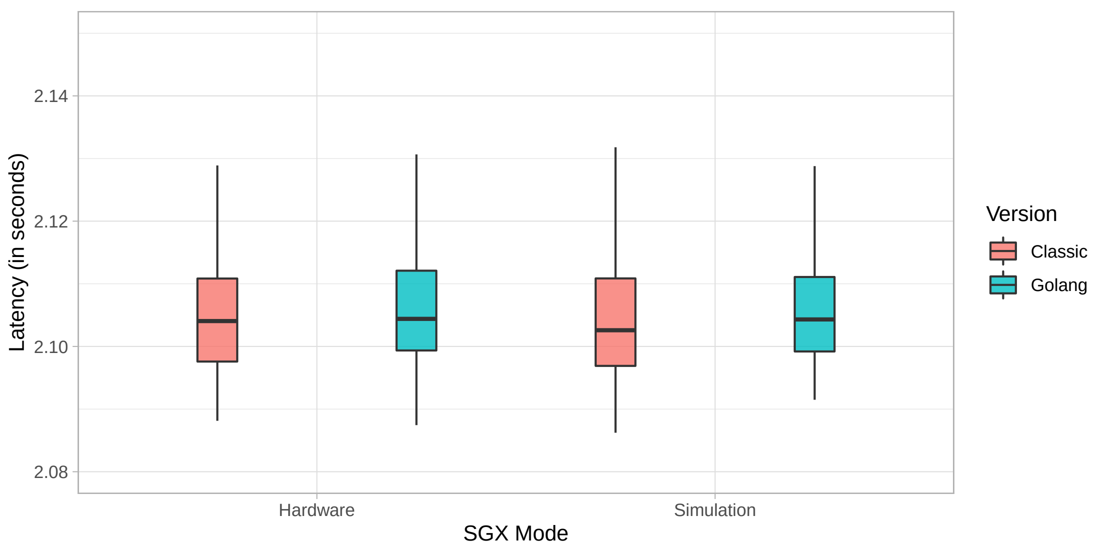

Figure 8.4: Results for `Invoke("create")` calls

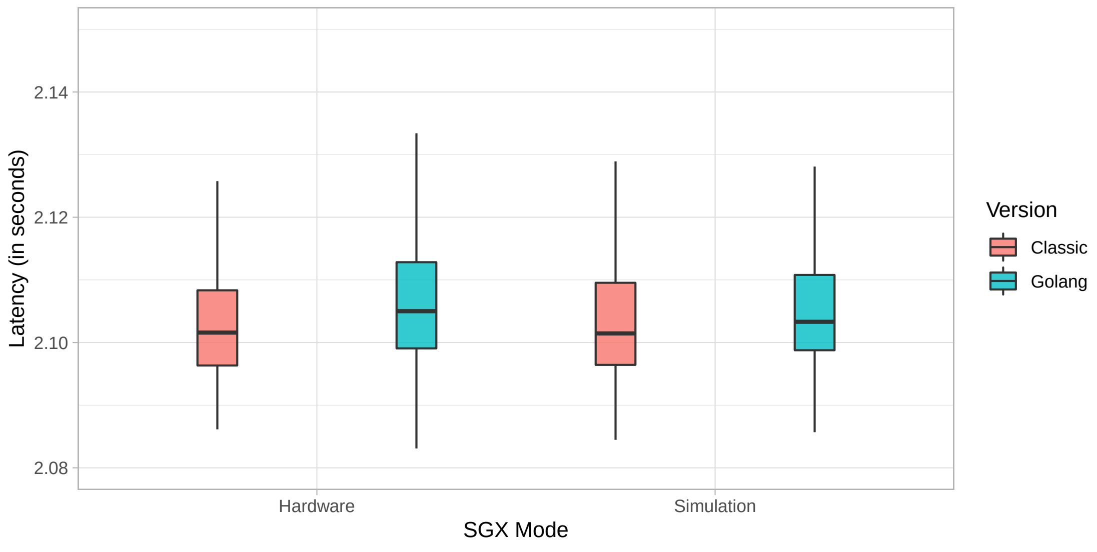

Figure 8.5: Results for all `Invoke("submit")` calls

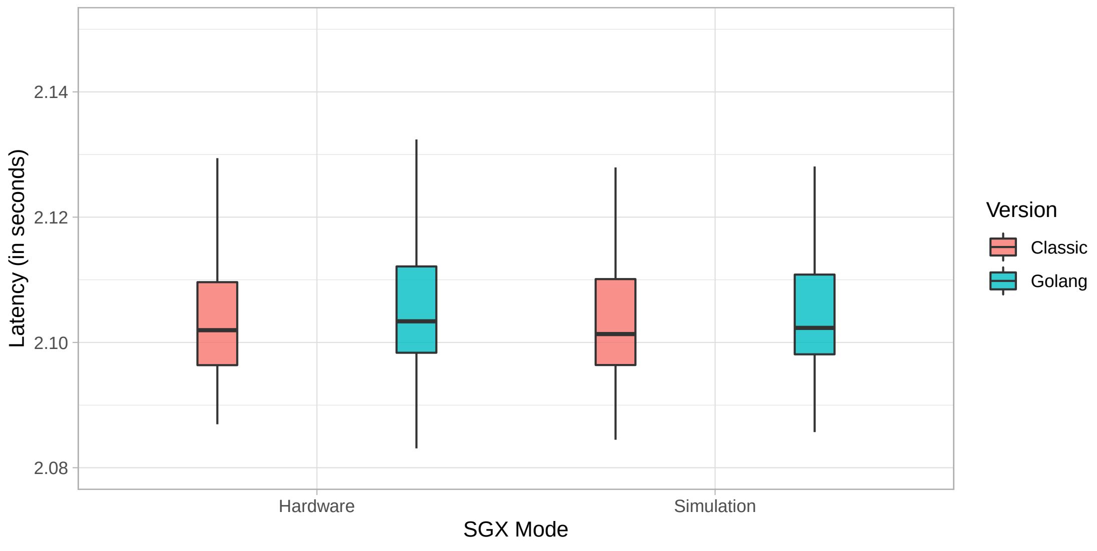

