High Speed Consensus with Trusted Execution Environments
Single Active Counter Byzantine Fault Tolerance

Master’s Thesis
Espoo, July 14, 2018

Supervisors:  Professor N. Asokan, Aalto University
Associate Professor Nicola Dragoni, Technical University of Denmark

Advisors:     Dr. Lachlan Gunn, Aalto University
Dr. Jian Liu, Aalto University
In recent years, Byzantine consensus algorithms have seen a surge in popularity with the rise of Bitcoin and blockchain technology. A major problem that hampers adoption of existing consensus algorithms in blockchain scenarios is their scalability. There has been much research in the past years aiming to optimize these algorithms and increase their efficiency. For example, recent work has shown that voting rounds present in many classical algorithms can be made drastically more efficient by the use of message aggregation techniques.

Another trend is towards the usage of trusted hardware to increase performance and lower resource requirements of these algorithms. Trusted hardware enables algorithms to reduce the lower bound on the number of replicas from $3f + 1$ to $2f + 1$, where $f$ is the number of tolerated faults. Currently, all existing Byzantine consensus algorithms either use no trusted hardware at all, or assume that all replicas have access to the same trusted hardware. This leaves a gap in the design space, neglecting scenarios where only some machines have access to trusted hardware.

In this work, we investigate the possibilities where only a subset of all replicas has access to trusted hardware. We introduce the SACBFT framework, consisting of two transformations that can be applied to existing Byzantine consensus protocols, increasing their efficiency by allowing them to make use of trusted hardware that exists in the system.

We apply the framework to PBFT and RePBFT to produce SACPBFT and SACRePBFT respectively, and show how to apply the framework to other protocols. We also evaluate a proof-of-concept implementation of SACPBFT, showing that it can dramatically reduce network usage and increase performance even when only a single replica has access to trusted hardware.
Acknowledgements

Between the start of my Master’s programme and now, I have learned so much more than I expected. Tasting the atmosphere and methods at two great universities has broadened the perspective with which I look at academia, and I am grateful for having been given the chance to do so.

I would like to thank my supervisors, N. Asokan and Nicola Dragoni for their support, and giving me the opportunity to work on my thesis at the Secure Systems group. Every day I felt like I learned something new, and being surrounded by such an intelligent and stimulating group of people has been a great experience. In particular I would like to thank my advisors, Lachlan Gunn and Jian Liu for their invaluable feedback and guidance, ensuring that I always progressed in the right direction. I would also like to thank Ricardo Vieitez and Fritz Alder for their suggestions and help regarding technical matters.

Over the last two years I have made many new friends in both Aalto University and DTU. I’d like to thank them for the many adventures and more generally providing me with an amazing time, giving me the motivation to keep working but also teaching me to balance work and life.

Finally, I am grateful to my parents for their never-ending stream of support, love, and inspiration, which led me to pursue a Master’s degree in the first place, and got me to where I am now – at the end of my thesis.
Contents

1 Introduction ................................. 1
   1.1 Motivation .............................. 2
   1.2 Contribution ............................ 3

2 Background and Preliminaries .................. 4
   2.1 Byzantine Fault Tolerance ............... 4
       2.1.1 The Byzantine Generals Problem .... 4
       2.1.2 Network model ...................... 5
       2.1.3 BFT Algorithms ................... 6
       Practical Byzantine fault tolerance ...... 7
       Resource-efficient Byzantine fault tolerance 8
       Zyzzyva .................................. 10
   2.2 Trusted Execution Environments ............ 13
       2.2.1 Intel SGX ........................... 13
       2.2.2 Trusted monotonic counters .......... 14
       2.2.3 TEE-enabled BFT algorithms ......... 15
       MinBFT .................................. 16
   2.3 Multisignatures .......................... 17
       2.3.1 Boneh-Lynn-Shacham signatures .... 18
       2.3.2 ByzCoin ............................. 18

3 Problem description .......................... 20
   3.1 Problem description ..................... 20
   3.2 Assumptions ................................ 21
   3.3 Fault tolerance goals .................... 22
   3.4 Performance Goals ....................... 23
   3.5 Summary .................................. 24

4 Framework .................................. 25
## CONTENTS

4.1 Design outline ........................................... 25
4.2 Communication ........................................... 27
  4.2.1 PREPARE phase ...................................... 28
    Why PBFT requires two voting rounds .................... 28
    SACBFT PREPARE phase .................................. 31
  4.2.2 Voting ................................................ 32
  4.2.3 REPLY ................................................ 35
  4.2.4 View-changes ....................................... 35
    MinBFT with one TEE .................................... 35
    View-changes in SACBFT .................................. 38
  4.2.5 Correctness ........................................ 39
4.3 Applying SACBFT ........................................ 39
  4.3.1 Applying the ordering phase transformation ........... 40
  4.3.2 Applying the voting round transformation ............ 41
4.4 Examples of transformed protocols ........................ 41
  4.4.1 SACPBF ............................................ 41
  4.4.2 SACRePBFT ......................................... 42

5 Implementation ........................................... 45
  5.1 Cryptography .......................................... 45
    Hashing ................................................ 45
    Multisignatures ........................................ 46
    Signatures .............................................. 47
  5.2 Key generation ........................................ 47
5.3 Software structure ...................................... 48
  5.3.1 Trusted computing base ............................... 50
  5.3.2 Communication ..................................... 51
    Message format ........................................ 51
  5.3.3 Transactions ....................................... 52
  5.3.4 Configuration ..................................... 52

6 Evaluation ............................................... 54
  6.1 Theoretical evaluation .................................. 54
  6.2 Experimental evaluation ............................... 57

7 Discussion ............................................... 60
  7.1 Fault tolerance ....................................... 60
  7.2 Performance .......................................... 62
  7.3 Future work .......................................... 63

8 Related Work ............................................ 65
Chapter 1

Introduction

Distributed consensus has recently gained popularity with the advent of blockchain technology. However, before Bitcoin [38] and blockchain as a paradigm existed, there was already an entire field of research dedicated to consensus algorithms, with a variety of protocols that facilitate distributed consensus in (often) closed groups of machines.

These algorithms can be categorized according to their failure models. The first group, the fail-stop model, assumes that participants either act correctly or crash. The second group, the Byzantine fault model, assumes that failing participants can exhibit arbitrary behavior. Tolerating Byzantine failures is essential to some distributed systems where safety is paramount, but proves to be a complex problem to solve efficiently. For both failure modes there exist upper bounds on how many failures can be tolerated in a system.

Some recent work in Byzantine fault-tolerant protocols assumes that each participant has access to some piece of trusted hardware that can be expected to either execute correctly or crash. These trusted components are then leveraged to contain some functionality of participants, providing increased fault tolerance and raising the upper bound on tolerated failures of the Byzantine fault model to that of the fail-stop model.

There do not currently exist any protocols that make the assumption that only a subset of the participants contain trusted hardware. In this thesis, we investigate this model and introduce SACBFT, a framework that can transform existing protocols into protocols that make use of the available trusted hardware to achieve improved performance and fault tolerance.
1.1 Motivation

While Byzantine consensus protocols exist that leverage trusted hardware, all of these require that every machine contains this trusted hardware. This is not always a realistic assumption.

For example, consider the scenario where machines in an existing consensus system are gradually upgraded to more modern machines, which often have access to trusted hardware. As long as not all machines are upgraded, it is currently impossible to run consensus protocols that make use of this trusted hardware. A protocol that could make use of the available trusted hardware could potentially increase performance already after the first machines in the system are upgraded. Additionally, it would lower the investment required to benefit from trusted hardware compared to existing protocols when deploying a new system.

Other examples of scenarios where such a protocol could provide value can be found in the areas of embedded systems and IoT\(^1\) devices. In these areas, some machines in a consensus system might have to be as lightweight as possible, and will only contain a minimal set of low-end components. These machines can be constrained on e.g. power consumption or weight. An example of such a scenario is one with a drone fleet flying above a number of ground stations. These stations can have high-end hardware with power-hungry processors, while the drones are more restricted in their choice of hardware.

With the increased usage of embedded systems in vehicles (especially self-driving cars), we see another use-case for consensus systems, which can aid in providing safety-critical functionality such as emergency brakes when certain sensors detect an object in the vehicles' path. In cars, the various control systems are typically independent of each other and communicate through special protocols. The control systems also have variable hardware specifications, depending on their function. The heavier systems could be equipped with trusted hardware, which can then be utilized in consensus protocols.

Also for already existing protocols that rely on trusted hardware, there is a need for some sort of transition protocol in order to deal with churn; old trusted components in a system are likely to be broken, in which case a protocol should be able to get by with relying only those machines that contain newer trusted hardware.

With these cases in mind, protocols capable of utilizing all trusted hardware available to the system (regardless of how many machines in

\(^1\)Internet of Things
the system contain trusted hardware) could prove to be valuable choices.

1.2 Contribution

In this work we present Single Active Counter Byzantine Fault Tolerance (SACBFT), a framework that provides transformations for existing, classical Byzantine consensus protocols to make use of trusted hardware available to the system. Additionally, the framework allows one to reduce network complexity of existing protocols from $O(n^2)$ to $O(n)$ by using message aggregation techniques to reduce the number of messages sent in voting rounds.

We apply SACBFT to PBFT [12] and RePBFT [19] to produce SACPBFT and SACRePBFT. We also provide instructions on how to apply SACBFT to other protocols.

To demonstrate the effectiveness of SACBFT, we developed a proof-of-concept implementation of SACPBFT, and compare the performance of this protocol with PBFT and Zyzzyva [35].
Chapter 2

Background and Preliminaries

In order to understand the material discussed in this thesis, it is important to have some background knowledge on relevant topics. This chapter summarizes the most important ones.

2.1 Byzantine Fault Tolerance

A fundamental problem in the field of distributed systems is that of getting the individual parts of a distributed system to reach agreement when some parts of this system might exhibit arbitrary behavior. This arbitrary behavior is known as Byzantine behavior. Many algorithms have been proposed to solve this problem, each probing the trade-off between resilience and performance.

Before we look at these algorithms in more detail, we will introduce common terms and concretely define the problem, allowing us to see where the complexity comes from and why all practical algorithms have to make certain assumptions.

2.1.1 The Byzantine Generals Problem

The Byzantine Generals problem was introduced by Lamport, Shostak and Pease [36]. They describe the problem by using the example of a Byzantine army laying siege to an enemy city. This city is so large that in order to surround it, the generals have to split up the army into smaller groups, each led by one general. The generals have to come up with a battle plan, but cannot leave their own soldiers, so they communicate using messengers. These messengers carry written messages signed by the
generals, meaning that they cannot be forged, which would be possible if messengers delivered messages orally.

The generals fear that some among them might be traitors however, so they wish to communicate in a way that ensures the generals can still reach agreement despite traitorous generals actively trying to sabotage them. Lamport et al. have shown that it is possible to tolerate $f$ traitors if and only if there are at least $3f + 1$ generals in total. In more abstract terms, the goal is that $n$ processes reach consensus, in the presence of up to $f$ Byzantine processes. Here "Byzantine" indicates the possibility of arbitrary behavior, which includes coordinated malicious behavior.

This setting has two interesting properties: Firstly, it is a no-one-trusts-no-one situation; it is unknown which processes are Byzantine, so processes have to operate with the assumption that any other process might be faulty. Secondly, there are only two-party communication channels. This makes the problem more difficult, as processes can equivocate, that is, send conflicting messages to different processes. In the Byzantine generals example a traitor could tell general $A$ to attack while simultaneously telling general $B$ to retreat.

### 2.1.2 Network model

Because the Byzantine Generals problem concerns remote processes communicating with each other, the network model plays an important role in problem complexity.

In order to be applicable to realistic scenarios, we look at an asynchronous network model. In this model, messages can get lost in transit, delayed, duplicated, or delivered out of order.

Fischer, Lynch and Paterson [20] have shown that distributed consensus in a completely asynchronous network model is not possible, because it is impossible to detect the presence of even one unannounced process death.

We thus need to modify the asynchronous model and make it synchronous by adding a synchrony assumption. Castro and Liskov introduced the weak synchrony assumption [12], stating that the delay between a process sending a message and other processes receiving this message cannot grow indefinitely. The resulting network model enables us to prove correctness of consensus algorithms, while maintaining realistic assumptions on network conditions.

