Master's Thesis

Resilient Byzantine Fault-Tolerance Using Multiple Trusted Execution Environments

Markus Horst Becker, B. Sc.

1. October, 2021

Institute of Operating Systems and Computer Networks
Prof. Dr. Rüdiger Kapitza

Supervisor:
Ines Messadi, M. Sc.

Statement of Originality

This thesis has been performed independently with the support of my supervisor/s. To the best of the author's knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text.

Braunschweig, 1. October, 2021

Abstract

Agreement protocols enable a number of connected machines to offer a service while tolerating some faulty members. The goal is to keep liveness and safety properties for as many faults as possible. Considering Byzantine Fault Tolerance at least $3f + 1$ machines are needed to tolerate f faults. Through trusted execution technology parts of code can be protected against certain attacks but we recognize that novel attacks and bugs will always present a weakness. We aim to provide a resilient solution for applying trusted execution technologies in this stronger fault model by splitting the PBFT protocol into separate compartments. By compartmentalizing an agreement protocol we split the trust and power of each compartment, limiting the effect of a faulty compartment, and withstand more total faults. Our SplitBFT protocol allows for faulty compartments on all replicas, provided that at most f of any type of compartment are faulty across all replicas. We design and analyze SplitBFT based on PBFT split into three compartments. Our evaluation shows that in real-world deployments the compartmentalization add relatively small overheads and the gained integrity and confidentiality come at a low cost.

Mr. Markus Horst Becker
Matriculation Number : 4808448
Email-Address : markus.becker@tu-braunschweig.de
Course of Studies : Master Informatik

Task Description of the Master's Thesis

Resilient Byzantine Fault-Tolerant Using Multiple Trusted Execution Environments

Robuste Byzantine-Fehlertoleranz unter Verwendung mehrerer Trusted Execution Environments

assigned to Mr. Markus Horst Becker.

Motivation

Byzantine fault-tolerant (BFT) systems can tolerate arbitrary faults, ranging from system crashes to attacker-induced arbitrary behavior. In classical BFT protocols like *PBFT*, $3f + 1$ replicas are needed to address f Byzantine faults. Hybrid BFT protocols, on the other side, use a small trusted subsystem that typically leverages trusted execution environments (TEE). It is assumed that it behaves correctly and can only fail by crashing, e.g., by implementing a rather simple trusted monotonic counter. As the protocol can use this trusted counter to prevent *equivocation*, i.e., a replica cannot send two conflicting messages to different participants, the number of replicas can be reduced to only $2f + 1$.

However, while with such a hybrid fault model the use of TEEs minimizes resource usage and improves BFT protocols' performance, it also raises concerns. Indeed, vulnerabilities have been identified in TEE-secured applications that may be exploitable by an adversary. Weichbrodt et al. [1] shows that synchronization bugs allow an attacker to gain control over an enclave. In a BFT setting, such bugs may corrupt an enclave of a trusted subsystem and break the system assumptions, e.g., a trusted subsystem can then send conflicting messages or return malicious execution results.

As part of this thesis, we intend to explore a different direction. Instead of assuming that code that is protected by a TEE can only fail by crashing – the protected code can arbitrarily fail but only to a certain extend. To do this, it should be explored how a BFT system can be partitioned in multiple TEE-protected compartments. These compartments should be in the position to validate their inputs analogous to quorum decisions of a classical BFT protocol such as PBFT. It is assumed that due to such an approach, a BFT system can withstand more individual flaws in the program code and, as such, become more resilient. However, this will likely only increase the integrity guarantees and do not influence or improve the system's liveness.

[1] Weichbrodt et al. "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves " Proceedings of the 10th European Workshop on Systems Security. ACM, 2016.

Technische Universität
Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Verteilte Systeme

Mühlenpfordtstr. 23
38106 Braunschweig
Deutschland

Prof. Dr.
Rüdiger Kapitza

Tel. +49 (0) 531 391-3294
Fax +49 (0) 531 391-5936
kapitza@ibr.cs.tu-bs.de
www.ibr.cs.tu-bs.de

Date: 01-04-2021

As an implementation basis, the PBFT Themis framework [2] should be used. Ideally, macro tests and continuous integration should be used to validate the framework integration and enable a comparison between a traditional PBFT implementation and the developed TEE-hardened variant.

Task Description

The thesis includes the following tasks:

- Getting familiar with the underlying technologies:
  - The Rust programming language, Themis framework, PBFT protocol, Intel SGX
- Designing a resilient BFT framework using Intel SGX
  - Analysing different splitting strategies to increase the resilience, considering attack scenarios, countermeasures and trade-off between performance and security
  - Provide a description of the partitioning scheme using pseudo code
  - Comparing the different strategies with a traditional hybrid protocol
- Applying the suitable partitioning scheme to the Themis framework
- Evaluate the performance of the implementation regarding throughput and latency
- Evaluate the security of the developed prototype

General Remarks

The duration is 6 months.

A 30 minute final presentation is part of the thesis and will be included in the final grade with 2 credits (Bachelor's thesis) or 3 credits (Master's thesis), respectively.

The remarks regarding student theses at the IBR have to be considered (see `www.ibr.cs.tu-bs.de/kb/arbeiten.html`).

Task description and supervision

Prof. Dr. Rüdiger Kapitza: _____

M. Sc. Ines Messadi: Ines _____

Thesis work

Markus Horst Becker: M. Becker _____

[2] Rüsch et al. "Themis: An Efficient and Memory-Safe BFT Framework in Rust" Proceedings of the 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers. ACM, 2019.

Contents

List of Figures

List of Tables

1. Introduction

Requirements on modern software systems often include high availability, integrity and confidentiality. Availability can be easily improved by running a system distributed across multiple machines. If a few machines fail the remaining machines can keep processing requests. This can be seen in paradigms like micro services, containerization, edge-computing and cloud computing in general. Availability is sustained as long as failures do not spread across the entire system. Typically replicated, these distributed systems require consistency depending on the application. Consistency can be achieved by ordering all incoming requests. By totally ordering requests the system gains strong consistency [32]. In safety critical applications, however, the possibility of a faulty or exploitable member of the distributed system needs to be tolerated [32]. State machine replication (SMR) is a general approach to achieve consistency and fault tolerance for a distributed system [40]. It is applied in several systems such as zookeeper [20, 31, 8, 1]. In those solutions an agreement component is added to gain fault-tolerance. The SMR system and agreement protocol can be specialized for the fault model to increase integrity and confidentiality. Depending on the application and fault model, different kinds of faults are expected and protected against. Byzantine faults, for example, allow members of these protocols to behave arbitrarily bad. Tolerating Byzantine faults requires complicated protocol steps to withstand these malicious members trying to disrupt the service. The protocols used to agree on the order of requests and reach consensus in these conditions are called Byzantine fault tolerance (BFT) protocols [32, 8]. BFT protocols have been studied since Lamport's Byzantine Generals Problem from 1982 [32, 8]. Variants of BFT protocols have been developed to become faster, more flexible, resource-efficient, safer and more specialized [5, 26, 10, 33, 38]. BFT protocols recently gained interest due to their relevance as a consensus algorithm in permissioned blockchains. They can also be used in other secure transaction applications in which linearization and safety are important. As such, more possible applications are being found in industries like logistics, retail and power as well as the administrative domain [21, 18, 30]. Setting up these systems is a technically difficult task requiring the balance of safety and performance. This has lead companies to offer Blockchain-as-a-Service (BaaS) running on their infrastructure [35, 11]. The advantage is that a user does not have to implement the complex back-end operations necessary, and the provider benefits from the economy of scale. Using one of these offerings introduces a large risk by requiring trust in the provider.

Removing the need for trust can be partially achieved by enabling trusted execution in the cloud [5]. Hardware-based trusted execution provides isolated and protected environments that removes the need to trust the provider. Trusted execution environments (TEEs) can be used to supplement BFT protocols in the cloud to protect against the provider abusing their local privileges and access. However, due to the complexity of agreement

protocols, further work is required to ensure a practical application of trusted execution in the cloud for use with BaaS. Notably, large trusted computing bases (TCBs) present a larger attack surface. While some applications can be put into TEEs entirely, accessing the network or other hardware devices requires the application to be split into a TEE part and an untrusted part. BFT protocols are complex and require network communication. Therefore, BFT protocols need some interactions with the operating system to communicate with other participants. We realize the opportunity to increase robustness for integrity and confidentiality by separating the protocol into trusted compartments on each machine. Reducing the size of the individual trusted compartments reduces TCB size. Protecting the trusted compartments allows fine-grained control over safety-critical application code. We utilize memory access protections on the compartment's secure data to protect confidentiality. However, we do not ignore the possibility of bugs or exploits being present inside of the TEE's code.

This thesis presents the SplitBFT approach which separates BFT protocol components into their own independent compartments. In this thesis we apply the SplitBFT method to the PBFT agreement protocol [8]. For this, we keep the structure and architecture of PBFT but split its phases into multiple TEEs. We remove local trust, even between local compartments on the same machine to allow independent failures of these compartments. This allows us the tolerate more Byzantine faults than typical BFT protocols, as we tolerate up to f faults of each compartment type for integrity. Which replicas these $3f$ faults are spread over is inconsequential for integrity. We also gain stronger confidentiality as all sensitive data is only present in once compartment on each replica. The design and implementation of the SplitBFT protocol is presented. We detail the process behind structuring a complex but robust software system split across multiple TEEs and their interaction with the untrusted software on multiple machines. Furthermore, practical deployments of protocols need robustness to withstand bugs and exploits within TEEs, which we show our compartmentalization improves. We will show this by presenting the gain in safety and confidentiality by splitting the protocol into independent compartments.

We implement SplitBFT in the Themis agreement framework written in the Rust programming language. Compartments are also implemented in Rust using Intel's Secure Guard Extensions (SGX) [14] and the Teaclave SGX SDK [2]. A modular implementation of the compartments and their interface code enables easy diversification, even in different programming languages. While using TEEs in SGX enclaves incurs an overhead, our implementation reduces this and achieves competitive performance.

Running both benchmark and real-world applications we evaluate SplitBFT in a distributed environment to measure the impact of compartmentalization. By tailoring the system to reduce the total overhead added by SGX using modern practices to speed up BFT protocols we will compare SplitBFT with PBFT in Themis.

1.1. Thesis outline

This remainder thesis is structured into the following chapters:

- **Chapter 2** ***Background*** introduces technologies and methods used to design and implement the SplitBFT protocol. The focus is on the BFT-SMR background as well as the TEE mechanism and limitations. A short overview of the Themis framework is given.
- **Chapter 3** ***Design*** presents the design decisions in SplitBFT and informs the following implementation in Themis. We lay out the compartmentalization method for the PBFT protocol. Using a structured approach to obtain the compartments for SplitBFT, we analyze necessary safeguards and boundaries for the TEEs. The messaging, signing and encryption schemes are presented, explained and discussed.
- **Chapter 4** ***Implementation*** covers the realisation of the design in the Rust programming language as a module in the Themis framework. Implementation details that require special attention because of performance and safety implications or limits of the TEEs are discussed. This chapter also contains worked examples of selected components of the larger system.
- **Chapter 5** ***Evaluation*** shows the benchmark results of the final system as well as preliminary benchmarks informing implementation decisions during the design and implementation process.
- **Chapter 6** ***Related Work*** outlines relevant previous and ongoing research and other publications in this field. We use this opportunity to outline the main differences of past research with the new SplitBFT approach.
- **Chapter 7** ***Conclusion*** closes this thesis by summarizing SplitBFT and the state of its implementation within the Themis framework and gives an outlook on further possible work using the presented methods.

2. Background

In this chapter we present the technologies and resources necessary to pursue the aim of this thesis. As outlined in the introduction, this thesis aims to combine agreement protocols with multiple TEEs to improve safety and confidentiality using a practical approach.

We begin with the motivation behind agreement protocols and their applications. To reason about the effects of BFT protocols the fault model needs to be examined. Then we define Byzantine fault tolerance from the perspective of our stronger fault model. Finally, this thesis makes use of SGX TEEs which have their own features and limitations important for the remainder of the thesis.

2.1. Goals of Agreement

Modern applications often require large amounts of storage and compute power. Users expect services to be accessible all the time and from around the world. Instead of building arbitrarily powerful monolithic machines and applications, the industry has moved to spread large applications across multiple distributed weaker machines. This makes it easier to react to changing requirements, as seen in cloud computing. More machines are added to the pool if required by the demand or application. By ensuring machines fail independently the availability of the system is also improved. When splitting the execution of any application across multiple separated machines it is a common goal to achieve consensus or agreement. This agreement is used to ensure consistency. One way to reach consistency is to order all requests across all machines. This is linearizabiliy. Ideally, the distributed system behaves identically to a single monolithic deployment. Members of an agreement protocol can guarantee that under good conditions the majority of other members will agree with them on the current order of requests and their execution. This can be used to construct a replicated service using deterministic state machines sharing the same state. Most replicated services have three main goals, or attributes, to maintain and preserve:

1. **Availability**: The replicated service is reachable by its users and accepts incoming requests. Deadlocks need to be avoided. The delay between a request and its response, and the throughput of the system may vary but never completely stop. The system is guaranteed to reach a state in which it is able to continue processing requests at some point. This means liveness is preserved.

2. **Confidentiality**: The contents of messages including requests and responses between the client and service are not leaked to a third party. The usual method to increase confidentiality is applying encryption schemes. We assume that strong

cryptography cannot be attacked by an adversary directly. This means the adversary cannot guess keys or shared secrets, or reverse the message from its ciphertext. The adversary is also not able to fake a signature or find a collision in a cryptographically secure hashing function. However, if an adversary has taken over a participant and can read its secrets, messages in the network can be decrypted or forged using the leaked secret. This lets the adversary impersonate the compromised participant. This is the case for both asymmetric and symmetric encryption, the scope of the consequences is proportional to the number of members sharing the secret. In practise, both asymmetric and symmetric encryption are used in different situations, using asymmetric cryptography like RSA or symmetric encryption like message authentication code (MAC) for 1-to-1 channels [7].

3. **Integrity**: Messages and the shared state are tampered-proof. This is usually achieved using cryptographic hashes and signatures. In the context of TEEs memory access protections also aid in ensuring state is not tampered with by a third party. Message integrity also includes the guarantee that messages cannot be forged. Even members of the system have to be unable to create a message that looks like it comes from another member. This is necessary to be able to find a true majority for achieving consensus. The common solution for this is also asymmetric cryptography for digital signatures or MACs. It is important to note that a tamper-proof message can still contain incorrect or misleading data if the sender is malicious.

These three properties, combined with agreement allow a distributed system to outwardly behave as a single fault-tolerant application.

2.1.1. State Machine Replication

SMR describes a systematic method of building a distributed system [40]. Each member is modeled as a deterministic state machine. With the same initial state and deterministic operations, all machines can be synchronized. For this to work it is important to ensure that all replicas perform all operations in exactly the same order. This method can be used to define a very general mechanism for efficient replication, checkpointing and recovery.

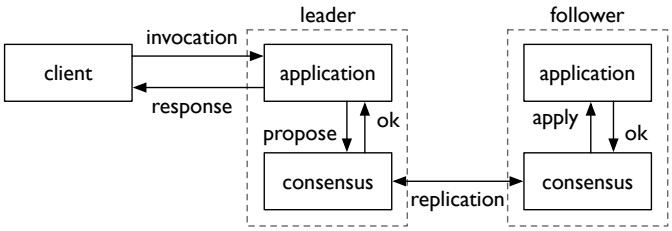

Figure 2.1.: Overview of example State Machine Replication system [1].

Another advantage of SMR systems is that already deterministic applications can be replicated using SMR with minimal changes [1]. Figure 2.1 shows an example architecture with an application and the necessary consensus protocol for replication. In general, the architecture of SMR systems consists of a small wrapper around the deterministic application and a consensus component on each replica. Clients send their request, as usual, to a single machine called the leader. The leader then distributes (replicates) the request to all other machines, called the backups using the consensus component. After successful linearization using an agreement protocol, the requests are executed deterministically on all machines and the client receives the responses. Because the initial state and the order of deterministic operations are identical all machines stay synchronized.

A large problem in SMR systems are failures on the replicated machines. This has created a field of study on how to deal with faults in SMR. Depending on the fault model, SMR systems must have different safety precautions to operate correctly. The strongest type of fault these systems may need to tolerate are Byzantine faults.

2.2. Byzantine Fault Tolerance

Byzantine fault tolerance describes a safety property of a distributed system. Members of the system are required to reach a consensus even though a certain fraction of participants are allowed to behave incorrectly [32, 8, 47]. BFT can be combined with SMR, creating BFT-SMR systems [40], combining the distribution attributes of SMR with the safety of BFT. In these systems the consensus protocol is a BFT protocol. Behaving incorrectly in BFT protocols is defined as the participant experiencing a Byzantine fault that leads to arbitrary behaviour. A faulty malicious member could pretend to behave correctly or lie about the information it has received from others. This makes BFT one of the most general fault models. Byzantine faults are the hardest type of failures to tolerate. The Byzantine Agreement Problem originated from the Byzantine Generals Problem proposed by Lamport, Shostak, and Pease [32]. Broadly, the Generals Problem is about communicating across an unreliable channel. The Byzantine Generals Problem extends the problem to allow participants to exhibit faulty behaviour. This is often personified as the Generals acting maliciously, or in the worst possible way. The generals cannot trust each other or their communication channel, but have to try to agree on a decision by sharing their *opinion*. They, importantly, have to be able to form a majority of *loyal* participants as long as this majority exists. The assumption about their communication channel is, in practice, that sent messages do arrive, but can be arbitrarily delayed. BFT protocols, as presented here, require participants to be known in advance and authenticated in some form. With an arbitrary, but also finite, amount of authenticated clients. This means that BFT protocols can be used in permissioned blockchains to agree on the state of the ledger.

2.2.1. Fault Model

There are two different fault models commonly used in consensus protocols.

- **Crash fault:** when a process encounters a fault it stops operating entirely. Crashed participants will not take part in any further communication. This crash can be caused by a hardware failure, a software bug, or the network connection being lost entirely. When a node gets taken over under crash fault assumptions the adversary will be unable to make the node perform any other actions. In practice, this requires further assumptions on the adversary or additional protections. Some hardware, for example, is often assumed to only fail by crashing [29], instead of sending arbitrary data. If we are also only considering *accidental* adversaries, turning off the power for the datacenter, for example, presents itself as a crash fault to the system. Some systems that do not expect to be targeted by malicious attackers at all can consider it sufficient to be protected against crash faults. The lower stakes allow a more efficient implementation. However, crash fault tolerant protocols open themselves up to novel attacks on the trusted subsystem [45].

- **Byzantine fault:** when a process encounters a fault it can behave arbitrarily. The Byzantine fault model is more general than the crash fault model. Faulty members can continue operating as part of the system, but the behaviour can be adversarial. As such, faulty members can leak information, forge messages or tamper with messages on the network. The exact abilities of a faulty node is dependent on the deployment and possible additional safety measures.

As Byzantine failures allow a participant to behave arbitrarily, it is a useful fault model for computer systems, which can be hacked or interfered with. Faulty behaviour includes, for example, delaying communications or lying about information concerning others. Faulty members are also able to equivocate (see 2.2.3), presenting different information to different participants. The goal is to preserve consensus even if the faulty machines do everything in their power to try to break consensus of the correctly behaving machines.

A replica encountering a software bug or hardware failure is also considered faulty, as further operations might also be arbitrarily bad as a result of the failure. This leaves the correctly behaving participants, which are not taken over by an adversary, and have not encountered a software bug or hardware failure. Software bugs or flaky connections might still be present on correct replicas, just have not caused a fault yet.

BFT protocols are characterised by the number of faults they can tolerate and still reach a consensus within the correctly behaving participants. In a system with n members, where a majority is required to reach consensus a BFT protocol tolerates f Byzantine faults. For the general Byzantine Problem the condition $n \geq 3f + 1$ holds [32]. This formula states a minimum number of replicas necessary to take part in the BFT communication to tolerate up to f failures.

Through the use of additional new technologies, like trusted execution (see 2.4), hybrid models are protected against some faults. Hybrid and crash fault tolerant protocols may only need $n \geq 2f + 1$ machines to tolerate f faults.

- **Hybrid faults:** offer better performance than BFT protocols at an overhead comparable to crash-fault tolerant protocols [5, 26]. This is achieved by assuming that Byzantine faults are avoided in a certain parts of the protocol. A trusted subsystem might only be expected to fail by crashing. Proving the correctness of the trusted subsystem is often required and very difficult. Furthermore, for correct operation the implementation of the protocol must be void of exploits.

2.2.2. BFT Stages

Correctly functioning state machines will reach the same decision given the same input, and they need to be able to share their decision. The need to be able to form a majority. If a replica is not faulty, then it will also send the correct response, and behave correctly throughout all the BFT stages.