Figure 8.6: Results for `Query("submit", "Auction", "John")` calls

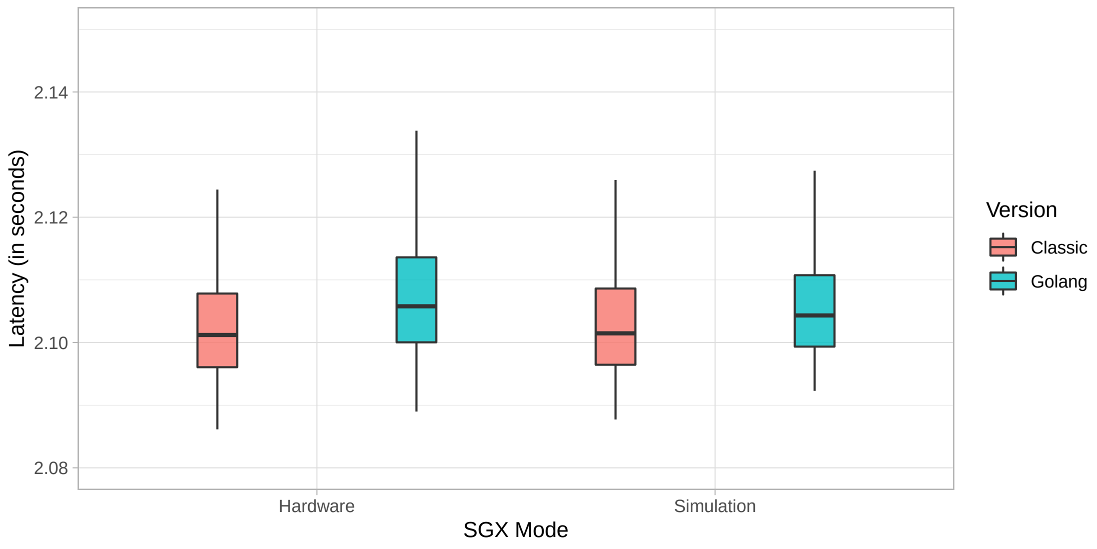

Figure 8.7: Results for `Invoke("submit", "Auction", "Jane")` calls

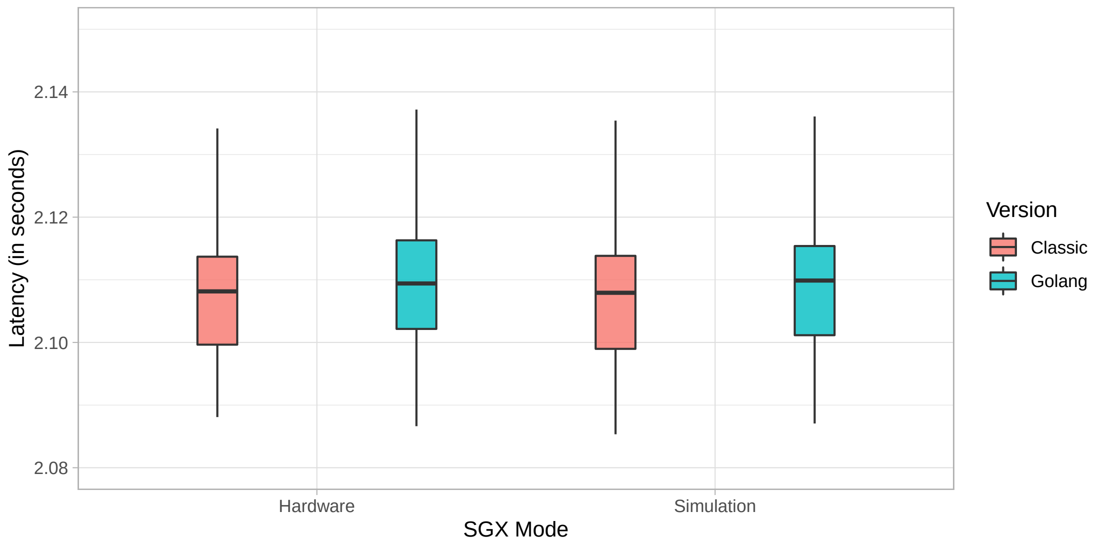

