University of Stavanger

IVER B. BOLSTAD AND ROBIN AASAN

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Simplifying Smart Contract Execution and Trusted Environments with EGo

Bachelor's Thesis - Computer Science - May 2023

```go
func (m *Manager) NewConfiguration(opts ...gorums.ConfigOption) (c *Configuration, err error) {
    if len(opts) < 1 || len(opts) > 2 {
        return nil, fmt.Errorf("wrong number of options: %d", len(opts))
    }
    c = &Configuration{}
    for _, opt := range opts {
        switch v := opt.(type) {
        case gorums.NodeListOption:
            c.Configuration, err = gorums.NewConfiguration(m.Manager, v)
            if err ≠ nil {
                return nil, err
            }
        case QuorumSpec:
            // Must be last since v may match QuorumSpec if it is interface{}
            c.qspec = v
        default:
            return nil, fmt.Errorf("unknown option type: %v", v)
        }
    }
    // return an error if the QuorumSpec interface is not empty and no implementati
    var test interface{} = struct{}{}
    if _, empty := test.(QuorumSpec); !empty && c.qspec == nil {
        return nil, fmt.Errorf("missing required QuorumSpec")
    }
    return c, nil
}
```

We, **Iver B. Bolstad and Robin Aasan**, declare that this thesis titled, “Simplifying Smart Contract Execution and Trusted Environments with EGo” and the work presented in it are our own. We confirm that:

- This work was done wholly or mainly while in candidature for a bachelor’s degree at the University of Stavanger.
- Where we have consulted the published work of others, this is always clearly attributed.
- Where we have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely our own work.
- We have acknowledged all main sources of help.

"The computer was born to solve problems that did not exist before."

– Bill Gates

Abstract

IoT has had significant growth during this last decade. However, its adoption on the edge isn't obviously feasible as it poses risks concerning data privacy, integrity and accountability. This is particularly true in situations involving multiple competitive stakeholders or when deploying devices in remote areas without proper surveillance as data manipulation by intruders may occur. Trusted Execution Environments (TEEs) have the advantage of isolating sensitive operations by separating themselves from the host operating system, ensuring confidentiality and privacy. Together with blockchain technology, TEEs can help establish trust between stakeholders. ChainBox is one such framework that enables trusted computing on the edge by utilizing TEEs and blockchain. Furthermore, ChainBox incorporates the use of WebAssembly for executing smart contracts providing additional isolation. However, Chainbox has a complex programming model. Thus, this thesis aims to evaluate and examine a framework called *EGo* that provides a straightforward programming model for developing Trusted Execution Environments (TEEs). Accomplishing this is done through reimplementing ChainBox, leveraging the capabilities of the EGo.

We show that EGo is a user-friendly and straightforward framework, most relevant for developers with limited experience with more advanced SDKs such as the Intel SGX SDK. With our reimplementation of ChainBox, using simplified smart contracts and non-secure connections, EGo performs with a throughput of about 3 to 3.5 times worse than the standard Go compiler.

Acknowledgements

We would first and foremost thank our sophisticated and enthusiastic supervisor, Leander Jehl, for his guidance throughout this project. He has been extremely helpful with both technical and theoretical questions regarding this thesis. We would also like to extend our appreciation to Hanish Gagoda and the Edgeless System team for helping with technical complications. Last but not least, we would like to thank family and friends for their overall encouragement towards ever-higher achievement.

Contents

Chapter 1

Introduction

1.1 Motivation

Internet-of-things (IoT) keeps getting more attractive to smaller businesses as it has a significant possibility for automatization and increased efficiency. However, its adoption isn't easily implemented for all businesses when taking privacy, data integrity, cost and ownership into account. For instance, in agriculture, monitoring food such as fruits may be of interest as it can be used to ensure consumers the quality of the fruit. Sensing-, data-, or analytics-as-a-service can ease adoption with the monitorization, but they have significant drawbacks mainly due to the fact that these models may belong to different competitive and untrusted stakeholders [1] wanting to manipulate the data for their own profit. These models, i.e., sensor infrastructure, data, and analytical models, in reality only serve their purpose if there is reciprocal trust between the various competitive stakeholders.

ChainBox [1] is one example of an application that enables trusted edge computing and data sharing between users. Unlike other systems that rely on distribution to facilitate trust, such as Hyperledger Fabric [2], ChainBox uses Trusted Execution Environment (TEE) technology to achieve trusted edge computing. This is accomplished through the use of SGX, a type of TEE that ensures the confidentiality and integrity of sensitive data. Additionally, ChainBox utilizes blockchain together with smart contracts for secure data sharing. The Blockchain is employed on a single-edge device, which minimizes costs and is reliable and secure by leveraging a TEE, contrary to systems that rely on the distribution.

ChainBox is a sophisticated application constructed primarily using C/C++. However, the implementation with Intel SGX has a complex programming model and limited adoption [3]. It would be of great interest to have a framework that effectively addresses the same issues as those addressed by ChainBox, but with a simpler programming model. Such a framework would make confidential computing more feasible for a diverse range of developers. It is therefore this thesis aims to look at and evaluate a relatively new framework called *EGo*. By using the design of ChainBox as an example, we will evaluate and test EGo by reimplementing ChainBox with this framework. EGo supports the use of enclaves using Intel SGX, but has the advantage of letting almost any Go-written application run inside enclaves without needing to partition the application into untrusted and trusted components, compared to how ChainBox is created.

1.2 Objectives

This thesis consists of two main objectives. We are first and foremost going to evaluate the *EGo* as a framework for building confidential applications. With the knowledge of how the framework works and what its limitations are, we will look to reimplementation a smart contract framework called: *ChainBox* [1]. The following objectives will be used to achieve this:

- Describe and study EGo focusing on its potential and limitations.
- Study ChainBox and the possibility of reimplementing it with EGo
- Implement and evaluate the reimplemented version of ChainBox

1.3 Approach and Contributions

In this thesis, we adopted an approach that consisted of studying the functionalities of both ChainBox and EGo, followed by the subsequent reimplementation of ChainBox using EGo. The EGo SDK was downloaded and tested both in a correctly configured cloud virtual machine (VM) and on-premise, giving insight into the process from installation to execution as well as the quality of the documentation. After a clearer understanding of how to execute a variety of EGo-compiled applications was obtained, we began the journey to reimplement ChainBox which further broadened our knowledge of the framework. Our approach led to a variety of findings regarding EGo. To summarize, our contributions were:

- Showing how EGo compromises user-friendliness with performance
- Evaluating EGo through multiple experiments, including a comparison of the framework to the standard Go compiler
- Showcasing how WebAssembly modules can function within an EGo enclave
- Proving how a sophisticated system such as ChainBox successfully can be implemented using EGo

1.4 Outline

In ch. 2 we will cover the necessary and relevant background information for this thesis. We will mainly introduce technologies and concepts such as Trusted Execution Environment, focusing on Intel SGX, Webassembly and Blockchain. Ch. 3 will cover the framework used to build our application - EGo, which will provide clarity and justification for our implementation in ch. 5. In ch. 4, we review related works to contextualize our research. In ch. 6, we show our results using EGo through a variety of experiments. We discuss EGo overall as a framework and elaborate on potential areas for improvement regarding our implementation in ch. 7. Lastly, we present what was not implemented from ChainBox in ch. 8 before finally concluding and summarizing our thesis.

Chapter 2

Background

The technologies within trusted execution environments include both hardware and software, which cooperates to make a secure and isolated environment within the computing system. Hardware-based security systems such as Intel SGX are designed to provide security through a specific isolated environment in the physical processor. This is also combined with software-based security technologies and is also essential in our system. To understand how and why this works as a whole, several different features and mechanisms need to be acknowledged. Hence, in this chapter, we explain the key technologies that make up our system.

2.1 Trusted Execution Environment

A trusted execution environment (TEE) is a trusted area inside the main processor on a device, often on server-side machines. Trusted environments are often used for sensitive operations and computations, and to store sensitive information.

TEEs often combine both hardware and software to create isolated areas. At the core of the environment is the specialized processor, as we will explain more in the chapter below. Hardware-based solutions also need additional features including attestation, encryption, and digital certifications to add further security to the environment. This is achieved with software-based solutions, which we also discuss further down in this chapter.

The importance of these environments has increased over the past few years as the demand for digital trust grows rapidly. The concept itself is not new, but it is no longer used only in high-end devices [4].

2.1.1 SGX

In general, Intel Software Guard Extensions (SGX) enable applications to execute code inside their own trusted environment, known as *enclaves*, allowing them to store secret data. Personal information, encryption keys, and passwords are examples of such secrets. The SGX SDK gives the developers direct control over the security of the system. Additionally, there are frameworks that use the SGX SDK as their underlying technology, such as EGo, giving developers indirect access to SGX capabilities. Enclaves are isolated and encrypted memory regions in RAM. These regions are non-addressable and can be used by the application when needed, often when secrets must be retrieved or modified. Secrets inside an enclave will be kept protected even when the application, BIOS, and OS are compromised. This allows for high integrity and security for the disadvantaged.