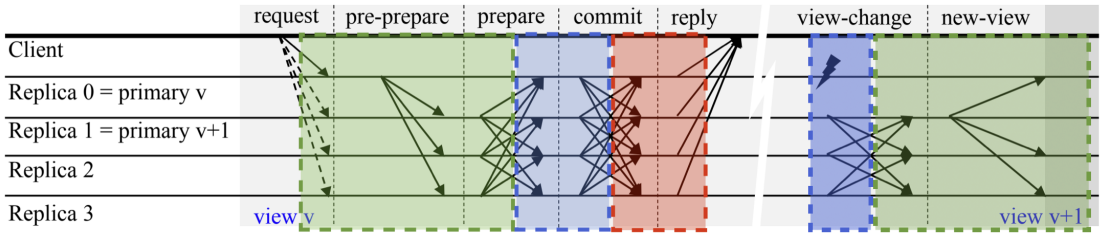

Figure 2.2.: PBFT protocol steps [38] annotated with proposed compartment type: 1. green = Preparation, 2. blue = Confirmation and 3. red = Execution

Figure 2.2 shows all stages part of common agreement stages [32], including the separation proposed later in this thesis. The protocol usually composes multiple inner protocols for certain tasks. First the ordering protocol is used to order and agree on requests, but checkpointing and view-change protocols are commonly used to provide garbage collection and maintain liveness. Additional protocols can be included to, for example, achieve rejuvenation of previously faulty machines.

Ordering

The communication in BFT protocols often occurs in two rounds. First, the agreement stage. After a request from a client is received, the request is broadcast across all replicas and the order in which the replicas are processing the outstanding requests is agreed upon by all replicas. This is the ordering part of BFT protocols. Ordering is necessary for linearization. It ensures that a majority of correct machines agree on the order in which requests are to be executed. Then the requests are executed in the so-called execution

stage and each replica sends the result of its computation back to the client. The PBFT protocol uses three phases ⟨`pre-prepare`, `prepare`, `commit`⟩ to achieve this, as shown in figure 2.2.

Checkpointing

In addition to ordering and execution of requests, additional functions of BFT protocols are necessary to keep real-world deployments from running out of memory. Checkpointing provides an opportunity for garbage collection as it marks a clean cut in the protocol at which no references or memory of old data is necessary. It is also required because the network is often assumed to be unreliable and checkpointing allows correct but delayed machines to catch up again. Checkpointing allows a form of recovery from these network-based failures. This involves the replicas sharing all the necessary data for new replicas to join in, and also creates a state, which allows recovered replicas to continue participating. For this to work a proof that a majority of correctly functioning replicas agree on this state is necessary. These checkpoints are created at regular intervals, usually based on the number of processed requests.

View-change

Another point of failure, which endangers reaching a consensus, is which replica is contacted by the clients. Instead of contacting a random one, the client often refers to a publicly known leader. The leader is one of the replicas which receives the requests from clients and initiates the ordering process. During the execution of the BFT protocol, the replicas can request to change the leader if they suspect the leader to be faulty. Changing the leader is done by changing the current view. Once enough requests to announce a new leader have been collected by a replica it will assume the leader position has been changed. This is called a view-change. The order in which the leader position rotates is usually predetermined.

A faulty leader needs to be dismissed as leader because otherwise it could just delete user requests. Clients who do not receive any response from the BFT members, therefore, must timeout and try another replica or broadcast their request to all replicas [32].

2.2.3. Equivocation

A faulty member of an agreement protocol who is able to send conflicting messages to others is performing equivocation [33]. If an agreement protocol was able to prevent equivocation, the Byzantine fault tolerance would be improved, requiring only $n \geq 2f + 1$ machines for f faults, instead of the more general $n \geq 3f + 1$ [33]. However, current research only allows for non-equivocating BFT protocols using either a (central) trusted party or crash-fault TEEs. As our fault model allows TEEs to contain bugs, those solutions would not be transferable to our model.

In our fault-model equivocation is expected to be possible.

2.2.4. Application

From the perspective of the client, only one message is sent to the replicas. This message contains the request for the application to perform an operation. After the message has been processed by the BFT protocol the application will perform the ordered requests and generate replies. The client will accept a number of replies, at most one from each replica. At some point, the client decides that it received enough responses and finds the majority in the responses. It considers that majority to be the correct result. The condition $n \geq 3f + 1$, then, means that more than $\frac{2}{3}$ of the distributed system behave correctly. The additional correct replica is necessary in case the client misses the last response from a correctly behaving replica. When the client performs the majority voting process on the responses, it will still be able to find a majority of correct responses. This means the client in a BFT-SMR system performs almost identical operations to the non-replicated version. All the client needs to be aware of in addition is which replica to send requests to and how many requests to wait for before voting starts.

This procedure is more complex than finding a consensus when the type of fault the system needs to tolerate is limited to crash faults, as those are generally characterized by $n \geq 2f + 1$, requiring a lot fewer resources when the expected number of faulty replicas is high.

2.2.5. Deployment environment

It is also assumed that the replicas differ in all ways that enable contamination from one faulty replica to the other functioning replicas. Byzantine faults occur one-by-one and replicas, therefore, fail independently.

The lengths to which Byzantine Fault tolerant systems go to ensure independence of faults is great. The largest contribution to ensuring failures are independent is diversification. On the software side, diversification of the source code and the operating system can be applied. On the hardware side, the platform, e.g. network card, CPU and other parts and vendors, as well as the networking itself need to be diversified. The network must be resilient to keep a faulty hardware component from disrupting the entire network.

Diversification can be achieved by using tools to generate different versions from source or writing different version from scratch, either by the same or even different teams of programmers [37]. For third-party software, different versions can be used to limit the chance of a bug or weakness being present in multiple installations. Configuration of software also needs to diversified. Most essentially passwords and other secrets to gain access to the machines need to be diversified. Hardware can be diversified similarly. Where possible, a vendor should only be used once on a single replica. This would both ensure that we do not have to trust the vendors. The problem with this, besides the initial and higher maintenance cost of these highly diverse problems, is availability. Ideally, each replica has its own administrator ensuring independence from the machines of other administrators.

In practice, distributed BFT systems are contacted by outside clients communicating with the leader replica to send his request, the correct replicas of the distributed application then communicate following the BFT protocol, and responses are usually sent from each replica directly to the client. Leaving the client to handle the voting on the responses. In this model the replicas are all networked to each other. To stay independent all machines should not be present in the same physical location.

2.2.6. Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) is a Byzantine fault-tolerant replication algorithm [8]. It has been implemented multiple times [47] and analyzed for correctness [8]. Many BFT protocols begin by being variants or derivatives of the PBFT algorithm to make use of the past research [38, 47].

PBFT operates in phases: pre-prepare, prepare and commit to totally order requests. It tolerates up to f Byzantine faults if there are $n \geq 3f + 1$ total replicas.

The protocol includes checkpointing and view-changes for garbage collecting and maintaining liveness.

2.3. Robustness

While BFT protocols allow to make rigid guarantees they also require very precise fault conditions. Under the assumption that the protocol does not contain any errors, but instead only faults which are part of the tolerance model, we want to increase the likelihood of a practical system to preserve the goals of integrity, confidentiality and liveness. In theory, any faults that are able to happen independently are considered as occurring equally likely. In practise, we want a better understanding of how likely it is for replicated services, like BFT systems, to fail. We recognise that most protocol's resilience is a best-effort, but want to describe and follow a systematic method to increase robustness. Part of the consideration is also the cascade of faults which are hard or impossible to model in a fault model. Diversification already adds robustness to BFT protocols, by making bugs more likely to be consigned to a single instance.

We show that compartmentalization creates stronger encapsulation of bugs and enables us to extend the fault model to allow more replicas and compartments to be faulty as long as enough compartments of the same type are still functional.

Safety describes a protocol's ability to maintain operational qualities, integrity and confidentiality, even when a certain number of faults occur. However, protocols with the same safety requirements, e.g. $n \geq 3f + 1$ for BFT protocols, can show different robustness to faults. Robustness can be increased by verifying or hardening the entire protocol or parts of the protocol against common faults.

2.4. Intel Software Guard Extensions

Intel SGX adds a set of additional operations to the x86 instruction set of Intel CPUs. The additional operations allow the CPUs to run specially compiled units of code, called *enclaves*, in a TEE [14, 22]. The main goal of SGX is to add a layer of defense, reducing the attack surface of the system. Ideally, if it has no weaknesses and the enclave code has no bugs, SGX removes the need to trust the infrastructure owner completely. The data owner only has to trust the software provider [14]. Enclaves are protected from the rest of the machine they run on to provide confidentiality and integrity. SGX is enabled by a set of hardware features supported in the CPU itself.

Intel SGX features

Intel provides a feature-rich SDK for SGX with many higher-level features added on top of the fundamentals supported by the hardware. The foundation for features used in this thesis are software attestation, the secure memory regions and the separation from the host system:

- **Software attestation:** the user is able to verify that they are communicating with the exact enclave they expect to be communicating with remotely. Hashing the enclave and signing the result can be used both for attestation to users as well as proving to the infrastructure provider that the enclave was not tampered with. The entire TCB is covered by the hash.
- **Secure memory:** the CPU has a reserved region of memory only enclaves can access called the Processor Reserved Memory (PRM). Enclaves are loaded into the Enclave Page Cache (EPC) inside of the PRM. The CPU copies the enclave data from memory outside of the EPC. After loading, the CPU itself assures exclusive access to the portion of the EPC to a single enclave. This protection, however, limits the amount of memory the enclave can access this way because the EPC is limited in size. The size of the EPC has been increased over the versions of SGX. Practically, this problem has been solved in newer Intel platforms [24].
- **Separation from the host:** SGX enclaves run exclusively in Ring 3, the least privileged execution mode [14]. This enforces separation from the operating system, disallowing arbitrary `SYSCALLS`. Untrusted execution can only transition into the enclave using the new instructions added to the x86 instruction set. Transitions into the enclave are performed with ECalls and the enclave can call untrusted code with registered OCalls. This presents a very limited attack surface. Because of the secured memory region and address translation, even higher privileged processes are not able to read the enclave's memory.

Official Intel SGX libraries and Intel SGX SDKs are implemented, documented, freely accessible and partially open-source. Moving code inside enclaves is easy in most cases, however, interacting with the operating system or other devices requires additional work

[43]. One of the greatest limitations, not present in common environments is the limited EPC size (128 MB total, 93 MB usable) [22]. Additionally, the high cost of transitions into the enclave and out of the enclave need to be considered for high-performance applications. Special system calls are added to enter and exit the enclave with pre-defined entry points and argument structure. The function interface is defined inside an EDL file which is needed to compile the enclave code into a binary that SGX can load. The `ECALL` system call is used to enter the enclave, and an `OCALL` can be made from the enclave to execute code outside of it. The Intel SGX SDK toolchain generates wrappers around these system calls using the EDL definitions to hide some of the complexity.

However, there are known attacks on SGX [14, 45, 41, 6], such as physical side channel attacks leaking information from the enclave, or software attacks on peripherals like Direct Memory Access over PCI-Express. Furthermore, SGX makes use of Intel's microcode within the CPU to execute complex instructions, because microcode is inaccessible to third parties and microcode updates are encrypted it is hard to analyse the actual microcode instructions on a CPU for bugs. Therefore, using SGX and completely believing the guarantees given by Intel requires trusting Intel. However, through microcode patches some known attacks and issues were already fixed. For this thesis, we trust Intel services, but assume the protocol implementation may include bugs and exploits.

2.5. Themis BFT Framework

Themis is a high-performance agreement framework [39] written in the Rust programming language. It abstracts the client-to-replica and replica-to-replica communication using an asynchronous message-channel model. It manages an asynchronous event loop for the separate parts of a replica. The separate parts are kept modular. To accomplish this Themis enforces strict interfaces on implementation of the protocol, the replicated application and the client.

2.5.1. Agreement Protocol Interface

Themis makes it easy to interchange communication and encryption models, applications and consensus protocols [39]. The framework presents interfaces for each of the components to implement, however, does not enforce any internal behaviour. In the default deployment, there are asynchronous workers for the following parts of the replica:

- **Communication:** network messages from clients and replicas are received and initial cryptographic checks are done. This part can be used across many protocols and applications without altering.

- **Protocol:** the consensus protocol is passed incoming messages from replicas and requests from clients. This module then initiates the broadcast of further messages to other replicas or to the application module. The application module can also cause events which notify the protocol module. A full implementation for PBFT already exists.
- **Application:** this module receives requests after being ordered by the protocol module. Responses are sent to the protocol module. The application interface includes functions for checkpointing.

Themis' framework components are modular and opt-in. Registered modules in a replica using Themis are called with the event context from an event loop in Themis. By moving more code into the protocol module the communication and application module can be omitted or only partially used.

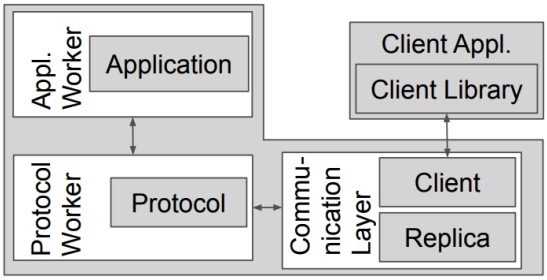

Figure 2.3.: Themis' high-level structure with asynchronous modules. [39]

The communication between the modules as well as the network communication are handled by Themis internally, as shown in figure 2.3. The advantage of this implementation is that values rarely need to be copied, instead, just the ownership of messages is *moved*. To allow the modules to make progress they are continuously polled.

2.5.2. Rust Programming Language

Rust is a systems programming language. Its goal is to enable programmers to write code for all domains with arbitrary requirements [28]. Rust is *systems-level* to allow programmers to take control of all low-level details necessary for the specific problem they are trying to solve, while allowing for useful abstractions and short-hand. The language design invites the programmer to use *good* memory management, sensible data representation and concurrency without leading to problems commonly present in other low-level languages like segmentation faults or data races. This makes Rust a powerful tool for writing code where low-level access is required but crashes, corruption or exploits need to be avoided. Reducing the likelihood of these kinds of errors and attacks makes Rust especially useful for writing BFT applications. The absence of a garbage collector also makes Rust a good match for networked applications, as fewer latency spikes should be noticeable.

Because Rust easily interfaces with the C Application Binary Interface (C-ABI) it is possible to write native Intel SGX enclaves. Enclaves are limited in size, having a small binary footprint as well as low memory usage is valuable. While Rust binaries can get quite large, the absence of a garbage collector and small runtime environment makes the language useful for writing trusted code. Rust performs exceptionally well under memory constrained conditions [15]. Rust also makes a commitment to reliability [27], and attempts to move as many exceptions to occur at compile-time or force error handling on the developer [28, 15]. Rust's programming paradigms make it practically impossible to encounter segmentation faults. The powerful type system allows for highly structured, easily readable, and less fallible code. Because applying a consensus protocol to any application necessarily adds a lot of overhead it is important to save performance whenever possible. Rust also advertises "Fearless Concurrency" [28], which in the context of a distributed system allows us to write a program which communicates with compartmentalized code and the network at large without encountering data races. This is one of the reasons Themis was written in Rust.

The ecosystem also adds incentives to write and package code in a way which makes code easy to reuse and test. Encouraging test-driven development, modularization and compile-time specialization makes old or foreign code easier to work with and improves trust into large code bases' correctness.

3. Design

This thesis' main focus is designing and implementing a compartmentalized BFT protocol based on the PBFT [8] protocol. We follow the SplitBFT approach to separate PBFT into multiple independent compartments. Our protocol is designed to reach consensus even if a number of compartments are malfunctioning. We use compartmentalization into TEEs to improve confidentiality and integrity. Using BFT and TEEs we define a stronger fault model than hybrid protocols. Our fault model needs to be detailed, as faults occurring inside and outside of certain TEEs can have different consequences. We present the design of the SplitBFT protocol and our method to gain robustness, confidentiality and safety using compartmentalization. Our goal is to tolerate as many simultaneous replica faults as possible. To do so we determine the scope and responsibilities of each compartment. Finally, the high-level structure of SplitBFT's design within the Themis framework is discussed.

3.1. System Model

We construct a distributed system of nodes. The nodes are connected by an asynchronous network which is allowed to delay messages, duplicate messages or deliver them out of order. Messages may be arbitrarily long delayed, but always arrive after a finite time. Each node is separated from the others and can fail independently. There are two flavours of nodes: clients and replicas. Replicas are further logically subdivided into one leader and multiple backups by the protocol.

We assume unforgeable digital signatures. Each participant is able to sign and verify messages if they have access to the necessary keys. Signing messages requires the author to have access to the private key. Verifying a signature requires the public key.

Replicas are required to support a form of trusted execution. The TEE must support secured memory regions to keep private keys and decrypted confidential data secret. Without memory access protections, an attacker on the untrusted side could read the private key or state and leak this data without taking over the TEE. This would allow the attacker to impersonate the compartment. Because we assume memory access protection in the TEEs the compartments may contain private information, such as asymmetric key pairs for authentication. The system running on each replica is separated from the operating system by the TEEs. The compartments in the TEEs are separated from each other, a faulty compartment cannot access another local TEE's memory. Each compartment on a replica performs a small part of the agreement process.

A replica can fail in the untrusted execution environment, which includes the untrusted part of our own system as well as any supporting software running on the machine, like the operating system or necessary drivers. With all safety-critical code and confidential data protected inside TEEs, a Byzantine untrusted side on a replica can only make the replica unavailable. The untrusted side may issue calls into the TEE, but the compartment or protocol code is able to detect incorrect calls using digital signatures. The untrusted side may also issue calls as a replay, which the compartment will detect as a repetition and ignore. A Byzantine untrusted side is also able to choose to operate correctly.

To differentiate SplitBFT from hybrid protocols we tolerate faults in TEEs. The TEE compartments are able to fail Byzantine as well. We assume faults in TEEs originate from bugs or exploits in developer code, not from the implementation of the TEEs platform itself. We also assume that once a TEE is taken over, it does not allow the attacker to take control of other local TEEs. Failures in TEEs are independent of each other: TEEs with similar tasks on different machines and different TEEs on the same machine fail independently. This can be achieved through diversification [37]. A malicious TEE is able to spread the fault to the untrusted side of the replica it is running on.

A failure in a client allows it to behave arbitrarily. The faulty client can then send arbitrary requests, but as the clients are individually authenticated, a faulty client alone cannot impersonate another member. The effect of a faulty client's requests on the application's state is not considered, as the application's implementation and access control is separate from the SplitBFT protocol. This was the focus of previous related works and could be combined with our approach [10].

The general SplitBFT approach demands that quorum decisions are protected by compartmentalisation. The protocol requires a quorum of $2f + 1$ messages out of the n compartments of a type to proceed. Security-sensitive code must be moved into compartments. Compartments must be independent on the local replica and not share a common state with other compartments. Therefore, there can not be a *store* or *state* compartments, which other local compartments need to rely on. For the PBFT derived implementation of SplitBFT we use three different types of compartments per replica. PBFT's phases provide an intuitive boundary to split the protocol into compartments. The phases are independent, data flows using quorums of the preceding compartment type. The compartments are *Preparation*, *Confirmation* and *Execution*. Each compartment is present exactly once on each replica.

3.2. SplitBFT Protocol

SplitBFT is a BFT SMR protocol based on PBFT [8]. We copy the general flow of the PBFT algorithm in normal-case operation. SplitBFT divides the safety responsibilities across the three compartments. Each compartment is responsible for one of the phases of PBFT: pre-prepare, prepare and commit. A compartment does not trust another compartment, just because it is local. This follows directly from our system model, as only

some compartments might be malicious. Therefore, any action in a compartment must be caused by either a quorum or a liveness decision combined with the log of verifiable past messages. Compartments are allowed to *trust* liveness data from the untrusted side of the replica for two reasons. First, the compartment is unable to generate liveness data, like a timer or interrupt because those require cooperation with the untrusted side, and second, a malicious untrusted side is already capable of attacking liveness by dropping messages. The untrusted side of the system may be unaware of the BFT protocol's local state, messages are forwarded to compartments by type. In regards to the BFT protocol, the untrusted side can be stateless.

3.2.1. Normal-Case Operation

We split PBFT along its phases and quorum decisions as shown in figure 2.2. Applying the SplitBFT method to the PBFT protocol we identify three possible individual compartments: *Preparation*, *Confirmation* and *Execution*. This leads to this abstract flow for a request to pass through the SplitBFT system:

1. Clients send requests to what they assume to be the leader replica. Clients start timers after sending a request and broadcast their message on timeout. The untrusted side of the replica accepts the message over the network and forwards the request to the local compartments.

2. The leader checks requests in the preparation compartment and then broadcasts both the request and a pre-prepare message for the request to the backup replicas. This requires that the message is signed by an authenticated client. The leader advances the clients last timestamp.

3. Backup replicas forward the incoming requests and pre-prepares message to their preparation compartment, verify its authenticity and broadcast a prepare message for the pre-prepare.

4. Prepare messages are routed into the confirmation compartment. Once a set of $2f + 1$ prepares from any set of prepare compartments are present in the confirmation compartment they broadcast a commit message. Unlike the PBFT protocol, the conformation compartments are unaware of the state of the local preparation compartment. This means we require $2f + 1$ instead of $2f$ messages, which may include the local prepare compartment.

5. Commit messages get sent to the execution compartments. When $2f + 1$ different commit messages are present in the execution compartment the request is executed and the reply is sent directly to the client. As the commit only refers to the request by digest, the request's data was transmitted to the execution compartment in the first step.