In addition to this, all communication channels in our model provide transferable authentication. This means that every message sent through the channels comes with an authentication token that allows any-
one to verify the authenticity (and integrity) of these messages. Thus, MACs cannot be used to implement this authenticated channel, since they typically only provide authentication between two parties, and verification requires knowledge of a shared secret. Digital signatures on the other hand work fine, as verification requires knowledge of a public key. We require transferable authentication as it is a precondition for providing non-equivocation [17].

2.1.3 BFT Algorithms

Byzantine fault tolerant (BFT) algorithms are distributed algorithms that are able to reach consensus between \( n \) replicas in the presence of a prescribed number of tolerable Byzantine faults \( f \), where \( f \leq \left\lfloor \frac{n-1}{3} \right\rfloor \). All algorithms we consider in this work use the network model with the weak synchrony assumption as described in section 2.1.2. Additionally, all algorithms discussed in this thesis are a form of state machine replication [46], and we will in the remainder of this thesis refer to processes as replicas, as is standard in the literature.

Processes interacting with a BFT algorithm are called clients. They send request messages to the system, and the system will perform deterministic operations specified in these requests. Clients wait for a reply to arrive from the system, which contains the results of their request. Note that for some algorithms, the client requires a certain amount of replies from individual replicas in the system in order to consider a request to be completed. The complete process of a client sending a request, the system processing it and sending a reply to the client is called a transaction.

By definition, BFT algorithms must satisfy two correctness properties:

- **Safety**: This property states that operations are executed one at a time, and in the same order across all correct replicas. This implies satisfying the linearizability property [26].

- **Liveness**: this property states that clients will always eventually get a reply to their request messages, even in the presence of up to \( f \) faulty replicas. This guarantees that the system will eventually reach a state where it can make progress (i.e. handle transactions).

In the following subsections we will describe three BFT algorithms. The Practical Byzantine Fault Tolerance (PBFT) algorithm, as well as the Resource Efficient Byzantine Fault Tolerance framework, and Zyzzyva.


Chapter 2. Background and Preliminaries

Practical Byzantine fault tolerance

Practical Byzantine Fault Tolerance [12] is a BFT algorithm described in a breakthrough paper by Castro and Liskov. This algorithm was the first BFT algorithm capable of reaching acceptable performance levels in real-world systems, and introduced several important concepts, which have since been used in many BFT algorithms. We describe BFT algorithms using these concepts as classical BFT algorithms.

In PBFT, a request is sent to the primary replica. This is a replica with the special role of receiving requests and assigning a sequence number to them. A sequence number is a unique identifier in the form of a monotonically increasing counter value. When it assigns a sequence number to an incoming request, it broadcasts this in a PREPARE message to all other replicas in the system. When a replica receives a PREPARE message it checks whether the sequence number is valid, and if so it broadcasts its “acceptance” of the sequence number to all other replicas. This is called the PREPARE phase. This broadcast is necessary in case a primary replica is Byzantine and sends different sequence numbers to groups of replicas in the system. After the PREPARE phase, all correct replicas in the system have arrived at the prepared state.

The PREPARE and PREPARE phases together form the ordering process of PBFT; once a replica is in the prepared state for a particular request r and sequence number n, we know that n cannot be assigned to a different request anymore. This PREPARE phase is essentially a voting round to accept the sequence number proposed by the primary in the PREPARE phase.

For actual execution of the request, PBFT initiates another voting round called the COMMIT phase. After the commit phase, all correct replicas are in the committed state, will execute the request operation, and send their reply to the client.

When the system has f or fewer faulty replicas, a client will receive at least n - f replies to a request. After f + 1 of these replies, it knows that the system has reached the next state and the request has been successfully processed.

These two voting rounds provide safety to the algorithm, but not liveness. For that, PBFT relies on views. The protocol moves through consecutively numbered views. Each view is a configuration where one replica is chosen to act as primary. Additionally, each replica keeps a log of all transactions in a view. When the primary fails, a view-change is initiated where a new primary is elected in a deterministic way, and all replicas share their logs of the view, ensuring that all replicas are up-to-date with
all transactions before continuing to the next view.

A primary can fail at any time, but broadly speaking we can discern two cases: either the primary will fail between transactions, and the client will not be able to reach the primary, or the primary will fail during a transaction and replicas will stop receiving messages from the primary.

In the first case, the client will broadcast a notification to all replicas stating that it cannot contact the primary, and in the second case correct replicas will wait for a specified amount of time, after which they consider the primary to be faulty.

Once a replica considers a primary to be faulty, replicas will attempt to change the primary by initiating a view-change. A view-change brings all correct replicas up-to-speed with the committed requests by sharing a request log, and elects a new primary for the next view. After a view-change has been completed, the client will send its requests to the new primary. Since there are only assumed to be a maximum of $f$ faulty replicas, the system eventually will pick a correct replica as primary, which ensures liveness.

Figure 2.1 shows the communication pattern for a successful transaction in PBFT. As can be seen, both voting rounds (during the PREPARE and COMMIT phases, respectively) require considerably more messages than the other phases, since each replica broadcasts their vote to all other replicas. More formally, one voting round requires $O(n^2)$ messages to be transmitted, and we say that the communication complexity of PBFT is $O(n^2)$. These voting rounds form a bottleneck for the performance and scalability of this algorithm, as bandwidth requirements increase quadratically with respect to the number of replicas.

**Resource-efficient Byzantine fault tolerance**

Resource-efficient fault tolerance [19] (ReBFT) provides a framework to minimize the amount of replicas required during optimal functioning of the system, and is based on the observation that in the absence of faults, protocols such as PBFT send more messages than strictly necessary.

ReBFT divides replicas into two sets: the set of $2f + 1$ active replicas and the set of $f$ passive replicas.

In normal execution, a transaction involves all active replicas, who act as if the passive replicas are not part of the system (i.e. when they broadcast a message in the original protocol, they will broadcast this to only active replicas in the ReBFT variant). In the REPLY phase, all active replicas then send their reply to the client as usual, but also send an UPDATE message to all passive replicas. Once a passive replica receives $f + 1$ UPDATE
messages, it will execute the request operation, in order to stay up-to-date with the active replicas. One important observation that should be made is that this is not part of the transaction; passive replicas can execute requests asynchronously, and clients do not require them to be up-to-date in order to consider the transaction to be complete.

Once a fault is detected, the BFT system will start a protocol switch. First, the system will switch to a transition protocol which activates all passive replicas, and ensures that they are all in the same state, by agreeing on a global history. Once this is done, the system falls back to the original BFT protocol that includes $3f + 1$ replicas so that it can proceed even in the presence of faults. After a specified number of transactions in this fallback protocol, it will switch again to the normal-case protocol. In order to prevent denial of service attacks by switching to often (which is expensive), the number of transactions in the fallback protocol increases
exponentially with every switch, ensuring that switches do not happen too often.

![Diagram of communication pattern](image)

**Figure 2.2:** Communication pattern of a transaction in a RePBFT system capable of tolerating one fault. Here replica 3 is passive, and receives an update only at the end of the transaction.

Figure 2.2 shows RePBFT, the ReBFT variant of PBFT. As can be seen, the number of replicas participating in the protocol is reduced, which in turn results in a reduction of messages. Nevertheless, the number of messages is reduced by only a constant factor, and communication complexity is still $O(n^2)$.

We categorize algorithms separating active from passive replicas as optimistic BFT protocols, since they make the assumption that faults are very rare occurrences.

**Zyzzyva**

Zyzzyva [35] is a BFT protocol where replicas speculatively execute requests. That is, replicas execute requests before they reach agreement on the order. As with PBFT, there exist views and primary replicas in
Zyzyva, and the primary replica proposes a sequence number to all replicas when it receives a request from the client. However, instead of initiating a voting round, replicas simply execute the request operation and send the result to the client. In an optimal situation (i.e. there are no failures of any kind) replicas do not learn whether agreement actually has been reached on the order of the request; only the client has this knowledge.

There are four possible cases for the replies obtained by the client:

a The client receives \(3f + 1\) consistent replies. This is the easy case: the client simply considers the transaction to be complete, and no further communication ensues.

b The client receives between \(2f + 1\) and \(3f\) consistent replies. In this case, the client creates a commit certificate from \(2f + 1\) consistent replies, and broadcasts this in a \texttt{COMMIT} message to all replicas. Some replicas might then have to roll-back their state, and have to ensure that their state is consistent with other replicas. They do this by finding out which sequence numbers are missing and requesting the corresponding requests from the primary through a \texttt{FILL-HOLE} message. If a replica encounters inconsistent view numbers for the same requests during this phase it will initiate a view-change. If a replica manages to make its history consistent with other replicas and execute the request, it will send a \texttt{LOCAL-COMMIT} message to the client. When a client receives \(2f + 1\) of these it considers the transaction to have completed successfully.

c The client receives fewer than \(2f + 1\) consistent replies. The client will then broadcast the request to all replicas, which in turn will attempt to forward it to the primary. If it turns out the primary is faulty, replicas will then act on this and initiate a view-change.

d The client receives replies with inconsistent sequence numbers. This is an indication of a faulty primary, and the client collects the inconsistent replies and broadcasts these to all replicas, as a proof of misbehavior of the primary. Replicas will subsequently roll-back their state to what it was before this transaction, and initiate a view-change so a different primary will be elected.

Even though this only gives a high-level overview of the protocol, we can see that while it initially may appear as simple, complexity increases considerably as soon as faults are encountered.
Two important observations can be made as well: firstly, replicas require the ability to roll back already executed operations. This is an important disadvantage, and rules out Zyzyva as a contender for certain use-cases for BFT algorithms that involve operations that cannot be reverted (e.g. the decision to jettison fuel from an aircraft). Secondly, while in PBFT (and most other classical BFT algorithms) the client is allowed to be Byzantine, in Zyzyva a Byzantine client can force the system into an inconsistent state. For example, if (just) the primary replica is faulty, it might equivocate on the sequence number of a request, observable by the client from replies (case d), but if the client fails to notify replicas about this they will permanently end up with inconsistently ordered requests, violating the safety property.

![Communication pattern of a transaction in a Zyzyva system](image)

**Figure 2.3**: Communication pattern of a transaction in a Zyzyva system, capable of tolerating one fault. In this figure, replica 3 is faulty. Solid arrows indicate messages sent during optimal execution, while dashed arrows indicate messages that are only sent when faults are detected.

Because voting rounds in Zyzyva use the client as an intermediary, replicas do not need to broadcast their vote to all other replicas. Therefore, communication complexity is $O(n)$. Moreover, as Figure 2.3 shows,
in the optimal case Zyzyva requires only 3 communication steps, thus minimizing the processing time for a single transaction.

Since in Zyzyva, replicas do not reach agreement in the common case, we categorize it as being a speculative BFT protocol.

2.2 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a piece of hardware that provides a secure code execution environment, isolated from other parts of the system. In particular, TEEs guarantee that other software running on the system is not able to read or modify the TEE memory. This makes them suitable for security-critical tasks. TEEs may also provide other features, such as sealed storage, allowing for data to be encrypted and stored on long-term storage, without having to reveal it, and remote attestation, which allows remote processes to verify the authenticity of the TEE and the integrity of the code running on it.

TEEs are becoming increasingly common. Intel provides the Intel Software Guard Extensions (SGX) technology [29] on most modern x86 processors, and ARM TrustZone [2] is available on many ARM devices. Both Intel SGX and ARM TrustZone enable developers to write their own applications running in a secure environment. There exist TEEs with fixed functionality as well, for example hardware implementing the TPM [48] standards.

For the purpose of this thesis any TEE satisfying the following properties suffices:

- The TEE provides a secure code execution environment, isolated from the rest of the system
- The TEE provides means to remotely attest the authenticity of the TEE and the integrity of the code running on it
- The TEE provides persistent storage that is both isolated and secure.

2.2.1 Intel SGX

Intel SGX provides trusted hardware components and CPU instructions in modern x86 processors that enable developers to run code securely, protected from even the operating system itself. Developers can write sensitive code as code that runs inside “enclaves”, which are isolated environments that ensure that other parts of the system cannot read or modify
their memory. Enclaves provide strictly defined interfaces to the encapsulating application, which allow developers to communicate with the secure portion of their program.

Additionally SGX provides means to remotely attest code.

2.2.2 Trusted monotonic counters

In this work, we require an ordering primitive to uniquely tag messages with. We describe a simple counter primitive that is able to guarantee uniqueness through use of TEEs.