Figure 2.1 describes how an SGX application runs, we can see it runs in two different components: trusted and untrusted. The application runs in untrusted memory until it needs to retrieve or modify secrets. When this happens, the application will create an enclave in the trusted memory and calls, using enclave calls (E-CALL), a trusted function used to work within the trusted part. When the E-CALL is made, the application will see the enclave's code and data as clear plain text. Calls to the trusted part from any other part of the system will not work, as can be observed at the bottom of figure 2.1. Functions can also call the outside of enclaves with *outside calls* (O-CALL). This architecture is just a standard for trusted execution environments using Intel SGX and may differ from other types of frameworks, as we will discuss in ch. 3.

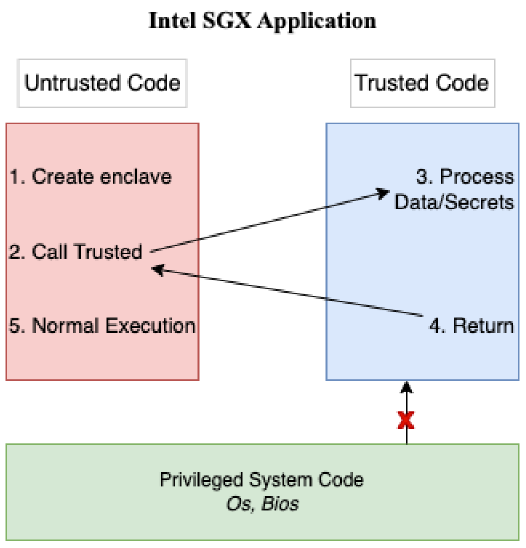


Figure 2.1: Intel SGX Application

Whenever enclaves are run on servers, connecting clients often need insurance that they are connected to the expected enclave. *Remote attestation* is used to verify the authenticity and integrity of the TEE by including a third party. This ensures the relying party that it is attested to an authentic TEE. Intel SGX currently supports two different solutions to remote attestation, elliptic curve digital signature algorithm (ECDSA) and enhanced privacy ID (EPID) based attestation [5]. Transport layer security connections can be established through both these attestation methods. Intel states that remote attestation gives the relying party increased confidence that the software is running [5]:

- Inside an Intel® Software Guard Extension (Intel® SGX) enclave
- On a fully updated system at the latest security level (also referred to as the trusted computing base [TCB] version)

2.2 WebAssembly

WebAssembly (WASM) is a low-level programming language with a binary instruction format. It was originally designed for web browsers, but later on became supported for a wide range of platforms. WASM provides a variety of security features. For example, when WASM modules are executed, they are not run freely on the host system, but rather in a sandboxed execution environment [6]. This means that the compiled code has access to neither arbitrary code, access files, or other resources on the host machine. WASM modules are also memory-safe and isolated from making network requests.

In figure 2.2 you can see how WASM, compiled from a preferred programming language, may be used in web-embedded or non-web-embedded contexts.

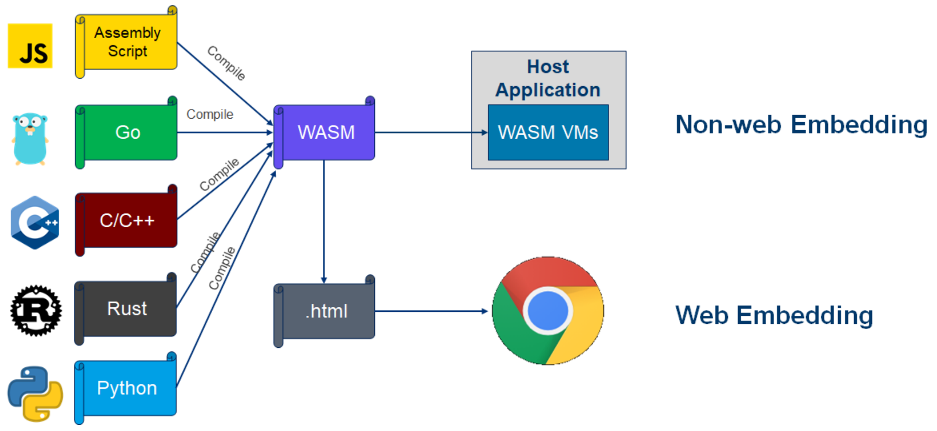


Figure 2.2: Compiling code from standard programming languages to WASM for web-embedded and non-web-embedded contexts. Figure retrieved from [7].

A Go package bringing the required API for executing WebAssembly code called *Wasmer-go* [8] has been incorporated for our implementation. Wasmer-go is based on a runtime called Wasmer, a WebAssembly runtime that enables the utilization of lightweight containers [9]. Wasmer-go also supports WebAssemby System Interface (WASI), a standardized interface between WASM modules and the host operating system, providing a secure and portable way for WebAssembly code to interact with the underlying system. WASI is used in ChainBox which is something our implementation therefore can use in a similar manner. Wasmer-go needs an engine and a corresponding instance to run the code WASM code. The Wasmer-go instance itself is an executable instance of the WASM module. Moreover, it contains all exported WASM functions that allow calling into the code itself from outside the sandboxed environment [10]. Note that a WASM runtime is integrated into an application, such as an application written in Go.

It's important to clarify that figure 2.2 demonstrates the compilation of an application into a WebAssembly module, which should not be confused with the integration of the runtime itself.

2.3 ChainBox

ChainBox is a TEE-based smart contract execution framework that tackles privacy, integrity, and confidentiality. It is meant to substitute systems realizing replication and distribution, e.g., *HyperLedger Fabric* [2] which ChainBox is partially based on. This makes their system more feasible for deployment in fields with limited resources and solves multiple bottlenecks in for example Internet-of-Things.

2.3.1 Architecture and Design

ChainBox mainly consists of three different components;

- **The ordering service**
- **The blockchain storage**
- **The runtime enclave**

The overall architecture is displayed in figure 2.3.

The ordering service is the core component of the system. It validates transactions and adds them to the blockchain as shown under point ① in figure 2.3. The ordering service runs inside an enclave and signs each block from within, ensuring the integrity of the created blocks. Communication between the **runtime enclave** and the ordering service with regards to the transactions and blocks, respectively, is attested both ways ensuring that both the components work correctly. This inter-enclave communication is also encrypted adding further security.

Next up is **the blockchain storage**, which can be seen under point ② in figure 2.3. In ChainBox, the blockchain storage is located on disk in *Protobuf* format. As mentioned above, each block is signed by the ordering service. Upon initialization of the ordering service, each block is validated with the signing key, if there are any existing blocks. If there is no existing one, a genesis block is created. Each block in the blockchain is considered public information.

Lastly, the figure shows **the runtime enclave** under point ③. The runtime mainly consists of a WASM module, referred to as a smart contract, and offers some API calls listed in 2.1. It is responsible for loading, instantiating, and executing smart contracts within the WASM module and sending transactions to the ordering service. The runtime allows for parallel execution of smart contracts and notifies the other runtimes when changes occur.

SET	Update a key-value pair
GET	Returns a pointer to memory and the length of the present value
FREE	Free memory used by the smart contract
REGISTER	Register the contract for callbacks on a specific key

Table 2.1: ChainBox Runtime API calls

The orderingservice and runtime, ① and ③, are run inside enclaves, explained in more detail in ch. 2.1.1. When applications run inside enclaves it is isolated from the outside, but they can still be reached by non-encrypted connections if they expose open ports to allow for HTTP communication, for example. To make the system more secure, the connection between enclaves needs to be secured as well. Chanbox uses TLS connections between the enclaves which can be seen in 2.3 as the blue and purple arrows.

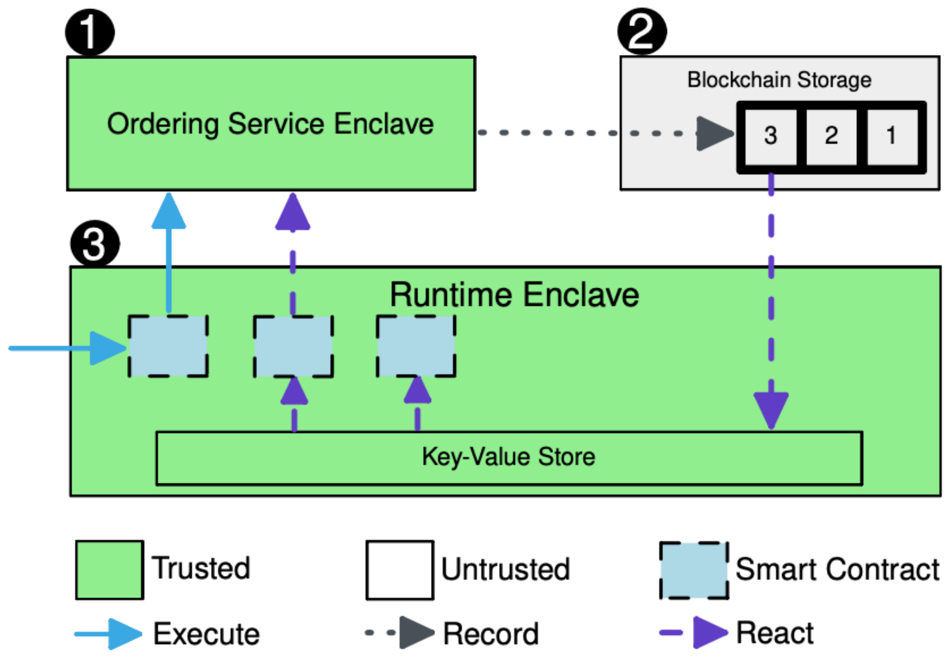


Figure 2.3: ChainBox Architecture [1]

2.3.2 Functionality and Use Case

One example of ChainBox being used in the field would be something that collects data from the real world. This could for example be weather data from a weather station. This weather data is being used to schedule flights at an airport. Thus, it is crucial that the data is correct and by means not changed with bad intent by the station administrator. The weather data would be sent through their environment and stored safely in the blockchain making the data considered trusted by all other stakeholders. This use-case is of course given that the device collecting the data functions properly and is not tampered with itself.