The state of a single replica is no longer represented by a single phase out of ⟨`pre-prepare`, `prepare`, `commit`⟩, as the compartments we split our protocol into have their own independent state. The independence of the compartments requires us to treat local and remote compartments equally. This means, unlike PBFT, we need $2f + 1$ messages for a strong quorum because we cannot access the state of the local preceding phase.

3.2.2. Compartments

After applying a split of the PBFT protocol into *Preparation*, *Confirmation* and *Execution* compartments we need to examine the function of each compartment to ensure independence. We transform PBFT's phases into quorum decisions of compartments of the same type. The untrusted side can only forward, store and distribute the different kinds of messages that are part of the protocol. On the untrusted side message may also be filtered. Any operation which only affects liveness may be given to the untrusted side to reduce the need to call into TEEs. The other components on the replica that aren't part of the TEEs such as operating system, drivers and other support software are also part of the untrusted side. The compartments are intentionally kept small and reduced to their core purpose, to keep the TCB as small as possible. A small TCB reduces the likelihood of a bug and exploit in the TEE. Each correct compartment signs its messages using a private key only accessible inside the compartment. This allows other compartments to uniquely identify a message's author as one of the other compartments. Because of the key distribution method (see 3.2.5), this allows each compartment to identify both the replica and compartment type a message was sent from. The untrusted side presents itself as a message broker (see 3.3) to the compartments. It acts as a layer which forwards messages to the compartments based on the message type according to the routing rules shown in table 3.1. If a message is either signed and sent by or delivered to a participant which is incorrect according to that table, the message is discarded.

Protocol	Message Type	Correct Sender	Correct Recipient
	Request	Client	Preparation C.
Ordering	Pre-Prepare	Leader's Preparation C.	Preparation C.
	Prepare	Preparation C.	Confirmation C.
	Commit	Confirmation C.	Execution C.
	Reply	Execution C.	Client
Garbage Collection	Checkpoint	Execution C.	All C.
View Changes	View-Change	Confirmation C.	Preparation C.
	New-View	Preparation C.	Preparation C.

Table 3.1.: Correct routing for SplitBFT message types. "C." abbreviates "compartment". This table is analogous to information shown in figure 2.2.

A request message should ideally always be sent to the leader replica directly. The client, however, might not be aware of recent view-changes and not know who the leader is. If a client sends requests to the wrong replica, either the preparation compartment on that replica will re-transmit the request to the leader or the client will timeout and broadcast the message itself. Also notable is that PBFT's concept of a leader is only present for the preparation compartment. After a request is proposed by the preparation compartment on the leader with a pre-prepare message and a quorum of prepare messages from the backup preparation compartments is present, the rest of the protocol continues without the need for a leader.

3.2.3. View Changes

We keep the view-change protocol used in PBFT to provide liveness even when the leader becomes faulty [8]. The view-change protocol only needs to be slightly adapted to work in a compartmentalized context. As shown in table 3.1 the view-change message is sent by the confirmation compartment when it is notified of a timeout. The timeout originates from the untrusted side. Because view-changes are a liveness protection, the untrusted side's timer can be used inside the TEEs without losing safety. The view-changes are sent to the preparation compartment of the proposed new leader. Once a preparation compartment has a quorum of view-change messages it broadcasts new-view messages containing this proof to all preparation compartments on all replicas. Each compartment receiving a new-view can verify this proof by checking the signatures and advance their view accordingly. Afterwards the preparation compartments continue operating in the new view. Later stages and compartments can deduce the successful view change from the quorum of messages from the preparation compartments in the next view. This optimization shows how the leader replica is only performing a special task concerning the preparation compartments.

3.2.4. Garbage Collection

To allow compartments to free memory occupied by old messages we use checkpoints for garbage collection. After a configurable number of requests have been executed the execution compartment serializes its state and sends a checkpoint message to all compartments on all replicas. The serialized state has to be encrypted such that only other execution compartments can deserialize it to keep confidentiality. Because garbage collection is necessary in compartments, checkout messages have to be sent to all compartment types. Once any compartment has a quorum of valid checkpoint messages for the same sequence number and state by different execution compartments it can delete all older messages out of its log. Execution compartments can also use the verified checkpoint to update their state if they are behind. The preparation compartments also use the checkpoint messages to advance their water marks.

3.2.5. Encryption and Signatures

Compared to PBFT our SplitBFT protocol introduces more logical protocol participants which need to be identifiable and able to sign messages. The encryption and signature scheme is designed alongside the following goals and requirements:

1. We have a finite set of authenticated clients known from the start.
2. Any enclave which receives a request must be able to verify its authenticity.
3. Only enclaves which need to decrypt client requests are able to do so.
4. As long as no execution enclave is faulty confidentiality is preserved.
5. A client is able to identify replies from different execution enclaves.

To achieve these goals a combination of symmetric and asymmetric encryption is applied. The encryption and signing scheme cannot be directly taken from PBFT, as the compartments need to be able to digitally sign their own messages. Protecting confidentiality requires that only execution enclaves are able to decrypt certain messages.

Asymmetric Key Distribution

In our system model we assume to be able to identify the sender of a message cryptographically. This can, for example, be achieved using digital signatures that are based on public key cryptography or MACs. MACs have the advantage of being a lot faster to verify, however, are not able to be used to prove a messages authenticity to a third party. For MACs, each pair of communication partners need to obtain a unique MAC. As we have not only the $3f + 1$ physical machines but also on each machine at least 3 enclaves, which need to be able to sign messages for each other, the number of MACs required to store and maintain would skyrocket. For the sake of maintainability, simplicity and robustness we provide each logical participant in our network with a public-private key pair to use for digital signatures. However, this leaves the opportunity to selectively replace some signatures with MACs in frequent communication channels in future work [7]. We authenticate each client i by a public and private key (c_{i_p}, c_{i_s}). Furthermore, each enclave on every replica has its own unique public-private key pair: (e_{r,j_p}, e_{r,j_s}), where r is the replica id and j the enclave type. This is necessary to prevent another enclave, the untrusted side or even a third party to impersonate the enclave. The enclave type j will be noted as P for preparation, C for confirmation or E for execution. Asymmetric keys are annotated with subscript p for the public key and s for the secret, or private, key. Finally, the execution, or application, enclaves contain a pre-shared key K. This is necessary because the execution enclave will have to decrypt the request of the client. Only execution enclaves are meant to be able to decrypt the request for the sake of keeping confidentiality as long as no execution enclave is faulty. An alternative would have been to have clients send their requests encrypted for each execution enclaves unique key-pair. This, however, would have required more changes in our algorithm because digests over the encrypted

data for different replicas would be different. Therefore, the client would have to send the request to each replica itself, instead of the leader broadcasting the single message. This is possible, but adds an overhead to ensure each replica received the same request.

In a configuration with k replicas and l clients the following keys exist and are known to each participant.

- Client i for $i \in \{1, ..., l\}$:
  - (c_{i_p}, c_{i_s}) used for signing requests.
  - K used for symmetrically encrypting requests and decrypting responses.
  - e_{r,E_p} for $r \in \{1, ..., k\}$ used for verifying responses from the execution compartments.
- Replica r for $r \in \{1, ..., k\}$:
  - Untrusted:
    - c_{i_p} for $i \in \{1, ..., l\}$ used for verifying client requests. The untrusted side uses the authenticated client's public key to act as a firewall to reduce load on the compartments.
  - Preparation enclave:
    - (e_{r,P_p}, e_{r,P_s}) used for signing pre-prepares and prepares.
    - c_{i_p} for $i \in \{1, ..., l\}$ used for verifying client requests.
    - e_{r,P_p} for $r \in \{1, ..., k\}$ used for verifying pre-prepares from the leader's preparation compartments.
    - e_{r,C_p} for $r \in \{1, ..., k\}$ used for verifying view-changes from confirmation compartments.
    - e_{r,E_p} for $r \in \{1, ..., k\}$ used for verifying checkpoints from execution compartments.
  - Confirmation enclave:
    - (e_{r,C_p}, e_{r,C_s}) used for signing commits.
    - e_{r,P_p} for $r \in \{1, ..., k\}$ used for verifying prepares from preparation compartments.
    - e_{r,E_p} for $r \in \{1, ..., k\}$ used for verifying checkpoints from execution compartments.
  - Execution enclave:
    - (e_{r,E_p}, e_{r,E_s}) used for signing responses.
    - e_{r,C_p} for $r \in \{1, ..., k\}$ used for verifying commits from confirmation compartments.
    - e_{r,E_p} for $r \in \{1, ..., k\}$ used for verifying checkpoints from other execution compartments.

- K used for symmetrically decrypting requests and encrypting responses.
- c_{i_p} for $i \in \{1, ..., l\}$ used for verifying client requests.

The color encodes the publicity of the keys:

1. Green: "Well-known" at the beginning of runtime. These are the public keys c_{l_p} for $l \in \{1, ..., i\}$ for clients and e_{h,j_p} for $h \in \{1, ..., k\}$ and $j \in \{P, C, E\}$ for the compartments.

2. Blue: Shared between a subset of trusted execution environments. Private until a member of that set becomes faulty. Used to preserve confidentiality as long as possible. This includes the shared private key K used to symmetrically encrypt the requested operation.

3. Red: Known only to the *owner* of the key unless faulty. These are the private keys c_{l_s} for $l \in \{1, ..., i\}$ for clients and e_{h,j_s} for $h \in \{1, ..., k\}$ and $j \in \{P, C, E\}$ for the compartments.

With this asymmetric and symmetric key distribution replicas can verify all necessary types of communication to perform PBFT split across multiple independent TEEs.

Client-Replica Communication

Client i can send a request with operation b encrypted with the preshared key K and signed with c_{i_s} to the leader. As confidentiality is a requirement in our fault model, we encrypt client requests $b_K = \texttt{Enc}(b, K)$ and include it in the packet. We sign the encrypted data using the client's private key and $b_s = \texttt{Sign}(\langle b_K, m \rangle, c_{i_s})$, where m is any necessary meta data sent alongside the request of the operation b. We sign the encrypted request data b_K instead of b itself to ensure all compartments can verify the signature. The meta data m contains additional metadata like the timestamp and client id. It is necessary to have this data accessible even to enclaves that cannot decrypt the entire message because it is needed for the ordering and agreement process. A correct leader distributes the request as well as a pre-prepare for the request, which puts the requests into a view and gives it a sequence number for ordering. Before broadcasting the leader can check on the untrusted side whether the message originates from an authenticated client. This does not hurt our safety assumptions, as only liveness can be disrupted by the untrusted side dropping the request. Because the operation itself is encrypted with K, which is only present in clients and execution compartments, no compartment or replica other than the execution compartment can decrypt the operation. A variation of SplitBFT could use MACs for this purpose, or use client-specific K_i keys for encryption to protect clients from each other. Because replies from the replicas are not needed as proof, they can also be sent using MACs for a gain in performance.

Request Verification

Any enclave as well as the untrusted broker can check that the signature b_s is correct. This proves integrity of the message from the client. Because we assume strong cryptography it also means that the message truly originated from the client as long as the client is not faulty. Therefore, the request came from an authenticated client and can be processed further. If a client is faulty its secret keys could have been leaked which allows other parties to send requests as the client. If any misbehaviour is detected within the message the client can be blocked. In the described version of the protocol, this blocking would occur as part of the application in the execution environment, which is able to implement its very own access control on top of SplitBFT's authentication scheme [8]. However, the execution enclave could also be used similarly to the untrusted side in checking whether a request *should* be processed further, similar to a firewall [49]. It needs to be the execution enclave, as it is the only compartment on a given machine which is able to decrypt the request and possibly check the application's ACL [32]. We use the untrusted side to act as a firewall by checking the validity of the signatures. This at most disrupts liveness, if the untrusted side is faulty and discards correct messages. In our fault model, a faulty execution enclave is also able to disrupt liveness, but by having it check the semantics of incoming requests a better firewall could be built. The execution enclave could then even cache the decrypted message data or preprocess request data depending on the application.

Ordering

When following the normal-case operation (3.2.1) messages are required to be signed by the correct TEEs or clients. This allows TEEs later in the pipeline to check for a quorum of correctly signed messages in the stage before them.

- Leader replica's with non-faulty preparation enclaves will verify the request's authenticity, then broadcast signed pre-prepare messages to the backups' preparation and all execution enclaves. The execution enclave stores the request for later.
- Once a valid pre-prepare message is received, each preparation enclave broadcasts a signed prepare message to the confirmation enclaves.
- When the confirmation enclaves have a quorum of prepare messages they broadcast a signed commit message to the execution enclaves.
- After a commit quorum is reached in the execution enclave, it interprets the request after checking its signature itself. It can decrypt the request using the pre-shared key. The requested operation is performed inside the execution compartment. At this point, the decrypted data could also turn out to be nonsense, which the execution enclave could see as a reason to block the client. If the message cannot be decrypted, nothing is executed.

- Any reply to the client is signed using e_{r,E_s} instead of using K to ensure the client can determine that a sufficiently large set of different execution environments have responded. The data is encrypted using K, which means that the digest over that data is identical for the responses of all execution environments and, therefore, easier to count for the client.

Each of the agreement messages contain at least the sequence number, the hash of the message b_s as well as necessary identification of the sender and recipient for signature checking. Requests, garbage-collection and view-change messages can be verified as well. Requests are verified by the signature of the client, this can be done in any TEE.

3.2.6. Fault analysis

Normal BFT protocols without TEEs are able to tolerate f faults provided they have $n \geq 3f + 1$ members. If the fault model only allowed crash faults in TEEs, this means a TEE only fails by crashing, the BFT protocol could withstand f faults with only $n \geq 2f + 1$ members. This is because the TEEs prevent equivocation in these BFT systems, making them equivalent to crash-fault tolerant protocols. This changed fault model is commonly used for hybrid models.

Our fault analysis needs to be more detailed than other BFT protocols, because faults in different compartments and on the untrusted side have different consequences. We keep f to count the number of replicas with *any* kind of fault. The TEEs can fail independently as: f_p, f_c and f_e count the number of faulty compartments for preparation, confirmation and execution compartment types respectively. A fault outside of a TEE is noted as f_u, for a faulty untrusted side. A faulty compartment implies a faulty replica, but a faulty replica does not imply faulty local compartments: $f_u \geq f_i \geq 0$ for $i \in \{p, c, e\}$. A replica may also have multiple faulty compartments. Practically, this represent that the adversary can break out of the compartment, or needed to compromise the untrusted side first to get access to the compartment. However, just compromising the untrusted side alone does not give control or access to data inside of the compartments. This means: $\max\{f_p, f_c, f_e\} \leq f = f_u$. We maintain liveness as long as $n \geq 3f_u + 1$ and $n \geq 3f_i + 1$ for $i \in \{p, c, e\}$. Confidentiality is kept as long as $f_e = 0$. Integrity is given as long as $n \geq 3f_i + 1$ for $i \in \{p, c, e\}$. One result is that an arbitrary amount of faults outside of the TEEs can be tolerated to maintain safety. In practise, this means each compartment type needs to be able to have a majority of correct members to form a strong quorum. This represents each stage, or phase of PBFT, to be able to form a correct quorum for the next stage to use.

Example

Figure 3.1 shows a possible deployment of SplitBFT containing faulty nodes. In replicas 0, 1 and 2 we find faults in the preparation, confirmation and execution compartments respectively. The faults are counted in table 3.2.

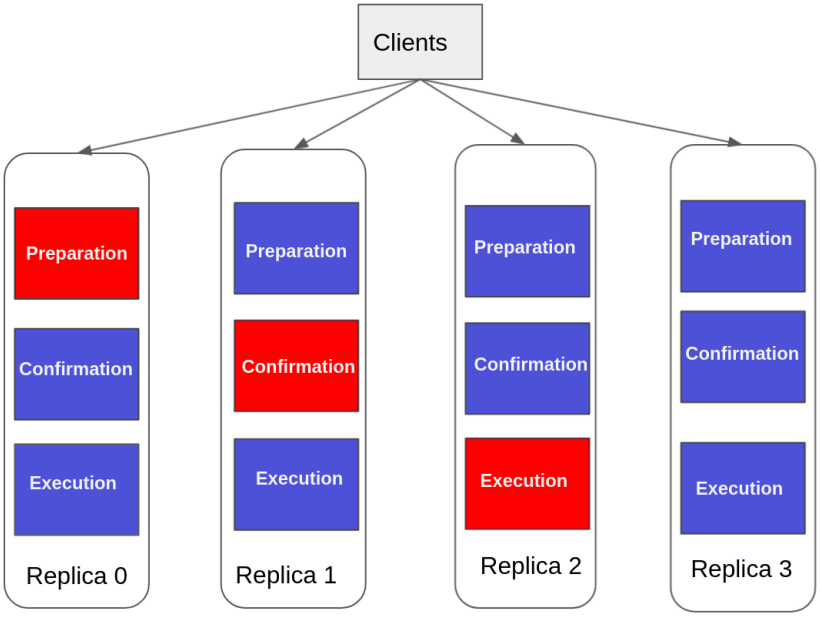

Figure 3.1.: Partitioning BFT Protocol into multiple compartments.

f	f_u	f_p	f_c	f_e
3	3	1	1	1

Table 3.2.: Number and type of faults for figure 3.1

In this example liveness can not be guaranteed because $4 = n \ngeq 3 \cdot f_u + 1 = 3 \cdot 3 + 1 = 10$. The faulty compartments on replica zero, one and two can delay the entire replica arbitrarily. If the adversary, however, only stops the faulty compartment liveness is maintained as there is only one fault per compartment type. With $f_i = 1$ for $i \in \{p, c, e\}$, if only the compartment itself was shut down $4 = n \geq 3 \cdot f_i + 1 = 3 \cdot 1 + 1 = 4$ is true and liveness of the system is maintained. Confidentiality is not kept because of the single fault in the execution compartment ($f_e \neq 0$) in replica two. This faulty execution compartment may leak the internal state or decrypted requests. If only preparation or confirmation compartments were affected, confidentiality of requests and the application state would be maintained. Integrity is maintained as each enclave type is still able to form a strong quorum of correct members: $4 = n \geq 3 \cdot f_i + 1 = 3 \cdot 1 + 1 = 4$ for $i \in \{p, c, e\}$. This shows how separation increases safety, as in classical PBFT having three out of four replicas faulty breaks the integrity of the protocol. Robustness is increased because we assume that finding a way to cause a fault in a compartment is harder than it is to cause a fault on the untrusted side.

3.3. Compartment Broker

The compartment broker, or SGX broker, is the main module that is added to the Themis framework to enable the use of SGX enclaves. Because the Themis framework is modular and asynchronous this broker layer needs to implement the necessary interfaces and adapt them to the TEE's interface. The general structure of a replica using the Themis framework is shown in figure 2.3. Different than the example structure shown in figure 2.3 the application code resides inside the execution enclave which is managed by the broker as part of the protocol module. Replacing only the protocol module with the broker layer reduces the amount of changes necessary to call into the compartments. Following the example design more closely, i.e. presenting an application interface which also interacts with the compartments used by the protocol would duplicate interfaces and make the software system more complex.

The broker layer interprets incoming messages from the network and routes them to the corresponding compartments based on type. The broker is designed for SGX enclaves as TEEs. This means its design is based on the specific interface for SGX enclaves, which are ECalls and OCalls. Enclaves can enqueue messages to be sent, either over the network or to other local enclaves. Sending messages locally is required, for example, to send prepare messages from the preparation compartment to the confirmation compartment. These messages are automatically routed to both the local compartment and the remote ones by the broker layer with only one OCall from the sending compartment. Other actions to be performed by the untrusted side can be enqueued using OCalls as well. Whenever executed by Themis, the broker consumes the queues and acts on the messages inside.

This queue design limits the number of ECalls and OCalls performed. Both the broker and the enclaves can collect messages and actions in a list, before posting the entire list as one collection of ECalls or OCalls. Furthermore, this queue buffer enables more flexibility to execute enclave-handlers asynchronously in their own thread or place a thread in the enclave using SGX switchless calls [42]. Moving the data from the OCalls into the queue requires a copy. This, however, is unavoidable, as the data for the OCalls needs to be created within the enclaves, and any out-buffer in the OCall is copied by the SDK.

Because the broker layer is implemented completely outside of the TEEs, our design makes sure to not give it safety-critical functions. An adversarial broker can attack liveness, by refusing to deliver messages or perform OCalls.

3.3.1. Compartment Interfaces

Because there are a large number of different kinds of messages that need to enter and exit the enclave, we chose to have a simple interface that accepts the serialized messages instead.

Therefore, the broker and enclaves communicate using exactly one ECall and one OCall, defined in the EDL file. The Ecall is used to enter the compartment and the OCall to exit.

```
enclave {
    from "sgx_tstd.edl" import *;
    from "sgx_stdio.edl" import *;
    from "sgx_backtrace.edl" import *;
    from "sgx_tstdc.edl" import *;
    trusted {
        public sgx_status_t handle_ecall_compartment(
            [in, size=len] const uint8_t* ecall_mp, size_t len);
    };
    untrusted {
        sgx_status_t handle_ocall_compartment(
            [in, size=len] const uint8_t* ocall_mp, size_t len);
    };
};
```

Additional ECalls and OCalls will be present from the SGX SDK, but those will not be used during protocol execution. The ECall and OCall introduced by SplitBFT for each enclave receive serialized data. However, the serialized data is to be interpreted as a selector over the available functions in both environments. As such, the ECalls can contain data for the following purposes:

- **Initialization**

  Like the PBFT implementation in Themis some runtime configuration is loaded from a configuration file [39]. To keep the SplitBFT implementation flexible, it allows the enclaves to request configuration data from disk. This data could be signed by a key present in the enclaves at boot to verify a trusted administrator signed the configuration file. The initialization ECall contains information from the configuration file as well as the command line arguments the replica was launched with.

