UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Hardware-Backed Confidential Blockchain

André Filipe Agostinho da Silva Cruz

Mestrado em Engenharia Informática

Dissertação orientada por:
Professor Doutor Bernardo Luís da Silva Ferreira

2022

Agradecimentos

I would like to thank all of those that have supported me over this long and hard journey, starting by my adviser, Bernardo, who has had the patience to accompany this document from the very start. I am very grateful to my family, who has supported all of my studies thus far, and helped me in the long days it took to finish this Thesis. Finally, I would like to thank all of my close friends who have taken the steps next to me the entire time, these include, but are not limited to, André Correia, Afonso Gonçalves, Rafaela Rodrigues, Fábio Estanqueiro, Carolina Sá, André Mendes, Filipa Almendra and Rafael Ramires. These and many more supported moments of work, but also much needed fun interactions over this difficult year.

Dedico este estudo a todos os que me apoiaram e suportaram neste caminho difícil,a quem devo toda a minha educação e formação.

Resumo

Nos últimos anos, a possibilidade de execução de sistemas computacionais em ambientes de execução que não pertencem ao grupo que desenvolveu o sistema tornou-se cada vez mais popular principalmente devido à sua total disponibilidade, que permite os seus utilizadores de escalar os serviços às suas necessidades sem a necessidade dos elevados custos à partida de construir os seus próprios servidores. No entanto, a utilização de infraestruturas de terceiros na forma de ”outsourcing”implica que os donos destes ambientes de execução tenham completo controlo sobre os mesmos, que permite a possibilidade de os donos destes ambientes de execução utilizem informação sensível do sistema, como a informação de possiveis clientes para o seu próprio benefício. Os últimos anos já mostraram que este conjunto de informação guardada em ambientes Cloud se encontra visível para os donos dos mesmos, que coloca os dados privados dos diversos utilizadores em risco. Recentemente, com a passagem de computação para a Cloud com novos serviços de IaaS (Infrastructure as a service), tornou-se cada vez mais importante a exploração de possíveis protocolos para computação distribuída confidencial, visto execução ser feita em ambientes adversariais, devido à falta de controlo sobre os mesmos. Uma das principais soluções propostas para este problema seria manter a informação totalmente encriptada enquanto presente na Cloud, mas isso necessitaria de algoritmos de procura extremamente eficientes e capazes de procurar informação sobre um dataset encriptado, ou então a necessidade de total desencriptação da informação do lado do cliente. Em sistemas com grandes quantidades de informação ou grande quantidade de clientes, ambas as soluções produzem uma grande quantidade de overhead, ao ponto de abrandar um sistema deste tipo ao ponto de ser impossível a sua utilização. Outras soluções propõem a possibilidade do sistema desencriptar a informação apenas para execução de computações necessárias, mas isso iria obrigar informação sensível a estar visível, mesmo que durante curtos prazos de tempo. Isto torna-se especialmente um problema quando o grupo que desenvolve um dado sistema não é o dono da infraestrutura que o suporta. Assim, tornam-se necessárias garantias de integridade, disponibilidade, consistência forte e confidencialidade, mesmo sob a possibilidade de utilizadores maliciosos. Assim, esta tese procura encontrar uma possível solução para este problema, visto que até agora todos os sitemas BFT-SMR com garantias de confidencialidade demonstraram-se ineficientes para grande escala ou serviram apenas como protótipos de nível académico. Utilizando uma tecnologia fornecida

por parte da Intel nos seus processadores desde a 6ª Geração, Intel SGX, que permite execução fiável e isolada de código escrito por um utilizador. Utilizando esta tecnologia, esta tese procura desenvolver um protocolo que seja o mais prático possível para uso diário, usando como base o protocolo de BFT-SMR conhecido como BFT-SMaRt. Fazendo uma união entre um dos protocolos de BFT-SMR mais eficientes atualmente com um ambiente de execução fiável, na forma da tecnologia Intel SGX, surge a possibilidade de criar um sistema que disponibilize um protocolo BFT-SMR totalmente confidencial. Este documento visa demonstrar a estrutura base de tal protocolo e o desenvolvimento necessário para o mesmo, juntamente com as tecnologias usadas, avaliando também o nível de praticalidade da solução apresentada. Utilizando a Linguagem $C++$ para criar o código utilizado pelo Enclave e a Framework Java Native Interface (JNI), passa a existir a possibilidade de serem feitas chamadas nativas a código escrito utilizando a linguagem C, capaz de fazer pedidos ao Enclave , sendo assim possível passar informação fornecida pelo utilizador para o Enclave, permitindo assim a utilização de primitivas de encriptação dentro deste ambiente de execução fiável de modo a ser possível garantir a confidencialidade desta informação, visto que chaves associadas a esta tecnologia da Intel se encontram fisicamente nos processadores, o que tornaria necessário ataques físicos para revelar estas chaves. Considerando que maioria dos ataques são feitos pela rede, esta tecnologia seria capaz de minimizar este problema. Algoritmos de BFT-SMR acabam por não resolver este problema, não garantindo a confidencialidade da informação que passe por sistemas que utilizem estes algoritmos. Assim, desenvolvendo um algoritmo de BFT-SMR com garantias de confidencialidade, passa a existir a possibilidade de abstrair a dificuldade de desenvolvimento destes protocolos, poupando tempo e recursos a equipas de desenvolvimento. Para este objetivo, desenvolveu-se uma biblioteca "Java-SGX", que disponibiliza uma interface capaz de fazer chamadas nativas a código C que se encontra ligado a um Enclave, permitindo assim programadores desenvolverem apenas os seus sistemas sem este nível de preocupação, sendo apenas necessário garantir a ligação à biblioteca desenvolvida. Embora a utilização desta abstração aumente a latência envolvida com o sistema, esta biblioteca foi desenvolvida com a usabilidade em mente, permitindo o desenvolvimento de sistemas distribuídos utilizando apenas a linguagem de programação Java, omitindo a lógica de gestão de memória e ligação de bibliotecas nativas. Este tipo de privacidade torna-se possível devido à divisão feita entre lógica de negócio fiável e não fiável, na qual a lógica de negócio fiável é apenas executada dentro do Enclave, local onde as primitivas de uso possível estão bastante restritas, devido a possiveis falhas de segurança. Assim, o sistema pode apenas usar o Enclave para computações cujos cálculos envolvam garantias de segurança, como encriptação ou gestão de chaves. Assim, esta tese apresenta o algoritmo CON-BFT, que se baseia na ligação do protocolo já existente BFT-SMaRt a um Enclave Intel SGX, que será utilizado para fornecer garantias de segurança ao sistema já existente. Este algoritmo fornece criação de chaves dentro

do Enclave, e selagem de informação utilizando a chave específica do processador, perimitindo que a informação seja guardada em memória não fiável de modo seguro, só sendo possível remover esta camada de proteção dentro do Enclave. Este protocolo visa também garantir integridade da informação partilhada entre réplicas, usando encriptação no modo Galois Counter Mode (GCM) de modo a permitir a réplicas verificar a integridade da informação original durante desencriptação. Assim, fornecendo Diffie-Hellman, encriptação AES, funcionalidades de MAC e HMAC juntamente com Hashing SHA-256, a biblioteca "Java-SGX"utilizada para garantir a ligação ao Enclave e a definição destas primitivas de segurança e privacidade, foi também desenvolvida para esta tese com o objetivo de abrir possibilidades para outros sistemas que queiram usar estas primitivas. Desenvolvido com a divisão em área fiável e não fiável, torna-se possível a utilização da Java Virtual Machine (JVM) para criar um simples ficheiro header que será ligado a ficheiros .C de modo a permitir as chamadas nativas. Isto permite utilizar o enclave para adicionar integridade a todas as funcionalidades do sistema, desde simples escritas e leituras a memória, até protocolos de state transfer para passar informação para réplicas que necessitem de recuperação. Assim, esta tese foca-se na possibilidade de desenvolvimento de um novo protocolo de BFT-SMR capaz de ser extendido com qualquer lógica de negócio pretendida pela equipa de desenvolvimento, mas garantindo que as garantias de confidencialidade de informação são mantidas sem a procupação por parte da equipa de desenvolvimento, permitindo a mesma de não ter necessidade de programar em linguagens de mais baixo nível para manter o acesso a estas primitivas de segurança, Para isso, avaliaram-se tecnologias que fornecessem ambientes de execução fiável de modo a permitir execução isolada das bibliotecas desenvolvidas, permitindo esconder a informação utilizada dentro deste ambiente isolado.

Palavras-chave: Intel SGX, BFT-SMaRt, BFT-SMR, Confidencialidade, ambiente de execução fiável.

Abstract

In the last few years, distributed computing has shifted towards execution using adversarial environments, mostly due to the rising costs of owning proprietary data centers. Due to privacy concerns, confidentiality guarantees are now es- sential to distribute computation of sensitive information. As such, this thesis focuses on those demands, delivering Con-BFT, a SMR-BFT protocol with confidentiality guaran- tees, using Intel SGX primitives, which allow for trusted execution. This trusted execution also allows for equivocation prevention, which can be used to improve the underly- ing consensus protocol, creating a more efficient SMR-BFT framework, while also providing a layer of confidentiality over the stored user-driven information. This thesis focuses on a possible solution for developing such a protocol, evaluating its overall level of usability for practical everyday use, and how Intel SGX affects the development of similar systems.

Keywords: Intel SGX, BFT-SMaRt, BFT-SMR, Trusted execution environment, Confidentiality

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introduction

1.1 Motivation

In the last few years, Cloud applications have started to gain popularity at an exponential speed. As beneficial as cloud computing can be, in its ability to spread computing power to multiple machines around the world, this also opens the possibility of sensitive information being taken away by other parties, due to the computation of said information being done in adversarial computers. This is even more problematic if we consider the rising costs of owning proprietary data centers. As such, creating a cloud computing system that can protect data from outside and inside threats is essential. With more and more applications for the web executing in adversarial environments, i.e, cloud platforms, more and more major corporations have been caught in some high-profile data breaches, to the point of selling personal information to third parties [2, 14]. Any system whose objective remains to be confidential currently has few options when it comes to cloud execution, with the development of such a system becoming a necessity.

1.2 Problem Statement

BFT-SMR systems replicate user-driven data over multiple replicas, having each replica execute deterministic functions independently, guaranteeing integrity (data not being altered from its original state), availability (guarantee a system is functional to execute tasks when requested) and strong consistency (all replicas contain the same updated information at all times), even when faced with nodes that may have succumbed to byzantine faults or malicious attackers. BFT-SMR systems with confidentiality guarantees are, usually, systems that are intended as prototypes for academic study or not efficient enough to be used in commercial-scale systems[6]. Solutions using data encryption have been developed, but secret-sharing mechanisms complicate said solutions[18]. Shared secrets are required to be of the same size of the original secret the shares are trying to protect, meaning that for large secrets and a sizeable number of replicas, there might be a large sto-

rage overhead. Due to secrets being separated into shares over the network, it also means that there will be a considerable message overhead in order to rebuild a large amount of secrets. While encryption measures are efficient, these complicate searching over encrypted information stored in adversarial environments. Unfortunately, encryption schemes that are built in order to allow for searching or executing complicated computations over encrypted information will either end up leaking said information or be too inefficient for large systems, making this a complicated trade-off [12]. As such, a system that can tolerate byzantine faults and protect the confidentiality of the information whilst not sacrificing performance is essential, especially when taking into consideration the high costs of owning proprietary data centers. This means that all information that is processed is done so in machines and environments that are not under direct control of the developer, running the risk of malicious attacks or data breaches by the very owners or administrators of the data center holding the information. This thesis focuses on the possibility of developing a distributed system that will be able to fully run in adversarial environments, meeting the current demands for privacy and confidentiality of information. By using Intel SGX, each node will be able to deploy its own enclave, creating a safe location for all sensitive computation to be executed. Having this computation isolated, it becomes possible to obfuscate what data and commands are processed inside the trusted execution environment, which allows data to be protected even outside the isolated environment by using the processor's secret and unique key. This protection over the information makes its unsealing impossible unless done by the same trusted execution environment, meaning that stealing the information becomes useless. This is extremely important since it can be used to safeguard against malicious attackers or even data breaches by the proprietaries of the adversarial environment where the system is executed. By using Intel SGX, and having each node deploy its own enclave, where all sensitive computation will be executed, it is possible to obfuscate what data and commands are processed inside this secure area, safely storing the processed information using the processor's secret unique key, in order to safeguard against both malicious attackers and data breaches by large enterprises that may control the adversarial environment where the system is being executed, making possibly stolen information useless.

1.3 Objectives

This thesis was developed with the objective of developing a new confidential BFT-SMR framework based on distributed network of Intel SGX enclaves. For this, a popular and known non-confidential BFT-SMR framework known as BFT-SMaRt [6] will be used. Another objective would be integrity guarantees for all messages, by having an Intel SGX enclave run integrity checks on all of them. To develop, test and evaluate a functioning prototype of such a system, comparing it to similar systems or frameworks.

1.4 Contributions

The contributions of this document are considered the following:

1. CON-BFT. A Confidential BFT-SMR Protocol that is capable of executing similar loads to BFT-SMaRt while mainting Integrity and Confidentialty guarantees due to the inclusion of an Intel SGX Enclave.

2. Java-SGX. An external Library that can be imported by any Java project to make native calls to C code connecting to the Enclave, allowing Enclave primitives to be included in Java projects by matching libraries.

3. Development of a Con-BFT prototype, alongside its testing and experimental evaluation, with the results included in this document, such as throughput and latency.

4. A reflection on the future of protocols that use Intel SGX, considering the benefits, drawbacks and whether said use in the future is one that should be employed, or if different alternatives should be explored.

Capítulo 2

Related Work

2.1 BFT and SMR

2.1.1 Byzantine Generals Problem and Byzantine Fault Tolerance

One of the oldest problems in Distributed Computing is known as the Byzantine Generals Problem, originally defined by Lamport. For this problem, multiple Byzantine generals that can only communicate through messengers face a large enemy army, and must all decide together the next course of action, whether it be attack, or retreat. There may be, however, disloyal generals, that seek to invent or omit some of the messages, to the point of sending possible conflicting information. This problem is not only applicable in real-life combat scenarios, however. This problem translates quite well into Distributed computing. If we consider each general to be a node in a Distributed System, we can possibly understand the problem at hand. Nodes giving out erroneous information to other nodes in the system with the objective of corrupting the system's current state is something that needs to be taken into consideration when designing a Distributed System. Nodes that fail in this way are considered faulty nodes. Byzantine Faults specifically are the ones that are not detected with failure-detection mechanisms, and as such, a node may be faulty for a plethora of reasons. The problem becomes then on how to ensure consensus on a value considering the possibility of faulty processes. Byzantine consensus may vary depending on a multitude of assumptions that we take from the system. Are all nodes connected to each other? Are the communication delays bounded? Are the channels reliable and secure? Are the messages in FIFO order? Note that these assumptions serve as constraints of what a distributed System can and cannot do, meaning that these assumptions should be carefully defined, in order for the wanted coverage in real-world problems to persist.

Any algorithm that has as an objective to solve the problem has a few requirements that it has to follow, such as termination (All correct processes decide eventually), agreement (All correct processes come to the same decision) and integrity (If commander is a correct process, all other processes follow his proposal). The protocol used for this is what is called Byzantine Agreement, which is an algorithm that works in multiple rounds

of synchronous communication, with the generals, or nodes, sending messages to one another. For this algorithm, the loyal nodes must all agree on a correct value, despite whatever messages are sent by the disloyal generals, or faulty nodes. If we consider an example with 3 different nodes, with one of them being a faulty node, if the faulty node sends conflicting messages to the correct processes, this means that by going through with a simple majority, the two correct processes would decide differently, and that would, therefore, break the agreement requirement of the algorithm. We can therefore conclude that $2f + 1$ nodes are not enough to guarantee Byzantine Consensus. However, by adding one correct process, and employing the idea of reliable broadcast, consensus can be guaranteed. For every sender process p, the remaining processes will exchange process p's message in order to agree on its value and finish the algorithm. From this, there are two possible outcomes from the algorithm, either the original sender is a correct process, or the original sender is a faulty one. If the original sender is a correct process, the correct conclusion would be the original sender's proposal, which as long as the number of faulty nodes does not exceed $(n - 1)/3$ will always be met. However, if the original sender is a faulty sender that could send incoherent messages to the other processes, a correct conclusion can only be reached by exchanging the values received by the other processes. If it is found that a node has sent different values to different nodes for this algorithm, his answers can be excluded, being therefore possible to find consensus.