The counter is required to be trusted, in that it is not possible for unauthorized code to read or modify the state of this primitive, and given a counter value, it has to be possible to remotely verify that the value was produced by some specific counter.

The second property it has to satisfy is monotonicity. That is, the counter will never assign the same value twice; it enforces a strictly increasing order on its state, which is updated with every query to the counter.

A TEE lends itself well for a counter with these requirements, as the trusted component is provided by the environment itself, and monotonicity can be guaranteed through remote attestation.

In order to be able to tag messages, we require a mechanism to bind a counter value $c$ to some message $M$. To this end, the TEE maintains an asymmetric key pair, and with every query accepts some message $M$ as input. It then signs the tuple $(c, M)$ with its private key, effectively binding $c$ to $M$. Monotonicity guarantees that this $c$ is unique and therefore it is impossible to bind the same $c$ to a different message.

To allow for remote verification, the TEE needs use remote attestation techniques to prove to a verifier that a specific key pair is maintained by this TEE. Whenever the TEE then signs a $(c, M)$ tuple with its private key, verifiers can assert that this value was indeed generated by the TEE.

What remains is to ensure that the TEE does indeed maintain a trusted monotonic counter, and does not simply pretend to do so towards verifiers. This can be done by either fixing the TEE functionality during the manufacturing process (i.e in hardware), or by remotely attesting the TEE code itself, ensuring that a trusted monotonic counter has been properly instantiated on it.

As stated, the TEE maintains a key pair $(s_i, p_i)$, where $s$ denotes the private (secret) key, $p$ the public key, and $t$ uniquely identifies the key pair. It also has access to a $\text{Sign}(x, s_i)$ function that signs message $x$ with private key $s_i$, producing $x_{\sigma_i}$, where $\sigma_i$ denotes a signature belonging to key pair $i$. 
The counter interface must expose at least one function, $\text{Increment}(M)$, which takes as input some message $M$. It then increments the monotonic counter and binds the newly obtained counter value $c$ to this data through a signature. When successful, the signed tuple $⟨c, M⟩_{σ_t}$ is returned to the caller.

**Algorithm 1** TEE Increment function

1. procedure $\text{Increment}(M)$ ▷ Increments the counter and binds it to $M$
2. State: $c$ ▷ counter value
3. State: $(s_t, p_t)$ ▷ Asym. key pair
4. Require: Atomicity of this algorithm
5. 2: if $c = \text{MAX INTEGER}$ then
6. 3: abort
7. 4: end if
8. 5: $c ← c + 1$
9. 6: $\text{Sign}(⟨c, M⟩, s_t)$
10. 7: return $⟨c, M⟩_{σ_t}$
11. 8: end procedure

The pseudocode for $\text{Increment}$ is listed in Algorithm 1. As can be seen, a check is included to ensure that the same value is not used twice in case of an integer overflow.

It is important to mention that this algorithm is only secure if the key pair is unique per instantiation of the TEE; when initializing the TEE, $c$ will be set to some initial value, and the if TEE is shut down after a few increments to the counter, it can be re-initialized to sign different messages for the same counter values unless the key pair is unique per instantiation as well. The algorithm can be extended to include a session-id that uniquely identifies the instantiation, to allow re-use of keypairs. We do not include this in this algorithm however, as the details of this vary between TEEs and we do not rely on it for our implementation in Chapter 5.

As a final note on trusted monotonic counters, when implementing this on a TEE, atomicity of any operation that modifies the state of the TEE has to be ensured in order to guarantee monotonicity of the counter value.

### 2.2.3 TEE-enabled BFT algorithms

A number of BFT algorithms utilize the power of trusted hardware to increase efficiency. One benefit of using TEEs is that it becomes possible to reduce the Byzantine fault tolerance problem complexity to that of crash
CHAPTER 2. BACKGROUND AND PRELIMINARIES

tolerant problems, lowering the minimum amount of replicas required to
tolerate faults to be just $2f + 1$ [18, 16, 50, 37, 32]. We describe one of these,
MinBFT [50], in more detail.

MinBFT

MinBFT is an algorithm based on PBFT\(^1\), with a similar communication
pattern. It also employs views for liveness, and utilizes a primary replica
for assigning a sequence number to requests.

Due to the possibility of equivocation, it is in general not possible to
achieve consensus with less than $3f + 1$ replicas in the presence of $f$ faults.
When we allow for the usage of TEEs however, we can eliminate the possi-
bility of equivocation. This is because if we require messages to be signed
by a TEE, this TEE can then add some unique tag to the message. Repli-
cas receiving a message will then compare this tag to what they expect
the next tag value to be. If these values are not equal, the receiver has
reason to suspect the primary of equivocation, and will not process the
message any further.

In MinBFT, a tamperproof trusted counter service similar to the one
described in Section 2.2.2 is assumed to exist in every replica, that is able
to produce a signed certificate proving that a certain counter value is
uniquely bound to some message. By using this counter value as a se-
quence number for incoming requests, it is able to eliminate the voting
round required by PBFT in its PREPARE phase: when replicas receive a
message from the primary with a counter value that is bound to a request
through a certificate, they are sure that this counter value can never be
used for another request.

Since equivocation is no longer an issue when it comes to ordering re-
quests, it becomes possible to decrease the minimum number of replicas
to $2f + 1$, and the voting round to agree on the request order is no longer
necessary.

Figure 2.4 shows the communication pattern of MinBFT. In order to
enable replicas to properly detect a faulty primary, the client broadcasts
requests to all replicas, instead of only the primary as in PBFT. Then, in
the PREPARE phase, the primary obtains the signed certificate binding a
sequence number to the request and broadcasts this to the replicas. In
the remaining two phases, replicas vote to commit and then send their
replies to the client.

\(^1\)The paper also introduces MinZyzyvyva, but we will not discuss this algorithm here.
Figure 2.4: the communication pattern of a transaction in a MinBFT system capable of tolerating one fault. Note that with MinBFT only $2f + 1$ replicas are necessary to tolerate $f$ faults. Replica 0 acts as the primary, while 2 is faulty and does not send any messages. As can be seen, the PREPARE phase no longer includes a voting round, because of the tamperproof assumption of the trusted counter service.

Even though only $2f + 1$ replicas are required, the COMMIT phase still includes an all-to-all broadcast, leaving the communication complexity at $O(n^2)$.

In Section 4.2.4.1 we examine the view-change sub-protocol in MinBFT, to illustrate the necessity of TEEs in this process.

2.3 Multisignatures

A multisignature scheme\cite{[4]} is a signature scheme that allows $N$ signers to collectively sign a message $M$. These $N$ signers have individual private keys $s_1 \ldots s_n$ that are not known to the public or even other signers, and individual public keys $p_1 \ldots p_n$ that are publicly available. The public keys
can be aggregated into a collective public key $p_{1\ldots n}$, and individual signatures can be aggregated into a collected signature $x_{1\ldots n}$. The public key $p_{1\ldots n}$ can then be used to verify that signers 1 to $n$ all participated in $x_{1\ldots n}$. Multisignature schemes aim to make this more efficient by requiring only a single signature to be transmitted to and verified by a verifier, rather than requiring $n$ transmissions and verifications. The aggregated signature can be larger in size than an individual signature, but it is smaller than the collection of $n$ individual signatures.

Multisignature schemes are slightly different from signature aggregation schemes, in that signature aggregation schemes enable efficient aggregation of $n$ signatures on $n$ distinct messages, while multisignature schemes enable efficient aggregation of $n$ signatures on one message [9].

### 2.3.1 Boneh-Lynn-Shacham signatures

The Boneh-Lynn-Shacham [9] (BLS) aggregate signature scheme relies on pairing-based elliptic curve cryptography to deliver compact aggregated signatures. In this scheme, signatures consist of a single $x$ coordinate on an elliptic curve, and their aggregation amounts to multiplication of these coordinates. The resulting aggregate signature is therefore also an $x$ coordinate and of the same size as the individual signatures.

This is an aggregate signature scheme, and explicitly does not allow the signing of one message, as it makes signature forgery possible. However, it can be turned into a multisignature scheme by requiring each signer to publish a zero-knowledge proof of the discrete logarithms belonging to their public keys, before signing any message [9, page 6].

Two key characteristics of BLS aggregate signatures are that they require only one communication round trip to generate one aggregate signature, whereas most other schemes require two round trips (e.g. Schnorr multisignatures [40]). A second characteristic is that the generated signatures are very small, making them ideal for situations where limiting network traffic is deemed important.

### 2.3.2 ByzCoin

ByzCoin [34] is a BFT protocol designed to solve scalability issues with existing consensus schemes used by cryptocurrencies. To do this, it builds on PBFT, but (among other improvements) reduces network complexity to $O(n)$ by using collective signing techniques [47] to eliminate the need for all-to-all broadcasts during voting. Collective signing uses multisignature schemes, and requires all participants to send their signature (i.e.
vote) to one designated leader, which then aggregates all votes and broadcasts these to all participants. This is a trade-off: it requires more communication steps per voting round than an all-to-all broadcast, but fewer messages overall and thus less bandwidth. The added latency introduced by the extra communication steps can be seen introducing as constant overhead to the protocol, but in return the bandwidth requirements grow in $O(n)$. Therefore, for larger $n$ it will perform better.
Chapter 3

Problem description

As we have seen in Chapter 2, BFT protocols need to provide safety and liveness properties in order to be considered correct. To achieve these requirements entails considerable complexity, which has a negative impact on performance. In Chapter 2 have seen how protocols aim to increase efficiency by various means. Often these means rely on some extra or stronger assumptions, such as rollbacks and non-Byzantine clients for Zyzyvia, and a trusted counter for MinBFT.

In this chapter we elaborate on the various assumptions our work is built upon. We then identify several fault-tolerance goals that we aim to satisfy, after which we describe a set of non-correctness-critical properties, which we formulate as performance goals.

3.1 Problem description

All existing BFT algorithms make either the assumption that all replicas have access to a TEE, or assume that none have a TEE. Neither of these assumptions cover the situation where a strict subset of all replicas has access to a TEE. There are legitimate reasons why only a subset of replicas might have TEEs available. Neither assumption accommodates for this, and one will have to fall back to the no-TEE assumption in order to achieve Byzantine fault tolerance. This wastes the potential of the available TEEs however, and passes by on performance improvements that may be gained from these.

Additionally, many classical BFT algorithms operate with $O(n^2)$ communication complexity. Previous work (see Section 2.3.2) has already shown that this can be improved upon to achieve $O(n)$ complexity instead. There is currently no general transformation that can be applied to these
algorithms to achieve better communication complexity.

Based on the above observations, we motivate our work with the following question:

- Can we achieve Byzantine consensus with fewer than \( n \) TEEs a single voting round, and \( O(n) \) communication complexity?

### 3.2 Assumptions

In this work, we make the following assumptions:

- **There can be up to** \( f = \lfloor \frac{n-1}{3} \rfloor \) **faulty replicas in the system.** This corresponds to the standard BFT adversary model.

- **Communication satisfies the weak synchrony assumption.** We rely on the traditional weak synchrony assumption introduced by PBFT [12] (as described in Section 2.1.2) in order to be able to guarantee liveness.

- **It is rare for a replica to fail.** When designing BFT algorithms, this is an often-made assumption, and is realistic, as many use cases for BFT algorithms involve situations where under normal circumstances all replicas behave correctly. This assumption suggests that algorithms should optimize for the "common" case, where there are few or no faults. For example, algorithms such as Zyzzyva [35] use speculative execution to reduce the number of voting rounds in the common case, and optimistic BFT algorithms such as RePBFT [19] reduce the number of replicas actively participating in voting rounds. Additionally, this assumption allows us to exercise some freedom in the efficiency of view-changes; typically these are much more expensive than processing an operation, but when a view-change is a relatively rare occurrence, the performance cost remains acceptable.

- **A subset of replicas has access to a TEE.** This is a weaker assumption than the one made by all existing TEE-reliant BFT protocols: they assume that TEEs are available to all replicas. This weaker version gives us more flexibility and allows for optimizing BFT protocols for new scenarios.

- **TEEs are secure.** With "secure" we mean that they cannot be compromised, their code can be attested remotely, and that it is impossible to obtain any data from the TEEs, except the values that are
returned by the programming interface of these TEEs. We also assume that TEEs can crash.