- **Identity updates**

  After receiving information about the system in the initialization, identity ECalls provide public keys for the networked peers. These keys can also be signed by an administrator key burned into the enclaves.

- **One or more received protocol message**

  Network messages arrive in the broker layer, and after parsing need to be routed to the enclaves. The serialized network message is passed to the enclave through this ECall.

- **Timers**

  The untrusted side sets timeouts and timers for different purposes. It can decide to inform an enclave about timer expirations for liveness decisions.

We allow timers and interrupts to be controlled by the untrusted side. They are reported to the compartments without proof. This is because this information can at most attack liveness, but a malicious untrusted side of a replica can already drop all messages.

The OCalls on the other hand allow the enclave to request actions from the broker layer:

- **Requesting identity information**

  After configuration data has been loaded into the enclaves, they are aware of the network of peers, however, keys of the peers are not hardcoded into the enclaves. This means, each enclave needs to request public keys from disk.

- **Send one or more signed protocol message to a list of protocol members**

  To send messages over the network the enclave has to serialize the messages and then pass them to the untrusted side to interact with the network. For verification at the receiver, all serialized messages need to be signed before they are given to the untrusted side to route.

Due to the overhead ECalls and OCalls add to every call, the implementation will aim to issue as few calls as possible. Batching of ECall and OCall data into a single ECall or OCall to reduce the number of individual calls is performed whenever possible. The untrusted side may choose to batch ECalls by waiting.

3.3.2. Asynchronous Operation

Switchless SGX is a method of using SGX while avoiding the high cost in CPU cycles of switching into, and out of, the enclaves during an ECall. The naïve solution is very simple: one thread spins inside the enclave reading a part of memory shared with the untrusted side, and the untrusted side writes to that memory, instead of issuing an ECall. OCalls can be implemented analogously.

The downside of this approach, in general, is that there is a chance of wasting CPU cycles by having these threads spin in the enclaves with no work to do. This is necessary, as there is no way of blocking and waking up a thread inside the enclave. However, for applications that expect to have very frequent need for ECalls and OCalls switchless calls can save the cost of the ECall and OCall almost entirely.

The Intel SGX SDK supports switchless SGX calls with the additional feature of a reactive worker model. By using worker threads it reduces the amount of wasted CPU cycles spinning in the enclave by meeting a user-set efficiency limit [42]. This assumes that similar ECalls will take a similar duration each call. This could be a potential problem for our implementation, as time spent executing the same ECall will spike once a quorum is reached, and otherwise the enclave will almost always immediately return. An optimization, which also fixes this problem, would be collecting enough messages for a quorum outside of the enclave and only switching into the enclave once with all necessary data.

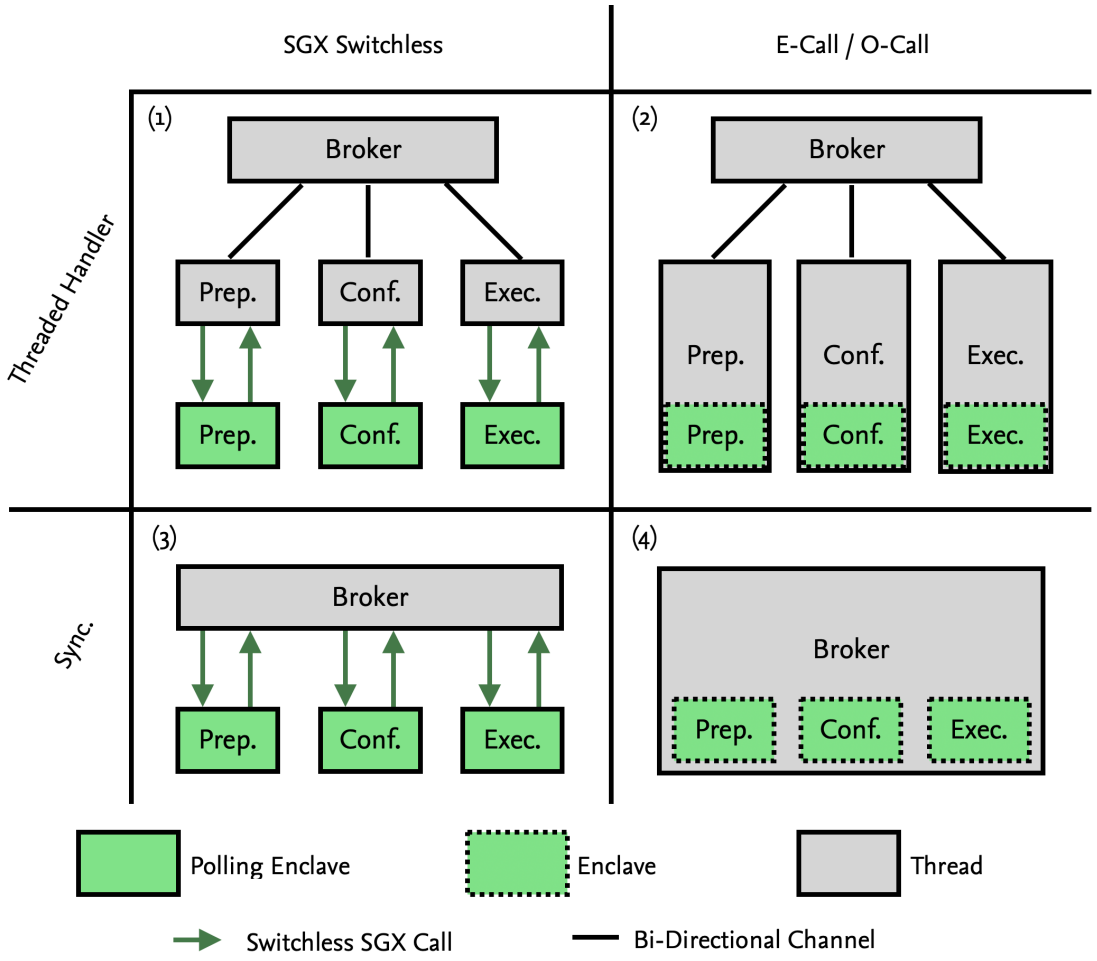

Figure 3.2.: Decision matrix of using SGX switchless calls or ECalls and OCalls to call into the enclave, and whether to have threaded handlers for each enclave or call them directly from the layer above.

Using the Intel SGX SDK, an ECall can be marked as switchless in the EDL file defining the enclaves interface. This is taken as a hint by the SDK, and calls will not necessarily be executed switchless. This means that it is possible to selectively mark our ECalls as switchless, and iteratively find the optimal settings for performance on our machines by benchmarking. We will present the synchronous classical broker interface. The extend to which full switchless SGX and threaded asynchronous enclave handlers are mixed prescribe a different execution model, as shown in figure 3.2. The left column in figure 3.2 shows the calls into and out of the enclave to be switchless, this is no requirement and even mixing is possible, though produces harder to maintain code. The top row of figure 3.2 applies a threaded enclave handler, which communicates with the broker using message channels. Instead of busy-waiting inside of the enclave, a thread which spends the time waiting in the untrusted side manages communication with the enclave beneath it and the rest of the system above it asynchronously. By mixing these two variations of asynchronous execution the four broker designs follow:

3.2 (1) Every enclave has a busy-waiting thread for switchless ECalls and a handler on the unsafe side in another thread is polling for switchless OCalls. Each handler thread is talking to the managing broker through bi-directional channels. This setup allows for using more available threads and theoretically minimizes latency out of all the configurations. It also allows enclave code to block or be very costly while still allowing other enclaves, enclave handlers and the broker itself to continue computation. However, many busy-waiting threads are wasting cycles and context switches are costly.

3.2 (2) Every enclave gets a thread of execution but does not poll inside of the enclave and does not use SGX switchless calls, but instead uses ECalls and OCalls to enter and exit the TEE respectively. This still allows multiple threads to be in the trusted environment at the same time and the broker to accept and handle IO while computation inside of the enclaves happens. This also reduces the number of threads, but at least two more ECalls and OCalls are needed than in case (1) for each context transition.

3.2 (3) This is the most asynchronous configuration. The broker initiates work in the threads waiting in the enclaves by a switchless SGX call. The enclaves then respond by performing a switchless OCall which requires the broker to poll the enclaves switchless OCalls in between performing IO and calling into the enclaves. This could be simplified in the implementation by using the Future type and asynchronous primitives. This option minimizes the amount of threads while still allowing entirely parallel computation inside of the enclaves and outside of the enclaves.

3.2 (4) In configuration (4) the broker thread directly calls into and out of the enclaves using ECalls and OCalls respectively. Having only one thread minimizes the amount of threading-based context switches, but as no switchless SGX calls are used, the thread has to transition into the TEE every time an enclave needs to be involved.

One possible application for mixing is in version (3), with a synchronous broker thread switchless-calling into the enclaves. The broker could broadcast every message to every enclave with low-cost switchless SGX calls, and enclaves could initiate sending of a message of their own with standard OCalls. This way the broker avoids busy-waiting on multiple possible calls as well as waiting for incoming messages, but the context switches are still minimized as enclaves will not respond on most messages, only appending them to their log.

The initial implementation uses version (4), as it is the most straight forward to implement and will serve as a baseline for further optimizations and experiments with the other design alternatives in the future. Using multi-producer-single-consumer channels to handle communication from the enclaves to the untrusted broker, while not strictly necessary, as the design is single-threaded, allows for easier adaptation for the other designs.

3.4. Application Compartments

It is common for the execution part of the protocol implementation to be closely integrated with the application or service that is replicated. We recognize a possible variant of SplitBFT for PBFT which adds robustness under a slightly changed fault model. In this variation we separate the execution compartments protocol code from its application specific code by placing application code in its own compartment. Further compartmentalization in our fault model do not add any additional safety or robustness, as the application instance has to trust its BFT protocol member shown in figure 3.3.

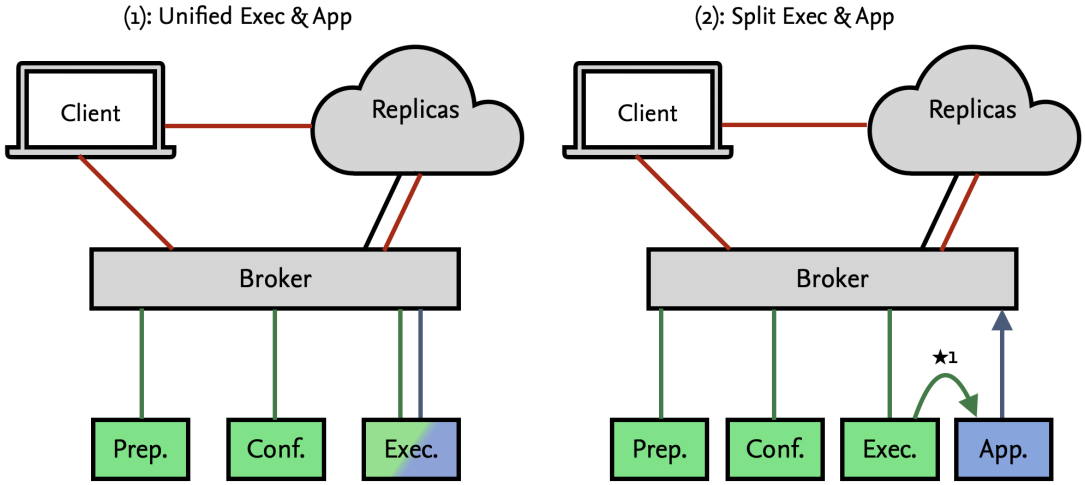

Figure 3.3.: Variant of SplitBFT with split of the execution enclave into BFT-protocol execution and application execution environments.

3.3 (1) Here code that is application specific resides inside of the execution enclaves, which shares protocol code. This makes assumptions about the execution compartment's TCB size harder as it is application dependent. A faulty execution compartment has full access to application data.

3.3 (2) The application code is contained in its own compartment. A faulty execution environment cannot directly access application state due to memory access protections between TEEs. However, no additional integrity or safety is gained, as a faulty execution enclave can make the application enclave do anything without requiring another quorum. Another upside of this split is that the protocol enclaves can be diversified, but an end-user deploying a system with their own application might choose to provide fewer application compartment implementations to save cost. This architecture could also be used for running the application on remote machines.

A faulty execution compartment will also induce an effective fault in the application environment, as it can introduce arbitrary delays. In the same way, a faulty application compartment will behave identically by being able to introduce arbitrary delays.

The existence and use of the symmetric K key is another argument for applying the possible execution and application split. In that case the application compartment would trust messages signed by the local execution enclaves key pair. The application then uses the symmetric key K instead of the execution enclave, and another set of key pairs is introduced the identify the application enclaves. Using the execution compartment's keys only makes message encryption more difficult, as hashes cannot be compared directly, but doesn't improve confidentiality as a single faulty application enclave can still decrypt and leak data. Removing the K symmetric key from the execution compartment means compromised execution compartments do not affect confidentiality at all. Assuming the protocol implementation in the execution compartment's TCB is larger than the application code, this could increase resilience.

A separation into an execution and application enclave would enable a design closer to the example Themis structure from figure 2.3. However, the additional message exchanges and ECalls would impact performance negatively.

Other approaches aim to increase robustness by separating machines physically [49], based on their semantically separate tasks [31].

4. Implementation

Following the aforementioned design we implement a new layer in the Themis framework to interface with Intel SGX and the framework's modular structure. The implementation is split into the untrusted and trusted sides. The untrusted side is very closely integrated with the implementation of Themis itself, but the SGX enclaves are implemented as separate individual binaries in a very different environment. First, we outline how the added translation layer fits in the Themis framework's structure. Then some critical implementation details about the layer are discussed. Its main functions are presented and their impact on the agreement protocol are outlined. Finally, the implementation of the SplitBFT protocol in the SGX enclaves in Rust using the Teaclave SGX SDK is presented. We outline some of the obstacles and problems which arise when implementing SGX enclaves in Rust.

4.1. Themis BFT Framework

The Themis framework is modular by design and allows combining different clients, protocols and applications. However, it is very flexible in allowing these modules to also perform tasks usually performed by other modules if necessary. For example, the protocol module can easily add additional internal layers of encryption, which would conventionally be implemented to the communication layer. We make use of this flexibility to realize the translation layer between Themis and SGX as a single protocol module. As the entire safety-critical implementation lives inside the compartments, the layer will act as a message broker and translation layer.

To interact with the modular Themis framework a new protocol module of SplitBFT needs to be created. Its main task is to act as a translation layer between Themis and the trusted enclaves. Network messages are translated to ECalls and vice versa. Figure 4.1 shows the structure of the added modules. Directly connected to Themis is the `sgx-broker`, which acts as the protocol module for SplitBFT.

For the most part, the broker layer acts as a shim between the Themis API and enclave interface. As such, it receives all network messages. To interact with the enclaves a generic `enclave-handler` crate is written with the task to provide a simple and safe interface to send and receive SGX calls. In addition to the API translation, the broker also ensures enclaves are started through these enclave handlers. It makes decisions about where to route incoming or outgoing messages. OCalls are also performed by the broker. Specific enclaves handlers are implemented for the exact interface defined in the EDLs for the enclave types. The handlers are created and stored in the broker. Finally, all the enclaves are implemented in their own environment. We follow the Teaclave SGX SDK

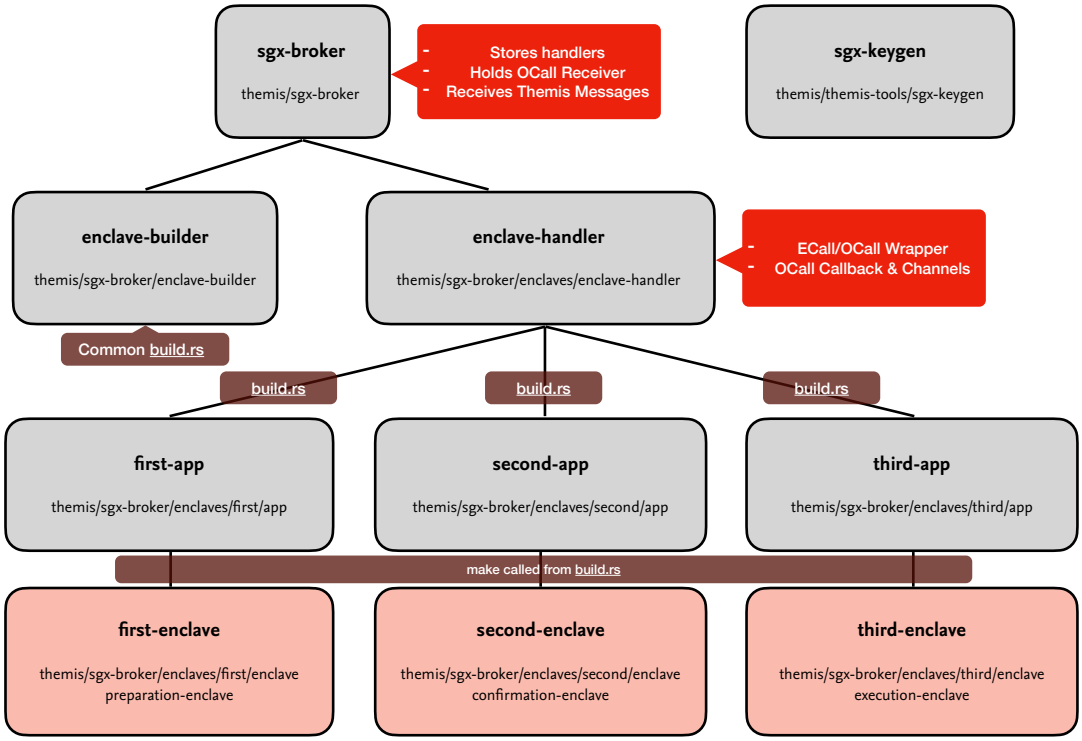

Figure 4.1.: Rust code structure added to Themis.

v1.1.3 documentation to implement the enclaves as no_std Rust library crates. To simulate the functionality of the Rust standard library the Teaclave SGX SDK's standard library replacement is renamed to "std". This allows the enclaves to use Rust code developed for a normal runtime environment for the most part. The replacement standard library does not match the Rust standard library entirely and misses some features. Furthermore, this means that only no_std dependencies can be used, or crates which have the exact same Teaclave SGX SDK library replacement.

The enclave handling code is designed and implemented with the intent of being flexible on where the threading-boundary is drawn, i.e. how many threads are used. This is, for example, the reason that the untrusted side of each enclave, which implements the `enclave-handler` interface, communicates with the broker using a multi-producer-single-consumer channel. This allows us to easily have the enclave handlers run in their own threads or using non-switchless SGX calls, and still allow Themis and the other enclaves to make progress at the same time.

Additional to the necessary code to run SplitBFT, we also added tooling to help with deployment and compilation. The `sgx-keygen` binary is used once to generate keys which are in the correct format for the OCalls of the enclaves. We use the Rust's ring library version 0.16.20 to generate PKCS#8 key pairs for each member of the protocol. This key generation needed to be altered from Themis' included generator, as more keys are needed.

Furthermore, in Themis with PBFT all keys authenticate the replicas equally, however, in SplitBFT the key is also used to identify the type of the enclave. Therefore, it is important that all replicas agree on the mapping of keys to enclave types. In practise, this is ensured by generating the keys in advance and deploying them to all machines and signing the configuration. The key $\leftrightarrow$ replica $\leftrightarrow$ compartment mapping is also represented in the directory structure and file name of the key on the file system.

The `enclave-builder` module contains generic code used in the `build.rs` files of the enclaves. This enables the use of `cargo build` and other Cargo commands in the Themis repository. An approach like this hides the internal use of make files, C-compilers and SGX tools necessary to compile the enclaves. Using only Cargo to build the entire project will also aid in testing and using CI/CD. One shortcoming of this approach, however, is that only the build process can be augmented using scripts this way. We would have made use of a `cargo clean` and `cargo test` hook to clean and test the artifacts created by the make files, which the cargo toolchain itself cannot do yet.

The same custom Rust build scripts are also used to make the Rust compiler link to the untrusted side of each compartment. Those custom build scripts also call `make` in the correct order to build the enclaves. The make files are altered versions of the Teaclave example code, which are based on the Intel SGX SDK. Changes were needed to be made to rename symbols inside the built enclave to avoid collisions when linking multiple different enclaves to a single binary. The colliding symbols originate from the trusted standard library's initialisation code in the Teaclave Rust SDK. Using the `objcopy` program the conflicting symbols are prefixed by the compartment name just before linking all artifacts together.