2.1.2 State Machine Replication

When a server has the objective of serving customer requests, how do you make it Byzantine fault tolerant? If we consider the server in question the previously mentioned system, the easiest way to do so is by adding multiple replicas of the same service, all the whilst making computation deterministic, in order for all replicas to have equal starting and ending states, which means that all correct replicas will always keep the same information and methods, i.e state. By using a State Machine to obtain fault tolerance, and guaranteeing the replication of computation as a whole, it becomes a guarantee that the computation is physically and electrically isolated, which allows to mask possible failures in the replicas, since the system as a whole can bring outputs to a vote. This could complicate the development of such a system, however, due to the fact that ordering messages, distributing inputs, collecting outputs and possibly putting them up to scrutiny in the form of voting cannot significantly increase the completion times. This technique also allows the developers to focus on the overall business logic of the system at hand, since the abstraction of the objects used in such coding can be employed into a previously built state machine. These state machines are nothing more than state variables, which contain the state and commands of the system. These commands contain deterministic implementations of system functions, and are executed in an atomic way, being therefore separated from any other commands. These commands are received in the form of requests, which

are ordered consistently with causality. Clients of these systems can therefore assume that requests issued by a single client to a given state machine are processed in the order they were issued, and that any request r made to a state machine by a client c that could cause a request r' to be made by a client c', will see the state machine process r before r'.[21] But what defines state machines is the fact that it specifies deterministic primitives, transitioning to different states based on the sequence of requests it processes, being independent of time or other variables. Any system will define a component as faulty as soon as its behaviour is not consistent with its specification. There are two main types of fault tolerance, Byzantine Fault-Tolerance and Fail-Stop Failures Tolerance, the former having the challenge of Byzantine faults not being detectable by failure-detecting mechanisms, as components can exhibit arbitrarily malicious or erroneous behaviour, possibly causing faults to other components. With the latter being faults that change the state of the component to make it easier to detect that a failure has occurred before stopping and restarting entirely. Byzantine failures can be especially destructive, meaning that any system development requires careful assumptions about how faulty components will behave and impact the system, as if those assumptions are not correctly satisfied, could jeopardize the entirety of the system. As such, the development of critical systems must tolerate Byzantine faults. This means that being an f fault-tolerant system satisfies its specifications and functions if no more than f of the system's components fail during an arbitrary interval of computation, with no guarantees if f or more components fail. An f fault-tolerant state machine can be created by deploying copies of the code in each one of the available processors for the Distributed System. Assuming that each copy is being run by a non-faulty processor with the same initial state of all other copies and follows the same requests in the same order, then all replicas will reach the same state and output. This means that if any one failure affects only one processor, by combining the states of all replicas in the form of consensus, the correct state for a f fault-tolerant state machine can always be achieved. With this, we can conclude that an f fault-tolerant state machine must have at least $2f + 1$ replicas, with the output being ruled by the majority of the replicas. We conclude this value due to the simple example of a given example state machine with 3 replicas. If one of them fails, the only way to guarantee majority rule with a consensus algorithm is by having a minimum of 3 replicas, being equivalent to $2f + 1$ replicas.
However, for the system to guarantee correct outputs there are assumptions that need to be met. In this case, agreement, the fact that every non-faulty copy of the state machine receives every request in the same order, which would represent these request being executed in said order [21]. Ordering is therefore extremely important in State Machine Systems, in which to guarantee the proper functioning of the system, since two requests are not necessarily interchangeable. This order can be implemented by clients assigning unique id values to the requests, with the state machine using these to guarantee total order. For agreement protocols, there is one arbitrarily chosen processor called the transmitter, to

disseminate a value to the other processors in such a way that all non-faulty replicas agree on the same value, and all the non-faulty replicas using the transmitter's proposal conclude that the transmitter is a non-faulty replica. The protocol begins as soon as a client makes a request, that ends up being sent to all the replicas of the state machine. From then on, the replicas will deterministically finish the computation and find consensus amongst themselves, not being required for the client to be the transmitter.
The client can be programmed to be simpler, by sending its request to one replica only, at the risk of sending the request to a faulty replica. However, there are a few agreement protocols that can be used, with different costs. For example, if digital signatures are used and processors can exhibit Byzantine failures with said signatures, then f+1 processors would be enough to tolerate f faults, due to it being possible to ignore faulty nodes. Otherwise, it becomes the problem of Byzantine Agreement, with a required $3f + 1$ replicas

2.1.3 PBFT

PBFT stands as a replication algorithm capable of tolerating Byzantine Faults, which assumes an asynchronous distributed system where nodes are connected by a network, which may fail to deliver messages, delay them or deliver them out of order. PBFT assumes that individual nodes may fail arbitrarily, which is only true if the all the replica's computations are independent from one another, i.e each replica works in different machines, having different processors running the operating system and service implementation. PBFT uses cryptographic techniques to pevent spoofing and replays to detect corrupted messages. Also using RSA primitives in the form of public-key signatures, message authentication codes (MACs), with every replica having access to the other replica's public keys, in order to verify signatures. PBFT allows and supports the possibility of very strong opponents that may be able to control or even coordinate faulty nodes to delay requests or replicas in order to damage the replication service.

Its algorithm can be used to implement any deterministic replication service with system state and multiple operations, not necessarily requiring them to be simple reads and writes of portions of the system state. This replicated state is implemented by n replicas, which are considered non-faulty if attackers cannot forge their signatures and if they execute the following algorithm [10]. PBFT's algorithm provides both safety and liveness if no more than $(n - 1)/3$ are faulty, meaning that this service behaves as a centralized implementation that executes operations atomically one at a time [10].

The algorithm stands as a form of state machine replication, modeled as a state machine replicated across the existing nodes in the network. Each state will maintain its service implementation and its operations. These replicas move through a succession of configurations called views. For each of these views, one replica will be elected its leader, with the remaining being considered backups. The primary replica of a view v is p, where $p = vmodn$, being n the total number of available replicas in the system, with view

changes being carried out when it appears that the leader has become faulty [10].PBFT's algorithm works as follows:

1. Clients will send requests to invoke system operations to the view v 's primary replica;
2. The primary replica will multicast said requests to the backups;
3. All replicas that receive said requests will execute them and return a reply to the client;
4. The Client will await at least $f + 1$ from different replicas with the same result, which if true, accepts it as the truly correct answer;

Like other State Machine Replication protocols, by ensuring that all messages are totally ordered, the algorithm can ensure safety, since all request execution is deterministic, and all replicas must start in an equal state.

A client c may request the execution of a given operation o by sending a request $< REQUEST, o, t, c >$ message to the primary replica. The timestamp t can be used to help with totally ordering requests, ensuring that lower timestamps lead to requests being executed earlier, with the mentioned timestamp possibly being the client's clock time. With every message, each replica will include the current view v on which the system currently stands, allowing the client c to calculate the current primary, to which messages will be sent. This primary replica will then atomically multicast each message for the existent backup replicas. To reply to the request, a replica r communicates directly with a given client c, with the format $< REPLY, v, t, c, i, r >$, containg the current view v, the request's timestamp t, the replica id i, and the request's result r. The client will then await $f+1$ valid signed replies from different replicas containing the same t and r, before accepting r as the valid reply to the given request, considering that at most f replicas can be faulty, as long as this stands true, the result will be valid [10].

However, a client may not receive its replies as fast as desired, if so, it broadcasts the request to all replicas in the system. If the request has already been processed, the replicas will just resend the reply, otherwise, they will relay the message back to the primary replica and await the primary replica p's multicast. If said multicast does not occur, backup replicas will suspect p of being faulty and initiate a view-change algorithm[10].

In PBFT's protocol, each replica's state includes the service state, a message log, and the replica's current view, denominated by an integer. When a primary replica p receives a client request, it will start a three-phase protocol in order to multicast the requests. The primary replica p in the pre-prepare phase will assign a sequence number seq to the request, and multicasts a pre-prepare message with the client's request m attached to it $<< PRE - PREPARE, v, seq, d >, m >$, transmitting the system's current view v, the sequence number, and m's digest, in the form of d. These pre-prepare messages are

used as proof that a given request was assigned a sequence number seq in view v . This allows the total-order protocol to be decoupled from the protocol to transmit requests to replicas. This means that the total-order protocol can be optimized for small messages and a transmition protocol optimized for larger requests.[10]. After receiving a pre-prepare message, a backup replica i accepts it if:

- The signatures in the request are valid;
- It is in the same view v as provided by the message;
- It has not accepted a pre-prepare message for the same sequence number in the same view with a different digest d;
- The sequence number is between a lower water mark h and a higher water mark H.

With the final condition, the backup replicas prevent a faulty primary from consuming the usable space of sequence numbers by selecting an extremely high value. If the backup accepts the given pre-prepare message, it will advance to the prepare phase, by multicasting a prepare message $< PREPARE, v, seq, d, i >$ before adding both the pre-prepare and prepare messages to the replica's log [10]. All replicas, including the primary, must accept the prepare messages and add them to their logs, provided their signatures are correct and their view numbers match. By ensuring that all replicas agree on the pre-prepare and prepare messages, it ensures that non-faulty replicas agree on total order of the given requests inside a view v. After this happens, each replica i will multicast a commit message $< COMMIT, v, seq, D(m), i >$ to the other replicas when prepared becomes true, which starts the commit phase. Replicas accept commit messages and insert them in their log as long as their signatures are valid, the view numbers match and the sequence numbers stand between h and H. After all commit messages are accepted, each replica i will execute the operation requested by m. This ensures that all non-faulty replicas execute requests in the same given order.

As mentioned previously, pbft also supports a view-change protocol. This provides liveness by allowing the system to make progress when the primary fails. View-changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute [10]. A backup replica starts a timer when it receives a request and is not executing it, if at any point, said request is being executed, the timer is stopped, but restarted if it is awaiting some other request. If the timer of a given backup i in view v expires, said replica i will request a view-change, in order to move to view $v + 1$. It multicasts a $< view - change, v + 1, n, C, P, i >$ message to all other replicas. This message contains the sequence number n of the last valid and stable checkpoint s known to i, a quorum of $2f + 1$ valid messages proving the correctness of s, in the form of C, and P, standing as a set P_m for each request m with a sequence number higher than n that was prepared in replica i. Each one of these sets of P_m contains a valid pre-prepare message, and $2f$

matching, valid prepare messages signed by different backups with the same view v, same sequence number seq and equal digest d of said request m. When the primary p of view $v + 1$ receives $2f$ valid $view - change$ messages for the same view $v + 1$, it will finally multicast a new-view message to all other replicas, terminating the view-change protocol [10].

2.1.4 BFT-SMaRt

BFT-SMART is a java Framework and Library developed in order to allow real-world use of State Machine Replication primitives, considering that most projects relating to the subject have been either not practical to use, or academic-developed prototypes. BFT-SMaRt was developed to tolerate non-malicious byzantine faults, which is defined by a pessimistic system model in which messages can be delayed, corrupted or even lost.[6] BFT-SMaRt also allows replicas to behave abnormally and indulge in any unsupported manners. The library was developed in order to take into consideration simplicity, avoiding optimizations that would have otherwise brought extreme complexity to the system's development, such as techniques that focus on resource efficiency. Even with this design choice, BFT-SMaRt is similar or even better than other highly-specialized and optimized systems.[6] The library implements a modular protocol with a well defined consensus protocol in its core, separating the SMR protocols from the consensus ones, which comes as a strong contrast against monolithic systems like PBFT [10, 6] that have the consensus protocols embedded into the SMR one, lacking clear separation between them. The library assumes the regular system model for BFT-SMR, with $n \geq 3f + 1$ replicas to tolerate f byzantine faults, alongside an unbounded number of faulty-prone clients, and synchrony during consensus protocols to ensure liveness. The system supports reconfiguration and state transfer primitives, being possible to change n and f at runtime. The system will, however, require reliable point-to-point links between processes for communication. These links are implemented using message authentication codes (MACs), over TCP/IP. The symmetric keys used for $< Replica, Replica >$ communication channels are generated using Signed Diffie-Helman. Diffie-Helman stands as a method of exchanging cryptographic keys,having been one of the first examples of public key exchange implemented in the realm of cryptography, which allows secret sharing over insecure networks, by using the other's public key to encrypt the key sent over the public network, requiring the receiver's private key for decryption. The keys for $< Client, Replica >$ channels are generated based on the ids of the given endpoints, in order for clients to not hold key pairs. While still able to protect against non-malicious byzantine faults, client authentication can still be enabled using signed requests, which serves as a strong deterrent against malicious faults. BFT-SMaRt uses multiple protocols necessary to its correct functioning[6]:

- **Total Order Multicast**. This is achieved using Mod-SMaRt, which stands as a modular protocol that is used to implement BFT-SMR using underlying consensus

primitives. An extension of a leader-driven consensus protocol during which clients send their requests to all replicas and await their replies, with the system then achieving total order through a sequence of consensus instances, with each one of them deciding over a batch of requests. Each consensus instance contains three communication steps, with the first one having the leader replica sending a $PROPOSE$ message to all other replicas, proposing the correct order of requests, containing the batch of requests to be decided. Afterwards, replicas will flood the network with $WRITE$ and $ACCEPT$ messages, containing only the cryptographic hash of said request batch. For these consensus primitives, syncrhonization is required, if synchronization is not satisified, in order to ensure liveness, Mod-SMaRt will switch to its synchronization phase, in which a new consensus leader is decided, forcing all the replicas to change to the consensus instance that is currently undergoing decision. This change might, however, trigger the state transfer protocol if faults do exist.

- **State Transfer**. Practical State Machine Replication will always require a state transfer protocol, since replicas should be able to be repaired and reintegrated into the system without restarting the system as a whole. The possibility of failures that can bring down more than f replicas of the system at once will also require stable storage to recover the system that has been affected. BFT-SMaRt implements these durability techniques to deal with the recovery of replicas or the whole system. The key ideas of such techniques would be to log the batches of request executions in a single disk, meaning that every replica would have access to its request history to share, taking snapshots at different execution points over the multiple replicas, and sharing all the acquired information, be it the given snapshots or request logs in a fully collaborative way, with all the replicas dividing the effort amongst them, to split the cost of the state transfer protocol.

- **Reconfiguration**. Unlike most previously built BFT-SMR systems, BFT-SMaRt does not assume a static system, assuming a system that can grow or shrink over time to fit the needs of the developer, system administrators and clients. As such, it provides a protocol to enable replicas to be added or removed from the system during runtime. Mod-SMaRt informs the replicas of the new replica that now requires addition or removal from the system. All correct replicas will adopt the same view as the system's current view at any given point during execution of client operations. This function will require an administrator's signature in order to ensure the request was submitted by someone privileged enough to modify the system. The view manager will then verify the administrator's signature and after the current view is updated, respective of whatever update came in the reconfiguration request. If the view manager was successful, it will then notify the system replicas of the

new updated view, before notifying the replica that was waiting to be added (or removed) that it can start (or stop) its execution. By adding a new replica, this new replica will trigger BFT-SMaRt's State Transfer protocol to bring itself up to date. Due to this, clients will require storing the system's current view, meaning that no replica will accept client requests made to outdated views

The biggest problem that was associated with BFT-SMaRt was how to break these previously mentioned protocols into an efficient architecture, considering that the library implements a high-throughput replication middleware. The system must be able to deal with hundreds of clients, and , more importantly, be able to deal with malicious behaviour from both clients and replicas. BFT-SMaRt manages client requests that are received through a thread pool that is provided by the Netty framework, which stands as BFT-SMaRt's option to manage hundreds of connections in an efficient way, with requests being separated between ordered and unordered requests, which are usually read-only commands and are delivered directly to the service implementation. If the requests are ordered ones, they are delivered to the client manager, which will then verify the request's integrity before adding them to the client's respective queue. The Netty Framework, used by BFT-SMaRt, is naturally a multi-threaded one, exploiting a processor's capabilities for higher throughput, splitting the work over multiple threads:

- **Proposer Thread**. This thread is responsible for assembling the request batch and and transmiting the $PROPOSE$ message of the consensus protocol, with BFT-SMaRt filling the request batch with pending requests until there is none left, or until there are no more requests to add. This Thread is only active in the replica that stands as the leader of the current consensus instance, in order to have only one replica proposing request batches.