Chapter 3

EGo

In this chapter, we carefully analyze the EGo framework and explain its potential and limitations. Having some background information on the framework will hopefully simplify as well as justify the way we have reimplemented ChainBox.

3.1 Introduction

EGo is an SDK (software development kit), that Edgeless Systems have created as an open-source solution [11]. This particular SDK has been designed to enable the use of Intel's SGX mechanisms in applications written in Go. Essentially, EGo can be viewed as an extension of the Go compiler, in which it creates binaries that can be run inside an SGX enclave.

3.2 Architecture

SGX has its own programming model where the application is partitioned into trusted and untrusted parts, as explained in sec. 2.1.1. In EGo, however, the transition between the trusted and untrusted sections is hidden inside the EGo runtime, making them transparent to the developer. As virtually, the whole program runs in an enclave, developers can assume that sensitive information in their program remains secure and unchanged throughout execution, even on untrusted computers. Figure 3.1, illustrates where important sections of an application such as the **Data**, **Code** and **Runtime** reside within an **unstrusted computer**.

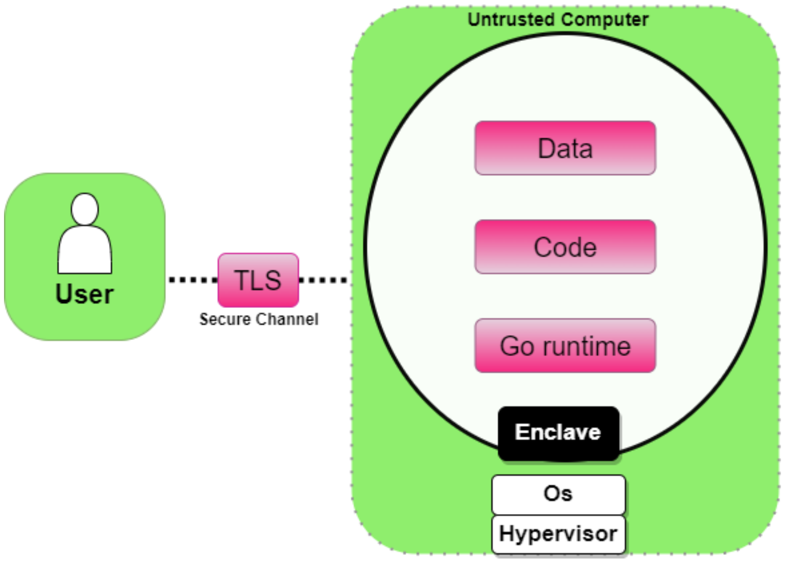


Figure 3.1: Illustration of an EGo enclave. Figure inspired from an illustration from Edgeless Systems [11]

3.3 Using EGo

Before one can begin using EGo, a few technical details need to be addressed. Thus, this section provides a description of the system requirements necessary to run an EGo application, as well as instructions for downloading and using the framework with its provided tools. While the easiest way to get started is to use a computer from a cloud provider that is already configured for SGX enclaves, this section will also briefly outline how to configure a computer for the use of EGo.

3.3.1 Prerequisites

Enabling SGX on the machine is essential for using EGo. While EGo does allow for running an application in **simulation** mode, i.e., without the need for SGX, our primary interest lies in running an application inside an Enclave with SGX. Without this capability, using EGo in the first place would be pointless.

Firstly, checking if the BIOS on the desired machine is SGX-enabled can be done as shown below. Note that this assumes the intel-CPU supports SGX which is detailed in Intel's documentation: link to Intel's website.

```
& sudo apt install cpuid
```

```
  $ cpuid | grep SGX
     SGX: Software Guard Extensions supported = true
     SGX_LC: SGX launch config supported      = true
  SGX capability (0x12/0):
     SGX1 supported                         = true
```

Listing 3.1: Terminal commands for checking if the hardware supports SGX using the **Cpuid** program.

If either of **SGX**, **SGX_LC** or **SGX1** returns `false` it means the BIOS is not configured for SGX which can be done through the BIOS.

3.3.2 Usage

Compiling a Go application to EGo can be done in a few steps. The first step is to download the DEB package through the snap on Ubuntu (version 18.04 and 20.04). The second step is to compile your go application into an EGo binary with the **ego-go build** command. The first time building the application, EGo will also create an `enclave.json` file with enclave-specific configurations. Thereafter, signing the executable can be done using the **ego sign** command which is required as every executable must have its own signature to function. The last step is to run the executable with the **ego run** command which will run the executable binary, loading it into and running it as an enclave [12]. An example of the whole sequence from downloading to executing an application written in Go is summarized as terminal commands shown below.

```
sudo snap install ego-dev --classic
ego-go build helloworld.go
ego sign helloworld
ego run helloworld
```

Listing 3.2: How to download and use the EGo SDK for an application written in Go named **helloworld.go**

The enclave-specific configurations in the enclave.json file created by the **ego-go build** command, have the fields as described in 3.3.2.

```
  {
  "exe": "helloworld",
  "key": "private.pem",
  "debug": true,
  "heapSize": 512,
    "executableHeap": false,
    "productID": 1,
    "mounts": [
        {
            "source": "/home/user",
            "target": "/data",
            "type": "hostfs",
            "readOnly": false
        },
    ],
    "files": [],
    "env": []
    }
```

Listing 3.3: Enclave specific configurations in the *enclave.json*.

Most fields are self-explanatory, therefore, we won't go through them all in much detail. An important takeaway from the enclave.json file is that it lets you include files in the **mounts** and the **files** fields. The mounts field specifies the files presented to the enclave from the host file system, while the files field defines the files embedded into the enclave binary and available in-enclave-memory filesystem. Both fields define a **source** and **target** indicating where the file is located in the host filesystem (source), and where the file can be accessed by the enclave (target). It's worth noting that the main difference between accessing files with either mounts or files is that the former allows the enclave to access the filesystem during execution, whereas the latter involves a copy of the file embedded within the enclave binary, meaning the file is not being edited on the host file system. The filesystem is in general regarded as untrusted with respect to the enclave which the developer should be aware of before allowing the enclave to access files. Other useful fields are **heapSize** which decides the amount of memory (in MB) the enclave can allocate, and productId for letting an attester distinguish between different enclaves signed with the same key.

A complete list of supplementary tooling has been itemized in the table below.

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

Table 3.1: EGo command-line arguments [12]

In addition to the commands above Ego offers a way to run an application in simulation mode with the `OE_SIMULATION=1` flag for the `run` command. This makes the application easier to debug as no data nor code gets encrypted during execution. Simulation mode is also useful when not having a CPU that supports SGX, but you still want to verify that an EGo application runs correctly. Note that simulation mode only works as long as the application doesn't rely on any SGX-specific functions such as remote attestation [12].

3.4 Attestation with EGo

EGo facilitates both local and remote attestation by providing Go packages for easier implementation of these features [13]. Remote attestation in general relies on external SGX services, which include a *Quote provider* that connects to Intel's Provisioning Certificate Caching Service (PCCS). With regards to EGo, the Quote Provider connects the enclave to the PCCS. This allows for remote attestation to be issued from a client, ensuring the authenticity and integrity of the enclave. Cloud providers like Microsoft Azure operate their own PCCSs, which simplifies the usage of remote attestation. However, if the application is to be run on-premise, the developer must host their own PCCS. To address this, Edgeless Systems has created a Docker image that can be pulled and run locally as a container to host a PCCS, provided that the user has an API key from Intel's PCK Certificate [14].

Local attestation is a term used by Intel SGX that describes how two enclaves can communicate through a secure channel after attesting each other's integrity. It does, however, require both enclaves to run on the same host. By utilizing two packages created for EGo called *Enclave* and *Eclient* [13], local attestation between two EGo-enclaves can be implemented by taking advantage of the functionalities predefined in the packages.