Rust's build scripts allow defining a list of files and folders to watch for changes. Only when mentioned files are changed the build scripts are run. This is used to limit the amount of `make` calls, which are the slowest part of the current build process. Similarly, using environment triggers the project is also rebuilt if some of the SGX environment variables change. This includes the location of the SGX SDK and whether or not to build in software emulation or hardware mode.

Also part of the framework at large are the applications replicated with BFT. We will be using two pre-existing applications and adapt them to run inside the execution enclave with SplitBFT.

4.1.1. Benchmark Application

To test and evaluate the implementation of SplitBFT we use different applications. A minimal application present in the Themis framework for PBFT is the so called "Benchmark Application". This application accepts any incoming request and responds with a small payload which can be ignored by the clients. A client is already present as part of the Themis framework. For the benchmark application, the requests only consist of no-ops, but the rest of the protocol needs to be executed as usual. This includes all communication rounds, signature checks and the correct encryption and decryption of requests. This

means once a commit quorum is present in the execution enclave the request is ignored and an empty reply is sent to the client. This implementation also acts as a foundation to implement further applications, as only the execution enclave needs to be changed in terms of how the request is to be interpreted when a commit quorum is present. Even though the requests and replies of the benchmark application are ignored by the replicas and clients the size of the requests and replies can be altered to simulate the load on the network and memory better.

4.1.2. Key-Value Store Application

The real-world application that is being replicated using SplitBFT is a key-value store (KVS) which implements the Yahoo! Cloud Serving Benchmark (YCSB) operations [12]. The data is stored in the KVS, similar to a hash map indexed by generic byte arrays. Values are also generic byte arrays. Operations include reading a record, updating a record and deleting a record. For the application in a BFT-SMR system the KVS also supports exporting and importing a state. The hash over the state needs to be identical on all replicas whose state is identical, which means no randomized hash maps can be used. Instead, a B-Tree is used internally to ensure a deterministic state. The open-source YCSB framework was developed to allow direct comparisons between data stores like Cassandra, HBase, PNUTS and MySQL. It has been used in similar contexts to benchmark KVS and database applications using trusted execution [34]. We focus on the most commonly used operations, the CRUD API includes insert, update, delete and read operations.

YCSB is used for benchmarking cloud services using configurable workloads. The workloads include different distributions of read, write and update operations. Workloads can also be executed in stages. Workloads begin with a load phase where data is inserted into the database. The next phase runs the actual specified workload. The workloads are designed to mimic data access in realistic patterns. YCSB supports various access patterns. They mirror the real world where a few records get requested more frequently than most others. This is similar to observed behaviour for in-memory caches as deployed by Twitter [48]. The access patterns can be mixed with the read-write frequency to create more complex workloads.

Figure 4.2 shows the starting point and necessary adaptations to compartmentalize an existing KV-Store application for the Themis framework. The red components of the diagram, namely the SGX broker and the E/OCall interface, are the main focus of this thesis. They contain all necessary parts for the SplitBFT protocol to run within Themis.

To use SplitBFT with a new application, like a key-value store, the client does not need to be changed. Only the replicated application itself needs to be placed into the execution enclave, which might require slight changes.

With an existing YCSB Themis client, application wrapper and minimal KV-Store implementation almost no changes are necessary. The data store was already implemented with minimal dependencies and a small footprint, therefore, only small code snippets needed to be adapted for the Teaclave standard library implementation. This was limited

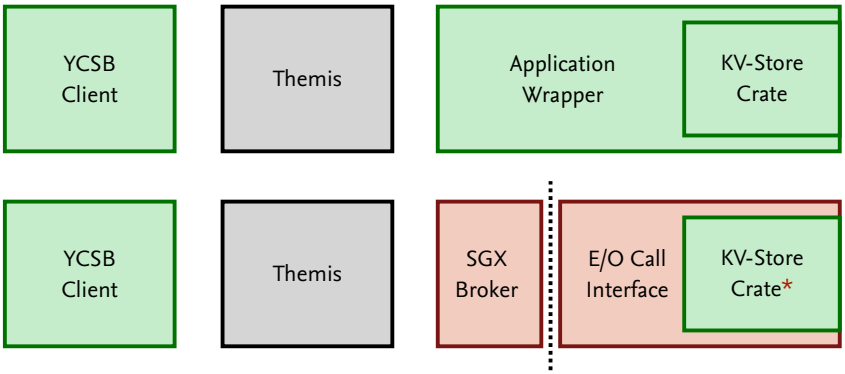

Figure 4.2.: Adapting KV-Store application for SplitBFT. The dotted line represents the the trusted computing base's boundary.

to changing and fixing dependency versions as well as renaming or aliasing types. Generally, common changes will be renaming the `Mutex` type to `SgxMutex` as well as including some traits manually. The reason traits need to be included is that the prelude in the Teaclave SDK's standard library includes fewer traits than the normal standard library. One of the missing traits, for example, is `ToString`, which exists in the Teaclave standard library, but is not included in the preamble. When preparing a crate to be used inside the enclave, another step can be eliminating allocations and introducing trait bounds which do not require allocations like `AsRef<str>` and `AsRef<[u8]>`. It is important to reduce allocations for performance reasons, but also because the enclave's memory is practically limited by the EPC.

The application wrapper needs to be reimplemented, as it is now split across the untrusted and trusted side. It needs to implement Themis' interface on the untrusted side, but perform protocol operations, especially cryptographic functions, in the TEE in encrypted memory. Because of the design of the broker no application module is deployed on the untrusted side. Instead, the execution enclave deserializes the operation into an enum of possible actions and then executes it accordingly. Responses are sent using OCalls directly from the execution compartment as part of the protocol module. The client benefits from the gained integrity and confidentiality without any changes, as it is completely oblivious to whether it is talking to a compartmentalized environment or a single application.

4.2. Dependency Versioning

Themis' master branch at the time of implementation used Rust version `nightly` from `2021-02-17`. The Teaclave Rust SDK's latest stable release was version 1.1.3, using Rust `nightly` but from `2020-10-25` [2]. The community added further patches on top of the stable release to support newer Rust nightly versions. To facilitate interoperability of the

Themis framework and the Rust SDK without back-porting the Themis framework Teaclave SDK's commit `b9d1bda6...` was used for development of the enclaves in Themis. The goal is to be able to update to the newest version of the Teaclave SDK, without having to change the enclave code. The reason the stable Teaclave SDK commit tagged with v1.1.3 could not be used is because of a deprecated and removed nightly trait in `core::alloc` called `AllocRef` used in `sgx_alloc::System` which is required for `sgx_tstd`.

Because `cargo`, the Rust toolchain's package manager and build tool, doesn't allow changing major versions of deep dependencies, and we needed to change even the dependencies' versions of the Teaclave SDK to the specifically chosen commit, and not the v1.1.3 tag. All used crates needed to be forked, changed and updated. This was trivial for the Teaclave SDK's internal crates, as they used path dependencies, and checking out the correct version suffices. The external crates, however, depended on the revision v1.1.3 which uses the git tag of the Teaclave SDK's repository to determine the code used. We built on the work of dinglish/sgx-world and forked it to mabecker/sgx-world.

A script was written and used to updated all mentions of the tag v1.1.3 to the chosen commit, and we then manually create and pushed the changed repositories to forks created on our GitLab.

4.3. Code Reuse

Sharing code between the untrusted side and the trusted enclaves is not straight-forward. Because the enclave and the untrusted side use different `std` libraries no code making use of the standard library could be shared. The workaround for this is writing all code, that is possible to write without using the standard library, as a `no-std` module. Small reusable Rust snippets often only need access to the `alloc` crate for common collections and heap allocations. However, to be included from an enclave a crate is not allowed to be part of the Themis module tree, as even including a `no-std` module out of a larger module tree triggers an error about two competing `std` implementations. In practise, we have found that a symbolic link between a version that is excluded from the Themis module tree but path-linked to the enclave, and a version that is included in the Themis module tree does work. This even allows there to be tests and documentation that are generated as part of the Themis workspace.

There are also other alternative approaches, which lack this integration but are easier and faster to accomplish. The first, which is used for code which is slightly different between enclaves, is the inclusion of Rust code containing a macro. By using the `include!` macro, a file can be included in rust code directly. If this file then defines a macro, the macro can be called and create slightly different code for each enclave. The downside of this approach is that the main file is not considered part of the workspace and tools and IDEs do not know how to treat it.

A second alternative is creating symbolic links to Rust files. This allows the parent to define a submodule in Rust which is then defined by the linked file. The advantage of this is a slightly easier to read code. The linked files are also considered part of the workspace and, therefore, work better with linters [17] and other programming tools which are usually well integrated in the Rust toolchain. One downside of this approach is that the symlinks do not always report changes when the target changed, which means changes might not be present everywhere in newer builds without cleaning old artifacts.

These approaches can also be mixed. We use all three methods to share code. We have a no-std module, which is included by both the untrusted and trusted side using the renaming scheme. Inside of this module we use the `include!` macro to create another macro definition from file. This macro definition is called directly in the shared code for the untrusted side, the trusted sides, however, call the included macro once each themselves. This macro contains most of the type definitions used to communicate over the network and across the ECall/OCall boundary. Finally, a symlinked file is used in each compartment to define the structure and global variables for the log inside each enclave.

4.4. Diversification

We want the SplitBFT protocol to be able to be diversified. The untrusted side of SplitBFT can be diversified like any other BFT protocol implementation. Compartments can be diversified in code as well as in the TEE technology used.

To diversify the compartments one approach is re-implementing the SGX enclaves by hand. This can also be done automatically [37]. Because the interface with the enclaves is pre-defined by the ECalls and OCalls, it is easy to make sure a different implementation has the same interface. This also allows re-implementation in different languages, as the function interface with the ECalls and OCalls is language independent. All data that is sent with ECalls and OCalls are either raw pointers to memory locations the enclave can use or serialized data. Both of which are also language independent.

Diverse binaries can be achieved using N-version development by different teams [8] or can be automatically generated from a single source [37]. Because Rust's foreign function interface (FFI) can be used with statically linked binaries from other C-ABI languages, diversification can implement parts of the enclaves in different languages and keep a small Rust wrapper. This could speed up development of the diverse enclave implementations. The wrapper would only have to handle ECalls and OCalls and then call the FFI with the deserialized messages. This wrapper would also need to be diversified.

4.5. ECall & OCall Interface

We choose to implement a generic ECall and OCall interface to aid diversification. By having only a single ECall and OCall, which accepts arbitrary data we have the ability to pass serialized data in a format which gets deserialized on either side. Using an enum

type we gain the ability to encode multiple ECalls and OCalls using the single defined physical ECall and OCall. This gives us great flexibility when choosing how to process a deserialized call as well as eases re-implementing in another language.

After enclaves are created using the Teaclave SGX SDK they are written to expect a special initialization ECall, which they only accept as their first ECall and only accept once. This `Init` ECall contains only a single field, an arbitrary sized byte array, which can contain arbitrary data specific to each enclave and application. In our implementation we chose to use this data for loading runtime information derived from the configuration files, like key paths and system information. The extreme alternatives would be either to load this data using OCalls on demand, or hard-coding this data into the enclaves. Hard-coding data into the enclaves would have the advantage that a compiled and signed enclave cannot be tampered with until started, thus the keys' integrity is not broken. If the untrusted side attempted to pass incorrect keys and configuration to the enclave, the enclave would be able to detect this by checking signatures and not accept incorrect data. This means the untrusted side can at most cause the enclave to not initialize, which only attacks liveness. In execution enclaves, the response to the client is encrypted using a symmetric key. Leaking this symmetric key from the execution enclave would allow an attacker on the network to read replies to the client from this enclave. This breaks confidentiality. Therefore, we need to ensure the symmetric key used for this purpose is also signed and verified by the execution compartment. Otherwise the untrusted side could pass a different key and cause the loss of confidentiality with a single fault on the untrusted side.

For these reasons it is necessary to sign all key-data with a key that is hard-coded into the enclaves. Keys can also be loaded into the enclave out of the untrusted file system as long as the keys from the untrusted file system are signed by a key which is hard-coded into the enclave. The key used to verify the configuration should only be held by the administrator of the replica. By compiling enclaves with different hard-coded keys for different replicas, administrators can also be limited to be able to affect single machines only. In a setup like that, we still have the flexibility to change the configuration without recompiling the enclaves, but know that an attacker is not able to convince an enclave to use incorrect keys, as the attacker is not able to break asymmetric ECDSA-25519 cryptography. Hashes over the data are created using the SHA-512 algorithm, which the attacker is also not able to find collisions in.

The trait definition in code listing 1 is added to a Rust file when the `make_shared` macro is invoked. The definition of the macro is inserted from a file. This is a work around we found to allow code to be included in both the untrusted and trusted parts of code while using the `std` library. The code generated by the macro invocation is around 2000 formatted lines of code including documentation. The snippet displayed in the following listing creates the interface for calling the ECall and OCall wrapper generated by the EDL file on any serializable type. Both of these functions are used on the inside and outside of the compartments:

```
pub trait SgxArgument<'de>:
    Deserialize<'de> + Serialize + fmt::Debug
{
    fn call<F, R>(
        &self, sgx_call: F, mutable_value: &mut R,
    ) -> Result<sgx_status_t, sgx_status_t>
    where
        F: Fn(*const u8, usize, &mut R) -> sgx_status_t,
    {
        let boxed_slice: boxed::Box<[u8]> = ::rmp_serde::to_vec(self)
            .map_err(/* ... */)?
            .into_boxed_slice();
        let retval = {
            let len = boxed_slice.len();
            let ptr = boxed_slice.as_ptr();
            sgx_call(ptr as *const u8, len, mutable_value)
        };
        Ok(retval)
    }

    unsafe fn from_raw(
        data: *const u8, len: usize,
    ) -> Result<Self, sgx_status_t> {
        let data = slice::from_raw_parts(data, len);
        ::rmp_serde::from_slice(data)
            .map_err(|_| sgx_status_t::SGX_ERROR_INVALID_PARAMETER)
    }
}
```

Listing 1: Generic ECall and OCall interface for serializable types.

When using the `call` function the argument type is serialized and the safe wrapper for the SGX call is called. This is done to be able to call the ECalls and OCalls with arbitrary types without using unsafe code to do so. Which ECall or OCall is called is determined by a pointer to the function as the first argument. This function is then called with the serialized second argument as a byte slice and its length. A mutable reference can be provided to allow the target function to return a value alongside the SGX return status. This translation is analogous to the Rust and C interface generated by the Intel SGX SDK.

On the receiving end, arguments are deserialized using the `from_raw` function. First a slice is constructed from a raw pointer and its length. This operation is unsafe and requires the function to be called with correct arguments. Therefore, the function is marked as `unsafe`, which requires calling code to wrap it in an `unsafe` block. This encourages callers to be cautious. The caller also determines the return type using type annotations.

The Themis framework already uses MessagePack encoding for serialization of protocol messages across the network [39]. MessagePack is also used to serialize data which is passed in and out of enclaves. The serialization format is independent, but MessagePack is a good fit for ECall/OCall arguments.

Figure 4.3 shows how messages are serialized and passed between clients and compartments. MessagePack is also used to serialize messages for communication between the replicas themselves.

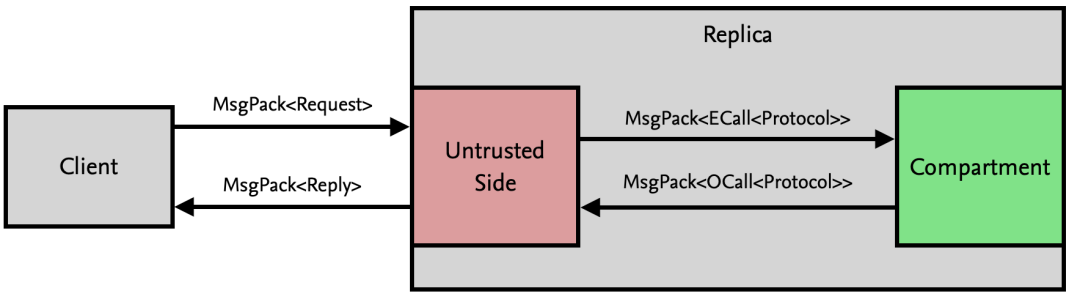

Figure 4.3.: Serialization and communication scheme of protocol messages.

When serializing, the signature of a message is considered part of the message. For each message type there is a method for creating a digest independent of the signature field. This independent digest is used to create and append the signature without changing the digest itself. This design decision does require deserialization for signature checks, however, deserialization is already required to determine which public key to check against. The advantages of MessagePack is that is is an efficient binary format, which is faster and smaller than JSON. For example, a ECall informing the enclave about an expired timer `Timer{MillisSinceLastRequest(1000)}` is serialized into 8 bytes as `[129, 9, 145, 129, 0, 205, 3, 232]`. The smallest JSON version of this struct is `{9:[{0:1000}]}` using 14 characters (112 bytes), which makes JSON 14× larger in this example. This format can be easily deserialized because the `ECall` and `OCall` type is an enum, the 9 in the serialization is the number of the enum variant, the contents of the contained array are the inner values. The variant number and interpretation of the inner attributes is based on the Rust implementation, which is identical on the untrusted side and the enclaves. Implementations in other programming languages have to have the correct mapping, but are not required to use the same types internally.

4.6. Broker Implementation

As mentioned above, the broker represents the translation layer between Themis and the SGX enclaves. It brokers and routes messages between the network and the enclaves using enclave handlers. To conform to the Themis interface requirements the broker is a single Rust type containing references to the handlers, which initialize and pass messages to the enclaves and can be polled for OCalls. The broker also supports configurable request batching from the communication module of Themis [39].

Themis provides two different interfaces a protocol module has to implement. The `Protocol` trait is used to provide a simple interface to define message and event types. The `Protocol2` trait ensures a type has all necessary callbacks Themis uses to call into a protocol module. The broker implements both the `Protocol` and `Protocol2` interfaces to be instantiated as a protocol in Themis. This ensures that the broker is able to handle messages from the other module types. This includes messages from the client, messages from other replica's protocol modules and messages from the local application module. Themis itself uses the tokio library version 1.8.2 and asynchronous Rust to schedule work and enable concurrent execution. It uses tokio's `mio` and `mpsc` for asynchronous IO and multi-producer-single-consumer message channels respectively. The interface that modules need to implement forces them to use the same libraries. We define a simpler network message type than the included PBFT messages, because messages need to be deserialized anyway and explicit tagging before deserialization is not required. For incoming messages, the broker determines the target enclave and routes the message into those enclaves using immediate ECalls. After performing an ECall, the broker queries its enclave handlers for new OCalls which could have been posted in the meantime. The enclaves can only post OCalls to the queue while they are called with and ECall. This means the broker only has to check for new OCalls after an ECall. Because the OCall generated by the Teaclave SGX SDK is static and not bound to an object created within the broker we need to register the handlers in a global list for each enclave type to receive OCalls. The complexity of this structure is handled per enclave type in its untrusted side and only linked to the broker and Themis as the final compilation step. When the enclaves OCall reaches the static code in the untrusted side it uses a global store behind a mutex to get references to the registered handlers. Each handler then enqueues the call data into an internal queue, but the call is not handled immediately. At a later time, after the initial ECall has been completed the broker queries each handler. If any OCalls exist, they are consumed and executed in order. Processing OCalls may require new ECalls to be issued, which may lead to longer execution chains. Outgoing protocol message OCalls always require both ECalls into the local target enclave types as well as network messages to be sent. When, for example, the preparation enclave wants to send a prepare message to the confirmation compartments $n - 1$ messages will be sent over the network to other replicas but one message will be passed to the local confirmation enclave using an ECall. Because we batch

OCalls as much as possible, a single OCall is performed to instruct the untrusted side to send all these messages. All OCall network operations are executed asynchronously by spawning a new worker in tokio.

Because the execution enclave also has to contain the application itself, the broker also performs the actions of an application module which is otherwise absent when running SplitBFT in Themis. The broker layer could also have implemented the application interface, which would allow it being used as both the application and protocol module in Themis. This would be closer to the intended Themis structure, but would lead to some problems. First, entirely technically, the modules are meant to be different to eventually be executed concurrently. But second, and more importantly for SplitBFT to function it would be required to instantiate Themis with the exact same broker as protocol and application module anyway. Having the broker implement both interfaces would make it seem as if it could be used individually as either, but because the execution enclave containing the application and participating in the protocol needs to be the same the broker could only be used with itself. Because this would lead to confusion, we decided to implement the application completely opaque to Themis within the protocol module.

When request batching is enabled the request batching scheme from the communication module in Themis is used. Instead of sending single requests to the preparation compartment batches are always used. These batches may only contain a single request. The preparation compartments check that all requests inside the batch are correctly signed before preparing. When ordering a batch only the digest of the batch is used. The execution compartment receives the batch from the preparation compartments and when a commit quorum is present all inner requests are decrypted and executed in order.

4.7. Enclave Implementation

The SGX enclaves are implemented as Rust library crates which compile to a static library. Afterwards, the usual Intel SGX SDK toolchain can be used to generate a signed enclave binary from the static library. It uses an EDL file to generate the wrapper for ECalls and OCalls which are used on the untrusted and trusted side. The untrusted side as part of Themis is then able to spawn the enclave using the untrusted side of the Teaclave SGX SDK. Which programming language was used to write the enclave is independent of the untrusted side as long as the same functions are exposed as ECalls.