- **Sender Thread and Receiver Thread**. The sender thread is tasked with getting every message m from the *out queue* to be sent by one replica to another one, serializing it, producing its MAC to be attached to the respective message m and sending it through TCP sockets. The receiver thread will then read the message, authenticate it, by reading its mac, deserialize it and add it to the receiver replica's *in queue*, where messages received from other replicas are stored before being processed.

- **Message Processor Thread**. This thread is responsible for processing messages from the BFT-SMR protocol. This thread will retrieve messages from the queue and execute them if they belong to the current consensus instance. If not, they are stored again and saved until the awaiting consensus instance is triggered, otherwise, the message will be discarded.

- **Delivery Thread**. After finishing a consensus instance, the decided batch of messages is stored in the decided queue, the delivery thread will be the one tasked with

getting batches from this queue, deserializing them, removing said requests from the client's queues and marking the current consensus instance as finished.

- **Request Timer Thread**. The request timer thread might be activated on ocasion to verify pending requests remained more than pre-defined timeout time t in the pending requests queue. If this timer expires, the request will be redirected to the current consensus leader, if a the timeout t expires a second time , the current consensus instance is terminated and the syncrhonization phase is triggered by Mod-SMaRt. A timeout can expire due to a client that did not send the request to the consensus leader or by a leader that did not order the client request. Due to the existence of many more clients than servers, it is usually suspected that clients are faulty, opposite to replicas, which are only suspected of being faulty if the problems persist over more consensus instances.

As mentioned above, BFT-SMART tolerates non-malicious byzantine faults by default, being able to be configured in order to tolerate other fault models.

- **Crash Fault Tolerance**. BFT-SMART is able to support configuration parameters that make the system strictly crash-fault tolerant, by being able to tolerate $f < n/2$ crashes, meaning that the system is able to support a simple minority of the replicas crashing. This, however, means that quorums in the protocols will have to be rearranged to take this into consideration, bypassing the WRITE step during the consensus execution, keeping the remainder of the protocols intact.

- **Malicious Byzantine Faults.** The use of public-key signatures makes it impossible for clients to forge MACs for sent messages, allowing the protocol to ignore false signatures, making the protocol much more resilient to malicious faults. BFT-SMART does not use public-key signatures by default, with the exception of establishing symmetric keys between replicas during leader change. One of the attacks that BFT-SMART cannot defend against are malicious view leaders degrading the execution of the system, usually by stalling message exchange and request execution. However, nodes can use reconfiguration tactics in order to change the current leader of a view in order to reduce the damage that is done by such malicious leaders.

2.2 Computing on Encrypted Data

2.2.1 Secret Sharing

Secret Sharing schemes allow a distributed system to break a secret into many shares, allowing it to be shared amongst the participants in the network. This secret is shared in a way that it requires a threshold k of n shares in order to reconstruct the secret, creating a

scene where less than n shares cannot recover it [9]. This scheme allows the system how to divide a secret S into n shares in order to make S reconstructable using at minimum a number of shares equal to a threshold k, giving away no information if $k-1$ shares are obtained [22]. Secret Sharing is an algorithm that was proposed in the late 1970's by the cryptographer Adi Shamir, which is an algorithm that by using polynomial interpolation, breaks secrets into multiple shares, requiring only a certain amount of shares, known as the threshold t in order to correctly recreate the secret. This means that an adversarial presence in the network would require s shares to meet the threshold t, making it *theoretically secure*, meaning that it is secure against attackers with *theoretically unlimited power*, considering that it is impossible to brute-force the required shares to meet the threshold [5]. This scheme is called $(k, n)thresholdscheme$. Protecting data can be done with data encryption, however, protecting the encryption key becomes another large problem, considering that using further encryption just escalates the problem rather than solving it. Storing the keys in well-stored locations can be a problem, since the more locations the key is stored in, the higher the risk of security breaches [22]. This method of secret Sharing, by using a $(k, n)thresholdscheme$ with $n = 2k-1$, a very robust key management system is obtained. Recovering the original key becomes impossible without exposing more than half of the system's replicas $k = (n/2) + 1$ [22]. Secret Sharing mechanisms are especially good when used in distributed systems, since these threshold methods are ideal where individuals (or replicas) with possibly conflicting interests must cooperate [22]. The scheme is based on polynomial interpolation, if the algorithm is given k points using 2-dimensional planes $(x_i, y_i), ...(x_k, y_k)$, with distinct x_i's, there will only be one polynomial $q(x)$ of degree $k-1$ such that $q(x_i) = y_i$, for all i. By picking a random $k-1$ degree polynomial $q(x) = a_0 + a_1x + ...a_{k-1}x^{k-1}$ in which $a_0 = D$, it's possible to allow the replicas to redo the interpolation process, as long as there is a subset of k values containing each of the D_i values and their identifying indices, considering that $D_1 = q(1), ..., D_i = q(i), ..., D_n = q(n)$, allowing the replicas to re-calculate the total secret, considering that $D = q(0)$ [22]. This technique is used to guarantee better security and protection, alongside privacy, even if one of the shareholders is compromised, be it by attack or fault, the revealed secret is useless on its own. This also means that requiring all shares to reconstruct a secret is not only dangerous, but could be considered counterproductive, considering that we are dealing with an untrusted network in which any of the shareholders could possibly go rogue at any moment.

Galois Counter Mode for encryption

The Galois Counter Mode, or GCM for short, is a block cipher mode of operation that is used to provide authenticated encryption. By taking a secret key K, an initialization vector IV and a plaintext P it is able to output a ciphertext C with the exact same length as P. This also produces an authentication tag T with a length t[16]. The authenticated

decryption takes K, IV, C, T as input and returns an output with the original plaintext P, or a given symbol **FAIL** if the input is not authentic. This encryption mode also supports authentication of important information that must be left unencrypted, such as addresses or network ports [16].

2.2.2 Trusted Execution Environment

A Trusted Execution Environment is a secure, integrity-protected processing environment, consisting of secure memory and storage capabilities [1]. As such, it stands as a tamper-resistant processing environment that runs on a separated kernel, guaranteeing the authenticity of the executed code and the integrity of the current runtime state, such as the CPU registers and memory [20]. It also provides remote attestation measures in order to prove its trustworthiness to third-parties [20]. These environments were designed to achieve secure computation, privacy, and data protection [20]. It isolates the execution, prohibiting other processes, including the operating system, of acquiring or altering data or code that is being executed. This execution environment runs alongside the operating system, and was created with the goal of preserving confidentiality of all the information that goes through it. It differs from the previous hardware solutions by introducing the possibility of securely updating the code and data inside the isolated environment, which improves from the previously available Trusted Platform Modules, which only supported a set of API's [20]. One of the foundations of the isolation provided by these environments is a separation kernel, introduced to enable the coexistence of multiple systems on the same platform, dividing the system into multiple partitions, each with its own set of privileges, guaranteeing isolation amongst them [20]. Unlike traditional kernels or Hypervisors, separation kernels are built to be as simple as possible, with four main security requirements [20]:

1. Spatial Data Separation, ensuring that data within one partition cannot be read or modified by other partitions;

2. Temporal Sanitation, ensuring that shared resources are not used to leak information into other partitions;

3. Control of information flow, ensuring that communication between the partitions is prohibited unless explicitly permitted;

4. fault isolation, ensuring security breaches in one partition cannot spread to other partitions;

All of these requirements can be attested by third-parties, usually by requiring signatures using keys fused directly into the processor, to ensure that not third party can simulate this process. This is especially important when a system that requires trustworthy computation

in untrusted systems, such as smartphones. Due to the installation of multiple applications and their accesses, mobile financial applications, or mobile banking requires the use of a trusted execution environment, to make sure that there are no malicious participants attacking the system execution. With the rise of cryptocurrency, these systems are used in order to implement crypto-wallets, as these offer the ability to store information safely in the computer memory, after being sealed with the embedded processor keys.

ARM TrustZone

ARM TrustZone is one of the more popular Trusted Execution Environments, next to Intel SGX, which separates any application developed with it into a *Normal World* and *Secure World*. These *worlds* are seen as virtual processing cores, with every processing core, being separated in two by the system, a trusted and an untrusted one. If an application needs to change betwen the two, it will use a privileged instruction, SMC (Secure Monitor Call), provided by a monitor component, ensuring storage of registers and protection of secure memory when switching from one world to another. Unlike SGX, which separates an application into trusted and untrusted components, TrustZone creates a new fully isolated environment to execute trusted applications. This fully isolated environment is guaranteed by a new CPU privileged mode, known as the monitor mode. When the SMC is activated, the processor will remove all sensitive information is deleted from the registers that are shared between worlds, since both *worlds* share a physical processor.

Due to TrustZone providing two different environments for execution, application development for TrustZone is much more flexible than SGX's enclave model. Whilst the simplest designs with TrustZone can be similar to sgx's primitives, with untrusted applications making synchronized calls into *secure world* libraries, TrustZone also supports a secure OS kernel executing in parallel inside the *Secure World*, but also creates a difficult problem for the programmer, leaving up to the developer on how to handle *Secure World* resources.

TrustZone has its system separated into two virtual cores, one for each *world*, and duplicated hardware models in order to provide physical isolation. The existence of physical memory management units, usually guaranteeing one for each virtual cores, allow for each *world* to have its own memory mappings, which efficiently remove the need to delete memory before switching *worlds*. Cache, however, is shared amongst the *worlds*, creating the necessity of differentiating what areas each *world* can access. For this, an additional bit is used in order to differentiate what memory is secure and what memory is not, eliminating the need of deleting the cache when changing *worlds*. Unlike SGX, TrustZone supports I/O access, in the form of a segregated bus, known as AMBA3 APB, which is used for secure peripheral access, guaranteeing that a peripheral that was connected securely cannot be accessed from the *normal world*. In order to support this, interrupt handling is provided inside the *secure world*, with exceptions being treated inside it.

Intel Software Guard Extensions (Intel SGX)

Intel Software Guard Extensions (SGX) is another widely used Trusted Execution Environment. Introduced with the skylake architecture a few years ago, the system allows the untrusted component of the application to create an enclave that will execute the trusted area of the application. This enclave will then be placed in protected memory. This memory is always protected from external processes, by creating trusted functions that can properly access the enclave. This enclave is a part of the process memory, however. This means that the application has its own data and code, which includes the enclave, that contains its own data and code as well. This means that any application that uses Intel SGX for its design is therefore divided into a trusted and untrusted component, with the trusted component being the aforementioned sgx-provided enclave. The processor will allocate a portion of the memory, limited at 128 megabytes of memory, in order for it to become enclave-specific memory, which will be then protected from external accesses, which also include the operating system's kernel. This memory is not impossible to access, however, the untrusted component does contain primitives in the form of enclave-specific instructions in order to request the enclave to do some form of computation. These primitives are known as enclave calls (ECALL), that are used in order to send information from outside to the inside of the enclave, location where computation does not have access to any external libraries, such as I/O, which would represent a possible security breach for the enclave [11]. These functions are accessible through outside calls (OCALL), which allows the enclave code to call on external libraries through the trusted component of the system. These functions, however, do have some cost associated with them, considering that the enclave will need to be exited in order to execute these functions, before the enclave is re-entered. These functions can be defined through specification files, commonly known as EDL files (Enclave Definition Language), in which functions are defined as trusted, meaning they belong to the enclave (ECALL), or untrusted, meaning they belong to the untrusted component of the application (OCALL). Intel SGX does offer confidentiality and integrity guarantees, using the enclave model. SGX also provides attestation tools, in which a user can confirm whether a remote enclave is working correctly and has not been tampered with [11].

The processor manages a memory region known as PRM (Processor Reserved Memory), which is programmed to be isolated from the rest of the operating system, with the processor being the only component connected to with. This memory region contains all of the enclave's data structures, data and code. In this memory region is also stored the EPC (Enclave Protected Cache), which stand as memory pages awaiting assignment to specific enclaves. With memory inside this area being limited, to allow multiple applications to work concurrently, sgx supports page swap between this area of protected memory and untrusted memory, moment in which the memory is blocked, meaning that no writes can occur over any of said memory addresses and encrypting said pages, so they

can be stored in untrusted memory. These swapped pages contain integrity proofs, in the form of Message authentication codes (MACs) and version numbers, in order to guarantee they were not tampered with. In order to reload any of these swapped out pages, the page will need copying back to the PRM before running any required security checks, ensuring that there was no tampering, alongside with verification of the page version, to ensure is the last one[11].

Attestation is also an SGX feature, which serves as intel's option to verify that no tampering has occurred with the newly deployed enclave. For this, a digest of the enclave's code and data is created and stored in the MRENCLAVE register, only after this register is filled can the attestation take place. Attestation can take two forms, either local, or remote:

- **Local Attestation**. A newly deployed enclave will perform an attestation request to the enclave to be attested. This enclave will then create an attestation report, containing it's identity, data structures, attributes and hardware it's running on, producing a MAC using a symmetric report key, which is owned by the processor and can be requested by the enclave. The enclave to be attested will then reply to the enclave attesting it with its attestation report. The attesting enclave can then use the report key provided by the attested enclave in order to create its own attestation report. Considering that the attestation is local and processor is the same, it means that the produced MACs will then necessarily be the same. The attesting enclave will then redo its attestation report and compare the MACs, before sending its report back to the attested enclave.

- **Remote Attestation**. For this, a request is submitted to the untrusted part of the remote application, which will connect to its enclave, requesting a report structure from the enclave, which will be created and then sent to a newly deployed enclave in said system, known as the quoting enclave which will sign said report with an EPID (Enhanced Privacy ID), producing what is known as a quote, before returning it to the request sender. This EPID serves as an asymmetric key owned by the processor,which can then be verified by the request sender, using the public key certificate of said EPID.

These are not all the primitives that are given by sgx. Sealing is a major one as well, in which the enclave uses its own key, which in turn is owned by the processor, to encrypt the information in a secure way, allowing it to be stored outside the enclave, ensuring integrity of the information that is sealed, considering that even if the enclave or the system as a whole shut down, a re-deployed enclave can always undo its seal over the sealed information as long as the processor remains the same. This is especially important when you consider that all information inside the enclave is destroyed if the application is closed. Considering that preserving data is the end goal in this scenario, intel sgx was

designed with mechanisms to ensure that the used and retrieved key are unique to said enclave, meaning that the used key can only be retrieved by this one enclave using this one processor. There are two options of this mechanism [11]:

- Sealing for the Current Enclave (**MRENCLAVE**). If this method is chosen, this means that the current measurement of the enclave is used for the sealing operation, meaning that for future removal of the seal, an equal MRENCLAVE value is required to generate the key used to seal the data. It is worth noting, however, that if any attribute, data, or value related to the enclave has changed, the MRENCLAVE will also change.

- Sealing to the Enclave Author (**MRSIGNER**). If this method is chosen, the identity of the enclave author is bound to a special enclave register (MRSIGNER) at initialization time, with the product ID also being bound to the generation of the sealing key at runtime. This means, therefore, that only enclaves with the same author (MRSIGNER) and the same product ID can undo the sgx seal. This type of sealing allows for enclaves written by the same author to share sealed data as long as the two enclaves belong to the same processor.

2.3 Confidential / Hardware-Based BFT-SMR

2.3.1 COBRA

COBRA serves as a confidential State Machine Replication System that is tolerant to Byzantine Faults. It was designed as a fully connected distributed system whose nodes could be divided into two infinite subsets, the clients $c = c_1, c_2, ...c_n$ and the server replicas $r = r_1, r_2, ...r_n$. Requests are sent to the system by sending each of the requests to a subset of the replicas (called a view). Each process, be it client or server, contains a unique ID that can be verified by any other process belonging to the system, usually through public key infrastructures. For this, each replica r_i contains public-private key pairs $< pk_i, sk_i >$, that is used for both message signature and encryption.

By design, it was assumed a partially synchronous model in which the entire network may behave in an asynchronous manner up to *an unknown instant* t after which the system changes to a synchronous one with time bounds for computation and communication. This is done to ensure liveness of the consensus protocols, that cannot be guaranteed for asynchronous systems.