3.3 Fault tolerance goals

We separate our goals into two classes: Fault-tolerance goals and Performance goals. Fault-tolerance goals concern the correctness of the system as well as its resilience against Byzantine faults. We consider the following fault tolerance goals:

F-1 Maintain correctness properties: If a protocol does not satisfy the necessary correctness properties, it cannot be considered to be Byzantine fault-tolerant. Since we aim to create a drop-in replacement for existing protocols, we must guarantee that our transformations do not break these properties when we apply them to existing protocols. Specifically, we need to guarantee that if a protocol satisfies the safety property, modifying it according to our framework does not violate this property. Similarly, if a protocol satisfies the liveness property, this property must still hold after we modify the protocol according to our framework.

F-2 Maximize the number of tolerated faults: One aspect in which TEE-powered BFT algorithms such as FastBFT and MinBFT stand out is that they tolerate $\left\lfloor \frac{n-1}{2} \right\rfloor$ Byzantine faults, whereas the maximum without TEEs is $\left\lfloor \frac{n-1}{3} \right\rfloor$. It is thus clear that using TEEs enables us to increase the amount of tolerated faults, although it should be noted that TEEs alone are not sufficient to achieve this, we also need authenticated communication channels between replicas [17]. This can be easily done with asymmetric cryptography or MACs, and it is standard to make this assumption in any case. Our goal is to increase the number of tolerated faults for a given number of replicas (or, equivalently, decrease the number of required replicas for a given number of tolerated faults) when not all replicas possess a TEE.

F-3 Minimize the number of TEEs necessary: We also want to minimize the hardware resources required in order to implement this framework. Having to invest in fewer TEEs makes the framework more easily accessible and more widely applicable.

F-4 Minimize the trusted computing base: Finally, we want to keep the trusted computing base as simple as possible. Requiring complex
code increases the probability of bugs appearing, and bugs in a trusted computing base can prove fatal for the overall trustworthiness of the system. Ideally, the code should be small and simple enough to allow for formal verification of the functionality. In addition to that, it becomes easier to include dedicated trusted hardware with the TEE-provided functionality in machines, as the TEEs will require less memory, computing power, and storage.

3.4 Performance Goals

Another important goal of our framework is to improve performance both by the use of a TEE and by other means. Specifically, we seek to achieve the following:

P-1 Minimize communication complexity: Firstly, we want to minimize the asymptotic communication complexity of algorithms through application of our framework. Many (older) BFT algorithms require replicas to broadcast their votes to all other replicas. This results in $O(n^2)$ messages being sent for one vote. Recent work shows that message aggregation using signature aggregation or secret sharing [34, 37] can reduce the communication complexity to $O(n)$, reducing bandwidth requirements and noticeably improving the efficiency of voting rounds. Since there are already protocols with these techniques in place, we should aim to reduce the communication complexity of any protocol to which our framework is applied to $O(n)$ as well, so that it can compete with the current state of the art.

P-2 Minimize the number of voting rounds: Existing BFT protocols that make use of TEEs require fewer voting rounds to complete an operation, which contributes significantly to their efficiency. While we cannot rely on all replicas having a TEE, we do want to improve the efficiency of existing protocols where possible. One of the efficiency goals of this research project is therefore to see if we can minimize the required number of voting rounds.

P-3 Minimize transaction latency: The latency of one transaction is a good way of evaluating performance. While the biggest factor in latency is the communication complexity, it is worthwhile to look at other ways of reducing transaction latency. For example, using computationally intensive cryptographic algorithms adds noticeably to
latency as well, especially when we consider small systems (e.g. 4 replicas). Therefore, we list this as a separate performance goal.

3.5 Summary

To summarize, we consider the following goals, to which we will refer throughout this thesis:

F-1 Maintain correctness properties
F-2 Maximize the number of tolerated faults
F-3 Minimize the number of necessary TEEs
F-4 Minimize the trusted computing base
P-1 Minimize communication complexity
P-2 Minimize the number of voting rounds
P-3 Minimize transaction latency
Chapter 4

Framework

Our proposed framework utilizes trusted hardware to maintain a "Single Active Counter" (SAC) at the primary replica. We therefore call it SACBFT.

The SACBFT framework can be split into two parts: the first is a replacement for the ordering process of existing protocols, while the second part modifies any remaining voting rounds (e.g. the COMMIT phase in many BFT algorithms) to make use of signature aggregation. In order to see why we need to replace the ordering phase, we will first look at how PBFT orders requests, after which we will show how SACBFT is able to replace this.

Afterwards, we look at voting phases in general, and how their communication overhead can be reduced by making use of signature aggregation techniques.

We then examine view-changes, where we discuss how MinBFT is able to perform view-changes with only $2f + 1$ replicas while tolerating $f$ faulty ones, and describe how SACBFT affects the view-change process of existing protocols.

Finally, we show how to apply SACBFT to existing protocols, and apply it to PBFT and RePBFT to produce SACPBF and SACRePBFT.

4.1 Design outline

The first SACBFT transformation replaces the ordering phase of an existing protocol by using a trusted monotonic counter accessible by the primary to bind counter values to incoming requests. It does this by signing a tuple consisting of the cryptographic hash of the request and a fresh (i.e. has not been used before) counter value. This is then sent to all
Figure 4.1: An illustration of the SACBFT transformations. The first transformation (top) removes the voting round from the ordering process, replacing it with a simple broadcast from the primary 0 (which now has TEE) as can be seen on the top-right figure. The second transformation lowers network complexity of voting rounds to $O(n)$, by aggregating messages at the primary. Communication arrows and phases affected by the transformations are colored blue.
replicas in a PREPARE message. Because the primary is the only replica that actively maintains a counter, we call this counter the “Single Active Counter” (SAC). The top half of Figure 4.1 depicts the SACBFT transformation to the ordering process.

The second transformation concerns voting phases. Many classical BFT protocols have at least one voting phase after the ordering phase, typically called the COMMIT phase. Since replicas need to learn about the votes of other replicas, they typically broadcast their vote to all replicas. This results in \( O(n^2) \) messages being sent in a single voting phase. In SACBFT, replicas send their vote (in the form of a signature) to only the primary instead, which then broadcasts an aggregation of all received signatures. With this optimization, communication complexity of voting rounds is reduced to \( O(n) \). The bottom half of Figure 4.1 depicts the voting round transformation made by SACBFT.

4.2 Communication

The communication pattern of most post-PBFT BFT protocols can be divided into a number of phases. For example, most protocols have a REQUEST phase, where the client sends its request to one or more replicas. SACBFT is no different in this respect. The subsequent phases can differ per protocol, but many of these BFT protocols rely on similar patterns, and SACBFT can be applied to these. In this section we discuss these common phases and the SACBFT-specific improvements that can be made to them. Note that SACBFT is a framework applicable to many BFT protocols, and we only discuss the elements that are required by SACBFT itself. Usually, a protocol includes more data per message, e.g. a view number or replica identifiers. When applying SACBFT to an existing protocol, we extend these original messages. That is, we leave the original data in a message as is (unless explicitly stated otherwise) and add the SACBFT-specific data to it.

Many BFT protocols agree on the sequence number to be used for a request by using a two-phase voting round. Typically these phases are called the PREPREPARE and PREPARE phases respectively. Since SACBFT determines transaction ordering by means of an attested monotonic counter, it does not require such an agreement. Instead, it just needs to distribute the sequence number obtained from the trusted counter (together with the request) to all replicas, each of which can then verify that this number was indeed issued by the trusted counter and meant for this particular request. We call this the PREPARE phase, and we will specify the
SACBFT-specific features in more detail in subsection 4.2.1. In most classical non-TEE BFT protocols, the PREPARE phase is followed by a COMMIT phase, where replicas attempt to reach consensus on the next state of the replicated state machine. As this does not concern the sequence number directly, SACBFT also requires this phase. We will see however, that some improvements can be made to any voting round, including the COMMIT phase.

4.2.1 PREPARE phase

In order to see how we can improve upon the PREPARE phase in existing protocols, we look at how ordering is achieved by PBFT. Many protocols use the same ordering procedure, meaning that this analysis holds for those protocols as well.

In PBFT, the primary assigns a sequence number to incoming requests, which is then broadcast to all other replicas in a PREPARE message. Replicas then agree on this sequence number by broadcasting a PREPARE message to all replicas. The authors of PBFT define \(\text{prepared}(m, v, n, i)\) predicate to be satisfied after a (correct) replica \(i\) broadcasts a PREPARE message for request \(m\), view \(v\), and sequence number \(n\), and has inserted both messages into its log. After the replicas have received enough PREPARE messages, the \(\text{prepared}\) predicate ensures non-equivocation of the sequence number [12, page 4].

The replicas then broadcast a COMMIT message which finalizes the ordering and allows replicas to process the request. Thus, ordering in PBFT is a procedure consisting of two phases. To see how a TEE can help us improve upon this, we first look at why two phases are necessary.

It is easiest to illustrate the necessity of a two-phase agreement by omitting the COMMIT phase in PBFT (thus allowing replicas to execute requests immediately after a successful PREPARE phase) and showing that the protocol is now vulnerable to certain attacks.

Why PBFT requires two voting rounds

In the weakly synchronous network model, an attacker can delay messages for any length of time, as long as they are delivered at some point in time (see Section 2.1.2). As we shall see, this assumption together with the fact that during the PREPARE phase the primary replica can equivocate on the request message for a particular sequence number, allows a malicious group of replicas to coordinate an attack and force the system
Figure 4.2: A possible attack on the PBFT protocol when the COMMIT phase is omitted. Each horizontal step separates the $3f + 1$ replicas into four groups. The red outline indicates a group of $f$ faulty replicas whereas a black outline indicates correct replicas. Blue arrows relate to request $r$, while red arrows relate to request $r'$. Lines ending in dots indicate that messages get delayed. A blue fill indicates that a group of replicas has executed $r$, while a red fill shows that $r'$ has been executed instead.

into an inconsistent state. Figure 4.2 shows how such an attack can be executed.

Each horizontal bar is divided into four sections, the first three each representing $f$ replicas, and the last bar represents the remaining replica. The first bar (colored with a red outline) consists of $f$ faulty replicas, including the primary replica. All the other replicas are assumed to behave correctly. Step A.1 shows how a Byzantine-faulty primary can equivocate during the PREPAPARE phase. The blue arrows indicate messages relating to request $r$, while the red arrows contain a different request $r'$. Once all replicas have received the PREPARE messages they will attempt to broadcast a PREPARE message.

However, in Step A.2 the attackers delay all messages in the system (shown as lines ending in . . . ) except for those with the rightmost replica
as their destination. This replica then receives $2f + 1$ (including its own) PREPARE messages for request $r'$. Since we have omitted the COMMIT phase, this will cause the replica to execute $r'$, indicated with the red fill in its bar.

The rightmost replica will now be “ahead” of the other replicas in its state, but the safety property is not yet violated; all requests thus far have been executed in the same order.

The remaining $2f$ correct replicas will initiate a view-change once they have received the messages that were delayed at Step A.2. View-changes are supposed to ensure that all correct replicas end up in the same state by requiring all replicas to share a log of their processed requests, with the necessary data for other replicas to be able to verify the legitimacy of these requests and execute them if necessary.

However, during the view-change in step A.3, the $f$ faulty replicas will pretend that they only sent a PREPARE for $r$, which together with the $f$ correct replicas that initially received a PREPARE for $r$ will form a majority of $2f$ replicas claiming that $r$ is the request that should be executed for this sequence number. The PBFT view-change protocol only requires $2f$ such messages and thus this is enough for all remaining $3f$ replicas to agree on executing $r$ (indicated with a blue fill in their bars).

As can be seen in the bottom bar, we end up with $3f$ replicas having executed $r$, and one correct replica having executed $r'$ instead. This is an inconsistent state and violates the safety property, and is thus a vulnerability in the protocol.

When we include the commit phase, this is no longer possible, as Figure 4.3 shows. We consider two possible variations on the attack, listed as A and B, and show that neither of them can be successful. Both scenarios start in the same way, with a malicious primary equivocating on the PREPARE message. In scenario A, once again all messages except for those sent to the rightmost replica are delayed, causing only the rightmost replica to arrive in a prepared state as indicated by the half-filled bar. However, in order for the replica to actually execute the request it will need to receive $2f + 1$ consistent commit messages as well. As no other correct replica has arrived in a prepared state, it will only be able to receive $f + 1$ consistent COMMIT messages (Step A.3), and thus the request will never be executed.