The general structure of the implemented Rust enclaves is very similar. They all contain definitions for the different types of ECalls and OCalls. To keep the EDL interface simple, they only expose a single ECall accepting arbitrary data which then deserializes into a variant of an enum. These ECalls are initially handled identically in all enclaves. This is because their initialization requires the same steps, in which they fetch the necessary configuration and keys. However, if the ECall contains a protocol message the way the message is handled changes depending on the protocol message variant and the enclave type.

Because of the unique runtime environment and dependency problems we are limited to a few possible dependencies to fulfill common tasks. We use the ring crate to perform cryptographic hashing and encryption. To serialize and deserialze on the trusted and untrusted side we use serde version 1.0, and more specifically the rmp_serde crate version 0.15.4 to work with the MessagePack binary format.

4.7.1. Protocol Messages

Protocol messages are represented as an enum of variants from table 3.1 containing all necessary data. The structure is taken from the PBFT protocol, however, because of the changed signing scheme the keys used for signing and encryption are changed as described in the design chapter. Protocol messages are created in the client and the enclaves. They travel over the network wrapped in `struct` containing information about the type of protocol used to create the contained message. When entering or leaving the enclaves, protocol messages are placed in a wrapper `struct` as well and the entire wrapper is serialized. The Themis framework enforces tagged messages for network communications. This means for network communication all messages are wrapped in a type containing the protocol message's serialized data alongside this tag. For SplitBFT we use a placeholder tag, as it does not need to be used. The tag is intended to closely mirror the notation of BFT papers like PBFT [8]. Because the tag cannot be trusted, as it is handled and added by the untrusted side and only the serialized inner message is signed by an enclave, tagging is not useful for SplitBFT. When a network message is exchanged it is deserialized in steps, first only the outer type containing the tag. This wrapper could contain additional information. For example, the PBFT implementation contains the option to add optional flags for `READ_ONLY`, `HASH_REPLY`, `UNSEQUENCED` and `NO_REPLY`. While these could be used for the SplitBFT implementation too, we instead would use the protocol message variant to encode these options. The implemented applications do not make use of these flags.

4.7.2. Message Log

All enclaves have their own global message log. This object is used to securely store past messages which are still relevant to the agreement process. The message log is stored as a `SplitBftLog` defined in code listing 2, which deduplicates messages as much as possible. Because only verified messages are added to the log at all, the untrusted side can only replay old messages to pass the signature check. This avoids the untrusted side performing a denial-of-service attack by filling the compartment's memory with replayed messages. The untrusted side cannot construct new messages that pass the signature check because the necessary key is only present in other compartments. All functions interacting or returning data from the log return references, this reduces the amount of copies. After a message is deserialized from the ECall it is added to the log once, following uses of the

message always use a reference. When a message is sent using data from the log, a reference from the log is used to serialize, to avoid another copy. Each deserialized or created message only takes up memory once in each enclave at most.

```
struct CheckpointIndex {
    sequence_number: Id,
    state: Data,
}

pub struct SplitBftLog {
    messages: Vec<Arc<SplitBftMessage>>,
    checkpoint_messages: HashMap<
        CheckpointIndex,
        HashSet<Arc<SplitBftMessage>>
    >,
    low_mark: SequenceNumber,
}
```

Listing 2: SplitBFT log format used inside of the compartments.

The `SplitBftLog` contains a second data structure for checkpoint messages. The checkpoint messages are also added to the general message log. This is the reason for using `Arc`s to store the messages. They allow shared ownership of the underlying message, once all references are dropped, the memory of the message is freed. When a checkpoint message is added to the log, it is also interpreted and added to the correct set in the `SplitBftLog::checkpoint_messages` member based on the sequence number and state. This additional structure is only used to speed up finding the newest stable checkpoint. Once a new checkpoint is stable checkpoint messages need to be removed from the hash map as well as from the general message vector. Otherwise, the memory would not be freed. This, however, is easy and fast as deleting all values in the hash map with a key of earlier sequence number is enough. Garbage collecting messages from the general message log is also linear to the length of the message log. A `DrainFilter` is used to remove all messages of an older sequence number. Using a `DrainFilter` like this also returns the messages it removes from the log. This allows us to check that messages are successfully removed from both the log and checkpoint hash map, because we can check the number of remaining references to the returned messages. As long as only one reference is left, the drained one, we can drop it knowing well that this frees the memory associated with the message. These data structures could also be used while having the memory for the actual message data located outside of the enclaves. Doing so, however, would require complex integrity checks. Because the EPC size can be extended using new multi-processor platforms and in our testing memory size never became a limit for the applications we are using we choose to keep the messages in the enclaves.

4.8. Continous Integration

We use the CI/CD capabilities of GitLab to ensure functionality of Themis. CI/CD checks every commit that our SplitBFT code does not break its own tests or any of Themis' tests. The Rust programming language includes tools for testing as part of the standard toolchain. It makes it convenient and easy to write unit and integration tests. Due to the different runtime environment of the enclaves the same tooling cannot be used inside the enclaves. However, the workaround we found to still test enclave code in CI is compiling using SGX software emulation mode and writing tests on the untrusted side which spawn an enclave and cause it to run a test. The Teaclave SGX SDK provides its own testing framework, however, we did not find it adaptable enough to run in CI. Having the ECall interface be as generic and flexible as it is allows us to declare tests on the untrusted which are executed in the enclave and results get reported back. Using a conventional ECall interface would require defining additional ECalls in the EDL interface specifically for testing.

Some unsafe Rust code has been verified with Coq in the past [25]. Coq performs mechanised formal verification. Creating a formal model for a BFT protocol is a complex and error-prone task. We, instead, only verified selected pieces of unsafe code present in all enclaves using `miri` [16]. Unlike of a formal verification, `miri` only performs runtime analysis and checks for some common types of runtime errors. For example, listing 1 was tested and `miri` confirmed that provided that all arguments are not maliciously constructed the call is safe. This gives us confidence that these pieces of code will not segfault or leak memory during normal operation. Especially with code that is inside the enclaves this ensures some common attacks like overflows are unlikely to be possible.

A CI-Pipeline tests both the framework and the added SGX-specific code. The pipeline uses the Gitlab CI runner, compiling and testing the entire Themis project, including the additions made in this thesis, inside a docker container made for this purpose. The CI docker container `mtibbecker/sgx-Rust` with tag `2021-02-17` is based on the Teaclave SGX SDK developer's `baiduxlab/sgx-Rust` container. The base container is an Ubuntu 20.04 container with a lot of additional dependencies needed for SGX and Rust. A result from that is the rather large size of the base container itself, reaching $\approx$ 650 MB when compressed. Changes made on top of this container for use in CI include updating the Rust and cargo versions to the versions used in Themis with SplitBFT. This increases the compressed container size to 844.46 MB. The CI runner does cache containers, therefore, the entire image does not need to be downloaded every time the CI is executed.

The CI pipeline is run on every push to the git repository and even runs tests which start and enter the enclaves using SGX's software emulation mode. Due to a race-condition in the compilation of the `sgx-unwind` dependency of the Teaclave SGX SDK a compilation failure can happen randomly. GitLab-CI supports retrying a CI pipeline on failure, up to three attempts. While this makes a false negative in practice less likely, there is still a chance the project's CI pipeline gets marked as failed, even though the build and tests would succeed.

Unit Testing In comparison to the PBFT implementation in Themis, testing is fundamentally different because of the compartmentalisation. This is because we are able to test each enclave on their own. In contrast, for PBFT testing the entire replica needs to enter a wanted state to test it's behaviour at that point, we, however, only have to convince a single enclave to enter a state we want to test without involving the entire rest of the replica. For example, we are able to test the execution enclave by sending it correctly signed commit messages in a test environment, without the need to have pre-prepares and prepares agreed on by the other enclaves. However, fully integration testing the entire replica requires spinning up all the enclaves and broker, making this part of testing slightly more difficult than testing *standard* Rust code for the PBFT implementation. We develop ansible playbooks and other scripts to deploy and test SplitBFT on real machines alongside the unit tests in GitLab's CI/CD.

4.9. Problems with the Themis framework

Working inside the Themis framework to implement SplitBFT is generally convenient and intuitive. However, during the implementation process some possible opportunities to improve the usability of Themis in the future became visible.

For components of Themis to be interchangeable they have to implement a lot of different traits, to enable Themis to use them. Because Themis is modular and uses callbacks, it requires its modules to at least implement those necessary callbacks. This can, for example, be seen in the `Protocol`, `Protocol2` and `Application` trait. We found that because we combined the protocol and application trait some required functions were not used at all. Implementing the `Application` trait allows Themis to take control and possibly provide some functionality for checkpointing and fast-track requests using `RequestFlags`. While this could be far more beneficial with further development of Themis, these provided functionalities might, on the one hand, not be wanted or required or, on the other hand, not be worth the overhead of implementing the necessary traits and including their dependencies. In our case specifically, implementing the `Application` trait for the execution enclave would entail deserializing the request buffer, interpreting it (whether it is a request, a checkpoint, etc.), then reserializing and executing an ECall which then deserializes again and performs the operation requested from the outside if the data can be verified. The problem, however, is that the enclave needs to deserialize, verify and interpret the data anyway, and as it can not trust the outside to interpret requests correctly, would need to do work twice.

Instead, for easier prototyping and implementing protocols that are less similar to PBFT a simpler version of the Themis traits would be preferable. This trait would ideally be agnostic over request and protocol data, always treating it as a simple byte buffer without tags. An `Application` trait like this would also allow writing a simple adapter for any running binary to be used with the Themis framework, as long as a socket or channel passing bytes is available.

Themis could still perform network and batching operations, but should not force protocol tags on every network message. The simple protocol trait should not show the user any of the inner workings of Themis, like the threading with `tokio` or pre-parsing of the messages for protocol tags. Instead, a simple interface, just passing received buffers over the network are handed to the implementer of the simple protocol trait. Batching can be done, however, needs to be configured at the start using flags and attributes which are sent with every message. Like configuration now, this should be possible to do using the configuration files.

4.10. Problems with the Teaclave SGX SDK

While working with the Teaclave SGX SDK extensively some problems kept coming up. They all seem to originate from small design decisions at the core of the Teaclave SGX SDK which have wide ranging effects. It starts with the SDK not being structured like other Rust libraries which interact with the operating system and hardware. Normally `*-sys` crates bind and link themselves to whatever library they require. To use the Teaclave SGX SDK, however, not only does the repository be checked out locally for EDL and Makefiles for the compilation of the enclaves, but also the untrusted side needs to be manually linked to several libraries. While other `*-sys` need to do this once, Teaclave SGX SDK defers this linking step to the application, for no apparent reason. This complicated the build step, and makes examples and tests much harder to write as they are not linked the same way by Cargo like applications are. We were able to write build scripts to simplify the build processes, but build scripts like the ones we constructed could be included with example code or made unnecessary be a restructuring within the SDK.

Due to the particular build requirements and project structure the otherwise very consistent Rust documentation cannot be used. While the Teaclave SGX SDK crates are published to crates.io and therefore are attempted to be built on docs.rs, this fails since the build cannot succeed without a very specific setup and hardware support. This leads to very outdated or non-existent documentation in the regular locations. By adding and setting feature flags for conditional compilation a minimal version of the Teaclave SGX SDK could be made to compile on the docs.rs environment and at least provide some documentation at the usual locations. However, to cope with this on of the developers of the Teaclave SGX SDK host their own docs for each of the inner crates. Because there is no virtual workspace all the docs are generated separately and this makes it impossible to search the entire documentation for a type or function without knowing the exact crate it should be contained in. Also, because this is not an automated process for each SDK version, the documentation hosted is for the current stable version, and is not visible for newer or older versions. Because we needed to upgrade the Teaclave SGX SDK to a newer version to support Rust language features Themis uses, we had to compile and host our own documentation for the version we used. During this process we found that some parts of the Teaclave SGX SDK are documented very little or not at all. This lack of up-

dated and useful documentation created an incentive to start our own documentation of the Teaclave SGX SDK concerned with its quirks and features on our own GitLab Wiki. To do this we developed a script, which most likely is similar to the script used by the Teaclave developers to generate their documentation. While it is understandable that Rust nightly versions will have breaking changes regularly, it would have helped with porting dependencies and deciding on a version to use if more versions were tagged and numbered in the Teaclave SGX SDK. At the time of writing the master branch's version is completely incompatible with the latest stable version but it has not received a version bump. Any dependency using it, even if the dependencies didn't replace the standard library, are not compatible with both versions.

An additional choice which lead to us spending considerable amounts of time and effort updating and porting dependencies is the design philosophy behind the "sgx-world" repository. This repository is a collection of forks of commonly used Rust modules with a few changes. Most importantly they are changed to compile without a standard library by the inner `#![no_std]` attribute. Then, to allow them to use the re-implemented standard library inside the enclaves, the SGX standard library is included and rename to "std", which replaces it in the rest of the project. This replacement means that a module like this can never be used together with another which doesn't have the exact same replacement. Otherwise the conflicting "std" definitions collide. This also occurs if the used standard library replacement is a different version of the SGX standard library. This could be made a lot easier to maintain if instead imports to the correct SGX standard library were added to the necessary files. While this would be very complex for large projects, it could be done semi-automatically, close to our approach presented which updates changes the version of the standard library used. Or different Teaclave SGX SDK could be included using feature flags from an automatically generated selection by a macro. However, most sgx-world crates are small or at least limited to a few files and a process like this would be very simple. Most of the difficulties of the replacement are also self-imposed by the Teaclave SGX SDK like the apparently unnecessary renaming of the "Mutex" and "RwLock" to "SgxMutex" and "SgxRwLock" Ideally, the sgx-world approach would be abandoned in favour of adding a feature flag or target profile to use the SGX types.

5. Evaluation

During the implementation of SplitBFT in Themis, we use benchmarks to guide our choices. Especially inside of the enclaves, where a different runtime environment is found to most normal software. We evaluate possible dependencies' performance for cryptographic operations. We use benchmarks of the system running locally and distributed on a set of networked machines to determine and compare SplitBFT's performance to PBFT. To determine the limits of the performance of SplitBFT, we use a benchmark application for maximal throughput. For the sake of representative results, we deploy SplitBFT with a KVS and run realistic workloads. We also examine the final SplitBFT enclaves using different metrics to measure the size of the TCB.

5.1. Hashing Algorithm

Hashing is used to create a smaller digest of larger request or protocol messages used to reference those messages. The hashes need to be cryptographically secure to ensure they cannot be forged for the application within a BFT protocol. This is necessary, because otherwise a malicious member could create and distribute forged messages with matching hashes to a correct request. This would disrupt the integrity of the system, because different execution compartments would execute different operations. Hashes also create the foundation for digital signatures, which we use to sign all messages. Therefore, hashing needs to be fast and secure.

Inside the enclaves we have access to a few hash functions. We will run benchmarks to aid in choosing the best hashing algorithm to use for SplitBFT. The following benchmarks and measurements are taken inside a Rust SGX Enclave. The host is running Ubuntu Bionic 18.04.5 LTS with `4.15.0-143-generic` kernel, on an Intel(R) Xeon(R) E-2176G, 12 Core hyperthreaded CPU running at 3.7GHz, boosting to 4.7GHz.

The benchmark measures the time taken to hash a list of realistic requests containing both primitive data and byte arrays. The measured hashing algorithms include:

1. `sgx_tstd_default`: A port of the default hashing algorithm in the Rust standard library. This hashing algorithm is allowed to change between Rust implementations, but in this instance is `SipHasher13` with zeroed keys. SipHash is meant to be performant first, it does permits keyed hashing for security, but while it is generally strong, it is not meant to be used for cryptographic hashing [3]. Another upside of this implementation is that it implements the `Hasher` interface from the standard library, and, therefore, is better integrated with general Rust code.

2. `sgx_tcrypto_sha1`: The Rust SGX SDK's `sgx_types` crate contains this undocumented Sha1 implementation, apparently present in the `libsgx_crypto` library v2.4, though also not documented in the Intel SGX Developer Reference [23]. As the benchmark will show, it is very performant, however, as it is neither documented nor cryptographically secure, should not be considered for this task.

3. `ring_sha512`: Implementation for SHA-512 in the Ring library. This implementation follows the specification FIPS 180-4. This is the fastest cryptographically secure hashing algorithm available in the enclaves.

4. `ring_sha384`: Implementation for SHA-384 in the Ring library. This implementation follows the specification FIPS 180-4.

5. `sgx_tcrypto_sha256`: A shallow wrapper around the `libsgx_crypto` library functions `sgx_sha256_init`, `sgx_sha256_update`, and `sgx_sha256_get_hash`. This allows a secure and hardware optimized implementation of the SHA-256 algorithm [23]. This is likely the reason why this version outperforms Ring's SHA-256 implementation.

6. `ring_sha1`: Deprecated implementation for SHA-1 in the Ring library. This implementation follows the specification FIPS 180-4. SHA-1 is not considered cryptographically safe and is only included here as a point of comparison. This is the only non SHA-2 family hashing supported by the Ring crate.

7. `ring_sha256`: Implementation for SHA-256 in the Ring library. This implementation follows the specification FIPS 180-4.

Hashing is performed using the "Init, Update, ..., Update, Finish" method. Another approach would be to serialize into `Vec<u8>`s and hash over the contiguous memory region. Serialization, however, is more expensive than the entire hashing, at approximately 20% more expensive than any of the tested hashing algorithms, therefore, becomes too much of an overhead, even without considering the unnecessary allocations and data duplication inside of the enclave. However, for messages to pass through the enclave boundary they need to be serialized, therefore, in the enclave, both the serialized and deserialized version of both incoming and outgoing messages are available.

To rank the mentioned hashing algorithms by performance and choose the optimal algorithm for this task we run the algorithms for different sized payloads inside the enclaves. The results of this benchmark is shown in figure 5.1. The enumeration above is sorted by the performance shown in the benchmark. Ring having the best performing secure hash function is a positive result for our application. It means that we can use the Ring library inside and outside of the enclave, as Ring is already used by Themis internally.

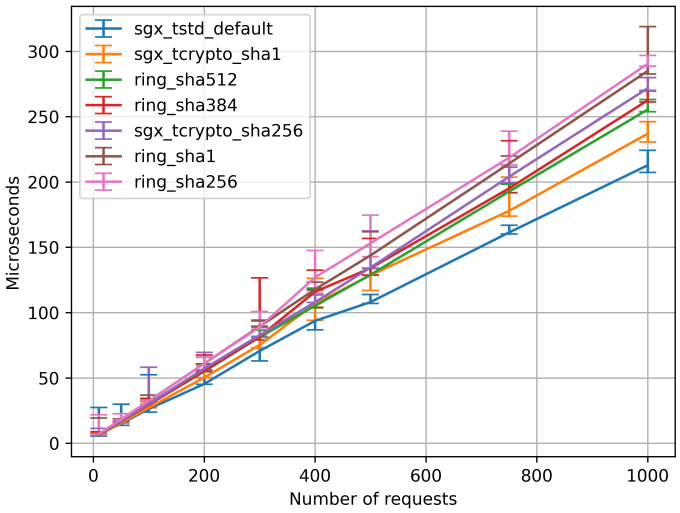

Figure 5.1.: Time spent in enclave hashing requests in a differently sized list using different hashing algorithms.

Because figure 5.1 is mostly linear, as is expected, and shows little variation we can determine the average hash time per request in the log for each of the algorithms. This cost per hashing operation is shown in figure 5.2.

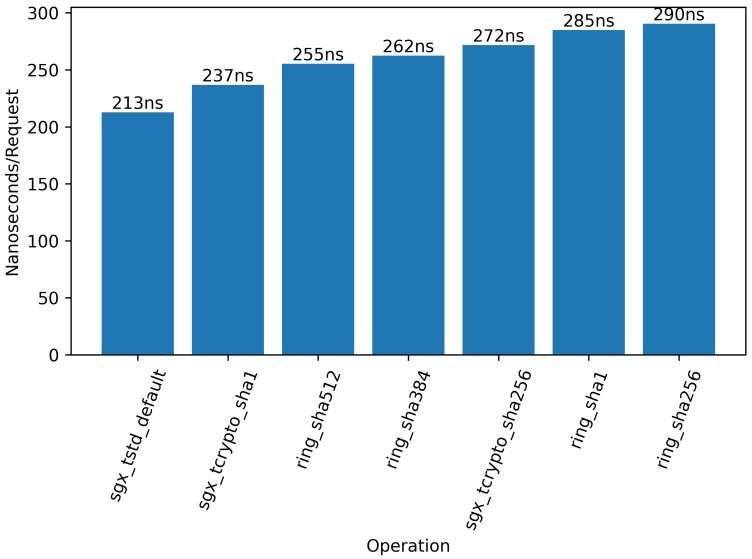

Figure 5.2.: Benchmark of hashing requests using different available hashing algorithms inside SGX enclaves.

The evaluation shows that Ring's SHA-512 implementation is the fastest hash from the SHA-2 family. This may be due to hardware support on the machine and might be different on different CPUs with other feature support. With SHA-512 being a common and secure hash to be used in all kinds of applications, it is likely to be optimized and supported further by many CPU vendors. Notably, the SHA-512 implementation even outperforms the smaller SHA implementations. This is most likely due to the interest in safer hashing algorithms and Ring as well as the CPU feature support aiming to optimize for SHA-512 hashing [19].