It was considered as a fully dynamic system where any node is able to join or leave at any point during execution by using reconfiguration techniques that can during execution time re-install the sequence of views in the system, each containing a sequence of replicas available at the time the reconfiguration algorithm is run. The reconfiguration request is also considered as a regular request, and also goes under the total-order consensus same

as all other requests. After the reconfiguration, all correct replicas will adopt the same current view V_{curr}, which represents the most recent view of the system. Until this is done, the only view that may participate in the execution of client requests is V_{curr}.

All of these replicas are prone to Byzantine faults, however. Any process that succumbs to any of these faults is from then on considered faulty. The system always considers the possibility of a system being able to control the network, by corrupting an unbounded number of clients or replicas in each view. A view can see a maximum of $(n-1)/3$ replicas corrupted, being n the total number of replicas in its view.

The state machine model defines that all correct replicas work with deterministic functions and contain the same state after executing all received requests, provided they were received in the same order. The use of secret sharing, however, means that no replicas will have exactly the same state, due to shares being scattered throughout the network. This is done in order to provide a confidentiality layer for a pratical BFT SMR system. No system comes without downsides, the system requires secret resharing mechanisms, in order to resupply any replica that has lost or compromised its own replicas, be in the form of byzantine fault or otherwise. Any replica r_i that recovers from failures will need to obtain the public part of its state, which is also contained in other replicas, using the state transfer protocol, and will also require an invocation of the share recovery protocol for each data entry D_i, in order to reconstruct its private state. Secret resharing is done with distributed polynomial generation, which allows $n \geq 3f + 1$ replicas to randomize a polynomial P of degree f, encoding the point (x, y), with y being the value 0 in order to maintain the secret. By the end of this protocol, every replica will maintain a point $(i, P(i))$ of P. For this, each server locally generates a random polynomial and distribute its shares to the group, with a set of these $f + 1$ polynomials being selected and byzantine consensus is used to ensure all replicas agree on this set. To end the protocol, the $f + 1$ selected polynomials are summed , resulting in the shares of a Polynomial P. This protocol ensures that at least $f + 1$ correct processes will obtain a valid point of the Polynomial. This, above all, allows the system to maintain secrecy of user-given information , by spreading the shares required to retrieve the secret over the network, in order to ensure that no faulty or compromised replica reveals sensitive information.

2.3.2 MINBFT

BFT protocols typically do require $3f + 1$ replicas in order to tolerate f byzantine faults. This is based on the idea that in the worst scenario, correct replicas can overcome faulty replicas when outputs are put to a vote. Papers in the past have proven that Byzantine Fault Tolerance algorithms cannot function with fewer than $3f + 1$ replicas [8], however, these proofs assume the possibility of faulty nodes sending conflicting information with malicious intent. The new Trusted Execution environments do provide an option in order to protect against these malicious attackers, by extending nodes with tamper-proof

components that would continue to produce correct information even if the node itself becomes faulty [25]. One of the most important metrics for for distributed systems is the number of communication steps, considering that latency does play a role in slowing down an algorithm. Tolerating disasters and DDoS attacks becomes increasingly difficult, making it necessary to deploy replicas in different sites, increasing communication delays [25]. In the algorithm proposed, MinBFT, the replicas will move through a succession of configurations called views [25]. MinBFT allows its primary replica to use trusted counters to assign both order numbers and a certificate for that number assignment in order to prove that the number was indeed assigned to said request, and not any other, while also making sure the counter was incremented. By using this technique, it is guaranteed that all replicas agree on the correct order for request execution [25]. For this protocol:

1. A client will send its request to all nodes;

2. The primary Replica will assign a sequence number to the client request, signing a certificate to prove it before sending the request to all other servers in the form of a $PREPARE$ message;

3. Each server will then multicast a $COMMIT$ message to the other servers as soon as $PREPARE$ message is received from the primary replica;

4. When a request is accepted, the request is executed locally and the reply sent to the client;

5. To finish the request, the client will wait for at least $f + 1$ matching replies to complete the operation.

By reducing the number of total replicas to $2f + 1$, the client can safely await only for $f + 1$ replies, since f replicas can indeed by faulty. This is possible due to MinBFT having protections against duplicity, i.e the possibility of faulty replicas sending messages with differing contexts to different replicas [25], more specifically, MinBFT can protect the system against faulty replicas that may want to send differing messages with equal identifiers. MinBFT uses a trusted subsystem, named USIG that prevents duplicity by assigning identifiers to different messages and does not assign the same number to different messages. By assigning a sequence number, it also assigns a certificate to the request, in order to prove that the sequence number was indeed given by the service.

The USIG (Unique Sequential Identifier Generator) service is a local service that exists in every replica, with the objective of assigning the value of counters and signing them. These subsystems ensure that USIG will never assign the same number to two different requests, i.e uniqueness, will never assign an identifier that is lower than a previously assigned one, i.e monotonicity, and will never assign an identifier that is not the sucessor of the immediately previously assigned one, i.e sequentiality [25]. These properties are

ensured even if the replica is compromised, so the service has to be implemented in a tamper-proof module. The interface provided contains two functions, one for creation of a USIG certificate and one to verify said certificate. $createUI(m)$ will return a new USIG certificate containing a unique identifer (UI), ensuring that this UI was executed inside the tamper-proof modlue. $verifyUI(PK, UI, m)$ verifies the UI is valid for message m, i.e if the certificate matches the message and the rest of the data in the UI.

This is important, because even if a faulty server decides to not send the message for the consensus algorithm, or worse, decides to send wrong information, it will never be able to send to conflicting information to different replicas, considering that two different messages will always contain different UI's. This means that the number of replicas required for consensus may drop from $3f + 1$ to $2f + 1$, due to it being possible to guarantee that a server will not send conflicting information to different replicas, meaning that the threshold may now drop to $n = 2f + 1$, just like crash-fault tolerant systems.

Because of the previously mentioned mechanism, from now on $f + 1$ correct replicas are enough to achieve consensus on anything that the system may require [25]. For this system, all client requests are signed with client private keys K, allowing for the server to identify and discard requests with invalid keys, since the system stores the requests with a sequence number larger than the last executed, creating a queue of requests. The client will then await the matching $f + 1$ replies. As previously mentioned, a primary Replica gives a request r a sequence number, guaranteeing that no two requests have the same sequence number. The main replica will create a PREPARE message with the sequence number of the request and broadcast it to all the other replicas, these may however fail, and if the sender is faulty, it may happen that the other replicas receive COMMIT and not PREPARE messages. If the sequence numbers are valid, the servers that receive the COMMIT message will replicate and broadcast it.

However, if primary Replicas are faulty, this could possibly hinder the algorithm, since these could possibly not assign sequence numbers to requests. As such, the algorithm allows for a change of primary replica in case this one is detected as being faulty. This is usually done throught timeouts. When a backup replica receives a request from one of the clients, it starts a timer which would end after T_{exec}, if the request is accepted using the previously mentioned mechanism, the timer is stopped. If the timer is exceeded, the replica will suspect a faulty primary and will start a view change. To start a view change, a backup will multicast a change-view message from view v to $v' = v + 1$ and to change states to the latest stable checkpoint c_{last}. As soon as it receives a total of $f + 1$ total change-view messages, it will fully change its view from v to v', moment in which it stops accepting messages directed towards view v. This change-view messages takes a unique identifier UI_i, in order to prevent faulty servers to send to different servers of the system messages with different checkpoints C_{last} and different server subsets for the new view. The servers will therefore only consider messages that are consistent with

the system state, meaning that the checkpoint certificate contains at least $f + 1$ valid UI identifiers.

2.3.3 BFT on Steroids (Hybster)

BFT replication services have come to employ 3f+1 replicas in order to ensure that a service is capable of accepting arbitrary faults from any of its replicas at any point. Hybster is presented with a hybrid fault-model where trusted replicas are assumed to fail only by crashing, with the number of required replicas therefore dropping to 2f+1[4]. This means that any hybrid replication protocol can require less diversification of the system and require a smaller amount of resources. Hybster stands as one of the first hybrid replication protocols to not depend fully on sequential processing, which has hindered systems so far, by prohibiting systems from taking advantage of multi-threading primitives, which could help BFT protocols reach new optimization levels. With new generation processors, new programming models can be used, taking full advantage of the enclave model proposed by intel SGX. Hybster is therefore a new hybrid-state machine replication protocol that is parallelizable. Ordering protocols of most existing state-machine replication systems is based consensus instances that are sequentially processed, which restrict the performance of such systems on most modern platforms. Most hybrid systems prevent equivocation by cryptographically binding sensitive messages to a unique monotonically increasing timestamp by means of the trusted subsystem. If faulty replicas make conflicting statements by sending multiple messages to different replicas in the same phase of the protocol, they can only do so by creating different timestamps for each message, meaning that other replicas are able to detect such behaviour. These unique timestamps, however, establish a timeline for each replica, which allows for induction of the sequential nature of these hybrid protocols. Hybster instead of having one single virtual timeline, proposes a system with each replica being equipped with a plethora of independent timelines, one for each trusted subsystem, that are as many as the level of parallelization required. This concept where each processing unit is assigned a subset of consensus instances prevents the opportunity for equivocation[4].

For this, Hybster was proposed as hybrid replication protocol designed around two-phase ordering and multiple trusted subsystems realized using Intel SGX. Opposite to other options, hybster does not force faulty replicas during a view-change to reveal all consensus instances for which they sent an order message, being based on the idea that it suffices to ensure that reveal is only required for consensus instances that are not guaranteed to be propagated by correct replicas, but have possibly led to execution of said requests. This means that Hybster allows faulty replicas to not present their messages as long as they are not critical to system safety. This not only significantly reduces histories, but also the complexity of the view-change protocol. Hybster is also capable to perform consensus instances in parallel, benefiting from modern multi-core platforms, opposite

to existing options that focus on sequential ordering of requests. As such, the hybster protocol can be divided into three parts, the trusted subsystem tasked with providing the mechanisms used by the protocol to prevent faulty replicas from making conflicting statement, the basic unparallelized replication protocol, which is the protocol that is performed amongst the replicas, and the parallelized realization, which right now serves as a view of the internal protocols that are performed by the concurrent cores from each replica's processor[4]. Hybster develops over a trusted subsystem called TrInc, developing its own subsystem, called TrInx, which tailored TrInc to increase performance. Considering that Intel SGX supports the creation of multiple trusted enclave instances, in conjunction with the hybster protocol, this allows a multiplication of the entire subsystem, therefore allowing a parallelization of execution of said trusted environments, bringing forth an increase in efficiency whilst not interfering with the trusted components.

As long as trusted environments are deployed by a trusty source, i.e a trusted administrator, before the group of available replicas is brought to life. These trusted instances are assigned unique identifiers, the number of trusted counters and a shared key amongst all the trusted components, the counters are set to 0. After being set up, these TrInx instances can create certificates for each message, with many ways of doing so [4]:

- **Continuing Counter Certificates** A TrInx instance can create a certificate for a message m using specified trusted counters tc, producing the new counter value tv'. When a certificate is requested , the TrInx instance tss_i will accept a new tv' value that is greater than or equal to the current tv of the counter tc, meaning that $tv' \geq tv$. It will then calculate a new MAC based on the secret key shared by all the trusted instances in the system, its own instance ID, the id of the counter tc, the new value tv', the current value tv and the message m itself. After doing as much, it will set the counter tc to the new value tv' and will return the newly calculated MAC. This means that a message will only have its certificate attached to it if said certificate is unique. Other instances of TrInx can be used in order to verify said certificates, requiring the id of the issuing instance, and the expected values of tc,tv,tv', alongisde the message m. The verifying instance can then use these values to redo the MAC calculation using the shared key, meaning that if the produced MACs do indeed match, the produced certificate is valid, considering that only these TrInx instances have access to the provided shared key.

- **Independent Counter Certificates**. In order to be able to verify a continuing certificate, the previous counter value tv has to be known to the trusted subsystem, but for a lot of the protocol's communication, it is often enough to ensure that the certificate is unique, meaning that it is often enough to make sure that there are no other certificates with the same value tv'. For this, the TrInx subsystem offers independent counter certificates that lack the previous counter value tv, but are only issued with new tv' values greater than the previous counter value tv.

- **Multi-Counter Certificates**. If a trusted subsystem is, at any point, required to prove the validity of more than one certificate, it is more efficient to issue certificates that encompass more than one counter instead of issuing individual certificates, which is an option that is provided by the TrInx subsytsems.

- **Trusted MAC Certificates**. While regular MACs can be used in order to guarantee the authenticity of messages, they do not provide non-repudiation. Since more than one party possesses the secret key, this means that a party can deny that it sent the message.With a digital signature envolved, this becomes impossible. The downside is the time required for computation, considering that private keys are far more expensive to compute than MACs. However, by including the secret key and its unique ID, TrInx can provide certificates that provide non-repudiation, which come as a middle-ground between MACs and digital signatures, only being slightly more computationally expensive due to the cost of switching to the trusted execution enviroment provided by TrInx. By mapping trusted MACs to continuing certificates where $tv = tv'$, it becomes possible to save on running time.

Hybster's basic protocol is a combination of other widely used core subprotocols for state machine replication protocols. Total-order, checkpointing, state-transfer and view-change protocols.

1. **Total-Order**. To establish total-order for requests pending execution, one of the replicas will be selected as the current leader l, which assigns order numbers for the given requests and proposes this assignment to all others, by awaiting for a sufficient number of replicas to accept to follow the proposal, consensus is reached and the requests can be executed. The system goes through consecutively numbered views v, in which the leader $l = vmodn$. If a leader is detected to be faulty, the view-change protocol will be invoked in order to elect a new leader.

2. **Check-pointing**. Hybster assumes that the network is unreliable, assuming that the network may lose or delay messages, and considering that byzantine-faults are very hard to detect, it is very hard to ensure if a replica is faulty or if the network itself is just slow or failing. When required, the system may resend messages that were lost or delayed, the system will keep a log of said ordered requests during a certain pre-defined amount of consensus instances, while deleting older messages when there is proof that enough replicas executed said requests. Replicas will regularly execute said protocols in order to achieve proof that the system indeed executed said requests. By sending $CHECKPOINT$ messages to one another, at some point, at least one of the replicas will be able to collect a quorum of said messages with equal checkpoints, said checkpoint becomes stable. It will then delete order messages to preserve memory storage, not affecting view-change protocol messages in the process.

3. **View-Change**. If at any point a leader is suspected of being faulty or fails to get its required quorum for whatever proposals it does, the current view is aborted, and a new view is created in which a sufficient amount of replicas follow the leader in order to make progress, i.e ordering said requests. It is worth noting, however, that this view-change protocol must ensure that any request with the order number o that has already been executed in a previous view, no request with that same order number can be executed. Assume that consensus has been reached with the previously mentioned checkpoint algorithm, with the replicas already having discarded their order messages, saved snapshots of the current system state and possessed a quorum certificate of $CHECKPOINT$ messages proving the correctness of the state. As soon as a client tries to issue a request r to the view's leader r_0 and has that message lost, after a timer has expired, the client will resend said request r. The client will have no way to know if the leader r_0 is faulty and refused to propose the client's request, or said request was just lost in the network. When the request r is sent, the request is sent via multicast to all the replicas in the view. Let us assume that r_0 is correct and that another replica, r_1 had temporarily disconnected and had not received r_0's order proposal. Even if other replicas received said order proposals, r_1 will now suspect that the leader r_0 is faulty, due to having received the request directly from the client via multicast instead of the leader's proposal. When this happens, r_1 will send a $VIEW - CHANGE$ request to the other replicas. These other replicas will announce to the system that they have abandoned the view $v_from = 0$ with the leader r_0 and want to join the new view $v_to = 1$ with the new proposed leader r_1. Due to View-change certificates, Hybster can rely on single correct replicas within quorums to announce consensus instances that have been executed by replicas that are not members of the aforementioned quorum.