3.5 MarbleRun

Microservice architectures have become an important and popular way of distributing applications for a variety of industries. Thus, begs the question of how an EGo application easily can be handled in a distributed and confidential manner. Edgeless systems' solution to this is *Marblerun*; a framework that works as a complement to EGo. Marblerun handles the whole distributed architecture on a Kubernetes cluster, enabled with intel-SGX. Marblerun is then able to verify the integrity of the services, as well as set up encrypted connections between them [15]. Although our implementation does not rely on Marblerun, it is worth mentioning for context and practical knowledge to reveal how EGo can work in a distributed system.

3.6 Limitations

EGo as a framework is designed to be as close to Go as possible. It gives users an easy way of transporting their applications for use in confidential computing. The developers of EGo suspect that it is likely that most apps in the future will run most of the code inside enclaves, especially considering how the world is moving their applications to the cloud [16]. This was one of the reasons for EGo's origin; it reduces the time developers have to port or refactor their code to C/C++ with the SGX SDK. EGo compromises being user-friendly with the control a developer gets over what is computed inside and outside an enclave. Performance has, therefore, the potential of being lost by context switches, i.e. switching between trusted and untrusted environments. This will be addressed later in the thesis.

Booting an EGo application also comes at the cost of time as EGo has to load the whole binary into the enclave, i.e., the larger the application is, the more time it takes to initiate it. A work using EGo, Porambage *et. al* [17] showed a difference of slightly over 8s to boot a process compared to Intel SGX in C++. Using EGo in such a manner that requires the application to often restart, would therefore be very inefficient.

Chapter 4

Related Work

In this section, we investigate works regarding TEE, works around the EGo framework or combinations of the two. ChainBox is to some degree inspired by Hyperledger fabric's [2] in terms of the overall design, which is also taken into account in part of the thesis.

Hyperledger Fabric Fabric is an open-source system for permissioned blockchain technology aiming at flexibility, scalability and confidentiality for distributed applications. It is hosted by the Linux foundation and is one of few blockchain systems that run distributed applications written in standard, general-purpose programming languages [2]. Fabric uses a *execute-order-validate* architecture in which the three steps can be run on different entities in the system. The execution step means that multiple peers execute a smart contract which implies also checking its correctness, thereby endorsing it. The transaction computed by the smart contract then gets sent to the ordering service i.e., the component responsible for the consistency of the immutable ledger. Lastly, a block is created from all transactions in the ordering service that get sent to all peer nodes. The peers will then append the validated transactions to their local copy of the ledger preventing inconsistency due to concurrency.

Secured Routines: Language-based Construction of Trusted Execution Environments Ghosn et al. (2019) [3] introduced a language-based construction of TEEs as an approach to fully integrate trusted execution by allowing *goroutines* (user-level abstraction of threads specific to the go programming language) to execute within an enclave. Channels are used to communicate between untrusted and trusted environments. The compiler and runtime called GOTEE is an extension of the Go language and serves the purpose of automatically extracting secure code and data, necessary to run the enclave. Function calls to an enclave from untrusted code are accomplished by running secure goroutines. Secure routines are run by the *gosecure* command, a keyword which is an extension of Go's standard go routine executed by the *go* command. Only a single annotation is therefore needed to distinguish trusted and untrusted execution. Through efficient compiler-driven code and data partitioning, GOTEE achieves up to 5.2x throughput and a 2.3x latency over the intel SGX SDK. Because of this improvement over the SGX SDK, the developers claimed the most effective use of TEE in general, is to have it execute only trusted operations while running the remaining part of the application outside of the enclave.

Go Language support in Hyperledger Fabric Private Chainode A work by Riccardo Zappoli [18] shares the interest in extending the language support for running smart contracts in TEEs with SGX, just like **Secured routines**. This is similar and partly the purpose of our reimplementation of ChainBox using EGo. This study mainly consists of continuing Hyperledger Fabric Private Chaincode (FPC), a program that enables execution of *chaincodes* (smart contracts specific to Hyperledger Fabric) using intel SGX for Hyperledger Fabric [19]. The reasoning for their study was to allow for Go-written chaincodes since FPC only allowed for C/C++-written ones due to compatibility requirements for SGX. A Go Chaincode Package was created and intended to be used and integrated into FPC. Like our implementation, they used the EGo SDK, which we will further discuss in ch. 7.

Chapter 5

Implementation

In this chapter, we will explain how our system is built component by component. How and where data flows through our program is also presented, giving further insight into the reimplementation. The combination of the Wasmer WebAssembly runtime (Wasmer-go), attestation, and SGX are key features in making the system secure. Thus, we will discuss how these technologies and mechanisms work together with EGo constructing a trusted execution environment.

5.1 Proposed Solution

Our product is a reimplementation of ChainBox, [1], using Golang with its extended framework EGo. As ChainBox, our system consists of a runtime, an ordering service, and a blockchain. Our solution also realizes blockchain on a single-edge device and relies on Intel SGX indirectly through EGo to ensure the program's integrity. Since our reimplementation is based on EGo, which is a more straightforward way of constructing Intel SGX applications than the Intel SGX SDK, our solution is just as, or even more, feasible than ChainBox. The system is specifically built to withstand a non-secure server side and provides vendors/users with a confidential log-based platform.

5.2 Implementation

We make the same assumptions as ChainBox when it comes to the design of the application. Firstly, we assume a consortium of independent vendors (stakeholders) taking part in the field with physical devices or software. Secondly, we enable the utilization of an immutable ledger to allow devices to generate data and perform operations, as well as consume data produced by other vendors. Thus, ensuring consistency among all vendors, meaning all users can access the same data. Thirdly, our **Threat model** seeks to specifically handle the following attempts to take advantage of the system:

1. The system is located on an untrusted field.
2. Vendors want to retract recorded data to gain an advantage over other vendors
3. Vendors may want to present conflicting information to other vendors.
4. The system administrator may want to suppress data or recordings from the vendors.

5.2.1 Architecture

In figure 5.1 you can see the overall architecture of our reimplementation. It consists of three parts, the runtime ①, ordering service ②, and blockchain ③. The green zones represent trusted environments, i.e., enclaves, contrary to untrusted grey zones. The figure also displays a blue zone, which represents a Wasmer-go runtime, running inside the trusted environment. The red arrows between the client and runtime, and runtime to the ordering service show secure connections while the dotted black shows write operations to disk. This architecture is equivalent to the architecture in ChainBox.

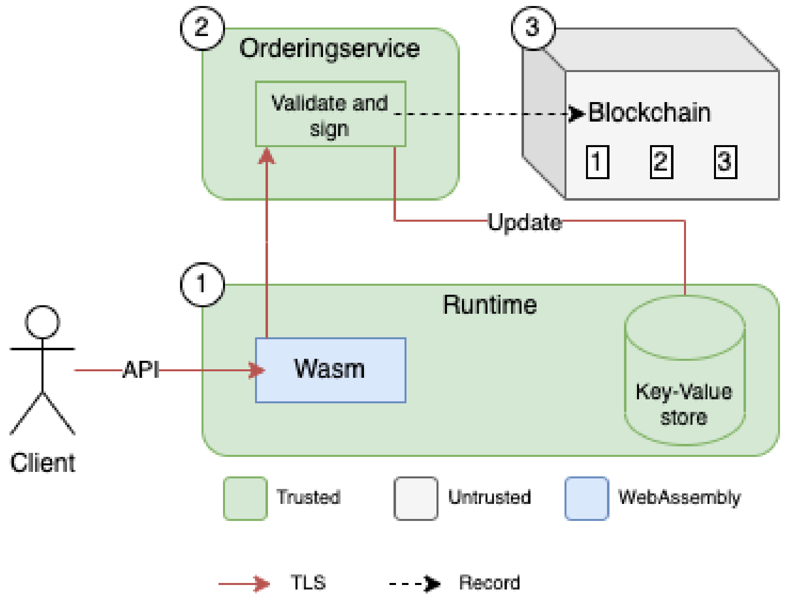


Figure 5.1: Application Architecture

We have chosen an asynchronous model for communication between the runtime and the ordering service. Through local attestation, as described in ch. 3, secure and encrypted communication is obtained between the runtime and the ordering service. The process of local attestation in our implementation is inspired by an example from the developers' GitHub repository [20]. Only after a mutual TLS connection is established in the form of a WebSocket, the runtime can begin handling, respectively, transactions from the clients and created blocks from the ordering service. The rationale behind this method is twofold: first, it allows for faster and more efficient asynchronous communication, and second, it ensures that all runtimes are notified promptly when a new block is generated, necessitating an open connection between the two components.

5.2.2 Runtime

The main task of the runtime is to act as an intermediate joint between the client and the ordering service that executes smart contracts. As we observe point ① in figure 5.1, this is where the Wasmer-go runtime and key-value store are configured.

The runtime listens on a port for API calls from the client and processes the request depending on the endpoints listed in table 5.1.