Figure 8.8: Results for all `Query("submit")` calls

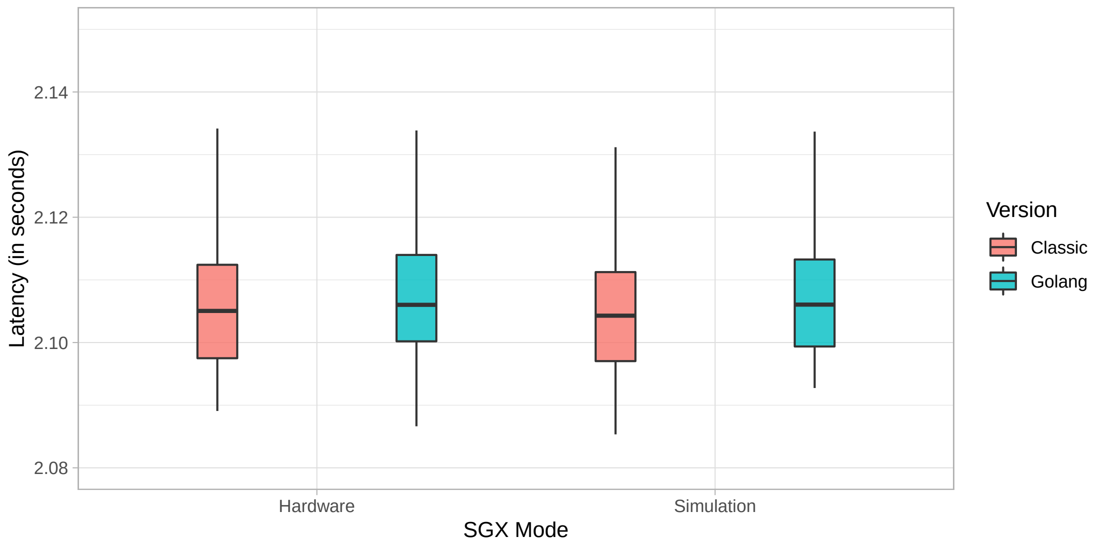

Figure 8.9: Results for `Query("submit", "Auction", "John")` calls

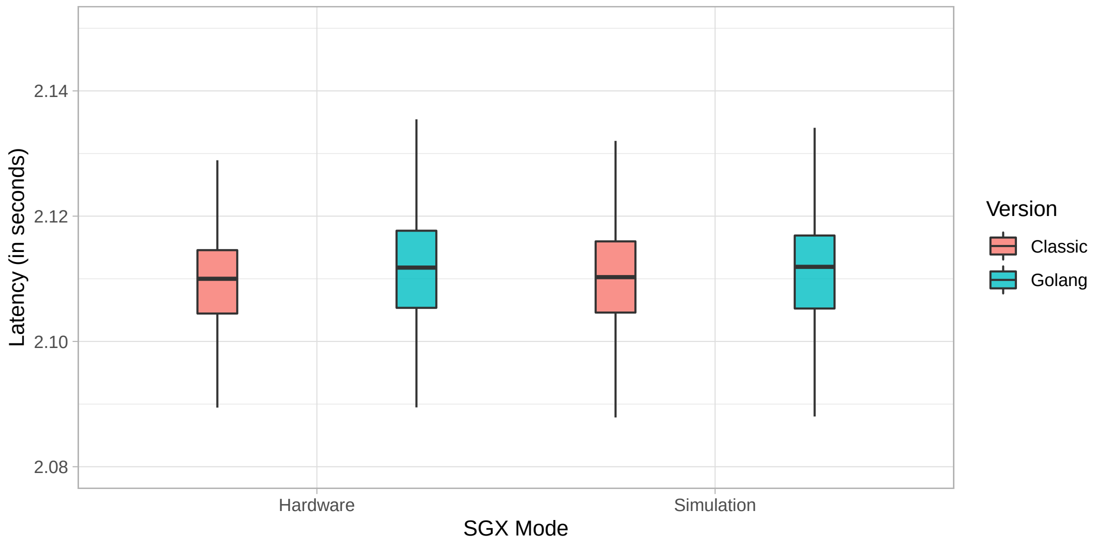