With features like SIMD SSE operating on larger words, the Ring library can perform optimal SHA-512 operations in CPU ring 3. The performance of the Teaclave SGX SDK's SHA-256 algorithm is still competitive and likely makes use of the same optimizations. Larger hashes benefit from this native support more than smaller hashes. However, the reason Ring outperforms the Teaclave SGX SDK's SHA-512 implementation is likely due to Ring's implementation's ability to be inlined and analyzed by the Rust compiler. The Teaclave SGX SDK links and calls the Intel SGX SDK's implementation in C, which can not be optimized in the same way.

5.2. Trusted Computing Base Size

Compartment	Binary size	LOC	Unique LOC
Preparation	2.9 MB	2444	525
Confirmation	2.6 MB	2291	372
Execution	2.9 MB	2423	504
Total	8.4 MB	7158	3320

Table 5.1.: TCB size of separate compartments in different metrics for the KVS application.

Table 5.1 shows the TCB size of the different compartments. The execution compartment contains an in-memory YCSB-compatible KVS implementation. Keeping the lines of code (LOC) as low as possible is desirable to reduce the likelihood of bugs. In the same vein, reducing the amount of code shared between compartments is advantageous, to guarantee independent failures. On the untrusted side, the broker layer defining the protocol module for Themis makes up 1074 additional lines of code. The remaining Themis framework itself contains 12240 lines of code. Once the replica code is compiled it creates a binary of 14 MB for the untrusted side. Some libraries, like libc, libpthread, libsgx_urts and libsgx_enclave_common are linked dynamically and configuration data is loaded from disk at runtime. The enclaves are linked statically. The LOC metric has to be interpreted cautiously, as automated diversification can achieve independent failures even with shared code. Most of the actual shared Rust code in the enclaves contains type definitions, which all enclaves need as well as the log structure, which is useful in and identical between all compartments. Not included in the LOC count are lines from the

Teaclave SGX SDK, serde and ring dependencies. The unique LOC almost entirely represent the protocol code, and how to handle parsed incoming protocol messages. It is also important to note that formatted Rust code is lower density than comparable system programming languages like C or C++.

5.3. Distributed Benchmarks

The most telling benchmarks of SplitBFT's performance are run across multiple machines. We use a `tmuxp` script and `ansible` playbook to build, distribute and run all benchmarks on a set of networked machines at once. Unless otherwise stated the benchmarks are run distributed on the machines from table 5.2. A bash script is written to automate benchmarking remotely with a range of different settings. During the benchmarks the configuration files are altered using `sed` and then re-distributed to the replicas and clients before the next benchmark is run. This way behaviour with different batching behaviour, agreement protocol and client numbers can be explored programmatically. A python script, using matplotlib and numpy, is used to automatically generate plots from the output of the benchmark bash script.

Amount	CPU	EPC Size	Bandwidth	Memory
1	Intel Xeon E-2176G	93.5MiB	1Gbps	32GB
4	Intel Xeon E3-1230 v5	93.5MiB	1Gbps	32GB
2	Intel Core i7-6700	93.5MiB	1Gbps	24GB

Table 5.2.: Systems used for distributed benchmarks

The general process includes using `ssh` to start an instance of Themis on one of the replicas. After giving the replicas a few seconds to initialize and establish connections between them the clients are started on the remaining machines. We use threading to achieve higher client numbers than we have physical hosts. The benchmarks are designed to not fill the clients with too large loads, to ensure that they are not the bottleneck of the system. The clients are split evenly across the remaining machines. To benchmark the performance of SplitBFT we both use minimal benchmark applications as well as the real-world YCSB benchmark to compare against PBFT [12].

5.3.1. Benchmark Applications

Figure 5.3 and figure 5.4 show the performance of a benchmark application. The benchmark application essentially performs no-ops after each request has been ordered. A batch is issued after at most 100 requests, after 500ms have elapsed or if the replica is otherwise idling. These limits are configurable in a configuration file. We run these benchmarks with different numbers of clients. The error bars show one standard deviation of the data across at least 10 runs.

By varying the number of clients we explore the performance of the replicas at different amounts of load. Especially with request batching it is expected that larger client numbers will make better use of the network bandwidth.

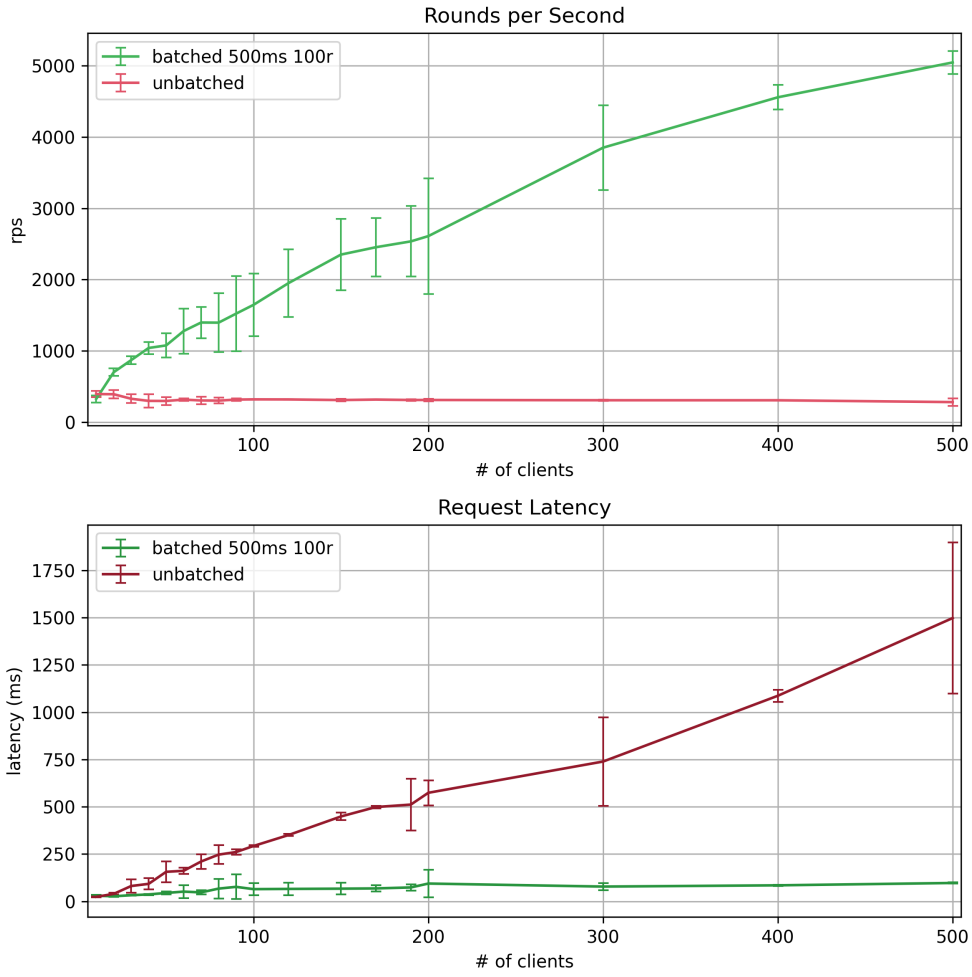

Figure 5.3.: Comparison of SplitBFT's local benchmark performance with and without request batching enabled.

Figure 5.3 shows a benchmark with all replicas and clients running locally on a single machine. On the one hand, this mostly eliminates the network speed as a variable, but increases the load on the machine dramatically. The performance is bottlenecked by either the CPU speed or the memory bandwidth, as the distributed benchmark of figure 5.4 performs better at higher client counts.

The performance difference, both in rounds-per-second (RPS) and average message latency, between batching and non-batching is shown. The difference is dramatic, as a batch of requests can be processed almost instantly, no matter the size. This would scale further, and is only limited on the network speeds and the memory bandwidths of the replicas. However, this shows that the preparation enclave, especially the leader, is able to create

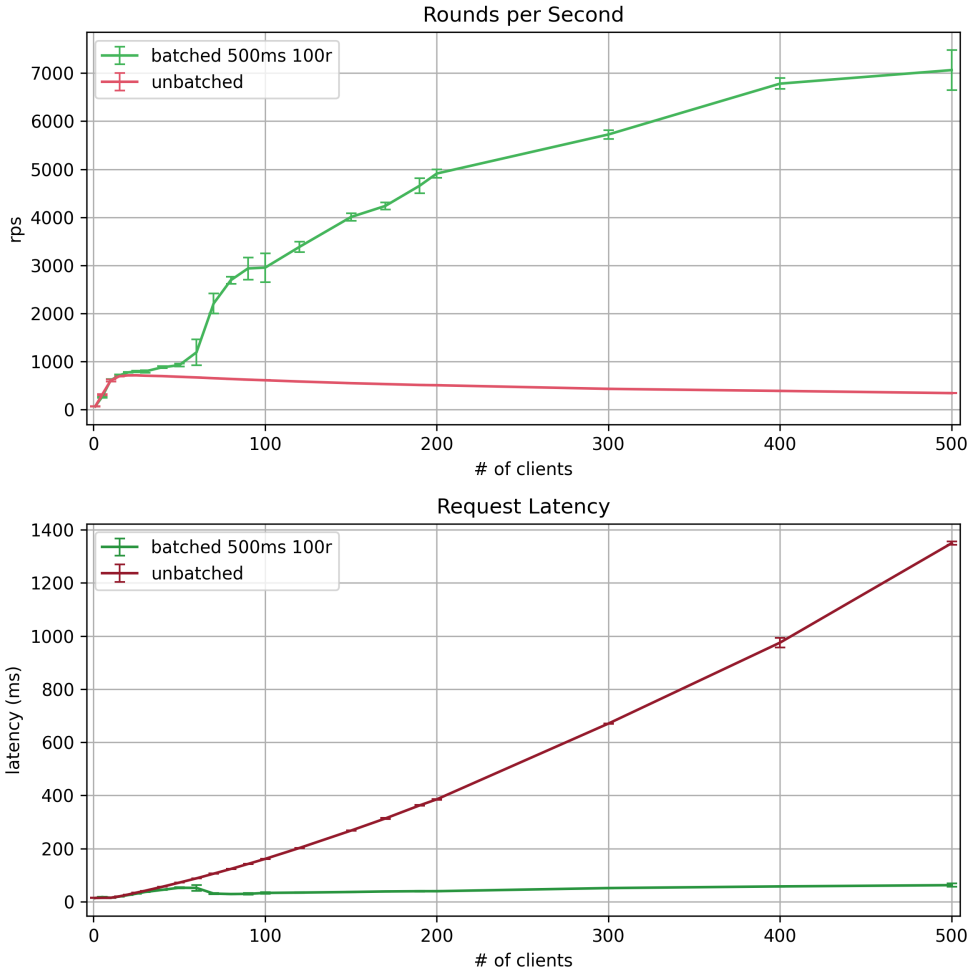

Figure 5.4.: Comparison of SplitBFT's distributed benchmark performance with and without request batching enabled.

a batch and distribute the requests inside it efficiently. Between 1 and 60 clients there is little difference between batched and non-batched configurations. This means that so few requests are issued that each request batch only contains one request. With more clients the number of requests per batch go up quickly and the batched configuration outperforms non-batching in every way. Because more requests get processed the average latency is lower too. The number of clients this system can operate with is technically limited by the number of allowed open connections. Clients and replicas maintain open TCP connections to each other at all times. The system at large correctly and quickly orders the batch while keeping a track of the contained requests. Over the runtime of 10s each, these benchmarks include all types of normal-case behaviour, including checkpointing.

The biggest load on the system, during these benchmarks, is right at the beginning. As all the clients are started at the same time and send their requests right away, the leader has to receive, parse and propose the incoming requests quickly. Because each client waits

for a response before sending another, after a short while the requests come in batches. If SplitBFT is configured to use request batching, the load follows the size of the request batches. Otherwise, the load is still distributed after some time, as new requests are only issued as pending requests are completed. Either way, the load is amortized after the initial spike of requests.

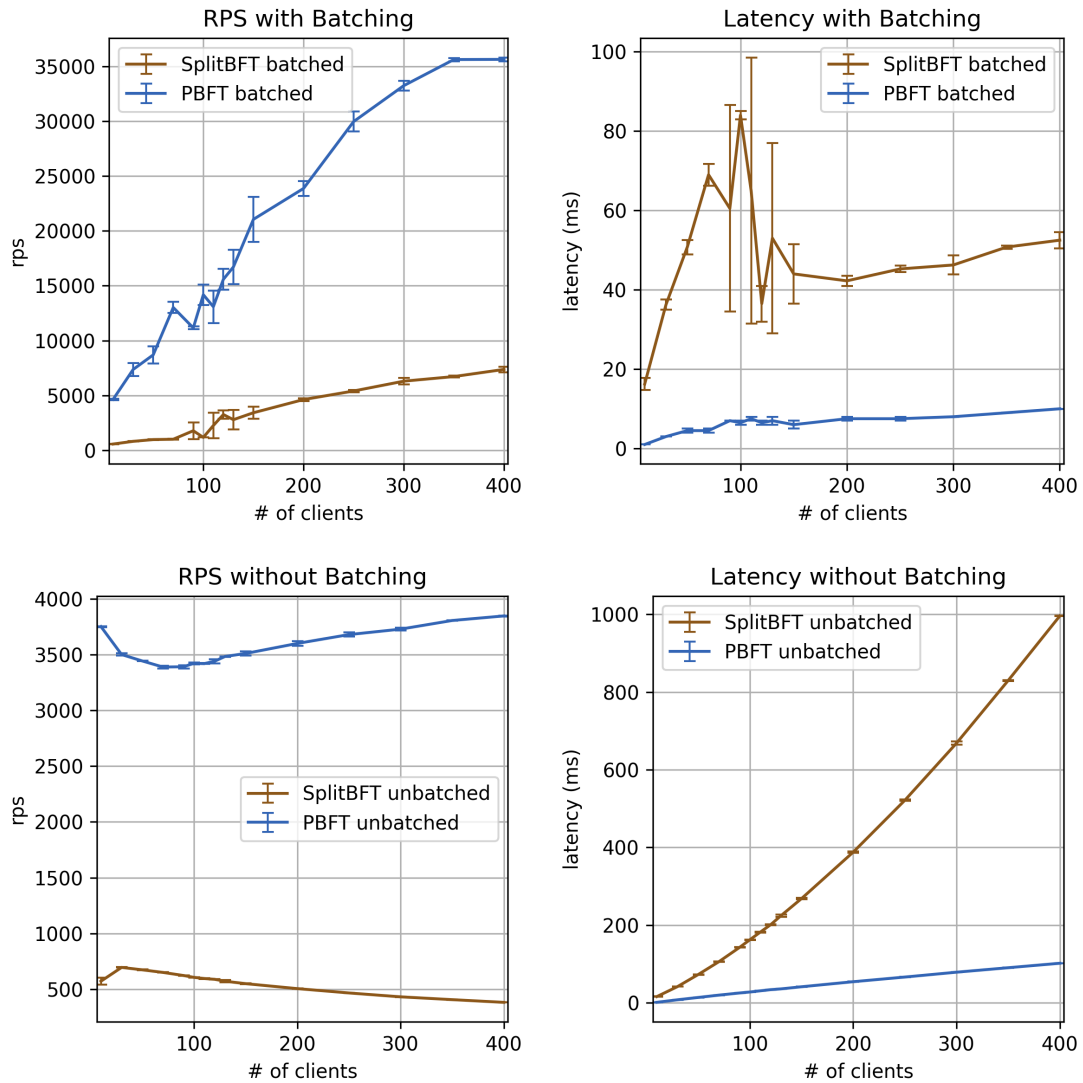

Figure 5.5.: Comparison of PBFT and SplitBFT in the benchmark application.

The figure 5.5 shows the comparison between PBFT and SplitBFT with and without request batching enabled. The shown metrics are reported by the benchmark application client. Request and response sizes are configured to be identical. In the request batched benchmarks both PBFT and SplitBFT exhibit the same behaviour around 110 clients. At this amount of load the latency and RPS fluctuate as not all batches are always full, but start to impact the performance positively. The exact number of clients at which batches

are created is dependent on the complexity of the protocol. As batches are created early if the replica is otherwise idling. This can lead to a system which could be sped up by not batching faster. Usually, using the benchmark application, the system will see a cyclic load, due to the way clients behave. One surprising result is that PBFT increases performance without batching at higher client numbers.

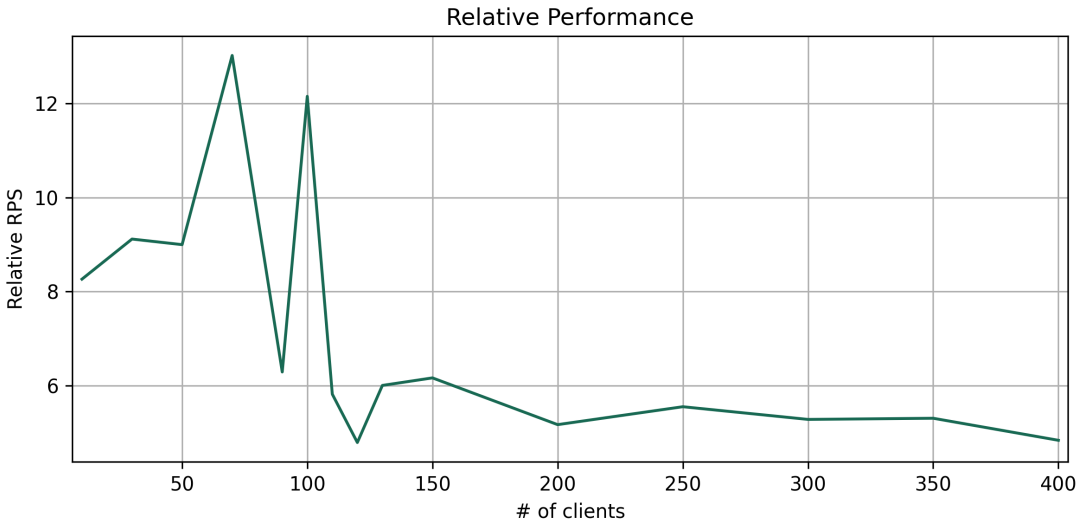

Figure 5.6.: Relative RPS performance of PBFT over SplitBFT

The relative performance of PBFT over SplitBFT is shown in figure 5.6. For low client numbers this performance fluctuates wildly as the efficiency of batching is dependent on the replica speed and affect the overall performance. With more than approximately 120 clients the relative performance of PBFT stabilizes at close to 5.5× that of SplitBFT. PBFT's performance advantage at lower client number is explained by the overhead of ECalls and OCalls. As the client number rises and batches are more likely to be full the ECall cost per request drops quickly as the protocol orders the request batch instead. SplitBFT's performance bottleneck at high client numbers is not the ordering phase but sending the replies. By ordering batches of requests, the overhead that ECalls and OCalls add per request diminishes. This is due to all the additional work that has to be done to send replies from the enclave, compared to PBFT. Whereas PBFT can construct, serialize and send the reply directly, the execution compartment has to construct the reply, construct an OCall batch and serialize to perform the OCall. On the untrusted side of SplitBFT the OCall is deserialized, interpreted and each reply itself is serialized and sent over the network. This additional work per reply is what makes the performance difference at large client numbers.

5.3.2. YCSB Evaluation

The YCSB program suite is a open-source set of tools for benchmarking cloud serving systems [12]. It generates workloads based on a specification to test systems for different access and usage patterns. The order and distribution of read and write operations can be changed to simulate real-world and worst-case access frequencies. Because our KVS implementation lives entirely in enclave memory access times are expected to be identical for different keys in any order. Therefore, we only examine a single representative example workload with 50% reads and 50% writes after the initialization of 1000 entries.

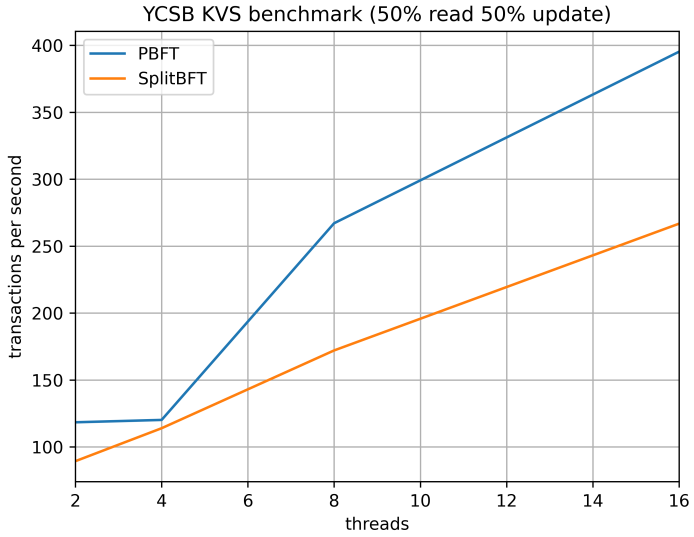

Figure 5.7.: YCSB benchmark results for PBFT and SplitBFT with YCSB default workload "a".