In scenario B, the attackers now delay all PREPARE messages except for those sent to the rightmost $f + 1$ replicas (Step B.2). Now, this will put all of those replicas in a prepared state. Step B.3 shows that it is then possible to have the rightmost replica receive enough COMMIT messages to execute $r'$. However, there are now $f + 1$ correct replicas that are in possession of a valid PREPARE message for $r'$ as well as $2f + 1$ valid PREPARE messages
for this same request. This is enough proof to convince all other replicas that \( r' \) is the request that belongs to this sequence number during a view-change.

Even without taking view-changes into account, it is impossible to force the remaining \( f \) correct replicas into a prepared state for a different request, as they will at most receive \( 2f \) PREPARE messages (\( f \) from themselves and \( f \) from the faulty group), while they require \( 2f + 1 \) such messages. The COMMIT phase is thus a necessary part of PBFT, and cannot be omitted without violating the safety property. This holds for many BFT protocols, and we will therefore need to look at ways of improving the efficiency of this voting round, rather than omitting it. We describe how to improve voting rounds after we specify the SACBFT PREPARE phase.

![Diagram](image)

**Figure 4.3:** Two variants on the attack depicted in Figure 4.2, that both get stuck, as they will need to be able to provide \( 2f + 1 \) valid PREPARE messages to put a replica in the committed state, but are unable to do so. Blue/red arrows indicate messages relating to requests \( r \) and \( r' \) respectively. Half-filled bars indicate that a group of replicas has reached the prepared state for the request belonging to their color. Full bars indicate that they have also committed the request.

**SACBFT PREPARE phase**

As we have seen in the previous subsection, two voting rounds are required in all protocols that do not rely on TEEs, because the primary is otherwise able to use the same sequence number for two different requests which ultimately leads to the system arriving at an inconsistent state.
CHAPTER 4. FRAMEWORK

In SACBFT, the primary replica has a trusted monotonic counter as described in Section 2.2.2, which we use to bind sequence numbers to requests. Moreover, since the primary replica is the only replica that assigns these sequence numbers, only the primary needs to have access to such a counter at any one time. When we bind requests to sequence numbers like this, replicas can simply verify the legitimacy of the resulting signatures to maintain a globally consistent ordering of requests. This observation forms the cornerstone of the SACBFT framework; as long as at least one replica has a trusted counter at its disposal, we can use this counter to reduce the number of voting rounds.

As in PBFT, a PREPARE message is sent to all replicas by the primary replica when it receives a request message $M$. First however, the TEE has to bind this request to a specific sequence number. This is done by calling the TEE-function $\text{Increment}(H(M))$ where $H(x)$ is a cryptographic hash function. The TEE will return a signed tuple $\langle c, H(M)\rangle_{\sigma_{PC}}$, where $c$ is the obtained counter value, and $\sigma_{PC}$ is a signature by the primary’s trusted counter. Consisting of the sequence number and this hash. The primary then has to share this counter with all replicas by broadcasting the message $\langle \text{PREPARE}, M, \langle c, H(M)\rangle_{\sigma_{PC}} \rangle$.

Every replica that receives such a PREPARE message must verify that the hash of $M$ is equal to the hash signed by the counter, as well as that $\langle c, H(M)\rangle_{\sigma_{PC}}$ is a valid signature. For signature verification, we assume the existence of a Verify$(X, P)$ function that returns true if $X$ is indeed signed by $P$, and false otherwise. When either of these fails, the replica must assume that the primary is malicious, and will request a view-change.

The pseudocode for these procedures is given in Algorithm 2.

4.2.2 Voting

While the improvements from the previous subsection replace the voting round in the PREPARE phase, many BFT protocols require at least one other voting round. Usually this is the COMMIT phase. Classical protocols require every replica to broadcast their vote to all other replicas. This results in $O(n^2)$ messages being sent, which hampers scalability. Recently, work has utilized signature aggregation techniques to reduce the communication complexity to $O(n)[34]$. With SACBFT we apply these techniques to all remaining voting rounds in the original protocol, reducing the overall communication complexity to $O(n)$.

For this, we rely on a multisignature scheme, which provides a $\text{MultiSign}(d, k_i)$ algorithm that will sign data $d$ with private key $k_i$ of some replica $i$, yielding $\langle d \rangle_{\sigma_i}$, where $\ast$ denotes that $\sigma_i$ is signed using a
Algorithm 2 SACBFT PREPARE phase

1: upon reception of request message $M$ at the primary replica $p$ do
2: \[ \langle c, H(M) \rangle_{\sigma_p} \leftarrow \text{TEE.Increment}(H(M)) \]
3: broadcast $\langle \text{PREPARE}, M, \langle c, H(M) \rangle_{\sigma_p} \rangle$
4: end

5: upon reception of $\langle \text{PREPARE}, M, \langle c, h \rangle_{\sigma_p} \rangle$ do
6: if Verify($\langle c, h \rangle_{\sigma_p}, p^C$) \neq true then
7: Request view-change and abort transaction
8: end if
9: if $c \leq$ latest successfully used counter then
10: Request view-change and abort transaction
11: end if
12: if $H(M) \neq h$ then
13: Request view-change and abort transaction
14: end if
15: continue with next phase in protocol
16: end

multisignature scheme. The scheme also needs to provide an
\text{Aggregate}($\langle d \rangle_{\sigma_1}, \ldots, \langle d \rangle_{\sigma_i}$) function that, given a number of signatures
on the same message $d$, produces an aggregated signature $\langle d \rangle_{\sigma_{1...i}}$ for
this message. Finally the scheme needs to provide a \text{MultiVerify}($\langle d \rangle_{\sigma_{1...j}}$
function that verifies an (aggregated) signature for some message $d$.

A voting round can be improved by having all replicas sign their vote
through the multisignature scheme and sending this to the primary replica.
The primary then collects all issued votes, and once it has received enough
valid votes (i.e. they pass verification) it will aggregate the received signatures
into one signature. If the primary replica can equivocate during the
ordering process, then the voting rounds will require at least $2f + 1$ votes
to tolerate Byzantine faults. When equivocation during the ordering
process is not possible, then $f + 1$ votes are sufficient to prove that at least
one correct replica has committed to the vote. This is because when the
primary cannot equivocate on the round number, Byzantine replicas will
not be able to convince a correct replica to commit to a different message
for this same round number either, even when banding together.

This aggregated signature is then broadcast to all replicas, which will
verify the aggregated signature to make sure that it does indeed represen
t enough replicas. To illustrate these techniques we have included
psudocode for an abstract improved version of a typical voting round
in Algorithm 3 and Algorithm 4. Note that if not enough replicas vote, the current transaction cannot complete successfully, and it will have to be aborted, and thus abstention is equivalent to a vote against executing the request operation. When voting, replicas need to account for the fact that the primary might have failed and will not reply with an aggregated signature. Therefore replicas must wait for a reasonable duration, after which they will request a view-change if they have not received an aggregated signature.

Note also that the vote simply consists of signing the received sequence number. Should a particular protocol require more than one voting phase, we can accommodate this by simply including a phase-tag in the signed data.

**Algorithm 3** SACBFT voting round (all replicas)

1. **when** replica $i$ participates in a voting round **do**
2. $c \leftarrow$ sequence number bound to current transaction
3. $(c)_{\sigma_i^*} \leftarrow \text{MultiSign}(c, k_i)$
4. set a timer to wait for a reply from the primary
5. send $(<\text{PHASE NAME}>, (c)_{\sigma_i^*})$ to primary replica
6. **end**

7. **upon** reception of $(<\text{PHASE NAME}>, (c)_{\sigma_{1,2f+1}^*})$ at replica $i$ **do**
8. if MultiVerify($(c)_{\sigma_{1,2f+1}^*}) \neq \text{true}$ **then**
9. Request view-change and abort transaction
10. **end if**
11. Cancel the timer waiting for a reply from the primary
12. **end**

**Algorithm 4** SACBFT voting round (primary)

1. **upon** reception of enough $(<\text{PHASE NAME}>, (c)_{\sigma_i^*})$ messages at primary replica **do**
2. if $\exists i \text{ MultiVerify}((c)_{\sigma_i^*}) \neq \text{true}$ **then**
3. **return** “Not all signatures are valid”
4. **end if**
5. $(c)_{\sigma_{1,2f+1}^*} \leftarrow \text{Aggregate}(((c)_{\sigma_1^*}, ((c)_{\sigma_{2f+1}^*})
6. broadcast $(<\text{PHASE NAME}>, (c)_{\sigma_{1,2f+1}^*})$ to all replicas
7. **end**

On Algorithm 4, line 3, an error message is returned. The actions taken based on this error can vary from implementation to implemen-
CHAPTER 4. FRAMEWORK

...tation. One possibility is for the primary to keep waiting until it has received replies from all replicas, thus allowing for a chance to still obtain enough valid votes before it considers the transaction a failure.

4.2.3 REPLY

The REPLY phase consists of one or more replicas sending the outcome of the request to the client. It is not necessary for the client to have any knowledge of the internal state of the system, which includes the counter value. Because all changes a protocol caused by the SACBFT transformation are purely internal (they only concern communication between replicas), nothing changes in the REPLY message of the original protocol. This means that the SACBFT transformations can be applied without modifying client software, which gives SACBFT an advantage over existing TEE-based protocols.

4.2.4 View-changes

Since in SACBFT we can make use of a TEE accessible by the primary replica, it makes sense to explore whether we can use this TEE to optimize view-changes. In order to see that this is not possible, we will look at what happens when we adapt MinBFT to have only one TEE (at the primary replica).

MinBFT with one TEE

MinBFT reduces the number of required replicas to $2f + 1$, however it is assumed that every replica has access to a TEE.

In MinBFT, every replica keeps track of whether incoming PREPARE messages have consecutive sequence numbers called Unique Identifiers (UIs). When a replica does not receive a PREPARE message for the next expected sequence number in time, it will broadcast a request for a view-change. It should be noted that in MinBFT replicas can execute requests without receiving a PREPARE message from the primary, if they receive $f + 1$ consistent COMMIT messages for this particular UI from other replicas. This is possible because the UI is generated inside a TEE, and if the verification procedure checks out it is assumed to have originated from the primary replica.

Now, if we look at a variant of MinBFT where only the primary replica has access to a TEE, it will be possible for the system to arrive at an inconsistent state if the attacker executes a well-timed view-change. Figure 4.4...
Figure 4.4: A possible attack on MinBFT, if it only required the primary replica to have a TEE. Blue arrows indicate messages containing information up to request $k$, while red arrows indicate messages carrying information up to request $k - 1$. Black arrows do not carry information related to any request. Half-filled bars indicate that a group of replicas have reached the prepared state for request $k$, and completely filled bars indicate that it has also been committed.

shows the execution of this attack. It starts with a client sending a request $k$ to all replicas. This starts a timer in every replica, which after a certain time period will result in the replica requesting for a view-change if the request has not been handled by then. In step A.2 and A.3, the request is executed in a normal way, except that the rightmost replica is not involved.
CHAPTER 4. FRAMEWORK

at all: the primary never sends a PREPARE message to the rightmost replica, and all subsequent messages to that replica are delayed by the attackers.

Because the rightmost replica has not received anything, it will request a view-change after some period of time. As step A.4 shows, the malicious replicas all participate in this, and will request a view-change as well. This results in all replicas receiving \( f + 1 \) REQ-VIEW-CHANGE messages, which is enough to trigger an actual view-change.

In MinBFT, VIEW-CHANGE messages include a log of all successfully executed messages in the last view, so that replicas that are not up to date can work through this list and reach the most recent state before entering the next view. However, all malicious replicas can work together to equivocate on this log. As indicated by the blue arrows in step A.5, they will send a log that includes \( k \) to all replicas, except for the rightmost replica. To this replica, they send a log that does not include this last request \( k \); it will only receive information on requests 1 to \( k - 1 \). Meanwhile, attackers will delay messages to or from the rightmost replica, except for the messages they send themselves.

With this, the view-change operation can be completed, but the rightmost replica will be left in a state where \( k \) has not been executed, and will no longer accept messages from the previous view so it will never reach a state where \( k \) has been executed. This violates the safety property.