Figure 8.10: Results for `Query("submit", "Auction", "Danny")` calls

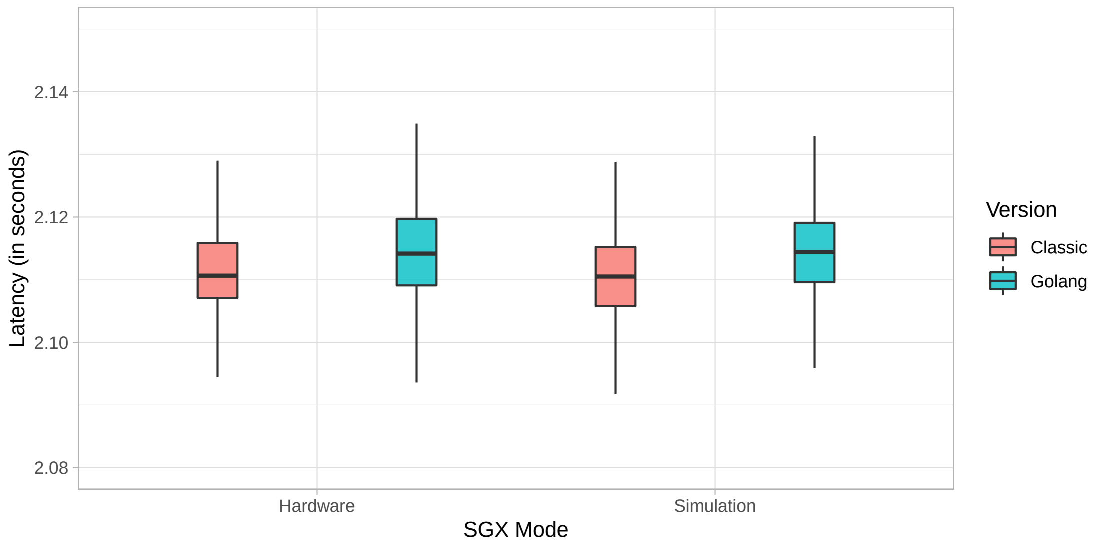

Figure 8.11: Results for `Invoke("close")` calls

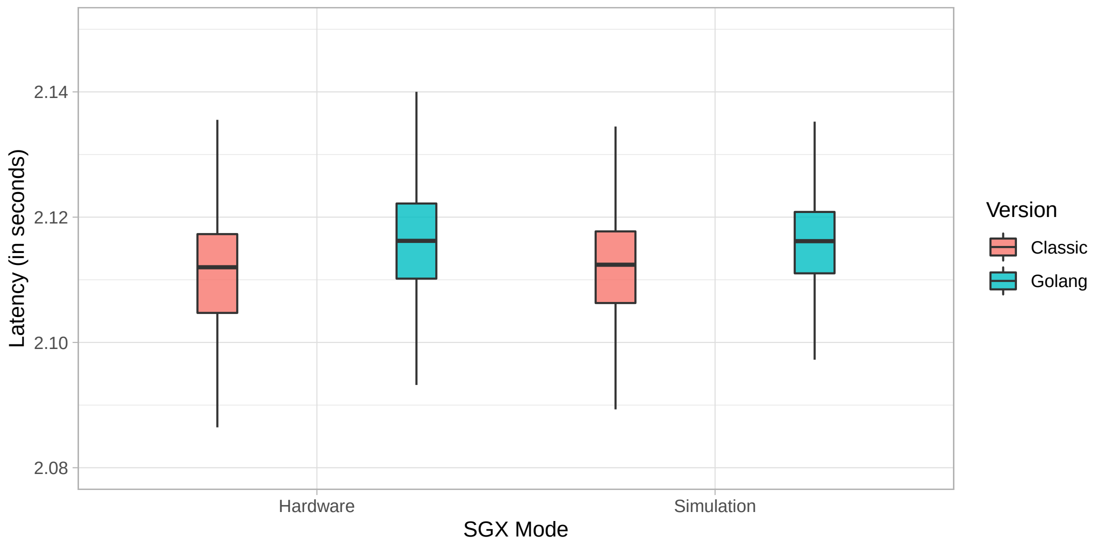

Figure 8.12: Results for `Invoke("eval")` calls