INIT	Initiate a specific client
UPLOAD	Upload a single smart contract
SET	Initiate specified smart contract with a key-value pair

Table 5.1: Runtime API calls

Currently, the Wasmer-go runtime is simplified to accept just one type of smart contract. This smart contract simply increments the value of a transaction by one. When a transaction is processed by the Wasmer-go runtime, it is sent to the ordering service through a secure WebSocket connection. We discuss the use of these connections in more detail in sec. 5.2.5.

As mentioned, the runtime is persistent by storing transactions on disk. They are specifically stored in a **.store** file, which is mounted in the runtimes enclave.json file. This in itself does not store the file in a secure way and is why we use a `seal` function from EGo's *ecrypto* package. This function encrypts plaintext within the file with the signer- and productid of the enclave.

Since one big focus in ChainBox is making cheap on-site deployment feasible, storing transactions both on runtimes and in the blockchain seems unnecessary and only consumes resources. This is mainly implemented because this thesis focus is the reimplementation of ChainBox.

5.2.3 Ordering service

The main task of the ordering service, point ② in figure 5.1, is logging the Wasmer-go generated transactions from the runtime to the blockchain. It is also responsible for updating the connected runtimes when a new block is added to the blockchain.

The ordering service writes to the blockchain using the *mount* configuration in enclave.json. The ordering service is programmed to create a new block for a fixed size of transactions. When it receives transactions from the runtime, it stores them temporarily until the number of transactions stored, i.e., a transaction counter is equal to the block size. To make the ordering service handle incoming transactions concurrently, the routine (lightweight CPU thread for Go) locks around the list containing all transactions to prevent race conditions for the other running routines.

The ordering service responds to the runtimes in two different ways, whenever a new transaction is stored or when a new block is created. An explanation of how the two enclaves communicate will be elaborated later in sec. 5.2.5.

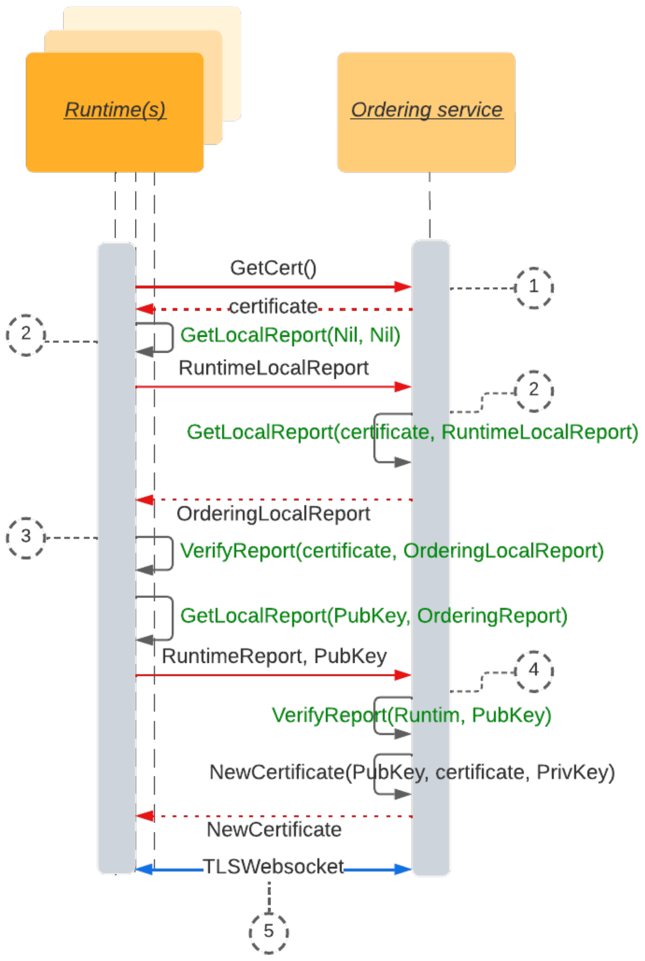
5.2.4 Blockchain

Our blockchain storage is currently, also as ChainBox states, assumed public knowledge. This is a simplified immutable ledger where each block is stored in JSON-format sequentially on disk. The block size varies, depending on the configuration of the ordering service. When the ordering service is initiated, a genesis block is created if there are no blocks (files) in the filesystem beforehand. The genesis block does not contain any transaction data, only the genesis hash and time of creation. Each block, excluding the genesis block, contains multiple data attributes. It contains a timestamp of creation, a unique hash with the corresponding previous blocks' hash, and the transactions themselves.

5.2.5 Sequence Diagrams and Data Flow

A clear explanation for attesting the integrity, both ways, between the runtime and ordering service enclave resolving in the aforementioned secure WebSocket connection will be provided in this section. Additionally, we will show how a transaction (i.e., smart contract execution) gets handled in every component within our implementation.

For the previously mentioned local attestation, two Go packages designed for EGo have been incorporated, called *Enclave* and *Eclient*. These packages provide handling so-called *Reports*, which includes important specifications and functionality [13]. In the figure below, the green text represents functions used from the forenamed Go packages.



Runtime(s)
Ordering service
GetCert()
certificate
GetLocalReport(Nil, Nil)
RuntimeLocalReport
GetLocalReport(certificate, RuntimeLocalReport)
OrderingLocalReport
VerifyReport(certificate, OrderingLocalReport)
GetLocalReport(PubKey, OrderingReport)
RuntimeReport, PubKey
VerifyReport(Runtim, PubKey)
NewCertificate(PubKey, certificate, PrivKey)
NewCertificate
TLSWebsocket
1
2
2
3
4
5

Figure 5.2: Sequence diagram showing the data flow between the runtime enclave and the ordering service enclave during local attestation, ending in a secure (TLS) WebSocket. Both enclaves verify each other *Report*-data by communicating *certificates* and a *public key*.

At initialization of the runtime, it retrieves a *certificate* generated by the ordering service using the *x509* format, together with a belonging private key ①. The next steps mainly consist of both enclaves generating reports ②, and thereafter verifying them with the other enclaves' report data ③ together with the initial certificate. Next, the runtime generates a *public key* which is sent and used by the ordering service to generate a *New certificate* ④. Lastly, the new certificate together with the private key is used for the TLS connection within the WebSocket ⑤.

After a secure connection between the enclaves is obtained, and a client has initiated and uploaded the smart contract (described in table 5.1), a connected client can execute the smart contract by using the **SET** API call with a key-value pair.

Diagram 5.3 illustrates a round-trip for the execution of a smart contract, i.e., how a transaction is handled for every individual component. The round-trip resolves with an acknowledgement to the initial client. *HTTP Long Polling* for the client was configured to achieve this.

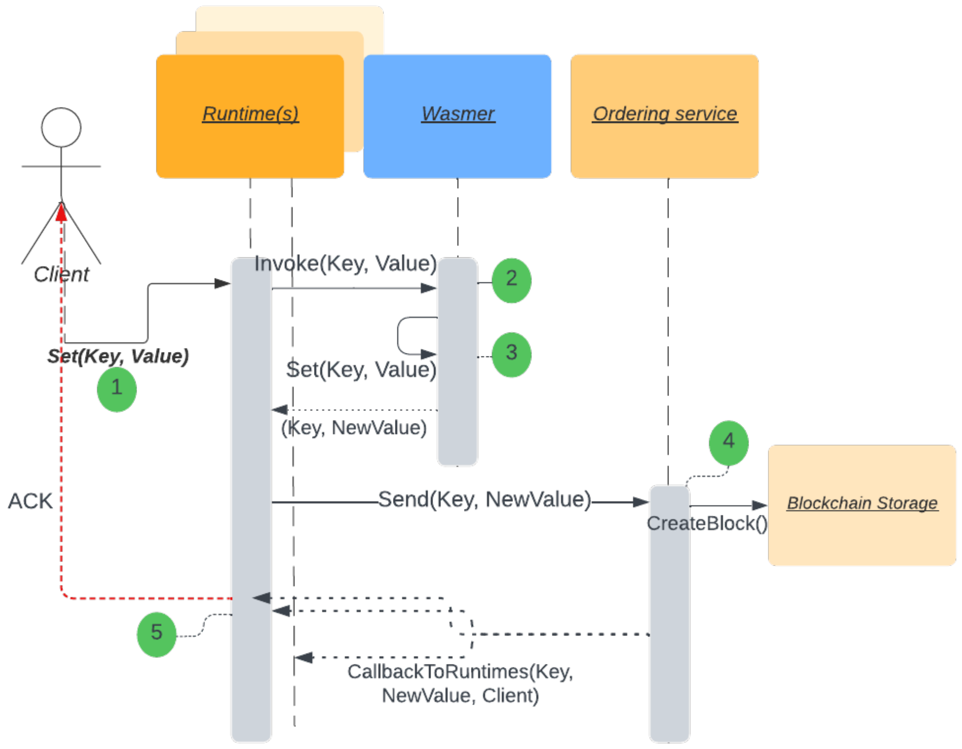


Figure 5.3: Data flow through every component during a round-trip of a single transaction.