Consensus-oriented parallelism schemes allow replicas that are composed of various similar processing units to communicate asynchronously using memory messaging only, operating independently from one another. With regard to ordering requests, this is achieved by letting each of the processing units responsible for executing a share of order numbers. Considering Hybster's design, this wouldn't be very effective, considering that like other hybrid protocols, no consensus instance $o' > o$ can be processed unless the consensus instance o has been concluded. By binding messages to of the protocol to values of trusted counters that can only directly increase ends up playing into this sequential limitation. In order to handle this limitation, Hybster allows replicas to certify messages for particular phases of the protocol with different trusted counters,therefore allowing the workload to be split amongst multiple independent processing units, each equipped with its own instance of the TrInx subsystem. Hybster allows a replica r to create message certificates of the format $c(r(u), ..)$ where $r(u)$ is the ID of the TrInx instance of the processing unit u in the replica r. By using consensus-oriented parallelization, consen-

sus instances are split over the processing units in a predefined manner, meaning that it is known before-hand which processing unit will process each consensus instance. This means that an order message with the certificate c will only be accepted if the associated TrInx instance is the one associated with the processing unit u that is responsible for the consensus instance o. Considering that both the number of processing units to be used per replica and the IDs associated to each TrInx instance are part of the system's configurations, it becomes possible to stop faulty replicas from using different TrInx instances to create conflicting order messages[4].

To keep the processing units all in the same page, checkpointing in a design with consensus-oriented parallelization has to be carried amongst all of them. The processing unit that will execute the protocol for the kth checkpoint is determined in cyclical fashion, using the round-robin algorithm to distribute additional load over the available processing units.

Using the parallelized version of Hybster, information about a currently ongoing consensus instance is distributed amongst processing units and each replica will contain multiple multiple trusted counters in order to certify order messages. However, the view-change protocol must still ensure that faulty replicas are forced to reveal the consensus instances that have been processed, and will not be able to explicitly work on views that have already been aborted. In order to achieve this while still maintaining access to the trusted subsystems local to each processing unit, hybster will split the view-change messages into multiple others, such as $VIEW - CHANGE$, $NEW - VIEW$,$NEW - VIEW - ACK$, resulting in each of these partial messages arriving to each processing unit. Still, replicas are only allowed to abort a view completely, meaning that if a replica abandons a view, all of its processing units must abandon it too. Therefore, receiving replicas only consider a view-change message only upon receiving all of its parts, with the number of parts necessary for view-change being equal to the replica's total number of processing units configured by the replica. As for ordering, partial messages must only contain $PREPARE$ messages for order numbers the processing unit is responsible for. As such, the system can ensure that the set of $PREPARE$ messages received by the replica are complete. Due to parallelized ordering, the combined sets can, however, contain gaps. As such, a proposed leader will fill the gaps by proposing empty consensus-instances.

2.3.4 Consensus-Oriented Parallelism

The execution of the instances of the consensus protocol can be parallelized, and not necessarily the execution of the tasks that have to be performed is the current state of the art [3]. Using the consensus-oriented parallelization, consensus instances are assigned to *pillars*, with each one of these being assigned to a specific thread with copies of all the functional requirements to perform the entire consensus protocol and client handling. This

allows the pillars to be separated and far away as possible from one another. For example, if a pillar from the leader replica r receives a new client request or enough requests to form a batch it will initiate a new consensus instance by informing all other replicas via its connections. [3]. Once finished, the outcome can be propagated to the still existing execution stage. Considering that the aforementioned pillars are executed in parallel, these cannot provide total order on their own, meaning that total order has to be enforced by using the instances' sequence numbers before invoking service implementation.

However, one possible extension allows to partition the service implementation, eliminating therefore, the execution stage. Creating Symmetry between the pillars helps not only with evenly distributing the computation over the threads, but also simplifies the system complexity [3]. However, asymmetry was presented in the previously mentioned solution, at least with the execution stage. With traditional state machines relying on deterministic executions to provide service implementations, parallelization of the service execution is therefore prevented [3]. But when parallelization of the consensus protocol is achieved, as soon as a consensus instance is complete, the requests belonging to this consensus instance are now eligible for execution. The problem is that total order has not yet been enforced. Meaning that until the pillar's result is propagated to the execution stage, the total order will not be enforced. [3]. At this point, requests can be pre-processed without losing the order that required to be recreated, this is done by using the pillars to process requests that do not rely on total order. Parsing requests and pre-validating them is work that a thread can do in order to save time. With client connections being handled by the pillars themselves, a the execution stage will not send replies itself, but instead delegate them to the pillar maintaining the connection with the specific client. In order to the alleviate the execution stage even further, a service implementation can carry out only the tasks within total order that are essential to ensure consistency, i.e when the request modifies state, leaving the the preparation of the final returns and replies to the pillars responsible for said connections [3].

Capítulo 3

Con-BFT's Protocol

Con-BFT is a new Byzantine-fault tolerant system built on top of BFT-SMaRt that uses Intel SGX in order to guarantee privacy, integrity and availability of the information that is given by a user and all computation that is done using said information. Intel SGX can also be used in order to provide privacy and confidentiality guarantees using an Intel SGX enclave and its sealing key. By using an Intel SGX Enclave it is also possible to ensure equivocation prevention, i.e preventing replicas from sending conflicting mesages to different replicas within th system, since Intel SGX can be used to create a trusted subsystem to attach message certificates.

3.1 Ensuring Privacy

By using an Intel SGX enclave, it becomes possible to protect the confidentiality of the information and computation done inside said enclave, considering that Intel SGX enclaves can be used to isolate the enclave's code and data from the outside environment and code [11]. Therefore, by creating a system where all sensitive computation and handling of information is shifted towards the inside of an Intel SGX enclave, it becomes possible to ensure obfuscation of client-driven information. For any given client request R, after being subject to total order, can have its sensitive computation handled inside of the enclave, using what Intel defines an enclave call (ECALL), which can be used to guarantee that all information given to the enclave is not visible outside of the enclave, even if said information needs to be stored outside of the enclave, as shown below.

Since Intel SGX enclaves contain what Intel defines as Sealing keys, i.e enclave-specific keys that are derived from the processor's key, these enclaves are able to ensure that information can be encrypted with these Sealing keys in order to be stored outside of the enclave. This enclave-specific primitive is known as sealing. This means that only the enclave that sealed a piece of information will be able to reveal it, allowing for reliable long-term storage, all whilst being protected by the symmetric key obtained inside

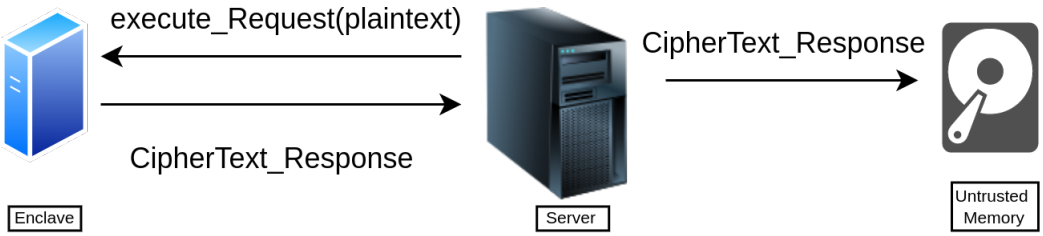

Figura 3.1: Con-BFT Storage of client-Information

the enclave with Intel's sgx-specific instruction $EGETKEY$ [11]. As such, as long as the information safely reaches the trusted environment, i.e the enclave, there is a safe assumption that the information will stay private, since all outside access to enclave data and code are protected. This means that information can be safely decrypted inside of the enclave, since the enclave allows confidentiality of information both inside and outside said enclave.

Therefore, by using Intel SGX, BFT-SMaRt's functions can be slightly altered in order to ensure this required privacy. For the development of this system, a simple Key-Value store was used:

- Put Operation. By including an Intel SGX enclave, the system can, through an ECALL, request the enclave to decrypt the incoming information, and seal it using the enclave specific key, which would force an attacker to physically attack the CPU in order to retrieve this information.

- GET Operation. By including an Intel SGX enclave, the system can do the exact opposite of the Put Operation, unsealing the information stored in unsafe memory and encrypting it with the given AES-Key for the connection being used at the moment.

3.2 Performance of Consensus Algorithms

By using Trusted Execution environments, it becomes possible to use trusted counters in order to reduce the number of messages required by a fault-tolerant system in order to achieve consensus. If trusted counters are introduced inside an Intel SGX Enclave, faulty replicas will not be able to falsify message sequences, due to being possible to associate each message with a verifiable HMAC and unique ID [4, 11]. This means that by using an Intel SGX Enclave to validate all messages that are sent between replicas, the number of required messages to ensure consensus can be reduced, due to wrongful messages being detected and ignored. These trusted counters are able to create and validate HMAC and counter values using previously exchanged keys, which then other replicas can verify

using their own enclave's trusted counters.

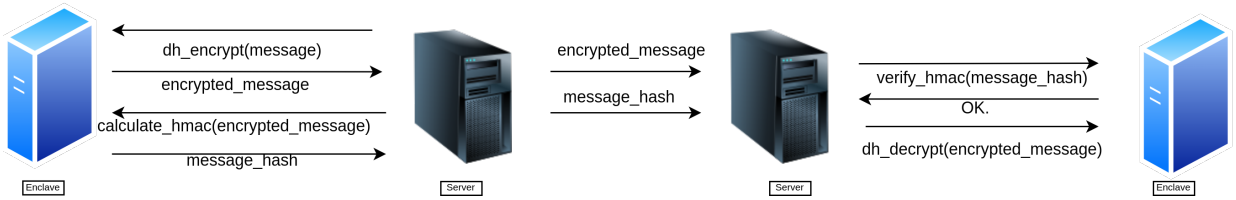

Figura 3.2: Con-BFT Enclave Integrity Guarantees

By using this trusted hardware in order to attach message certificates for all replica connections, it becomes possible to ensure that a faulty replica can no longer send conflicting information to different replicas with the objective of introducing wrongful information to the system [4]. By introducing a MAC (Message authentication code) with a symmetric key created inside the enclave or encrypting information with Galois-Counter Mode (AES-GCM), this means that information sent between replicas contains integrity guarantees. This means that in the case of a MAC, any enclave that contains the symmetric key used for the calculation of this MAC could use said symmetric key to verify it, or in the case of the AES-GCM encryption, any enclave that contains the encryption key, can therefore use said encryption key for the decryption and verification of the information. As such, by having a message sender use a MAC algorithm with a secret symmetric key and a message m, a trusted MAC tag will be produced that cannot be changed, since only the trusted environments contain access to these symmetric keys, which can then be used to reconstruct the MAC and verify the integrity of the message [11, 4]. This means that whenever a leader l proposes a request, it will need to either produce a MAC using its own symmetric key, or encrypt the information using the Galois-Counter Mode (AES-GCM) inside the enclave , which means that in either scenario, in no point will there be a visible key outside of said trusted environment, i.e the enclave. This means that by using trusted environments, it is possible to implement equivocation prevention, which remained as the reason that consensus algorithms required $3f + 1$ replicas in order to guarantee consensus[4]. This allows the $3f + 1$ replica requirement to drop to a $2f + 1$ replicas in order to guarantee consensus, only requiring the client to await a total of $f + 1$ necessarily correct messages [4, 3]

3.2.1 System Initialization

For the system to be initialized, a configurations file will have to be produced by the developer. This file will contain how many replicas r will need to be initiated for any operations to be executed by the replicas, and information required for replicas to connect to each other, like BFT-SMaRt. Each replica will then initiate its own Enclave, with these enclaves calculating Diffie-Hellman parameters for key-exchange between them as shown in

3.3.

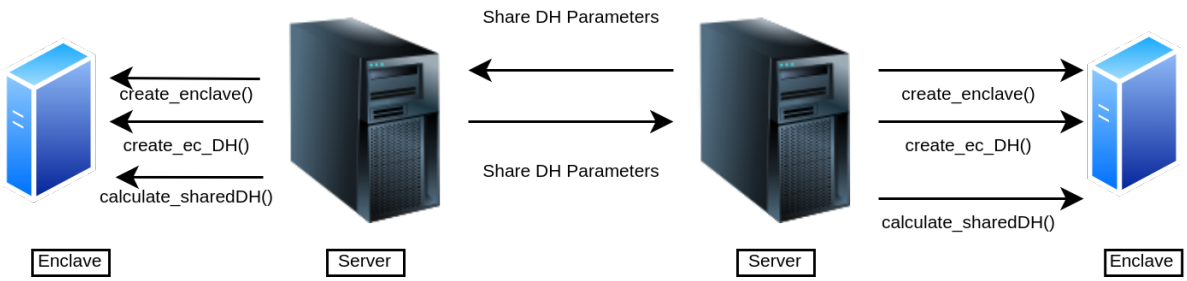

Figura 3.3: Con-BFT Enclave Diffie-Hellman

These keys will then be used for AES-GCM encryption, allowing the enclave to ensure integrity and authenticity for each message produced by an Enclave. Other keys can then be produced and exchanged if at any point Message Authentication Codes (MACs) need to be produced in order to certify any message. This key-exchange protocol allows these replicas to create a secure TLS channel for each $< Replica, Replica >$ pair, which also means that if at any point any Symmetric Key is compromised, only one communication channel is compromised, instead of the whole network. All of these keys are sealed by their respective enclaves in order to be safely stored, guaranteeing integrity, confidentiality and persistence for these keys, in case the enclave is destroyed or the system is abruptly shutdown.

3.3 Request Execution

In order to support client requests, clients will need to sign said requests before sending them for the system to process it, i.e being propagated to every replica r available on the system. This allows replicas to be protected against malicious clients, a feature already supported by BFT-SMaRt. The leader l of the current view will propose requests by attaching an order number o to each of the requests, before proposing them to other replicas in the form of a consensus message $PREPARE$. As soon as a quorum of $f+1$ $COMMIT$ messages are available with the leader-given order number o for the replicas, the replicas that have received said $COMMIT$ messages will start executing them. This is done by passing the requests toward the inside of the Enclave for execution. The overhead for ECALLS whilst not possible to ignore, is not significant to make the system unfeasible. After executing said request, the log file will be updated with the request information sealed with the Enclave key, to ensure that reading the log file is only possible inside the Enclave.

3.4 Data Storage and State Transfer

Considering that memory inside the enclave is limited, the system will need to store sensitive information outside of the enclave, i.e the Untrusted area. For this, Intel SGX contains a primitive known as sealing. By using this primitive, it becomes possible to encrypt information with the enclave's specific encryption key. This allows the use of sealing primitives in order to encrypt information that requires storage, be it in disk or memory, which means that the enclave ensures the persistence of information, even if the enclave shuts down unexpectedly. This application will manage a Key-Value store, allowing it to store information and current values provided by any client. The untrusted area of the system will be responsible for keeping this key-value store intact, by storing sealed information that can only be retrieved inside the trusted area, i.e the enclave. For this, the Key-Value store will support basic operations, such as PUT, GET and CONTAINS:

- **PUT**. When the Enclave reads the request, and will service it, it will start by sealing the value in question, using an OCALL, the enclave will return the pair $< Key, SealedValue >$ for it to be stored in the Key-Value store.
- **GET**. When the Enclave reads the request, it will service it by using an OCALL in order to retrieve the $SealedValue$ that is corresponding to the provided Key. After retrieving said value, it will undo its sealing and encrypt it using the client's symmetric Key, in order to return the correct value to the user.

In order to keep state of the current key-value store, considering that it is impossible to bring a key-value store into consensus, a log file containing all the totally-ordered requests a replica has executed will be used, delivering all the requests the replicas have received by their correct order.
By doing this, a new replica that has joined the system (or a recovering one), can achi-

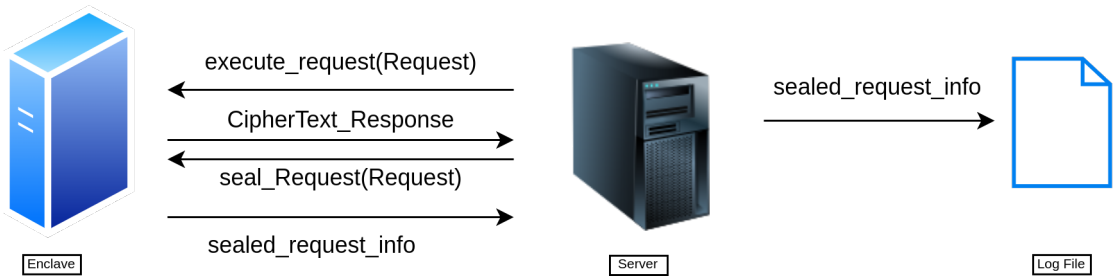

Figura 3.4: Con-BFT Request Logging

eve the same state that the other replicas currently hold. Just like BFT-SMaRt, this would be a combined effort, having the multiple replicas all contributing resources to the state transfer effort [6]. Symmetric keys will also require storage outside of the enclave. Considering that each replica will need to store a singular symmetric key for

each $< Replica, Replica >$ connection, this means that all the system's information that is sent through the network is encrypted using these keys, therefore, storing these keys outside of the Enclave becomes important in order to ensure persistence of information. As such, sealing said symmetric keys for external storage outside of the enclave becomes essential. By creating a simple Hash Table, it would be possible to store $< ReplicaID, SealedSymmetricKey >$ outside of the enclave, guaranteeing persistence. Considering that Symmetric keys are unique to each $< Replica, Replica >$ connection, these would not be considered part of the state-transfer algorithm, considering that either a new or recovering replica would need to create new symmetric keys, ensuring no leakage of information.