For access frequencies the benchmark in figure 5.7 uses the zipfian distribution. This distribution makes some records extremely popular and most other records unlikely to be accessed. After inserting data in bulk, read and update requests are sent to the BFT system. When compiling the execution enclave, the maximum size of the heap, which is used for data storage in this application, can be customized. In the figure 5.7, we can see that SplitBFT performs only slightly worse than the PBFT implementation. At this amount of load on the ordering algorithm the main difference is most likely caused by the ECall and OCall performance cost.

The figure 5.8 shows the performance of SplitBFT and PBFT for different read percentages. The remaining operations are update operations for records inserted before the benchmark started. The plots show a slight performance advantage for PBFT, however, both protocols seem to follow the same gradient. This can be explained by the ECall and OCall overhead.

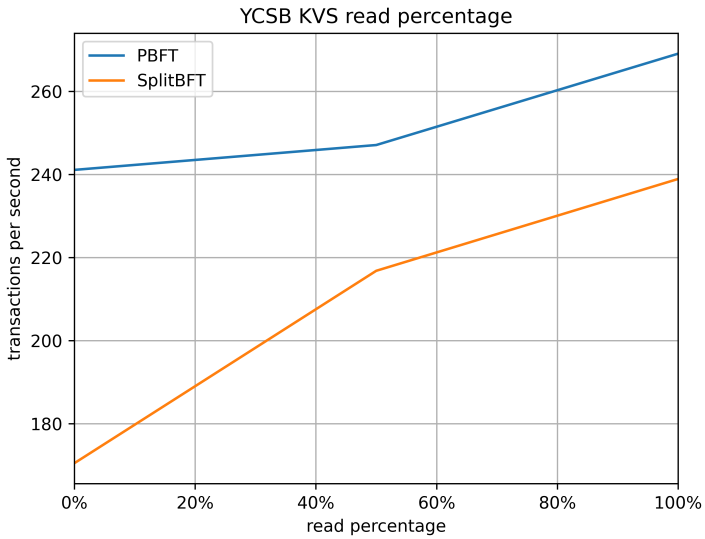

Figure 5.8.: YCSB benchmark results for read percentage sweep using PBFT and SplitBFT with 8 clients and zipfian distribution

6. Related Work

The realization that agreement protocols' rounds and phases allow for isolation is not entirely new. However, we present a new approach to use TEEs to isolate BFT protocols with the goal of increasing resilience and robustness. Lamport [31] already logically separates agreement participants into clients, proposers, acceptors and learners. Past research has explored many options to use separation to gain performance and scalability, or use trusted hardware to gain safety. Recent work has used TEEs to create hybrid models which use weaker fault models to increase performance and scalability. This is in contrast to SplitBFT which assumes the TEE can exhibit Byzantine faults. Yet SplitBFT still is able to use the separation to increase safety and resilience.

6.1. Physical Separation

A physical split within an agreement protocol is already usually done between the clients and the replicas. Replicas are only logically separated into representing proposers, acceptors (agreement) and learners (execution). Lamport [31] describes a system in which each replica is a server containing all participants. The proposer role is special, as a single proposer is distinguished as the leader. But even splitting agreement across more machines intuitive, as the message passing phases can be distributed across the network, spreading the load of agreement on even more machines. Traditionally, distributing the work of a single replica across multiple machines, however, does not increase safety or fault tolerance. Under the right circumstances increasing members will increase performance, provided the protocol has good scalability. Building and maintaining more machines is also costly, especially if large numbers of faults need to be tolerated. The risk with scaling to more networked participants is that the additional network communication overhead increases latency and reduces throughput. Commonly for BFT the number of participants also needs to be increased to tolerate more faults. Our approach achieves compartmentalization and tolerates more faults without increasing the number of hosts while also making each replica more resilient.

Separating Agreement from Execution

Yin et al. [49] show that separating the agreement operations with the execution environment physically achieves similar results to ours. The agreement layers themselves are less isolated than in our model, instead, the agreement protocol and execution hosts are protected by the network topology and firewalls. The isolated clusters can communicate efficiently and by using trusted boundaries they also achieve higher resilience. Their main

benefit is the gained confidentiality, and reduced replication cost of execution environments. The confidentiality gain is founded on an optimization which requires a firewall, which presents a single point of failure, that our protocol avoids.

MultiPaxos

Whittaker et al. [46] present MultiPaxos, a compartmentalized high-performance Paxos. They make use of the fact that crash-tolerant consensus only requires $f + 1$ proposer and $2f + 1$ acceptor compartments. Their work includes many variants and optimization strategies to gain performance and scalability from their flexible compartmentalization strategy. Like in SplitBFT, compartmentalization is considered a technique, and not a single protocol. MultiPaxos, however, is not just splitting Paxos, it restructures it into different amounts of proposers, proxy leaders, acceptors and replicas. They use compartmentalization to gain performance. SplitBFT on the other hand uses compartmentalization to gain safety and confidentiality. Starting from a conventional non-compartmentalized BFT protocol, SplitBFT only requires fencing existing code for compartmentalization. This leaves SplitBFT possibly with more compartments than MultiPaxos' compartmentalization technique.

PigPaxos

Charapko, Ailijiang, and Demirbas [9] alter MultiPaxos and add a piggybacking scheme to reduce communication overhead. This increases scalability and throughput of PigPaxos by introducing relay nodes. These relay nodes also present an approach that uses splitting of responsibilities as an improvement to BFT protocols. However, this is done to increase performance and not safety as in our case.

Tiered Distributed Systems

With the goal to improve performance, not safety, another way to achieve compartmentalization is by tiering the agreement as shown by Baldoni, Marchetti, and Tucci Piergiovanni [4]. Their three-tiered asynchronous distributed system moves part of the ordering step onto a mid-tier between clients and replicas. SplitBFT places all compartments of a replica physically on the replica. The tiered approach instead places nodes closer to the clients. Because of the communication rounds in SplitBFT, our compartments benefit from being close to each other, more than the client would benefit from a single compartment being closer to it.

6.2. Trusted Hardware

With the availability of general purpose processors supporting TEEs hybrid protocols become feasible. Past systems achieve similar safety qualities as crash-fault tolerant protocols by moving some parts of the protocol into a TEE. The hybrid fault models assume that TEEs can only fail by crashing. This allows protocols to make more assumptions and use the gained trust to increase performance and scalability.

TrInc: Trusted Counter

Levin et al. [33] use a similar approach to SplitBFT, creating a small TCB. Their versatile trusted counter compartment prevents equivocation, making it applicable in several applications to stop faulty untrusted side's worst case behaviour. The main difference between TrInc and SplitBFT is that our fault model allows for compromised TEEs. However, one advantage of TrInc is that the TCB can be smaller as it only needs to sign messages and maintain a counter. SplitBFT, however, places safety-critical functions into the TEE and gains more resilience than TrInc because of it.

MinBFT

Veronese et al. [44] improved on previous work requiring a Trusted Timely Computing Base (TTCB) for hybrid protocols. MinBFT uses trusted counters, like TrInc, to assign verifiable sequence numbers. Veronese et al. [44] use the tamperproof certificate of the TEE-signed sequence number to remove equivocation. This allows all replicas to assume that all other replicas receiving a message with the same sequence number have received the exact same message. By eliminating equivocation they reduce the number of necessary replicas to $2f + 1$ instead of PBFT's $3f + 1$. Because replicas cannot send inconsistent messages an entire communication round can be eliminated compared to classical PBFT [44, 49], providing lower latency. Which means they achieve the theoretical minimum number of communication steps. Similarly to TrInc, MinBFT requires the the trusted counter does not become Byzantine. A crash-fault in the counter implies a crash-fault in the replica. The main difference to SplitBFT is that we tolerate faults in the TEE and, therefore, use a stronger fault model and use TEEs to gain resilience.

Hybster

Behl, Distler, and Kapitza [5] present a hybrid state-machine replication protocol with extremely high performance. Through a highly parallel software design they achieve before unseen throughput, even on moderate machines. Similar to SplitBFT the Hybster protocol uses the Intel SGX platform as a TEE. They reduce the number of required replicas to $2f + 1$ by using TEEs for the agreement process. Other than preexisting hybrid protocols Hybster was also formally verified. The main difference to SplitBFT, other than the

compartmentalization, is that the execution is not protected by the TEE. Similar to other hybrid protocols, the TEE is used to gain performance and scalability, instead of safety and confidentiality.

CheapBFT

The CheapBFT protocol by Kapitza et al. [26] drastically reduces the required amount of replicas. By using a TEE to eliminate equivocation CheapBFT only requires $f + 1$ replicas to agree on and execute a client's request. The TEE performs a similar task as a trusted counter, implemented as a Counter Assignment Service in Hardware (CASH) subsystem on an FPGA. The reduction from $2f + 1$ to $f + 1$ required replicas is achieved by a internal CheapTiny protocol. While FPGA's are not as commonly available and affordable, the possibility to build a high-performance TEE without trusting a vendor, like Intel, is powerful.

7. Conclusion

Agreement algorithms are present in a distributed cloud or decentralized structure to maintain state across multiple machines. Together with permissioned blockchains, which can use BFT protocols for the ordering of transactions, the interest in resilient BFT protocols will continue to grow. To make use of cloud provider's infrastructure some form of TEE need to be used for these applications. In future these systems will work with more and more sensitive data. There are incentives for bad actors to target these BFT systems, which means deployments will have to be protected against Byzantine faults. Using TEEs, a layer of protection against common attacks on software systems can be added, but to gain resilience and be prepared for novel attacks compartmentalization and diversification is unavoidable. Deploying library operating system solutions like Graphene-SGX [43] provide a simple starting point to place software systems into a TEE. These library OS systems are, however, less resilient than compartmentalized systems. We show that with minimal changes to existing protocols, compartmentalized variants can be created which allow for altered fault models. The fact that TEEs sometimes have weaknesses bugs or exploits leading to Byzantine behaviour, reflects the real world better than those of some hybrid models. Protecting sensitive data in TEE will require the TEEs to be designed to be resilient, as some attacks are known. To achieve the safest and most resilient BFT protocol, future work will have to formulate more precise fault models and measurements of attack surfaces and their possible impacts. The biggest problem is to meaningfully measure the size of the TCB and likelihood and extend of faults within the individual TEEs.

Compartmentalizing PBFT is straight-forward and increases resilience in our stronger fault model. Our evaluations show that using TEEs and compartmentalization under the right circumstances provides great resilience for safety and confidentiality at relatively small cost. As pointed out by related work the overhead which is responsible for the performance difference might be avoidable by hardware and firmware improvements [42].

Using modern tools and environments, like Themis, the agreement framework written in the Rust programming language, and Rust SGX SDKs designing and implementing resilient replicated systems has become easier than ever. The presented SplitBFT method and example PBFT variant show that converting an agreement protocol into a compartmentalized version is possible through a well-structured process. The encountered problems and observed bottlenecks are possible to overcome, either by further software development or hardware improvements. By adding more traditional and modern agreement protocols and modules to the Themis framework a more comprehensive and representative comparison of different approaches will be possible. Examinations of other BFT protocols and their compartmentalized variants will help inform a stronger fault model

to be used in the future. Depending on the protocol, one might be able to increase scalability by not requiring every compartment to be present on every replica as there is no special trust between local compartments.

Future work can increase robustness further by updating used technologies to newer, more hardened versions. FPGAs have been used to speed up computation in a separate custom hardware environment for Byzantine agreement in the hybrid protocol CheapBFT [26]. However, for the use in SplitBFT they lack attestation and cryptographically secure memory. To gain the performance advantages FPGAs can provide to compartmentalized BFT future FPGA platforms would need to support more TEE features. FPGA-based platforms may be able to avoid the cost of ECalls and OCalls. A mixed approach could be followed as well [36]. Removing the requirement to trust Intel might be advantageous in certain applications. Even the untrusted code could be further hardened, by using new and upcoming features like the `memfd_secret()` system call [13] or deploying entirely inside VMs or containers. Diversification in future work could include mixing different TEE providers in a single replicated system. A formal verification of the SplitBFT method or compartmentalized PBFT would also provide additional confidence in their properties.

Bibliography

[1] Marcos K. Aguilera et al. *Microsecond Consensus for Microsecond Applications*. 2020. arXiv: 2010.06288 [cs.DC].

[2] Apache. *Rust SGX SDK*. https://github.com/apache/incubator-teaclave-sgx-sdk. 2021. URL: https://github.com/apache/incubator-teaclave-sgx-sdk/tree/c2698dc2685f8dcd9550086c62077bceff15ded0.

[3] Apache. *SipHasher SGX SDK Documentation*. https://dingelish.github.io/sgx_tstd/sgx_tstd/hash/struct.SipHasher.html. 2021. URL: https://dingelish.github.io/sgx_tstd/sgx_tstd/hash/struct.SipHasher.html.

[4] R. Baldoni, C. Marchetti, and S. Tucci Piergiovanni. "Asynchronous active replication in three-tier distributed systems". In: *2002 Pacific Rim International Symposium on Dependable Computing, 2002. Proceedings.* 2002, pp. 19–26. DOI: 10.1109/PRDC.2002.1185614.

[5] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. "Hybrids on Steroids: SGX-Based High Performance BFT". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 222–237. ISBN: 9781450349383. DOI: 10.1145/3064176.3064213. URL: https://doi.org/10.1145/3064176.3064213.

[6] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, 991–1008. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[7] Miguel Castro and Barbara Liskov. "Authenticated Byzantine Fault Tolerance Without Public-Key Cryptography". In: (Aug. 1999).

[8] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: *ACM Trans. Comput. Syst.* 20.4 (Nov. 2002), pp. 398–461. ISSN: 0734-2071. DOI: 10.1145/571637.571640. URL: http://doi.acm.org/10.1145/571637.571640.

[9] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. "PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus". In: *Proceedings of the 2021 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 235–247. ISBN: 9781450383431. URL: https://doi.org/10.1145/3448016.3452834.

[10] Allen Clement et al. "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pp. 153–168.

[11] ConsenSys. *Quorum Blockchain Service*. 2021. URL: https://consensys.net/QBS.

[12] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

[13] Jonathan Corbet. "memfd_secret() in 5.14". In: (Aug. 2021). URL: https://lwn.net/Articles/865256/.

[14] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086. 2016.

[15] The Rust Project Developers. *How Rust is Tilde's Competitive Advantage*. 2018. URL: https://prev.rust-lang.org/pdfs/Rust-Tilde-Whitepaper.pdf.

[16] The Rust Project Developers. *Mid-level Itermediate Representation Interpreter*. 2021. URL: https://github.com/rust-lang/miri.

[17] The Rust Project Developers. *The rustc book*. URL: https://doc.rust-lang.org/rustc/lints/index.html.

[18] DHL. *DHL AND ACCENTURE UNLOCK THE POWER OF BLOCKCHAIN IN LOGISTICS*. 2018. URL: https://www.dhl.com/global-en/home/press/press-archive/2018/dhl-and-accenture-unlock-the-power-of-blockchain-in-logistics.html.

[19] Jim Guilford, David Cote, and Vinodh Gopa. *Fast SHA512 Implementations on Intel® Architecture Processors*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-sha512-implementations-ia-processors-paper.pdf.

[20] Patrick Hunt et al. "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, p. 11.

[21] IBM. *Blockchain in retail solutions*. 2021. URL: https://www.ibm.com/blockchain/industries/retail.

[22] Intel. *Intel® Software Guard Extensions (Intel SGX) Developer Guide V2.8*. https://download.01.org/intel-sgx/sgx-linux/2.8/docs/Intel_SGX_Developer_Guide.pdf. 2020. URL: https://download.01.org/intel-sgx/sgx-linux/2.8/docs/Intel_SGX_Developer_Guide.pdf.

[23] Intel. *Intel® Software Guard Extensions (Intel SGX) Developer Reference V2.4*. https://download.01.org/intel-sgx/linux-2.4/docs/Intel_SGX_Developer_Reference_Linux_2.4_Open_Source.pdf. 2020. URL: https://download.01.org/intel-sgx/linux-2.4/docs/Intel_SGX_Developer_Reference_Linux_2.4_Open_Source.pdf.

[24] Johnson et al. *Supporting SGX on Multi-Socket Platforms*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html.

[25] Ralf Jung et al. "Stacked Borrows: An Aliasing Model for Rust". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371109. URL: https://doi.org/10.1145/3371109.

[26] Rüdiger Kapitza et al. "CheapBFT: Resource-Efficient Byzantine Fault Tolerance". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: Association for Computing Machinery, 2012, pp. 295–308. ISBN: 9781450312233. DOI: 10.1145/2168836.2168866. URL: https://doi.org/10.1145/2168836.2168866.

[27] Steve Klabnik and Carol Nichols. *Rust Book: Error Handling*. 2021. URL: https://doc.rust-lang.org/book/ch09-00-error-handling.html.

[28] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284. URL: https://doc.rust-lang.org/book/.

[29] Marios Kogias and Edouard Bugnion. "HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387545. URL: https://doi.org/10.1145/3342195.3387545.

[30] Tsung-Ting Kuo, Hyeoneui Kim, and Lucila Ohno-Machado. "Blockchain distributed ledger technologies for biomedical and health care applications". In: *Journal of the American Medical Informatics Association* 24 (Nov. 2017), pp. 1211–1220. DOI: 10.1093/jamia/ocx068.

[31] Leslie Lamport. "Paxos Made Simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (Dec. 2001), pp. 51–58. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[32] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[33] Dave Levin et al. "TrInc: Small Trusted Hardware for Large Distributed Systems". In: *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*. Boston, MA: USENIX Association, Apr. 2009. URL: https://www.usenix.org/conference/nsdi-09/trinc-small-trusted-hardware-large-distributed-systems.

[34] Ines Messadi et al. "Poster Abstract: A Fast and Secure key-value Service Using Hardware Enclaves". In: *Proceedings of the 20th International Middleware Conference (Posters)*. ACM. 2019, pp. 9–13.

[35] Azure Microsoft. *Azure Blockchain Service*. 2021. URL: https://azure.microsoft.com/en-us/services/blockchain-service/.

[36] Hyunyoung Oh et al. "TRUSTORE: Side-Channel Resistant Storage for SGX Using Intel Hybrid CPU-FPGA". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1903–1918. ISBN: 9781450370899. DOI: 10.1145/3372297.3417265. URL: https://doi.org/10.1145/3372297.3417265.

[37] C. Pu et al. "A Specialization Toolkit to Increase the Diversity of Operating Systems". In: 1996.

[38] Vincent Rahli et al. "Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq". In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 619–650. ISBN: 978-3-319-89884-1.

[39] Signe Rüsch, Kai Bleeke, and Rüdiger Kapitza. "Themis: An Efficient and Memory-Safe BFT Framework in Rust: Research Statement". In: *Proceedings of the 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 9–10. ISBN: 9781450370295. DOI: 10.1145/3366611.3368144. URL: https://doi.org/10.1145/3366611.3368144.

[40] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: https://doi.org/10.1145/98163.98167.

[41] Michael Schwarz et al. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 753–768. ISBN: 9781450367479. DOI: 10.1145/3319535.3354252. URL: https://doi.org/10.1145/3319535.3354252.

[42] Hongliang Tian et al. "Switchless Calls Made Practical in Intel SGX". In: *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. SysTEX '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 22–27. ISBN: 9781450359986. DOI: 10.1145/3268935.3268942. URL: https://doi.org/10.1145/3268935.3268942.

[43] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 645–658. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[44] Giuliana Veronese et al. "Efficient Byzantine Fault-Tolerance". In: *Computers, IEEE Transactions on* 62 (Jan. 2013), pp. 16–30. DOI: 10.1109/TC.2011.221.

[45] Nico Weichbrodt et al. "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves". In: (2016). DOI: 10.24355/dbbs.084-201611011011-0. URL: http://www.digibib.tu-bs.de/?docid=00064029.

[46] Michael Whittaker et al. *Scaling Replicated State Machines with Compartmentalization*. Dec. 2020.

[47] Chen Yang et al. "Review on Variant Consensus Algorithms Based on PBFT". In: *Artificial Intelligence and Security*. Ed. by Xingming Sun, Jinwei Wang, and Elisa Bertino. Singapore: Springer Singapore, 2020, pp. 37–45. ISBN: 978-981-15-8101-4.

[48] Juncheng Yang, Yao Yue, and Rashmi Vinayak. *A large scale analysis of hundreds of in-memory cache clusters at Twitter*. URL: https://www.usenix.org/sites/default/files/conference/protected-files/osdi20_slides_yang.pdf.

[49] Jian Yin et al. "Separating Agreement from Execution for Byzantine Fault Tolerant Services". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 253–267. ISSN: 0163-5980. DOI: 10.1145/1165389.945470. URL: https://doi.org/10.1145/1165389.945470.

A. Contents of the CD

The CD included with the thesis has the following contents:

- Source code for SplitBFT in Themis including git history. Packaged as a git bundle it includes all references to browse the implementation process.
- Thesis as PDF including thesis code as LaTeX project.
- Benchmark data used to generate plots in the thesis.

All this data can also be found as attachments to the most recent release on this GitLab repository: `https://gitlab.ibr.cs.tu-bs.de/ds-thesis/2021-ma-markus-becker-bft-split-multiple-enclaves`