A connected client can execute its uploaded WASM module using the **SET** API call which includes a key and value ①. Other data types besides integers will be rejected by the runtime, resulting in an error. The runtime will, subsequently, find the client's Wasmer-go instance, and call the exported function defined in the uploaded WASM module, hence, invoking the function ②. The Wasmer-go runtime will then execute the smart contract, which includes calling another function from within the sandboxed environment using the key-value pair as parameters. The resulting **Key** and **New value** will be returned to the runtime ③. Continuing the transportation of the created key-value pair; the created key-value pair is sent from the runtime to the ordering service in which a new block gets created if the amount of transactions exceeds the block size as described in sec. 5.2.3 (④). Once a block is created, the ordering service will distribute all transactions to all connected runtimes. Each runtime will store the transactions within their .store file as mentioned in sec. 5.2.2. Finally, an acknowledgement will be returned to the initial client ⑤. Note that an acknowledgement is still sent to the runtime if there is no created block, and thereafter to the initial client from the runtime.

Chapter 6

Experimental Evaluation

With our reimplementation, we wanted to simplify and cover all of the features ChainBox offers, while also preserving the same level of security. However, achieving the aforementioned goals simultaneously comes with a performance cost. This is due to EGo's programming model regarding the context switches as explained in sec. 3.6. In that regard, three experiments have been conducted, showing the system's performance.

We measured the total round-trip latency for the system when sending a transaction i.e., executing a single smart contract. Measuring the round-trip in this context implies measuring the latency for a transaction to get handled by, firstly, the Runtime (including the Wasmer-go runtime), and secondly, it involves the processing time within the ordering service. Subsequently, an acknowledgement is returned to the runtime before it is finally sent back to the initial client, completing the round-trip. A visualization of the round-trip can be found in sec. 5.2.5. Average total time and throughput measurements were calculated for every experiment as well.

All data points were obtained by sending a total of 5000 transactions per experiment, with three repetitions. For each experiment, we increased the number of transactions sent concurrently e.g., two transactions sent concurrently imply sending a total of 5000 transactions where two and two are sent concurrently. The block size for all experiments is 20, except for the first in which the block size varies.

We ran all our experiments on Microsoft Azure. The virtual machine comes with one physical core, 8 GB of memory, and 75 GiB of SSD memory. It's worth noting that all transactions from the experiments were sent from the same machine our application was hosted on, reducing the available recourses for the application.

Monitoring the virtual machine, displayed in figure 6.1, as we conduct these experiments shows that average CPU utilization lies around 55-65% independent of block size and concurrent transactions sent.

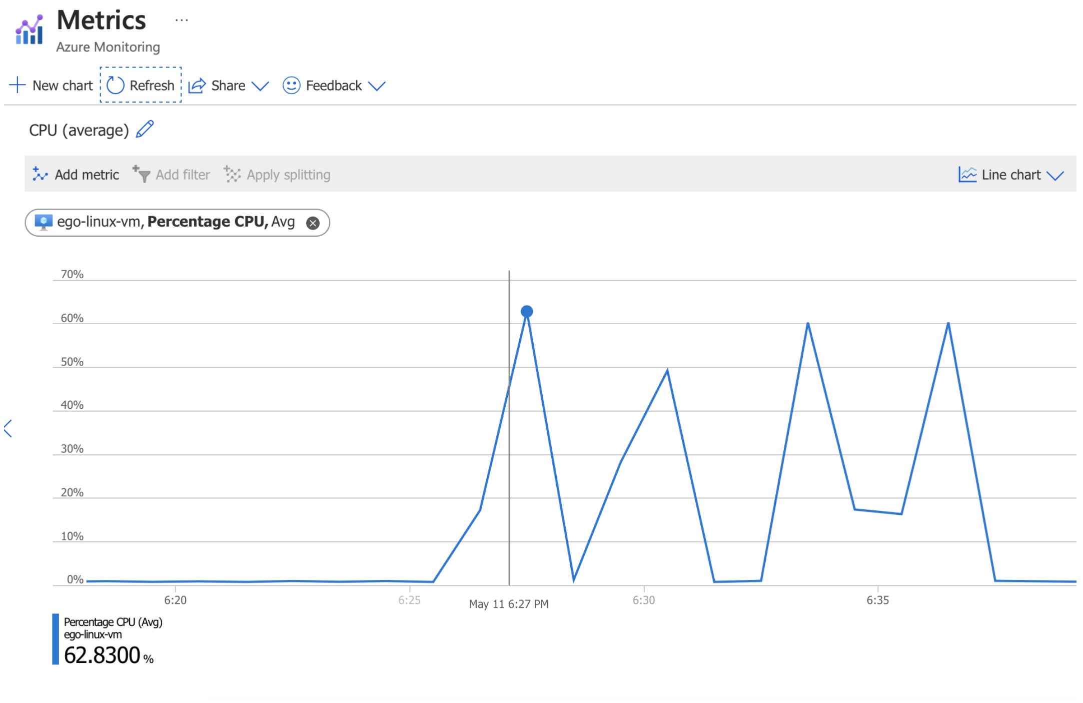


Figure 6.1: Azure VM metrics - Avg. CPU usage (%)

6.1 Testing EGo

In this experiment, displayed in 6.1 and 6.2, we increase the block size from 10 to 40 to detect if any significant change in the data occurs.

Requests sent concurrently	Total time (s)	latency (μs)	Throughput
1	37.10	7294	134.77
2	36.73	14512	136.13
4	36.77	28910	135.98
8	36.34	56579	137.59
16	35.97	111200	139.00

Table 6.1: Results using EGo with a blocksize of **10**

The table above shows the data points when the ordering service is configured with 10 transactions per block. If we observe the latency, we can see a doubling when the concurrent requests are increased. The reason for this is the list that stores the transactions temporarily in the ordering service. As mentioned in ch. 5.2.3, the transaction storage list is locked when the ordering service receives transactions. Incoming transactions thus have to wait before appending to the list, specifically, during `Send(Key, NewValue)` and `CreateBlock()`, in figure 5.3. This causes extra latency per transaction, and even more latency every time the list exceeds the block size because of block creation by the ordering service.

In table 6.2, the block size is increased to 40 transactions per block.

Requests sent concurrently	Total time (s)	latency (μs)	Throughput
1	36.07	7089	138.62
2	36.52	14407	136.91
4	36.21	28307	138.08
8	35.76	55268	139.82
16	35.37	108960	141.36

Table 6.2: Results using EGo with a blocksize of **40**

If we calculate the average throughput for all requests sent concurrently in respectively table 6.1 and table 6.2, we observe a 1.66% decrease. This suggests that the difference in block size imposes a marginal difference in throughput which may be because of fewer block creations, i.e., fewer write operations to disk. We still however see roughly, a doubling in latency for each doubling of concurrent requests.

6.2 EGo vs. Go

In this experiment, we examined our application running with EGo and Go. Compared to previous experiments, we removed local attestation between the runtime and ordering service because of incompatibility with Go. We further simplified the TLS connection between the runtime and ordering service, and furthermore, removed it completely from the client to the runtime to simplify the experiment. Keep in mind that the results reflect this change compared to the experiment conducted in 6.1 in which the TLS connections remained unchanged.

Requests sent concurrently	Total time (s)	latency (µs)	Avg. Throughput
1	5.15	1007.99	970.87
2	4.47	1751.85	1118.57
4	4.37	3428.95	1144.16

Table 6.3: Results using EGo

Requests sent concurrently	Total time (s)	latency (µs)	Throughput
1	1.68	318.17	2976.19
2	1.57	602.28	3184.71
4	1.56	1079.89	3205.13

Table 6.4: Results using Go

The comparison shows that Go in general performs roughly three times faster in terms of the **Total time**, **Average latency**, and **Average throughput**. The results demonstrate a slight decrease in total time and a slight increase in throughput for both EGo and Go when increasing the requests sent concurrently.

To further demonstrate the difference between using EGo compared to Go, a histogram is shown in figure 6.2, using the data sets as the first row in table 6.3 and 6.4.