When all \( n \) replicas have access to a TEE, this attack can be protected against by requiring every VIEW-CHANGE message to include a UI (unique identifier generated by the TEE). Because these UIs need to be sequential, equivocation can be detected by all replicas, so it will no longer be possible to equivocate on the log.

What this shows us is that while using a TEE to bind sequence numbers to requests stops Byzantine replicas from equivocating on requests themselves, we still need to take into account that the system can be put into an inconsistent state by selectively omitting (information in) messages.

One might be inclined to think that \( f + 1 \) TEE-enabled replicas are enough to prevent an attack like this from happening to a view-change, but in that case an attacker can violate the liveness property by controlling \( f \) TEE-enabled replicas and prevent them from participating in a view change, or having them participate with an incorrect message log. In the case where there are somewhere between \( f + 1 \) and \( n \) TEE-enabled replicas, an attacker will still be able to do this, as at least \( f + 1 \) consistent view-change messages are needed to be sure that at least one correct replica has participated in it.
View-changes in SACBFT

Looking at MinBFT, it seems as though we require $2f + 1$ TEEs for a successful view-change. Whether it is possible to reduce the number of required replicas when the number of TEE-enabled replicas exceeds $x$ for $1 \leq x < n$, remains an open question for now. We circumvent this by not relying on TEEs at all during a view-change, since we cannot make any guarantees on how many replicas have access to TEEs. We leave exploring possible improvements as a topic for future research.

When applying the SACBFT transformation to an existing protocol, view-changes can largely remain the same. While we do not modify any functionality in a protocol’s existing view-change algorithm, we do need to change the way a protocol decides on the next primary.

We cannot elect just any replica to act as the next primary when a view change occurs; we need to make sure that it has a TEE, otherwise we cannot make use of trusted counter. However, if we have fewer than $f + 1$ TEE-enabled replicas, this will not always be possible, since all of them can fail. Therefore, we change the selection procedure for primaries. Primary selection is done in a round-robin fashion, where all TEE-enabled replicas are first in line. So, if there are $1 \leq x < f + 1$ TEE-enabled replicas, there will be $x$ attempts to start a view with a TEE-enabled primary. Once we exceed $x$, we will fall back to the original protocol that includes two voting rounds to provide ordering. We must ensure that the order in which replicas attempt to elect primaries is deterministic, so that all correct replicas will choose the same primary. We can base the order on e.g. the replica id (assuming there exists ordering in their identifiers). This way, we maintain liveness of the protocol, even when all TEE-enabled replicas fail.

Falling back to the original protocol will negatively impact performance, but does not need to be catastrophic. Rules can be implemented to attempt retry the SACBFT protocol after a given amount of time, or after manual intervention has been taken by the system administrators. Moreover, the implementation of the protocol does not have to be that much more complex just because it contains a fallback protocol, as shown by [25], and it can even improve the flexibility of the protocol.

When we have at least $f + 1$ TEE-enabled replicas it is no longer necessary to consider replicas without a TEE as potential primaries for a new view, as we are guaranteed to eventually reach a TEE-enabled replica that has not failed. This way we will not have to fall back at all, since liveness is ensured even with just this set of TEE-enabled replicas as candidates for a primary.
CHAPTER 4. FRAMEWORK

As a final note on view-changes, we mention that while their functionality does not change, care has to be taken when applying the SACBFT transformations to existing protocols, in order to not break view-changes. For certain protocols (e.g. PBFT), view-changes rely on PREPARE messages as proof that a request was assigned a sequence number. When applying SACBFT, this will have to change to rely on PREPARE messages or simply the \( (c, H(M))_{\pi, c} \) tuple instead.

4.2.5 Correctness

The correctness properties of SACBFT rely heavily on the correctness properties of the underlying protocol. In most cases, the correctness arguments remain valid with minor changes.

We need to show that if the protocol satisfied the safety property without SACBFT, this still holds with the SACBFT changes to the PREPARE phase. Luckily, this is relatively easy to see: within a view, monotonicity of the sequence number is guaranteed because of the TEE used by the primary, and we assume that actions performed by this TEE are atomic and do not leak any information apart from their return values. Correct replicas check whether a sequence number has already been used before accepting an incoming PREPARE message, preventing replay attacks. Moreover, because we use a trusted monotonic counter, non-equivocation is ensured. This is a property that is often used in proofs of correctness (e.g. in [12] and [37]). Since with SACBFT this mid-point is provided, the proofs will still hold.

The message aggregation techniques used by SACBFT only modify the way messages are distributed, and moreover the primary is unable to modify or forge incoming votes because it does not possess the correct private keys. This does not change the safety properties satisfied by the original protocols in any way either.

Finally, in classical BFT protocols, view-changes are often needed to guarantee liveness. We only change the order the primary replicas are picked, which has no impact on liveness, as it is still guaranteed that eventually a correct replica will be picked to act as a primary in the next view.

4.3 Applying SACBFT

The two transformations described in this chapter can be applied to a protocol separately from each other. The ordering phase transformation has some requirements and is not completely trivial to apply to an existing
CHAPTER 4. FRAMEWORK

protocol. The voting round transformation on the other hand is fairly simple and has no special requirements.

For both transformations, one first has to ensure that the protocol includes a primary replica.

4.3.1 Applying the ordering phase transformation

In order to apply SACBFT ordering phase transformation to a protocol, one first has to make sure that the protocol satisfies the following:

1. Sequence numbers are "picked" by one primary replica, and distributed to all other replicas through this replica.

2. Its safety proof includes non-equivocation of the sequence number as a proof mid-point.

3. View-changes are only necessary for liveness.

4. At least one replica has access to a TEE.

While it is possible to apply the SACBFT ordering phase transformation to a protocol even if requirement 2 does not hold, one will have to provide a separate safety proof for the transformed protocol in order to guarantee correctness.

The transformation itself can be applied as follows:

1. Ensure that all TEE-enabled replicas are first in line to become primary in the primary election process.

2. Ensure that all replicas can remotely attest the TEE-enabled replicas, and can verify signatures generated by these replicas.

3. Identify the ordering phases of the original protocol.

4. When the current primary has a TEE, replace these ordering phases with a PREPARE message, broadcast from the primary, that includes a sequence number picked by the primary's TEE and bound to the current request (as described in Section 4.2.1).
4.3.2 Applying the voting round transformation

The voting round transformation is less involved. It can be applied to any PBFT-like protocol, as long as there is a primary replica involved. The steps for this are as follows:

1. Ensure that all replicas have an asymmetric key pair belonging to a multisignature scheme.

2. Ensure that all replicas have access to the public key of each replica in this multisignature scheme.

3. Identify all voting rounds in the original protocol.

4. Replace each voting round with a voting round based on signature aggregation as described in Section 4.2.2).

4.4 Examples of transformed protocols

4.4.1 SACPBFT

When applying SACBFT to PBFT, we replace the two voting rounds in the original protocol. Firstly we replace the first voting round (PREPARE and PREPARE) with a simple broadcast from the primary replica, informing all other replicas of the request and sequence number. The second voting round occurs in the COMMIT phase. As discussed previously, this is a voting round with \( H(M) \) as a ballot, and replacing it with the SACBFT voting scheme brings down communication complexity to \( O(n) \) at the cost of an extra round’s latency.
CHAPTER 4. FRAMEWORK

Figure 4.5: An illustration of the communication patterns of PBFT and SACPBFT with one faulty replica (3).

As can be seen in figure 4.5, the total number of communication steps nonetheless remains the same. This is because we manage to remove one step in the prepare phase, and this compensates for the extra communication step required for signature aggregation.

Algorithm 5 shows minimal pseudocode for SACPBFT. The REQUEST message contains client identifier \( c \), operation \( o \) and timestamp \( t \), while in the other messages \( v \) refers to the view number and \( i \) to the replica identifier. Note that line 21 line is open to interpretation. One can decide to have the protocol wait for a reasonable amount of time, in hopes of still obtaining \( f + 1 \) valid signatures, or notify the client that not enough commit messages have been received and abort the transaction.

4.4.2 SACRePBFT

As figure 4.6 shows, applying SACBFT to RePBFT is very similar to its application to PBFT. The first voting phase is replaced with SACBFT’s prepare phase, and the second voting phase with a signature aggregation scheme. This is analog to the changes made to PBFT. The only difference is actually introduced by RePBFT; replies are not only sent to the client, but also to passive replicas.

The pseudocode for SACRePBFT is not shown as it is mostly analogous to the SACPBFT code in Algorithm 5. The one difference is that the
“broadcast to all replica” messages will change to “broadcast to all active replicas”, and in the \textbf{REPLY} phase an \textbf{UPDATE} message will also be sent to all passive replicas. Additionally, if at any time an incorrect message is received by enough replicas, SACRePBFT will initiate a protocol switch to SACPBFT, analogous to how RePBFT switches to PBFT (described in [19]).

\textbf{Figure 4.6:} An illustration of the communication patterns of RePBFT and SACRePBFT with one passive replica (3).
Algorithm 5 SACPbFT

1. upon reception of request $M = \langle \text{REQUEST}, o, t, c \rangle$ at primary replica do
2. $\langle u, H(M) \rangle_{\sigma_p} \leftarrow \text{TEE.Increment}(H(M))$
3. broadcast $\langle \text{PREPARE}, \langle u, H(M) \rangle_{\sigma_p}, v, M \rangle$ to all replicas
4. end

5. upon reception of $\langle \text{PREPARE}, \langle u, H(M) \rangle_{\sigma_p}, v, M \rangle$ at replica $i$ do
6. if $v < \text{current view}$ then
7. ignore this message
8. end if
9. if $c \leq \text{last received signed sequence number}$ then
10. Request view-change and abort transaction
11. end if
12. $\langle u \rangle_{\alpha_i} \leftarrow \text{MultiSign}(u, k_i)$
13. set a timer to wait for a reply from the primary
14. send $\langle \text{COMMIT}, \langle u \rangle_{\alpha_i}, H(M), v, i \rangle$ to primary replica
15. end

16. upon reception of $f + 1$ $\langle \text{COMMIT}, \langle u \rangle_{\alpha_i}, v, H(M) \rangle$ messages at primary replica do
17. if $v < \text{current view}$ then
18. ignore this message
19. end if
20. if $\exists i : \text{MultiVerify}(\langle u \rangle_{\alpha_i}) \neq \text{true}$ then
21. return “Not all signatures are valid”
22. end if
23. $\langle u \rangle_{\alpha_{f+1}} \leftarrow \text{Aggregate}(\langle u \rangle_{\alpha_i}, \ldots, \langle u \rangle_{\alpha_{f+1}})$
24. broadcast $\langle \text{COMMIT}, \langle u \rangle_{\alpha_{f+1}}, v, H(M) \rangle$ to all replicas
25. end

26. upon reception of $\langle \text{COMMIT}, \langle u \rangle_{\alpha_{f+1}}, H(M), v, i \rangle$ at replica $i$ do
27. if $v < \text{current view}$ then
28. ignore this message
29. end if
30. if MultiVerify($\langle u \rangle_{\alpha_{f+1}}$) $\neq$ true then
31. Request view-change and abort transaction
32. end if
33. Cancel the timer waiting for a reply from the primary
34. $r \leftarrow$ execute operation $o$
35. send $\langle \text{REPLY}, v, t, c, i, r \rangle$ to client $c$
36. end
Chapter 5

Implementation

In order to evaluate the performance of the protocol transformations proposed in this thesis, we implemented a proof-of-concept version of the SACPBFT algorithm (Section 4.4.1) that is able to execute an arbitrary number of transactions in one view. Our software was developed in a combination of the C, C++ and Go [24] programming languages.

In this chapter we describe the structural choices and implementation choices of our proof-of-concept, after which we briefly describe the configuration parameters. First we discuss the cryptographic primitives we used.

5.1 Cryptography

SACBFT relies on cryptographic functions for secure hashing, signatures, and multisignatures. In this section we elaborate on our choices for these functions in the proof-of-concept implementation of SACPBFT.

Hashing

In Chapter 4 we rely on a secure cryptographic hash function in order to hash requests. We use SHA256 of the SHA-2 hash function family. This is a well known and widely deployed cryptographic hash function, and is considered to be secure by the academic community at the time of writing.