For the state-transfer algorithm, BFT-SMaRt's state transfer algorithm was used, changing the structure of the log file. By having request information sealed using the Enclave's unique key and then written in the replica's log file, it becomes possible to restore a replica's state even if the replica unexpectedly crashes, guaranteeing persistence, and confidentiality. As such, when state transfer is required, each replica will retrieve an equal slice of the total number of requests, encrypt them inside the enclave for safe communication, and send them to the recovering replica, which will then execute these requests in order.

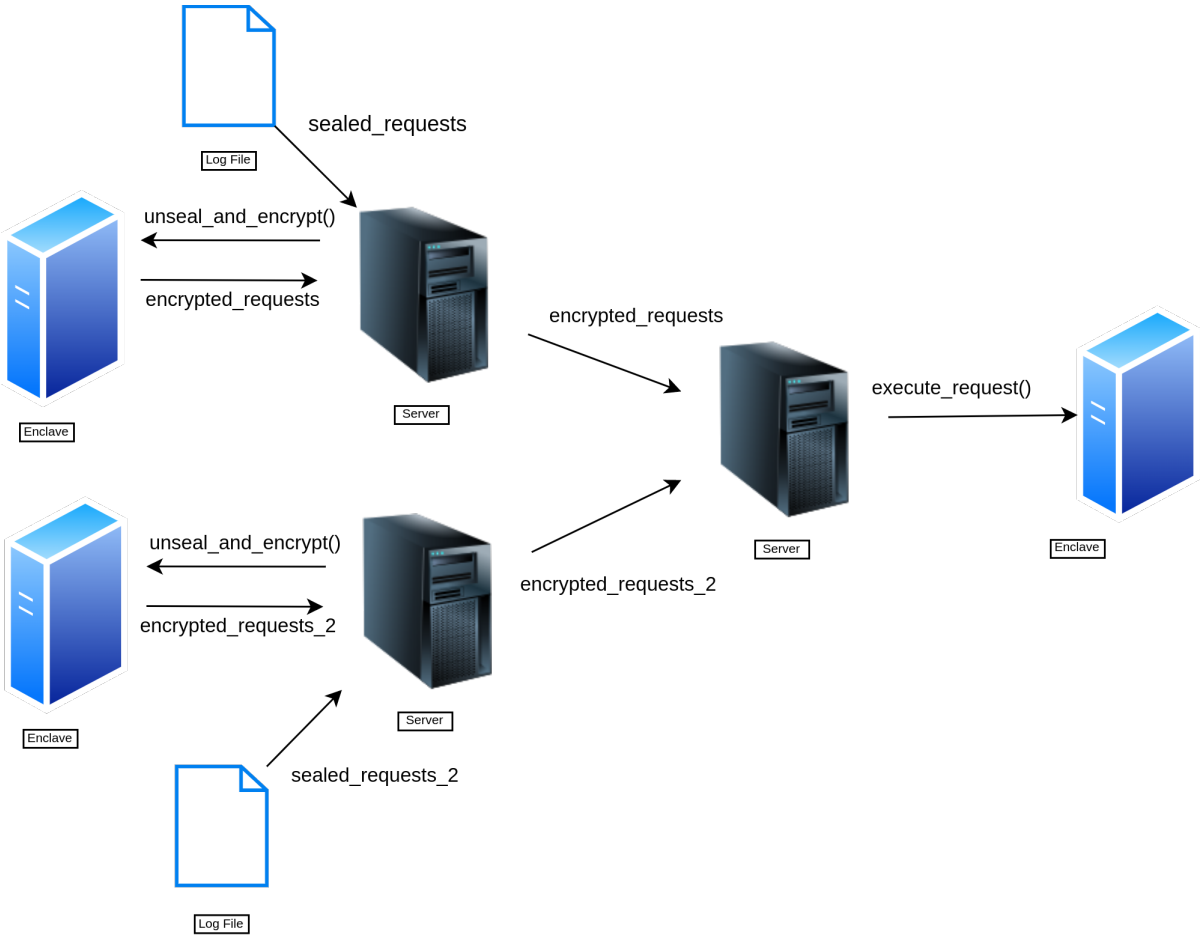

Figura 3.5: State Transfer Algorithm

Capítulo 4

Development

4.1 Java-SGX

Intel SGX is restricted to C or $C++$ applications, meaning that adapting BFT-SMaRt in order to accept $C/C++$ code required the use of external frameworks. The development was based on the Java Native Interface (JNI), which allows native calls to $C/C++$ APIs. The C code was split into trusted and untrusted sections, with the trusted section corresponding to the given Intel SGX enclave. The API was developed with the objective of each replica in the system containing its own Enclave for sensitive computation. With this in mind, the Java-SGX API (which contained the Enclave code and can be called by any Java project) was developed with the functionalities that follow:

1. Sealing and Unsealing of information;
2. SHA-256 Hashing;
3. MAC and HMAC functions;
4. Diffie-Hellman (With both EC and RSA);
5. 256 bit AES-GCM encryption;
6. Swapping between Sealing and AES encryption for communication;

These functions were developed with the goal of attaining confidentiality of user-driven information, with all sensitive computation being executed inside of an Intel-SGX Enclave. Due to the complexity of binding Intel SGX to a given project, especially when a project is not built with the C programming language in mind, this API was built with the objective of easily binding Intel SGX to a Java project, by matching C functions to an interface that can be called by Java functions.

4.1.1 Java-SGX's Architecture

With usability being the most important factor when designing Java-SGX, this means that ease of use had to be the priority during development. By using the Java Native Interface (JNI), the single use of the shell command $javah$, the Java virtual Machine creates a simple header file that can be matched with C files that contain the code written for the functions to be called from the Java class. These C functions would then be able to make ECALLS towards the inside of the enclave, which would then await results from the inside of the given enclave in order for these results to be returned for the Java code.

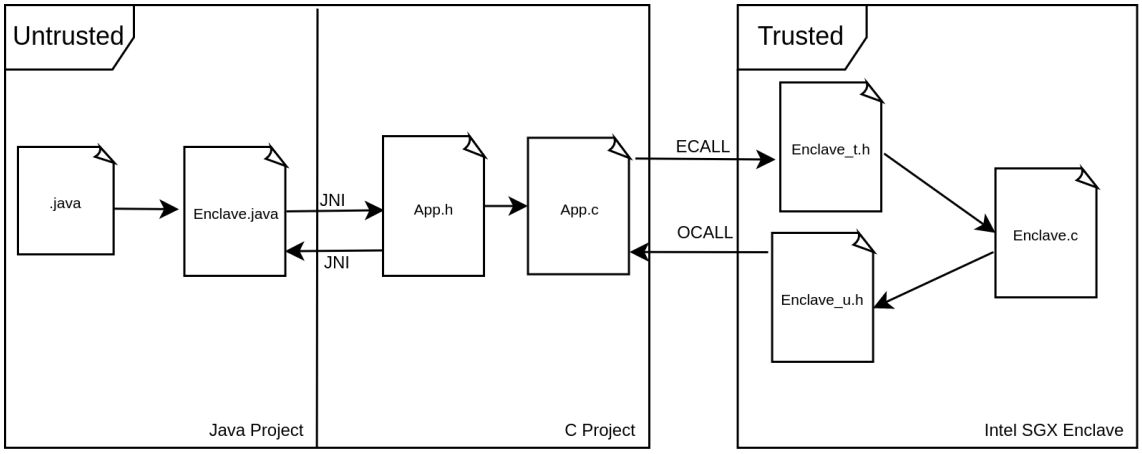

Figura 4.1: Example of Java-SGX Workflow.

As seen above, the JNI allowed for a streamlined workflow, allowing Java functions to make calls to native functions, which would then be responsible for the encapsulation of code that takes care of conversion of data and preparation of the ECALL in order to send information towards the enclave and code responsible in order to send the response back to the Java code.

4.1.2 Java-SGX's Untrusted Area

Due to incompatibilities between Java and C, such as charsets or Objects, the untrusted area was built with the objective of preparing information to be sent for the enclave and allocating memory addresses for enclave responses. These incompatibilites did cause some problems in development, considering that Java uses UTF-16 charset, versus the UTF-8 charset used by the C programming Language, and the modified UTF-8 charset used by the Java Native Interface. Considering that the JNI is considerably old, this did slow development.

As such, the untrusted area was built to serve as a bridge between the Java application and the Intel SGX enclave. By creating function signatures with the *native* keyword,

the javah command allows for the creation of header files that can then be matched and developed in either C or $C++$ files. These files were then compiled into dynamic libraries that could be linked during runtime using makefile.

Listing 4.1: SgxFunctions.java file containing native functions for JNI calls.

```
public class SgxFunctions {
        static {
                System.loadLibrary("Sgx");
        }

        public native int jni_initialize_enclave(int
           enclaveId, byte[] enclaveFilePath);

        public native void jni_sgx_destroy_enclave();

        public native byte[] jni_sgx_seal_info(byte[]
           toSeal);

        public native byte[] jni_sgx_createDigest(byte[]
           toHash);

        public native byte[] jni_sgx_unseal_info(byte[] c);

        public native byte[] jni_sgx_create_RSA_pair();

        public native byte[] jni_sgx_begin_ec_dh();

        public native byte[] jni_calculate_shared_dh(byte[]
            dh_params);

        public native byte[] jni_sgx_aes_dh_encrypt(byte[]
           sealedKey, byte[] toEncrypt);

        public native byte[] jni_sgx_aes_dh_decrypt(byte[]
           sealedKey, byte[] toDecrypt);

        public native byte[] jni_sgx_create_hmac(byte[]
           sealed_hmac, byte[] input);

        public native int jni_sgx_verify_hmac(byte[]
           sealed_hmac, byte[] hmac, byte[] input);

        public native byte[] jni_sgx_generate_hmac_key();

        public native byte[] jni_sgx_swap_sealed_aes(byte[]
            sealed_key, byte[] sealed_input);
```

```
        public native byte[] jni_sgx_swap_aes_sealed(byte[]
            sealed_key, byte[] sealed_input);
}
```

4.1.3 Java-SGX's trusted Area

The enclave requires multiple files in order to create the required dynamic library for linking, each with its own objective:

- The Enclave Definition Language file (.edl), which contains the definition of all *ECALLs* and *OCALLs*. This file can be structured inside two areas, *trusted* and *untrusted*, containing the signature and parameters of each function. This file is then used by Intel SGX, using the *Edger8r* tool to generate edge routines, these routines provide the interface between the untrusted application and the enclave. This tool will then generate four files by default:
  - *t.h, which contains the declarations for trusted code;
  - *t.c, which contains the definitions of the trusted code;
  - *u.h, which contains declarations for the untrusted code;
  - *u.c, which contains definitions of the untrusted code;
- Enclave xml configuration file;
- Enclave linker Script (.lds);

Java-SGX Enclave Definition File and access to the Trusted memory

Due to Intel SGX's restrictions, memory inside the enclave cannot be accessed directly from the untrusted area. As such, in order to retrieve information from the enclave, memory is allocated outside of it and a pointer is given to the enclave to populate with the result of the computation to be done inside the enclave. These enclave functions, both Enclave Calls (Ecalls) and Outside calls (Ocalls) are defined previously inside an enclave definition language file (.edl), inside the trusted and untrusted section, respectively. Which in compile time, the *Edger8r* tool will then create trusted and untrusted header files to be matched to the application file and the enclave file, respectively. This allows for trusted and untrusted code to be called from the opposite section of the code through header declarations.

Listing 4.2: .EDL file for the Java SGX Project with parameters removed for legibility.

```
trusted {

        //Sealing Functions
        public sgx_status_t seal();

        public sgx_status_t unseal();

        public sgx_status_t init_mbed();

        public sgx_status_t destroy_mbed();

        //EC Functions
        public sgx_status_t create_ec_key_pair();

        public sgx_status_t create_ec_dh_shared();

        //RSA Functions:
        public sgx_status_t create_rsa_pair();

        public sgx_status_t compute_rsa_shared();

        public sgx_status_t mbedtls_close_contexts();

        //AES Functions:
        public sgx_status_t aes_dh_encrypt();

        public sgx_status_t aes_dh_decrypt();

        public sgx_status_t swap_sealing_to_aes_enc();

        public sgx_status_t swap_aes_enc_to_sealing();

        //Hashing Functions:
        public sgx_status_t produce_hmac_key();

        public sgx_status_t createDigest();

        public sgx_status_t produce_hmac_digest();

        public sgx_status_t verify_hmac_digest();

        public sgx_status_t create_sha256_digest();
    };
```

Above we have the trusted functions that can be executed inside the enclave. These functions were split into different sections for better readability if needed, right now, these stand as:

- Sealing (Which include both Sealing and Unsealing of information);
- Diffie-Hellman functions (That allow for the creation of Diffie-Hellman parameters for key-exchange using elliptic curves or using RSA primitives);
- AES encryption (Which allow for encryption and decryption of information using the previously calculated keys using the Diffie-Hellman protocol);
- Message Digest and Hashing Functions (Which support SHA-256 Hashing, MAC and HMAC calculation).

These primitives serve as the primary computations behind the necessities that a confidential system requires. If we assume that each server replica contains an Intel SGX enclave, each replica can use the Java-SGX library in order to provide these functions, with no necessity of code written in C/C++.

Java-SGX in practice

In order to simplify the usage of Java-SGX, a .jar file was exported containing the class that imports the native library and exposes the native interface for JNI calls inside the Java code. If the paths are set correctly and the native library is found, all is required is that the Intel SGX Enclave is initiated, and all other functions can be called at any point from the Java code.

After two server replicas connect to one another, these replicas can use the Diffie-hellman functions in order to exchange symmetric keys. For this, each replica will request to its enclave the creation of an Elliptic curve and have its public parameters exported and delivered to the Java application in the form of a byte array so that these can be shared with other replicas so that the Diffie-Hellman protocol is completed. As soon as a replica receives a byte array containing the exported public parameters, it requests the enclave to finish the calculations for the shared key. From then on, these keys can be used for AES-GCM encryption, which guarantees not only privacy, but also integrity. Due to using Galois/Counter Mode, this means that if at any point information is corrupted during communication, this can be verified during decryption. This library also supports Hashing and HMACs if any java project requires it in the future.

By using the Galois Counter Mode (GCM) of encryption for Java-SGX , this means that all encryption stands as authenticated encryption, which guarantees integrity of the encrypted information. This could be protection versus bit-flipping attacks, for example

[16]. By having each replica contain access to an Intel SGX enclave through the Java-SGX library, this will allow replicas to ensure integrity of information by using AES-GCM encryption, without the requirement of Message authentication codes in order to ensure the integrity of a given piece of information.

4.2 Con-BFT

BFT-SMaRt uses Gradle as its automation build tool. BFT-SMaRt's build scripts already contained options for local deploymeent, building multiple copies of the project locally in order to use and test the system. The build scripts had to be changed in order to include the required Java-SGX files, alongside the class dependency that invokes the JNI calls for the project. The changes to the build script force the inclusion of multiple external scripts, mostly for the signing and inclusion of the Intel SGX enclave. In essence, Intel SGX requires all enclaves to be signed in order to be used, which forced BFT-SMaRt to undergo some changes when it comes to replica startup.

4.2.1 System Startup

For a replica to execute its startup, it is required that an Intel SGX Enclave is initiated. For this, the shared libraries have to be correctly matched in the gradle build script. If they have been correctly matched, the java program will be executing a shell call to a bash script that will then use the Intel primitives to sign an Enclave in order for it to be initiated. After a signed shared object (.so) file is created by the bash script, the enclave can finally be initiated. For this, Java-SGX includes a native call that takes the path to the signed shared object file and the given id for the Enclave. This happens since a CPU can initiate multiple Enclaves and requires a unique id for each enclave it instantiates.