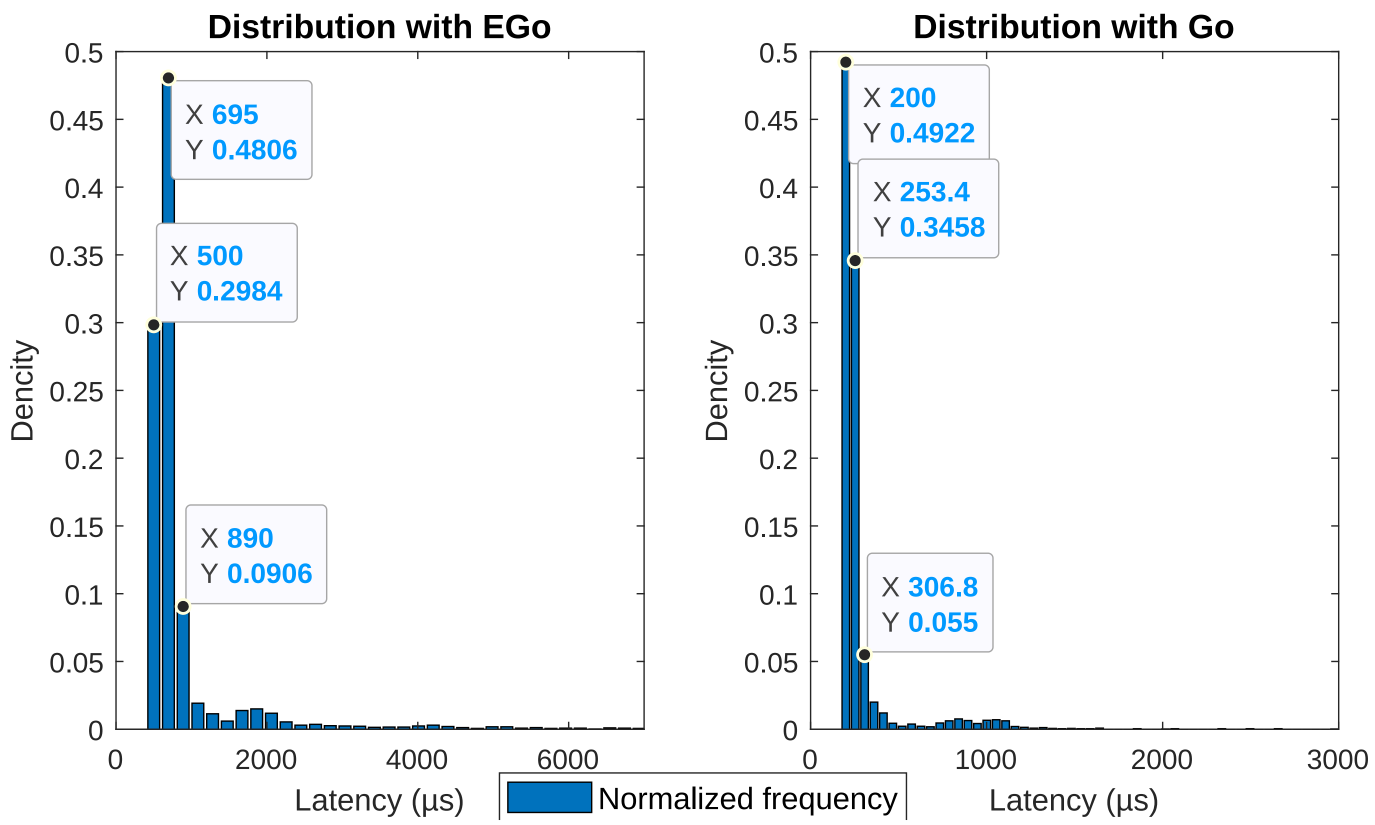


Figure 6.2: Distribution of latencies for EGo and Go sending all requests sequentially.

We can observe the difference in the overall distribution of latencies for EGo and Go for sending all requests sequentially (i.e., 1 request sent concurrently from table 6.3 and 6.4). Go shows a much less distribution of latencies, varying from roughly 200 to 1500 µs while EGo shows ranges from about 500 to 6000 µs. The wide distribution of latencies for EGo proves how it operates much less consistently than Go. Additionally, the top coordinates marked in both histograms indicate that Go performs almost 3.5 (695/200) times better than EGo, 48-49% of the time.

6.3 Mulitple Runtimes

We also wanted to observe our system using multiple runtimes at the same time. Multiple runtimes basically mean concurrent requests, but each transaction is sent from different runtimes. We can clearly see, in table 6.5, that the results follow the almost identical pattern as the other tests. This is because of the same lock as discussed in 6.1 in which the lock blocks other routines from appending to the transaction list.

Runtimes	Total time (s)	latency (µs)	Throughput
1	37.32	7220.30	133.98
2	38.59	15237.33	129.57
4	38.67	30346.00	129.30

Table 6.5: Results using 1, 2, 4 runtimes

Chapter 7

Discussion

This thesis serves two purposes, namely to evaluate EGo and to reimplement ChainBox using EGo. We will, accordingly, in this chapter discuss EGo as a framework for confidential computing by focusing on its architecture, user-friendliness, and documentation. Subsequently, we elaborate on certain aspects of our implementation both in and outside the context of EGo.

7.1 Discussion of working with EGo

The decision to reimplement ChainBox using EGo proved to be advantageous in terms of ease of implementation since the components i.e., ordering service and runtime, were almost built as if they were to be written and run with the standard Go compiler. Only a few packages i.e, *Eclient*, *Ecrypto* and *Enclave* from the Go package designed for EGo [13] were used. In our experience, porting code from Go to EGo and vice-versa requires minimal refactoring since using the latter typically requires only a few additional packages such as the aforementioned ones. For example, for attesting to and from an enclave, the Enclave and Eclient packages are typically used, while the Ecrypto package revolves around sealing and unsealing files to persist encrypted data to disk.

Moving on to another subject, the challenge of optimizing the code to run efficiently within an EGo enclave goes beyond the capability of the developer because of the transparency with the context switches related to SGX. As a result, while our reimplementation of ChainBox using EGo may be optimal for the problem at hand, it may also potentially be suboptimal without our knowledge. This highlights the tradeoff between the user-friendliness of EGo and the control the developer has over the execution of the code inside and outside of the enclave, as mentioned in ch. 3. It is therefore important for developers to carefully consider the potential performance implications of using EGo. Moreover, documentation detailing how EGo works under the hood, especially with regard to how it works with SGX is not something the developers have provided as far as we can tell. This aspect diminishes the appeal of the framework for developers who possess expertise in tools such as the Intel SGX SDK.

Continuing the exploration in EGo; we found EGo's documentation for getting started with it pretty straightforward, but this was only the case after getting access to an Intel SGX-compatible machine that had the required updated kernel. An attempt to run EGo on an on-premise server did not work, following a number of error messages due to the outdated kernel. We sufficed to use a pre-configured virtual machine image from Microsoft Azure which was much more convenient in our experience. On the former, the developers for EGo, Edgeless Systems, were very helpful in interpreting various error messages. However, it would have been convenient to see a supplementary system requirements list in the documentation.

Studies like *Secure routines* [3] and *Go Language Support in Hyperledger* [18] as mentioned in ch. 4 both integrate trusted execution in a way that doesn't require an application as a whole to be run inside an enclave. Similarly, we could have looked to rearchitected ChainBox in which the most vulnerable code was handled by EGo. It's hard to say if this would have outperformed our implementation as several secure channels would be required to communicate back and forth between the secure EGo enclave. One could also argue that it would have taken away the purpose of using EGo considering how it is meant to run an application as a whole within an enclave, implying that it also should be optimized for that purpose. This approach however stands in contrast to the findings with Secure routines where they claim that the best approach is to have the TEE only execute trusted operations as described in ch. 4. This raises the question of whether we will see new frameworks following the approach of Secure Routines, making the EGo framework obsolete in comparison.

7.2 Discussion of the implementation

The first detail worth addressing is with regard to the local attestation between the runtime and ordering service as this assumes the two enclaves are running on the same host. Having the option of running each enclave on separate hosts would be beneficial to allow scaling out our system with several vendors (runtimes), without taking away valuable resources from the ordering service. The reasons for sufficing to local attestation instead of something like remote attestation were the following:

1. Test how we could attest both ways that our enclaves were running as, specifically, EGo-created enclaves.
2. Take advantage of and thereby test the packages designed for use with EGo, i.e., the *enclave* and *eclient* package [13], giving a clearer overall impression of the framework.
3. Having a secure application that can be deployed with the requirement of only one server which lowers cost compared to Hyperledger Fabric, which requires distribution to facilitate trust [2].

As our primary focus was on working with EGo rather than developing a fully functioning blockchain, we implemented a simplified version of the blockchain architecture. The current blockchain in our application does include cryptographic hashes using *SHA-256*, with the input being the data and the previous hash, ensuring the immutability of the blockchain. It does not, however, include a signature from the ordering services' enclave like ChainBox does, which we will come back to in ch. 8. The blockchain is also simplified in terms of how the blocks are stored on the host's file system; all files, i.e., the blocks, are named using *Unix time* (in nanoseconds). Consequently, all blocks are stored sequentially in the filesystem due to the slight increase in time for each created file. This made the re-loading of all blocks into the ordering service enclave effortless since we could loop through the folder where the blocks were located. A finished product with our implementation would, therefore, be in need of a much more comprehensive and efficient blockchain implementation in terms of signatures and how the blocks are stored, than the current one.