In our implementation, we rely on the OpenSSL [41] SHA256 implementation, provided by the libcrypto [42] API.
Multisignatures

We also require a specific multisignature scheme for the SACBFT voting phase transformation described in Section 4.2.2. In principle, any multisignature scheme would suffice, although preference goes to one that has small signature sizes and minimal communication overhead. While we rely on the BLS signature scheme (see Section 2.3.1), we would like to note that recent work by Boneh et. al. introduces a multisignature scheme that could provide better performance [8]. Unlike BLS, it does not require special measures to allow for signing the same message, and the public aggregate key is sufficient to verify which replicas participated in the signature (unlike BLS, where a separate public aggregate key will have to be computed). Unfortunately, the work is mainly theoretical and no implementation exists at the time of writing.

We use the BLS aggregate signature scheme as a multisignature scheme, since SACBFT requires all replicas to sign the same message. For this, we require a zero-knowledge proof on the private keys used for multisignatures, to prevent signature forgery. For this proof-of-concept implementation, we have not implemented this, as the test environment is controlled by us, and signature forgery is not a concern. Moreover, the proof should be done when distributing the keys, and it does therefore not affect performance of the algorithm. In a real-world deployment, the zero-knowledge proof must be done before the system is used. For this, there exist ample zero-knowledge proof protocols (see e.g. [6, 11, 22]).

We use an efficient BLS implementation in Go, originally written for the Vuvuzela [51] project. The code is slightly modified, since the original implementation did not allow for aggregating signatures on the same message. To be able to interface with this implementation, we wrote C bindings for the Go BLS code, which we then used in our proof-of-concept implementation.

With the unmodified Go library, BLS operations generally take less than a millisecond (except the verification operation, which grows with the number of signatures), calling the same operations from within our C++ code, operations tend to take roughly 4 milliseconds. This is likely due to heavy context switching as well as page faults, considering the Go library spins up a garbage collector and calls many Go standard library functions (the entire Go Runtime had to be embedded in our program). Since this overhead is constant-time, we do not consider it to be a big issue as it will be negligible when the number of replicas increases.
Signatures
The trusted monotonic counter described in Section 2.2.2 relies on a signature scheme to bind counter values to messages. Since we use Intel SGX as a TEE, it is convenient to use a signature scheme that is already available in the intel SGX SDK API [28]. The SGX SDK implements and provides an interface for the NIST ECDSA standard [39], using Curve P-256.

5.2 Key generation
The keys are generated during deployment of the system. We discuss the generation of the two types of key pairs separately.

BLS key pairs
The BLS keys are generated through a Go program that generates the keys for all replicas in one run. This is done while setting up the system, and is done by one machine. While in a production setting it is much safer for each replica to generate their own key upon e.g. first execution of the program, and then communicate their public key with all other replicas, this adds complexity to the code that is unnecessary for the purpose of measuring performance. We therefore use a single program to generate all keys, and then distribute the keys from this central system.

In a production environment there should also be a protocol in place for changing keys after a replica has been compromised or has failed. This protocol should ensure that no replicas accept messages signed by the replica’s previous key and are aware of the replica’s new public key.

This program takes a parameter n to determine the number of keys it should generate, after which it writes n public keys to one file, and creates a subdirectory with n files each containing one private key (named r1 to rn).

Trusted counter key pair
This key pair is generated if the program is started, tries to take the role of a primary replica, and does not find an existing key pair on the system. It will then generate the key pair, save the public key to the harddisk, seal the private key with a key known only to the TEE, and saves the encrypted private key to the harddisk as well.

Again, in a production environment, there should be a protocol allowing each replica to communicate their TEE public key to all other replicas.
Since we do not want to add unnecessary complexity, and only use a single TEE for the entire system (since we do not test view-changes), this is sufficient for our purpose.

Unlike the BLS keys, this key pair cannot be generated on any system – the private key is encrypted with a key known only to the TEE belonging to the machine that generated the key pair. Therefore, we need to execute this on the system that will act as a primary replica for our benchmarks.

In a production setting, it should not generally be necessary to replace a TEE key pair, unless the security of the TEE itself is compromised – something we assume does not happen.

**Key generation steps**

In order to successfully generate and distribute the keys, the following steps should thus be taken:

1. Generate $n$ BLS keys on a central system.

2. Run the SACPBF program in server mode, taking on the role of primary.

3. Distribute the BLS public key file and the counter public key file to all machines.

4. Distribute the private keys to the relevant replicas and ensure their configuration file points to their private key (See Section 5.3.4).

Note that Step 2 has to be performed on the system that will also act as primary for the evaluation phase.

While this approach is not as safe as having each replica generate their own BLS key, it considerably simplifies key distribution, and the security risk is deemed acceptable in our test setup.

### 5.3 Software structure

The SACPBF code itself is mostly written in C++ using an object-oriented approach, with small parts in pure C for the interface between trusted code and untrusted code. The trusted code implements a trusted monotonic counter. It also provides several extra functions to generate an ECDSA key pair, and to save/store them securely to disk, in such a way that only the SGX enclave can read them.
Figure 5.1: High-level structure of the SACPBFT proof-of-concept implementation.

The majority of untrusted code can be divided into two modules, as Figure 5.1 shows. The first module is the network module, containing all code related to network connections and data transmission. The second module is related to transactions, containing a manager to order them and manage their execution, and the individual transaction types themselves. Both modules rely on a cryptography module for cryptographic functions.

Our program is able to function in server (i.e. replica) mode, or client mode. Both modes rely on the same network module for communication, but the server mode also requires the transaction module. Both modes also rely on a configuration parser module that parses configuration files and translates configuration values to their correct types.

This mode implements all replica behavior, and additionally relies on a trusted counter module that provides all trusted counter functionality.

The client mode allows the user to specify the request size as well as the number of requests to perform in succession. When the number of re-
quests is specified, the client will wait until it has received enough replies before sending the next request.

5.3.1 Trusted computing base

The trusted monotonic counter runs completely inside an SGX enclave. This counter exposes a function $\text{getCounter}(X)$, which corresponds to the $\text{Increment}$ function described in Section 2.2.2. In order to ensure atomacity of this function, the function obtains a mutex before it executes.

The pseudocode of the $\text{Increment}$ function requires $c$ and $s_i, p_i$ to be persistent between function calls. To achieve this, we simply define them to be global variables within the enclave. We keep the same key-pair throughout instantiations of the enclave, meaning this is not a secure implementation. It makes deployment on our test setup considerably easier however, and since key generation and distribution is not performance-critical, we consider this to be a permissible omission for our proof-of-concept.

Remote attestation

Remote attestation is needed to bind the key pair belonging to the trusted counter to an actual enclave. We briefly describe how to perform remote attestation using Intel SGX in a safe way, ensuring that a malicious actor will not be able to forge signatures that should come from the counter.

Intel SGX allows enclaves to produce "quotes" [27, page 15], that contain a number of data fields allowing users to verify the legitimacy and authenticity of the enclave. Relevant for our purpose are the MRENCLAVE field, which uniquely identifies the code running on the enclave, the MRSIGNER field, which uniquely identifies the developer key with which the enclave was signed, and a report field which will be used to include a hash of the counter’s public key $p_i$.

First, the developer needs to obtain a certified asymmetric key pair from Intel, which they use to sign the enclave itself when compiling the program for production usage.

During the development process, the developer will have to generate a quote from an enclave running the final (production-ready) enclave code, to obtain the correct MRENCLAVE value. This value then has to be made public so that integrity of the enclave code can later be verified.

During the deployment phase, the following steps need to be taken:

1. All TEE-enabled machines must generate an asymmetric key pair.
2. All TEE-enabled machines must produce a quote of their trusted counter enclave, including (a hash of) their public key generated in the previous step in the report data, to bind the public key to this enclave.

3. The quotes have to be verified. In particular, for each quote, the MR-SIGNER value has to correspond to the correct developer key, the MRENCLAVE value has to correspond to the MRENCLAVE value made public by the developers, and the (hash of the) public key included in the quote has to match the (hash of the) public key that is claimed to belong to this TEE.

4. If all verifications passed successfully, the public keys can safely be distributed to all replicas.

Note that the verification in step 3 should be done in a trusted environment. Alternatively, the quotes can also be distributed to all replicas and verified separately by their managers, as long as it is done before actual execution of the protocol.

5.3.2 Communication

The SACPFT prototype uses TCP connections to communicate between replicas. Each replica sets up a connection with the current primary during the startup phase. This way, TCP handshakes do not have to be performed for every transaction, reducing latency as a result.


Message format

We use Google Protocol Buffers [23] as a serialization format for the different messages sent by replicas. Protobuf is a popular data interchange format, allowing for efficient serialization of messages. Our prototype uses the following message types:

- **SIGNATURE**: This is simply a data structure to hold signatures generated by a trusted counter as well as BLS signatures. This message type is used as part of PREPARE and COMMIT messages.

- **REGISTER**: Registers a client to all replicas. This allows replicas to map the client ID to an actual host, and opens a TCP session with this client (used for sending REPLY messages).
• REQUEST: Contains the request message sent by a client to the primary replica.

• PREPARE: The PREPARE message is sent by the primary to all replicas. It contains the corresponding REQUEST sent by the client, together with a SIGNATURE message containing the counter value bound to this request.

• COMMIT: This message represents both communication steps in the commit phase of SACPBFT. Replicas send a COMMIT message to the primary, including two signatures: the signature they received from the primary in a PREPARE message, and a BLS signature representing the replica’s vote. Once the primary has obtained enough of these it will broadcast another COMMIT message, with an aggregated BLS signature in its body.

• REPLY: This contains the request operation result and is sent to the client by all replicas.

5.3.3 Transactions

Whenever a replica receives a message, it will hand this over to the transaction module. This module maintains a queue of currently open transactions, and ensures that incoming messages arrives at the correct transactions.

Each transaction runs in a separate thread, which gets destroyed once the transaction finishes. This ensures that the network thread does not get blocked for long periods of time when processing messages within a transaction.

Since the goal of this proof-of-concept implementation is to enable benchmarking of the performance of the algorithm, our transactions support just one operation: the NOOP “no-operation” operation. This operation does not actually do anything, but can take an arbitrary amount of data as operation parameters, to allow for testing the performance of different message sizes.

5.3.4 Configuration

The proof-of-concept program requires a configuration file containing necessary information for correct functioning of the system. This allows us to change the configuration during evaluation of the framework. Each
replica has its own unique version of the configuration file, as it also contains file paths to their BLS key pairs. Other important parameters in the configuration file are:

- The amount of replicas in the system.
- The ID of the primary replica.
- A list mapping all replica IDs to IP addresses with TCP ports.
- A list mapping all clients IDs to IP addresses with TCP ports.
Chapter 6

Evaluation

In order to evaluate the effectiveness of our proposed framework, we make an estimation of the data transmitted per transaction, which we use to compare the network usage of PBFT and SACPBFT. We also created an experimental setup that simulates a BFT system running SACPBFT in the optimal case.

6.1 Theoretical evaluation

We compare the quantity of transmitted data of PBFT and SACPBFT in the optimal case where there are no faulty replicas.

To make a realistic estimation of the size of the transmitted data per transaction, we need to estimate the sizes of all elements in each message. For each type of element, we provide a realistic size estimation, which we use to plot the growth rates of transmitted data as a function of the amount of replicas in the system.

We define the size estimations for the message elements as follows:

<table>
<thead>
<tr>
<th>Element(s)</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>1</td>
<td>message tag (e.g. REQUEST or PREPARE)</td>
</tr>
<tr>
<td>o</td>
<td>0</td>
<td>request operation (size varies)</td>
</tr>
<tr>
<td>t</td>
<td>8</td>
<td>timestamp (UNIX Epoch time)</td>
</tr>
<tr>
<td>c, i</td>
<td>2</td>
<td>ID type (client ID, replica ID)</td>
</tr>
<tr>
<td>(v, n, u)</td>
<td>4</td>
<td>numerical type (view nr, seq. nr, counter value)</td>
</tr>
<tr>
<td>(h(x))</td>
<td>32</td>
<td>hash (SHA256)</td>
</tr>
<tr>
<td>r</td>
<td>0</td>
<td>operation result (varies)</td>
</tr>
<tr>
<td>(\sigma_x)</td>
<td>64</td>
<td>signature (ECDSA)</td>
</tr>
<tr>
<td>(\sigma^*_x)</td>
<td>32</td>
<td>multisignature (compressed BLS)</td>
</tr>
</tbody>
</table>
Some values warrant some extra explanation: we define the size of the message tag to be $S(e) = 1$ byte, which corresponds to the size of the message tag in our implementation. However, this is not a lower bound – there are only 4-5 different message types in this protocol, and is possible for an implementation to work without the tags. For the growth rate comparison it does not matter much; each message in both protocols contains the same number of message tags.