After initiating an Enclave, it will then calculate elliptic curve diffie-hellman parameters in order for them to be exchanged with other replicas upon connection. Only after these steps are executed can the rest of the BFT-SMaRt system startup code be executed. From then on, the SGX interface can be used in conjunction with the JNI for any required Ecalls for functions available inside the enclave.

4.2.2 Including Java-SGX in BFT-SMaRt

After the system finishes its usual startup and the various replicas exchange symmetric keys using the diffie-hellman algorithm, replicas can use these keys for communication at any point by requesting said information to be encrypted by the Enclave. The given encryption key is also encrypted by the Enclave's Sealing key. Since this key can only be acquired inside of the enclave, this can be used to ensure the confidentiality of the encrypted information, since it can only be decrypted inside the Enclave, which ensures

that third parties whose objective is to listen in on communications cannot decipher them. If we also take into consideration the fact that the encryption is AES-GCM, we also know that the encryption functions also ensure the integrity of the information, which means that the enclave can be used to ensure both authenticity and integrity of a given message, alongside its confidentiality.

Capítulo 5

Evaluation

After implementing Con-BFT, it became imperative to understand what was the actual impact of an Intel SGX Enclave over the system, comparing it to the BFT-SMaRt version used to build Con-BFT. These tests consists mostly of:

1. Operation Benchmarking in order to evaluate the raw throughput of the system;
2. Calculate System's latency;
3. Performance comparison with the Original BFT-SMaRt;
4. Availability testing, ensuring that the addition of an Intel SGX Enclave does not affect system availability.

5.1 Experimental testing Environment

For the experiments described in the following sections a testing environment was assembled of two SGX-enabled computers, using Intel i5-6440HQ CPU and Intel i5-7400 CPU respectively, with both machines containing 8 gigabytes of RAM at their disposal. Both computers were running version 20.04 of Ubuntu for ease of use, whilst being deployed in a local network. Each of these machines contained two Con-BFT replicas ready for request execution.

5.2 Con-BFT's Throughput

To evaluate Con-BFT a Java Class was created which started a new Thread for every client that was to execute requests. By having a number of Clients C execute a number N of random operations PUT, GET, $DELETE$, $GETKEYS$, each execution of this script executes a total of $C.N$ operations, which allows the calculation of the average time a request takes to execute. Results were organized in sets from 1 to 100 simultaneous

Number of Clients	Number of Operations	ops/s	Total Time (Seconds)
1	1000	126	7.8833
10	1000	179	55.6238
25	1000	434	57.5405
50	1000	820	60.9532
100	1000	1606	62.2664

Tabela 5.1: Operation Benchmarks for Clients requesting 1000 Operations.

clients, all executing N operations ranging from 100 to 1000 operations. The average time for the execution of these requests was then calculated, with the results in 5.1.

The first test was based on the execution of this script for 1000 operations over multiple simultaneous clients, with the results in table 5.1. By calculating the total time required to execute the script in its completion and averaging said time over the total amount of requests allowed for the calculation of throughput by calculating the time taken to execute one operation. The same was then done for 500 and 100 operations respectively, with results in 5.2,5.2, 5.3 and 5.3 respectively.

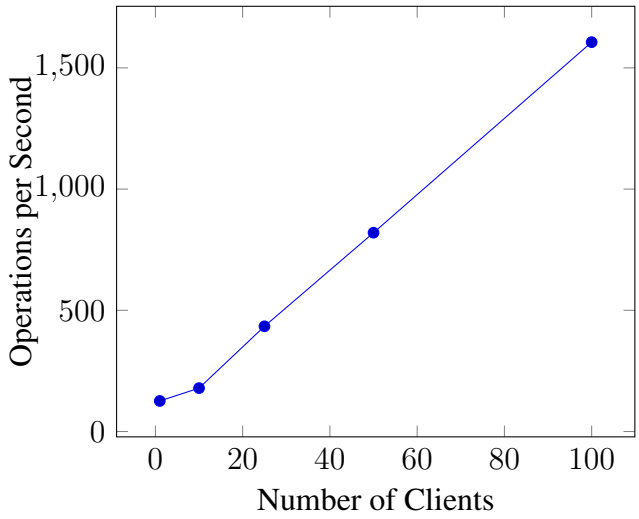

Figura 5.1: Operations per Second for an increasing number of Clients for 1000 operations.

Graph 5.1 was produced so that it is possible to evaluate how the average execution time per operation scales with a larger pool of operations to execute due to the larger client count connected to the system. Similar graphs were produced for different number of operations N 5.2,5.3,5.1.

Number of Clients	Number of Operations	ops/s	Total Time
1	500	123	4,0632
10	500	92	54,1246
25	500	222	56,3287
50	500	428	58,3691
100	500	841	59,4171

Tabela 5.2: Operation Benchmarks for Clients requesting 500 Operations.

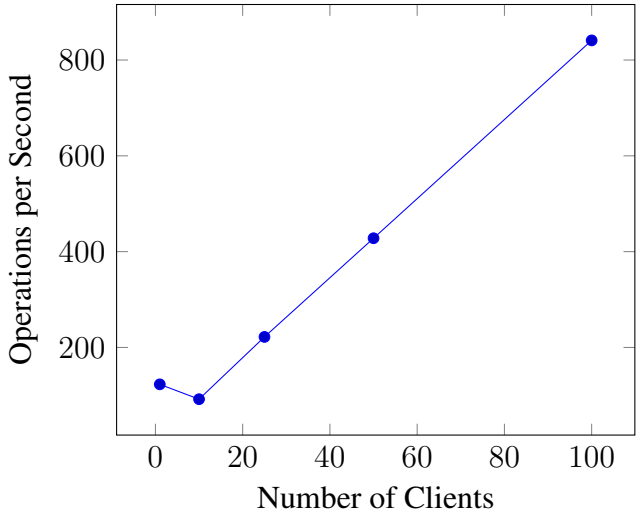

Figura 5.2: Operations per second for an increasing number of Clients for 500 operations.

Number of Clients	Number of Operations	ops/s	Total Time
1	100	88	1.1375
10	100	18	55.4165
25	100	43	57.3486
50	100	82	60.7327
100	100	160	62.3005

Tabela 5.3: Operation Benchmarks for Clients requesting 100 Operations.

By looking at the graphs produced from the execution of the script, regardless of N random operations, the average time per operation remains somewhat consistent, which seems to indicate the system is capable of handling a somewhat higher number of requests from concurrent clients at the expense of higher execution times with a lower loads. This is further emphasized by the total execution time of this test script, in which, on average, answering requests from 100 concurrent clients will take close to 4 seconds longer than 10 concurrent clients, which translates to an extra 4 seconds of execution to deal with a load of ten times the total number of requests.

The graph 5.5 represents the average value for the system's throughput in operations per second. Ignoring the scenario of a single client, which stands as a clear outlier, this

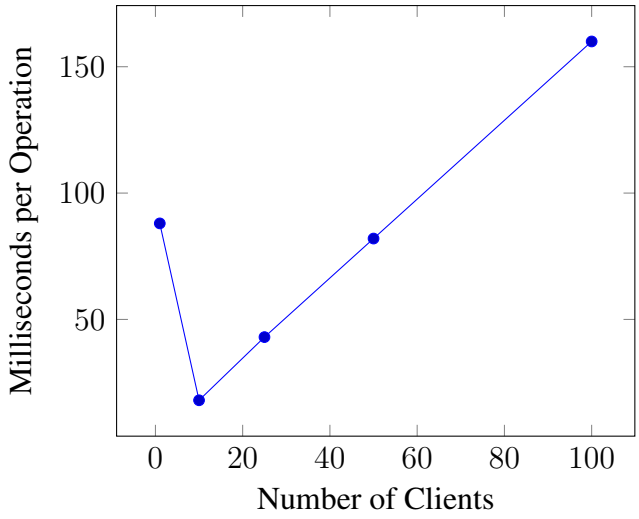

Figura 5.3: Operations per second for an increasing number of Clients for 100 operations.

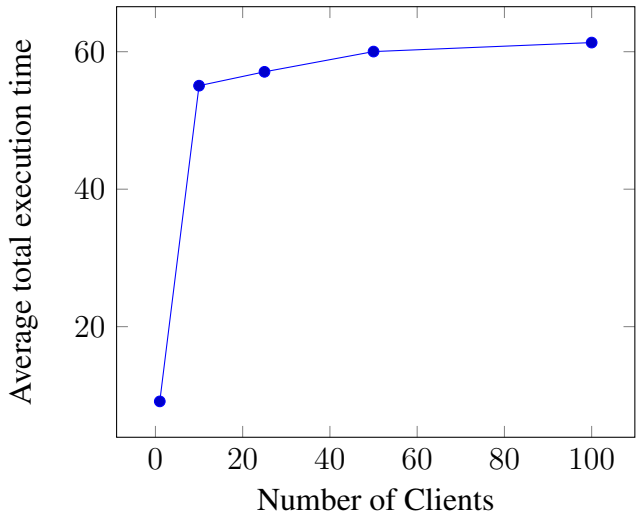

Figura 5.4: Total average execution time per number of Clients.

graph supports the idea that Con-BFT can deal with a larger number of clients, considering that the throughput falls by a very small amount between 10 and 100 active clients.

5.3 Latency

The same script used for throughput testing could be used in order to calculate latency for requests. This was done in order to measure the average time taken by the system to give a response to a given user request.

Latency calculation results were obtained using the aforementioned testing script. And while possible to observe a trend of increased latency when the number of clients increases, by looking at absolute values with increased number of requests per client, the

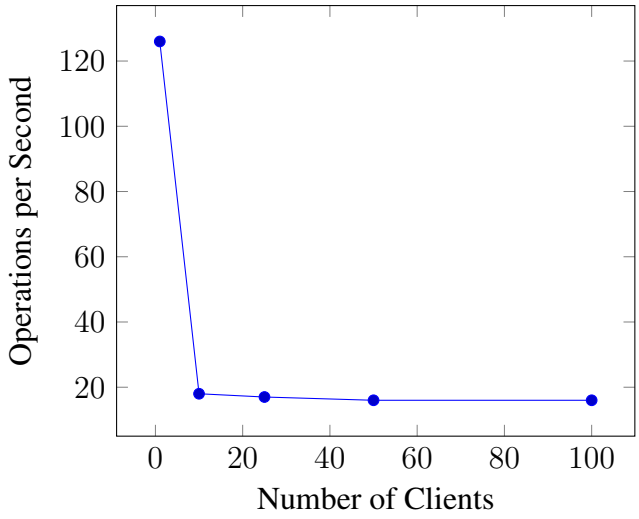

Figura 5.5: Operations per second when executing 1000 operations.

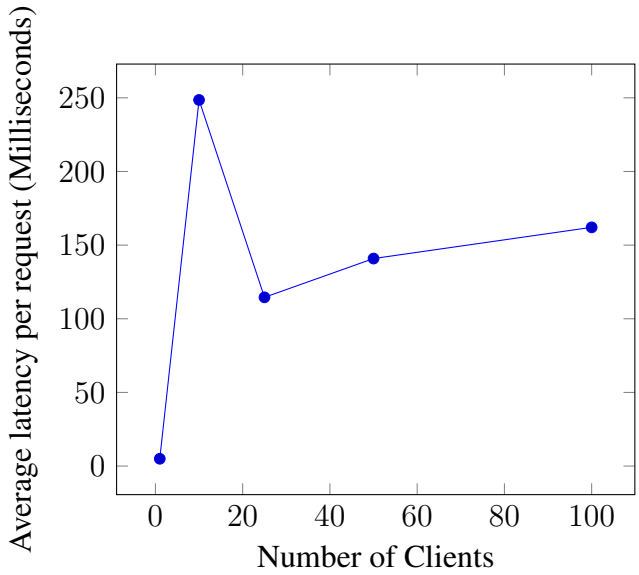

Figura 5.6: Average Latency per request when executing 100 Requests per Client.

numbers support the idea that increased loads on the system also increase overall efficiency, due to taking advantage of BFT-SMaRt's multithreaded capabilities.

It is possible to see that the average latency per request goes down when the number of requests per client increase, something that could be correlated with the system's throughput, since it was shown that the total execution time for the system remained somewhat constant, meaning that the system and especially the enclave are somewhat inefficient for low request counts.

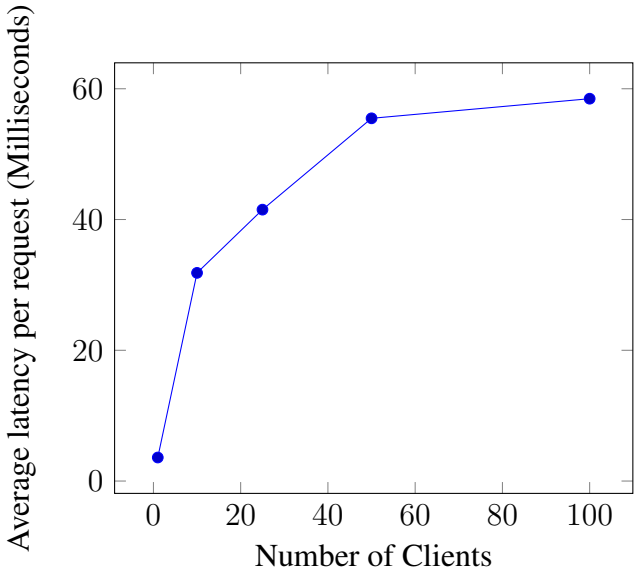

Figura 5.7: Average Latency per request when executing 500 Requests per Client.

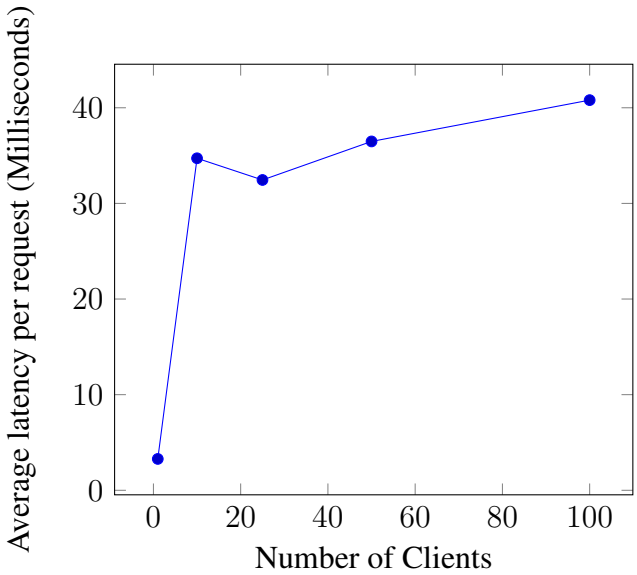

Figura 5.8: Average Latency per request when executing 1000 Requests per Client.

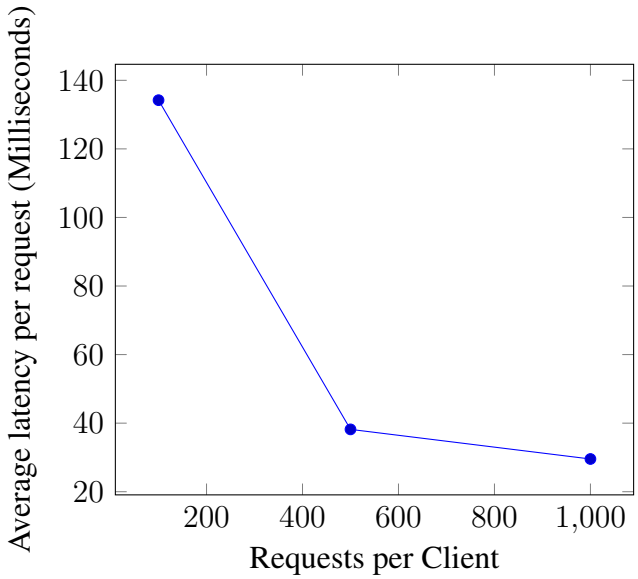

Figura 5.9: Average Latency per request

5.4 Comparison with BFT-SMaRt

Con-BFT adds an Intel SGX Enclave to BFT-SMaRt's protocol in order to add confidentiality guarantees. However, even though the original objective in mind with the Enclave was to add these confidentiality guarantees, the enclave became an essential part of Con-BFT's communication, considering that the AES-GCM encryption that Java-SGX provides with the help of the Intel SGX enclave is used to ensure integrity in messages between the replicas. This means that the enclave will slow down the system heavily, since all communications use it. This means that Con-BFT will stand as an alternative to BFT-SMaRt's protocol, exchanging efficiency with confidentiality.