Both the runtime and ordering service, handling the storage file (earlier referred to as the .store file) and the blockchain respectively, read and writes files to and from the host system. Once again, context switches come into play in which the enclave needs to perform system calls to the host operating system, leading to additional overhead for the application as a whole. This may be the reason for the 1.66% decrease in average throughput when increasing the block size from 10 to 40 in our first experiment in ch. 6. As blocks get created, based on a predefined block size constant, and the runtime also stores all created blocks to file while simultaneously decrypting the storage file, the application ends up using a lot of resources. Some of which may also be unnecessary as the storage file serves no purpose, as described in sec. 5.2.2, for our implementation. Thus, removing how the runtimes store all blocks, may be a feature to be removed in the future, unless it appears to be required for a refined version of our implementation.

Chapter 8

Future Work and Conclusion

Our implementation started to look something like ChainBox, but there are still some shortcomings. In that context, we will in this chapter present the most crucial missing features our reimplementation ought to be extended with before it could be considered finished, before finally concluding the paper by summarizing our key findings.

8.1 Future development

The connection between the clients and runtimes is currently not as secure as they potentially can be. Clients connect to the runtime with the enclave's *UniqueID*. The UniqueID is subsequently used to verify the integrity of the enclave. However, this is a simplified version of a Remote attestation that later should be included. Furthermore, a client chooses their own API key defined with the **INIT** API call, which can be any sort of string. A solution could be to implement predefined API keys for the runtime and clients instead.

Unlike ChainBox, the blockchain lacks a signature from the ordering services' enclave, hence the blockchain is never verified for its origin from the ordering enclave. Thus, re-loading a blockchain into the ordering service from the host's filesystem happens with no verification to ensure that it was initially created by the ordering service enclave.

In our current implementation of the ordering service, each block is only created when it has received a certain amount of transactions. This means that some transactions would be stuck if suddenly no more transactions were sent. This could easily be fixed with a timeout where a new block is created after, e.g., 5 seconds after the last received transaction.

ChainBox's WebAssembly runtime has more functionality than ours. For example, *Get*, *Register*, and, *Free* is currently not options in our runtime and should be implemented in future versions of the system.

Our webassembly runtime is executed inside an enclave, and also in a sandboxed environment. This makes integrity and confidentiality high, but our program does not check if the uploaded Wasmer-go module might drain the host system for resources. This has the potential of being extremely costly if the program was run on a cloud platform. One solution to this would be spawning a process solely for the purpose of running the Wasmer-go runtime and capping the resources for that single process. However, EGo does not support spawning new processes. A fix to this should be implemented in future versions.

Marblerun is another feature Edgeless Systems has developed, mentioned in ch. 3. Deploying our product with Marblerun as a Kubernetes cluster can potentially be much more efficient, especially if the program is intended to run on a cloud platform. Each node - the runtime, ordering service, and blockchain - would be run as different microservices. This would increase the elasticity and scalability of the program as a whole.

8.2 Conclusion

Our main focus in this thesis is researching EGo and reimplementing ChainBox as securely and efficiently as possible. We show that EGo is not far from plug and play, however, it has its limitations mainly when it comes to performance. Since EGo runs the entire program in one single enclave, it makes the development process much easier but can impose limitations on the developer's ability to customize and optimize the program to align with its intended use case. On the other hand, it means that the development becomes much easier than with other frameworks such as with the Intel SGX SDK. Thus, EGo is a type of framework that trades complexity for simplicity, sacrificing some optimization opportunities in favour of user-friendliness. When it comes to the documentation for EGos underlying architecture, we found it to be insufficient which undermines its trustworthiness. However, the documentation for using the framework was straightforward, especially when using an SGX pre-configured virtual machine

(VM). With our reimplementation of ChainBox, we found the throughput to be about 3 to 3.5 times worse than with the standard Go compiler. Although the aforementioned result was found with an experiment using simplified smart contracts and unsecured connections, it suggests that EGo may be feasible and used as a substitute for the SGX SDK.

Appendix A

Instructions to Compile and Run System

Refer to ch. 3 to see the system requirements and tutorial for downloading *EGo*. Pull our public GitHub repository here.

Firstly, two packages need to be installed for the attestation to work and can be installed as followed;

- `sudo apt install libssl-dev`
- `sudo ego install az-dcap-client` - Only if the system is hosted on Microsoft Azure (Recommended)

If the system is running on-premise, you have to apply for an Intel PCCS API key and host your own PCCS. Download EGo's docker PCCS image using this command;

```
docker run -e APIKEY=<your-API-key> -p 8081:8081
--name pccs -d ghcr.io/edgelesssys/pccs
```

Secondly, the go-environment needs some environment variables to be set before building the executables. This is done with the following two commands;

- $go\ \ env\ \ -w\ \ CGO\_CFLAGS = I/opt/ego/include$
- $go\ \ env\ \ -w\ \ CGO\_LDFLAGS = -L/opt/ego/lib$

The next step is to figure out which ports you want our system to run on;

1. Unsecure and secure ports in the ordering service's server connecting with the runtime (Orderingservice/main.go - line 135 and 151). Blocksize can also be specified on line 28.
2. Then insert the ports from the ordering service into runtime/main.go (lines 146 and 147) and insert the port you want the runtime server to run on (line 223)
3. Then you need to insert the runtime ports in the client (client/main.go - lines 32, 33, and 34)

Both `orderingservice/enclave.json` and `orderingservice/enclave.json` need their mounts updated, in `source` simply enter the directory you want your enclave to have access to. The enclave also needs the `Target` folder where it can write to.

Our program consists of two different parts that need to be **Built** and **Signed**.

1. `./buildordering` - Bash-script that builds, signs, then runs the ordering service
2. `./buildruntime` - Bash-script that builds, signs, then runs the runtime

The runtime should now have printed the *UniqueID* and established a connection to the orderingservice. Moving forward, the UniqueID now needs to be inserted into line 39 in client/main.go.

Everything should now be set up, and the final step is the following commands while in the client folder;

1. initiate a new client by writing the following command `go run main.go INIT <API-key>`
2. Upload a smart contract by writing `go run main.go UPLOAD <path to WASM module> <API-key>`

Transactions can now be executed by using `go run main.go SET <key> <value> <API-key>` and new blocks should appear in folder orderingservice/files/blockFiles whenever the amount of transactions exceeds blocksize.

More information regarding run and compile instructions can be found in our `README.md` file in our repository.

Bibliography

[1] Mahhouk M. Almstedt L. Jehl L. Bleeke, K. and R Kapitza. Chainbox: Using tees and webassembly to run smart contracts on the edge. 2022.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Adrien Ghosn, James R. Larus, and Edouard Bugnion. Secured routines: Language-based construction of trusted execution environments. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 571–586, Renton, WA, July 2019. USENIX Association.

[4] Corinne Bernstein. The usage of tee's, Accessed 2023. The usage of TEE's.

[5] Intel. Build an intel® software guard extensions ecdsa attestation service to strengthen enclave security, Accessed 2023. [Intel SGX].

[6] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: Analyzing the performance of webassembly vs. native code. In *USENIX Annual Technical Conference*, pages 107–120, 2019.

[7] MegaEasy. Extend backend application with webassembly, 2021. [WebAssembly architecture].

[8] Github repository. wasmer-go. `https://github.com/wasmerio/wasmer-go`, 2021.

[9] Github repository. Wasmer github repository. https://github.com/wasmerio/wasmer/#-language-integrations, 2021.

[10] MDN Web Docs. Wasm-instance documentation. https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Instance, Feb, 23. 2023.

[11] Edgeless systems. https://www.edgeless.systems/. Accessed: April 16, 2023.

[12] Edgeless Systems: EGO. https://docs.edgeless.systems/ego/. Accessed: March 16, 2023.

[13] Edgeless Systems. Ego enclave framework documentation. https://pkg.go.dev/github.com/edgelesssys/ego@v1.2.0, 2021. Accessed: April 17, 2023.

[14] Intel Corporation. Intel(R) Platform Certificate Service API - PCS Certificate v3. https://api.portal.trustedservices.intel.com/documentation#pcs-certificate-v3, 2021. [Accessed: March 15, 2023].

[15] Edgeless Systems. Marblerun documentation. https://docs.edgeless.systems/marblerun/, Accessed: 2023.

[16] Edgeless Systems. How we built ego. *Edgeless Systems Blog*, February 2022.

[17] Pawani Porambage, Yushan Siriwardana, Roshan Sedar, Charalampos Kalalas, Wissem Soussi, Huu Nghia Nguyen, Edgardo Montes de Oca, Vincent Lefebvre, Gianni Santinelli, Juan Carlos Caja, Antonio Pastor, Chafika Benzaid, Othmane Hireche, Yongchao Dang, Tarik Taleb, Geoffroy Chollon, Maria Christopoulou, Pablo Fernández, and Alejandro Molina Zarca. 5g security: New breed of enablers. Technical report, INSPIRE-5Gplus Project, March 2023.

[18] Riccardo Zappoli. Go language support in hyperledger fabric private chaincode. 2022.

[19] Hyperledger Foundation. Hyperledger fabric private chaincod, Accessed on 2023-04-23.

[20] Edgeless Systems GmbH. EGo. `https://github.com/edgelesssys/ego`, 2021. Accessed: March 15, 2023.