Number of Clients	Number of Operations	Con-BFT Total Time	BFT-SMART Total Time
1	1000	7.8833	5.2827
10	1000	55.6238	43.3547
25	1000	57.5405	43.9184
50	1000	60.9532	46.7362
100	1000	62.2664	46.6115

Tabela 5.4: Benchmarking for 1000 operations: Con-BFT vs BFT-SMaRt

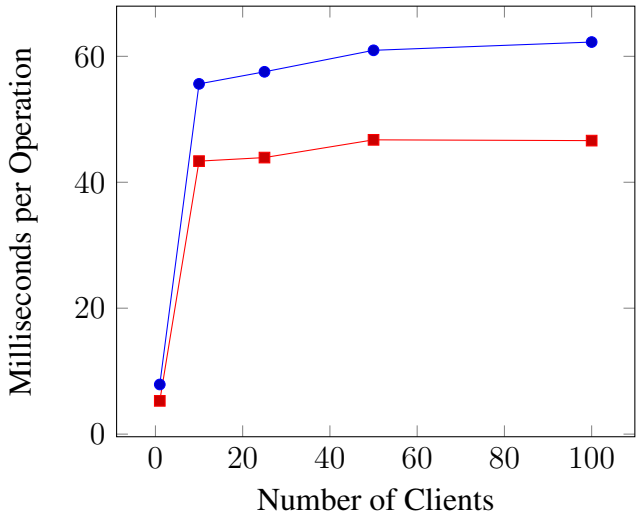

Figura 5.10: Milliseconds per Operation for BFT-SMaRt (Red) versus Con-BFT (Blue).

As we can see in the data displayed in 5.10 and 5.11, the enclave still has a considerable impact in the system's efficiency, which can be mostly, if not completely, attributed to the newly added Intel SGX Enclave. The fact that Java-SGX is not a multi-threaded API, the original BFT-SMaRt will have multiple threads awaiting enclave results. The larger the amount of requests, the larger the difference in execution time between Con-BFT and BFT-SMaRt, due to exactly this fact.

Another overlooked fact during development, is the amount of replicas. If the system manager desires to add more replicas to the system, this could severely affect the system's efficiency, considering that for every replica that is added, every other replica will require one extra diffie-hellman execution to ensure a new secure connection, but every replica will need to encrypt every message sent one extra time, since there will be an extra replica in the system. This means that the system's architecture did not take into account higher numbers of replicas, considering that the somewhat-high overhead that comes with using Enclave Calls to make multiple request to the Enclave.

5.5 Timeouts

One large factor that increases total execution time, are timeouts. For every $N.C$ combination, with N being the number of Requests and C the number of clients connected to 4 different replicas, it was possible to see 0 Timeout executions, in which the total time was far below the calculated average total execution time. As we can see in 5.11, the average number of timeouts increases with the number of active clients, regardless of number of requests executed by these same active clients.

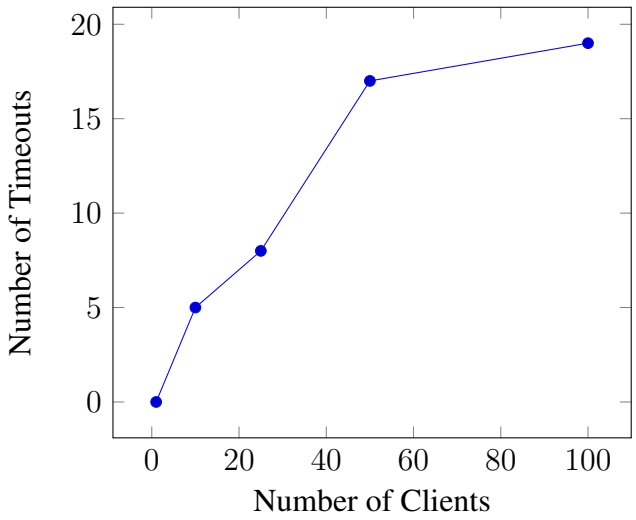

Figura 5.11: Number of Timeouts for 1000 operations for an increasing number of clients.

After testing for both 25 and 50 simultaneously active clients, it's possible to verify that the variance between the number of timeouts with these numbers is the largest. This shows that the system gets more resilient to timeouts as the number of clients increases, as we can see by the number of timeouts by total requests executed, as seen in the graph 5.11 .

The total number of timeouts increases as the number of requests increases, as would be expected of a distributed system dealing with larger numbers of requests. However, when taken into consideration that each client contributes a high number of requests, this

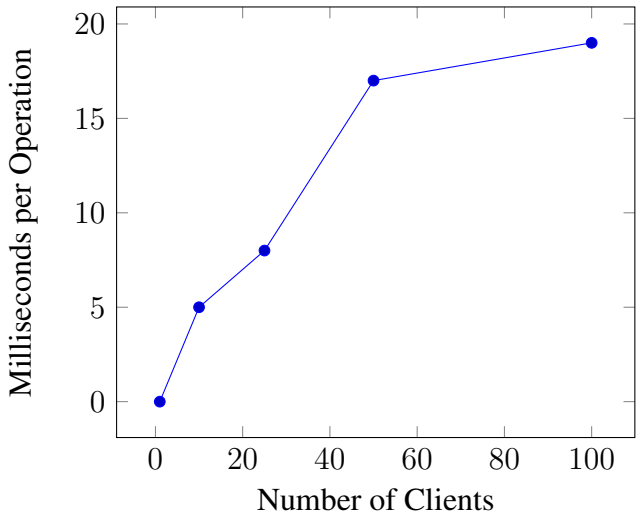

Figura 5.12: Number of Timeouts for 1000 operations for an increasing number of clients.

means that timeouts per client decrease as the number of clients increases, something that was measured in 5.13.

Figura 5.13: Average number of timeouts per number of requests.

As we can see, the total timeouts in relation to the total number of requests, on average, decrease when requests increase. Which allows us to conclude that the system's architecture, based on BFT-SMaRt's architecture, gets more efficient with a higher number of clients, possibly due to its multi-threaded nature, which benefits when used with processors with high number of physical cores, that may take advantage of such an architecture. As we can see in this graph 5.13, just like with the throughput, it becomes apparent that the system is more resilient, since the system will, on average, hit a lower number of timeouts when the number of requests is increased, which means that average availability increases as the number of requests increases. As possible to see in the graph

5.14, average availability increases sharply when the total amount of requests increases, which seems to support that the system points to a higher availability with higher workloads.

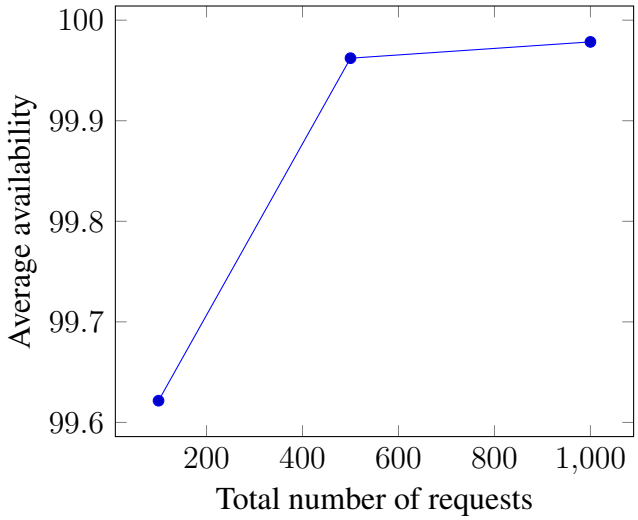

Figura 5.14: Average availability per number of requests.

Capítulo 6

Conclusion

As mentioned previously, keeping information private has become more important in the last few years, especially when taken into consideration breaches of private information for multiple distributed systems, especially in the cloud. A few options were explored, with the option of using trusted hardware being the selected option.

6.1 Conclusion

Right now, when using infrastructure as a service solutions, this means that the developers are outsourcing execution to adversarial environments to which they do not have physical access. This means that there are no guarantees that information is not stolen during request execution, since client information is usually visible in those moments. This means that the system either performs its computations over encrypted information, or uses trusted execution environments that hide this information during request execution. This however, contains its own drawbacks. These systems are closed source and hardware-dependant. This means that the developed solution, Con-BFT, will only work with Intel processors that support Intel SGX. Considering that Intel SGX is not open-source, this means that there is no community to support development and virtually no guides for new developers to get adapted to the technology. Considering that this technology has a high barrier of entry, alongside its dependency of hardware and most importantly, requires trust in the technology, this means that from a practical point of view, it is very unlikely for systems to be developed alongside these trusted-hardware technologies, considering that when they are, they need to build specifically with these in mind. The Java-SGX library built for this project helps alleviate this problem, by allowing developers who have no experience with Intel SGX to match the shared libraries in order to add Intel SGX to their projects. However, Intel SGX has been found to have more vulnerabilities than originally thought $CITATIONS$., which makes the trade-off between efficiency and scalability harder to justify, considering that any adaptation or new functions that developers want to add will require programming with a totally different language with most of its libraries

restricted due to the Enclave's security measures. This means that in order to develop for trusted area, i.e the enclave, a team will be dependant on usually outdated libraries compatible for Intel SGX, or develop a lot of useful functions from scratch. All of these problems led to Intel to no longer support the Intel SGX project, with the 10th generation of Intel processors being the last generation of processors that supports Intel SGX, this means that projects that are built with this technology in mind could be short-lived, as older hardware gets replaced for newer processors. Unfortunately for this project as well, the Java Native Interface was unofficially deprecated mid-development, with Oracle announcing an alternative to JNI in the form of the new project Panama. With all of these factors combined, it's hard to believe that any real and production-use will come for Con-BFT, considering it is now build on top of two unsupported technologies, which force developers into hardware-specific builds.

6.2 Intel SGX Vulnerabilities

In "A Survey of Published Attacks on Intel SGX"[17] by Alexander Nilsson, it is possible to review the published attacks on Intel SGX Enclaves broken down into multiple categories, exposing the extreme difficulty of writing secure software for these enclaves. [17]. In fact, in "Hacking in Darkness: Return-oriented Programming against Secure Enclaves", by Jaehyuk Lee and Jinsoo Jang [15], it was possible to completely "disarm"Intel SGX Enclaves, exfiltrating secure enclave data, bypassing both local and remote attestation, and successfuly decrypting the sealed data. This was done by finding buffer overflow vulnerabilities in secure enclave code. Since the Enclave program runs as a user-level program, it cannot handle processor-level exceptions, meaning that when these exceptions happen, control is handed back to the untrusted operating System in what is called an Asynchronous Enclave exit, which can be used to bypass security of an Intel SGX Enclave. [15].

One popular type of attack versus Intel SGX would be cache-timing attacks [17], of which one of the most concerning was [13], a scientific article where it was proven that the AES-encryption protocol could be broken when running inside an Intel SGX Enclave , as done for this project, in which secure encryption keys were found by attackers [13]. Considering that this project is built on the assumption that keys built and sealed by the enclave are secure, this means that Intel SGX might not be the way to finish a project with confidentialty guarantees.

6.3 Future Work

6.3.1 Parallelization of the System

As previously mentioned, the number of replicas can be dropped to a total of $2f + 1$, with every replica r being composed of equal processing units, which other systems have called *pillars*. These processing units do not share state, and only communicate via asynchronous in-memory message passing, whilst operating completely independently [4, 3]. This scheme of consensus-oriented parallelization supports the ordering of the requests by splitting the available order numbers by the *pillars*, making each *pillar* responsible for the execution of the consensus instances for predefined set of order numbers, decided by a developer-provided formula. The disadvantage of this method is the fact that order number o' can only be executed if all order numbers $o < o'$ have been executed already [4]. This is not without solution however, as proven by hybster, by certifying messages with different trusted counters for different protocol phases, it allows the system to split the work amongst multiple independent pillars.

As such, our system will allow each replica r to create message certificates denoting the ID of the of trusted instance of the processing unit p inside replica r, in the format $C(r(p), ..)$. This becomes enough to prevent faulty replicas from conflicting order messages, since consensus instances will be distributed over the processing units in a predefined way, meaning that the system knows which processing unit is responsible for each instance. As such, an order message from replica r will only be accepted if the attached certificate C was issued by the trusted counter belonging to the processing unit p responsible for the current order number o [3, 4].

By giving each replica r's processing unit p a set of order numbers to process, the parallelization becomes complete. The leader l of a view (Assuming view $v = 0$, for simplicity), decides to propose three requests in parallel with order numbers $o = 0, 1, 2$ in its processing units p_0, p_1, p_2. The leader l will issue $PREPARE$ messages equipped with the aforementioned certificates C, awaiting them to be accepted by other replicas. Other replicas will only accept these certificates if the given order number o is an order number associated with the processing unit p_i and the signed MAC tag is valid. Once a processing unit collects a quorum of $COMMIT$ messages with the original certificate for a certain instance, it will finally send the request to its replica's execution stage, where it will await the execution of all previous requests before being executed inside the replica's enclave.[4, 3]

6.3.2 Making Java-SGX Multi-Threaded

For this system, the Enclave clearly stands as the bottleneck. Based on BFT-SMaRt, Con-BFT takes advantage of a multi-threaded architecture in order to efficiently deliver the results of Enclave computations. These may range from encrypting and decrypting

messages to sealing information to be stored in untrusted memory. However, due to the high amount of threads and connections, the higher the number of replicas, the higher the number of requests that will be made towards the enclave, considering that the Enclave is used to ensure integrity of all communication between the replicas. If we consider that the Enclave operations are executed sequentially, this means that the higher the number of Replicas, the higher these will have to wait for resource time. Adding a Thread pool to work with Java-SGX and the Enclave would allow to optimize this and would help mitigate this problem.

Bibliografia

[1] N Asokan, Jan-Erik Ekberg, Kari Kostiainen, Anand Rajan, Carlos Rozas, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. Mobile trusted computing. *Proceedings of the IEEE*, 102(8):1189–1206, 2014.

[2] Ethan Baron. Google selling users' personal data despite promise, federal court lawsuit claims. Tampa Bay Times, 2021.

[3] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference*, pages 173–184, 2015.

[4] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 222–237, 2017.

[5] Kamal Benzekki, Abdeslam El Fergougui, and Abdelbaki Elbelrhiti Elalaoui. A verifiable secret sharing approach for secure multicloud storage. In *International Symposium on Ubiquitous Networking*, pages 225–234. Springer, 2017.

[6] Alysson Bessani, Joao Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[7] Guilherme Rosas Borges. Practical isolated searchable encryption in a trusted computing environment. Master's thesis, Faculdade de Ciências e Tecnologia (FCT), 2018.

[8] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162, 1984.

[9] Renato M. Capocelli, Alfredo De Santis, Luisa Gargano, and Ugo Vaccaro. On the size of shares for secret sharing schemes. *Journal of Cryptology*, 6(3):157–167, 1993.

[10] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[11] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[12] Fulano, Cicrano, and Beltrano. A paper on something. In *The 7th Conference on Things and Stuff (CTS 2009)*, Lisbon, Portugal, May 2009. Accepted for publication.

[13] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[14] Mireya Jurado and Geoffrey Smith. Quantifying information leakage of deterministic encryption. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 129–139, 2019.

[15] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, 2017.

[16] David McGrew and John Viega. The galois/counter mode of operation (gcm). *submission to NIST Modes of Operation Process*, 20:0278–0070, 2004.

[17] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *arXiv preprint arXiv:2006.13598*, 2020.

[18] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[19] Dominic Rushe. Google: don't expect privacy when sending to Gmail. The Guardian, 2013.

[20] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[21] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[22] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[23] Joao Sousa and Alysson Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference*, pages 37–48. IEEE, 2012.

[24] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. Cobra: Dynamic proactive secret sharing for confidential bft services. In *submission.*, 2021.

[25] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.