ID fields are 2 bytes. This is because we evaluate the transmitted data size for up to 300 replicas, meaning that we need 9 bits to uniquely represent all replica IDs. Since in most architectures bytes are the smallest allocatable unit, we use 2 bytes instead. Other realistic choices would have been standard sizes for integers (4 bytes or 8 bytes).

We specify the request operation and operation result size to be 0. This is because the possible operations depend on the implementation and scenario. Moreover, in both PBFT and SACPBF these are only sent once by each replica, meaning that they would add the same amount of transmitted data to each transaction for both protocols. We can therefore safely ignore them when inspecting the growth rate of the transmitted data.

With the message elements defined, we then define a size function $S$ for messages in PBFT and SACPBF as follows:

<table>
<thead>
<tr>
<th>PBFT</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>Size</td>
</tr>
<tr>
<td>$a_1 = \langle \text{REQUEST}, o, t, c \rangle$</td>
<td>$S(a_1) = 11$</td>
</tr>
<tr>
<td>$a_2 = \langle \text{PREPREPARE}, v, n, h(m), a_1 \rangle$</td>
<td>$S(a_2) = 52$</td>
</tr>
<tr>
<td>$a_3 = \langle \text{PREPARE}, v, n, h(m), i \rangle$</td>
<td>$S(a_3) = 43$</td>
</tr>
<tr>
<td>$a_4 = \langle \text{COMMIT}, v, n, h(m), i \rangle$</td>
<td>$S(a_4) = 43$</td>
</tr>
<tr>
<td>$a_5 = \langle \text{REPLY}, v, t, c, i, r \rangle$</td>
<td>$S(a_5) = 19$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SACPBF</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1 = \langle \text{REQUEST}, o, t, c \rangle$</td>
<td>$S(b_1) = 11$</td>
</tr>
<tr>
<td>$b_2 = \langle \text{PREPARE}, \langle u, h(m) \rangle_{\sigma_C}, v, b_1 \rangle$</td>
<td>$S(b_2) = 112$</td>
</tr>
<tr>
<td>$b_3 = \langle \text{COMMIT}, \langle u \rangle_{\sigma}, h(m), v, i \rangle$</td>
<td>$S(b_3) = 75$</td>
</tr>
<tr>
<td>$b_4 = \langle \text{COMMIT}, \langle u \rangle_{\sigma_1 \cdots n}, h(m), v \rangle$</td>
<td>$S(b_4) = 73$</td>
</tr>
<tr>
<td>$b_5 = \langle \text{REPLY}, v, t, c, i, r \rangle$</td>
<td>$S(b_5) = 19$</td>
</tr>
</tbody>
</table>

From these equations, it might look like SACPBF requires more network traffic at first glance, since the message sizes are larger than those of PBFT. While this might be true for individual messages, PBFT transmits considerably more messages per transaction. We construct functions $PBFT(N)$ and $SACPBF(N)$ that capture the communication required for
one transaction, where \( N \) is the number of replicas.

\[
\text{PBFT}(N) = S(a_1) + (N - 1)S(a_2) + (N^2 - N)(S(a_3) + S(a_4)) + NS(a_5)
\]

\[
\text{SACPBFT}(N) = S(b_1) + (N - 1)(S(b_2) + S(b_3) + S(b_4)) + NS(b_5)
\]

Figure 6.1 plots these functions for increasing \( N \). SACPBFT transmits much less data per transaction, especially when the number of replicas increases, indicating that it is more suitable for low-bandwidth environments than PBFT.

Note that the functions presented in this section are merely an estimation and do not provide an exact lower bound. In a real setting other things need to be taken into account as well, for example the overhead of the varying underlying network protocols, as well as possible overhead from authenticated encryption protocols used to set up secure channels between machines.
6.2 Experimental evaluation

We assess the performance impact of our improvements by simulating a BFT system with one client and $n$ replicas, and measuring the average time it takes for one transaction to complete.

To assess the performance impact of our improvements we deployed proof-of-concept implementations of PBFT and SACPBFT to a test cluster. As SACPBFT is a modification of PBFT, it makes sense to compare these.

Test setup

![Network layout diagram](image)

**Figure 6.2:** The network layout of the cluster used in our experimental setup.

Our test setup consists of five SGX-enabled machines, operating in an isolated switched network (See Figure 6.2). Each system runs on an Intel Core i5-6500 processor, featuring 4 cores and no hyperthreading. The first system (SGX-1) has 16GB RAM, and all others have 8GB RAM.

We distribute the number of replicas in our system evenly over the available machines (e.g. when the system contains 30 replicas, each machine will run 6 of these). In addition to that, the first machine also runs the client. Each replica runs in its own process.

Testing

For each protocol, we measure the time it takes for a transaction to complete for increasing values of $n$. For each sample point, we execute ten requests, each carrying a 1024B payload, and calculate the average time per transaction based on the obtained data.

The bandwidth of each machine is limited to 8Mbit (up- and downlink separately), in order to more accurately simulate a constrained network
environment (e.g. a number of embedded devices communicating with each other using light-weight RF protocols).

For the SACPBFT implementation, only the process managing the primary replica has access to an SGX enclave.

Results

![Graph showing latencies for a 1024B payload](image)

**Figure 6.3**: Transaction latency vs. number of replicas.

Figure 6.3 depicts the average transaction latency for a 1024B payload for increasing values of $n$. As can be seen, PBFT’s performance drops fast compared to the other two algorithms when the number of replicas increases. This is due to its all-to-all broadcasts. When $f \approx 30$, transactions take around 550 milliseconds to complete.

SACPBFT achieves considerably better performance. With the same amount of replicas, transactions take only around 150 milliseconds to complete. When the number of replicas is small however, PBFT outperforms SACPBFT. This due to the overhead introduced by calls to the LBS
signature library, as explained in Section 5.1.0.2. There is room for efficiency improvement in the SACPBF implementation however, so it is possible to reduce this gap.
Chapter 7

Discussion

The evaluation done in Chapter 6 allows us to compare the transaction latency of various protocols to that of SACPBFT. However, in Chapter 3 we defined more goals than just minimizing transaction latency. In this section we look at the extent to which each of these goals is satisfied. We also propose alternatives and point out areas of improvement that might further increase performance. Finally, we discuss possibilities for future research.

7.1 Fault tolerance

Requirement F-1 states that the framework should maintain safety and liveness properties of the underlying protocols. As described in Section 4.2.5, our framework does not break liveness arguments in existing protocols. Safety arguments will also still hold as long as the underlying protocol uses non-equivocation as a mid-point in the proof. All protocols investigated in the scope of this thesis do so, indicating that this is an acceptable limitation. This allows the framework to act as a drop-in replacement, in that no extra effort is needed to ensure the correctness of the transformed protocol. Moreover, if an existing protocol does not use non-equivocation as a proof mid-point, this does not necessarily mean that the safety property is violated, but rather that manual work is needed to verify that it still holds. Overall, the goal is thus satisfied, although manual verification might be necessary in rare cases.

The second goal, F-2 is to maximize the number of tolerated faults. The SACBFT framework requires $3f + 1$ replicas, like all other solutions that do not rely on trusted hardware. The reason we have not been able to improve this so we can tolerate more faults, is not because the trans-
actions within a view require it (a single transaction can function with \(2f + 1\) replicas if both the \texttt{PREPARE} and voting round transformations are applied), but rather that view-changes are not modified by the framework and still require \(3f + 1\) replicas. If a system has \(2f + 1\) TEEs, it is possible to bring down the replica requirement to \(2f + 1\) for the entire system, as then a view-change sub-protocol where \(2f + 1\) TEEs participate can be used, as in e.g. minBFT [50]. Whether view-changes can be improved upon in other cases remains an open question for now.

We also stated that we want to minimize the number of TEEs necessary (F-3), with the intent to increase performance with minimum investment. Our framework is able to increase the efficiency of transactions with as little as 1 TEE. We can thus consider this goal to be satisfied. The caveat is that with only a single TEE, the performance benefits exist only for as long as the TEE-enabled system does not fail. This is a limitation that cannot be circumvented, but increasing the number of TEEs decreases the likelihood of this ever happening. Once the system has \(f + 1\) or more TEEs, the performance benefits are permanent. This is still an improvement over the \(2f + 1\) requirement of other protocols that rely on TEEs.

The final fault tolerance goal states that the trusted computing base should be as small as possible (F-4). The SACBFT framework relies solely on a trusted monotonic counter, the pseudocode of which is listed in Algorithm 1. The length of this algorithm suggests that it is possible to implement the counter in only a few lines of code. However, the complexity of the signature algorithm has to be taken into account. If the \texttt{Sign} operation requires complex code then this goes against goal F-4. On the other hand, many popular signature schemes have battle-tested implementations that have been reviewed and improved by many contributors.

With SGX, the standard libraries provide an interface to perform signature operations, which we use to keep the codebase small. In our proof-of-concept implementation, the codebase for the trusted counter is slightly over 100 lines of code, including auxiliary functions to initialize the cryptographic context, generate key pairs and seal/unseal them. This is still relatively small and it is manageable to formally verify this code. All in all, the trusted computing base has the potential to be very small and its functionality is well defined, meaning that it can be thoroughly tested for correctness.
7.2 Performance

One of the main goals of SACBFT is to increase performance of existing protocols. An important factor in this is goal P-1: minimizing communication complexity. The painful spots in existing protocols are traditionally the voting rounds, where each replica needs to inform all other replicas of its vote. The $O(n^2)$ communication complexity this brings, has been reduced to $O(n)$ with multisignature techniques in earlier work. The SACBFT transformations allow one to replace classical voting rounds with multisignature voting rounds as well, thereby reducing the communication complexity. This $O(n)$ complexity is on par with the current state of the art in BFT algorithms, and we can thus say that the communication complexity is minimal.

The second performance goal (P-2) aims to minimize the number of voting rounds. While Zyzyzyva can be considered to have the minimum number of voting rounds in the optimal case (i.e. none), it operates in a non-standard adversary model and requires rollbacks. We therefore consider this unrealistic and look at the minimum round number in the traditional model instead. In the traditional model, TEE-based algorithms require the smallest number of voting rounds: these algorithms can omit one voting round by relying on trusted components, and we based the SACBFT ordering process transformation on this concept. This transformation removes the ordering phase voting round and as a result SACBFT is able to bring down the number of voting rounds to the minimum seen existing algorithms (with Zyzyzyva as an exception).

The final performance goal is to minimize transaction latency (P-3). There are 3 approaches to this: firstly, one can minimize the number of communication steps for a transaction. For most protocols, the amount of communication steps stays the same after the SACBFT transformations, when a multisignature algorithm is used that requires a single round trip. Some schemes require two round trips for one voting round (e.g. Schnorr multisignatures as used in CoSi [47]). If such a scheme is used with SACBFT then the amount of communication steps can increase.

The second way in which one can attempt to minimize latency is by reducing communication complexity. This saves bandwidth and will reduce latency especially when the number of replicas increases. This is defined as a separate performance goal (P-1) and has already been discussed. The all-to-all broadcast voting model is more efficient for very small groups of replicas, so one possible improvement here is to modify the algorithm to use the all-to-all broadcast for all systems with fewer than $x$ replicas, and use the aggregation method otherwise.
The third way is to restructure the network topology of the system to optimize bandwidth usage and allow for faster propagation of messages. Some protocols such as FastBFT [37] and Byzcoin [34] experiment with different topologies such as a tree topology with the primary as a root. This allows messages to propagate through the system in $O(\log n)$ time. In this thesis we do not explore different topologies, since it adds complexity to the fault tolerance problem (e.g. in a binary tree, the protocol needs to ensure that all replicas are reachable even when one of the internal nodes fails), but this is certainly a subject worth exploring in future work.

As the results in Chapter 6 show, SACBFT is capable of significantly reducing transaction latency, although it only focuses on one out of three ways to minimize latency. To really minimize latency, the implementation plays a major role as well. In our proof-of-concept implementation of SACPBFT, we can identify several areas where we could increase performance and reduce latency. One such example is the way the multisignature scheme is implemented. While the BLS implementation used in the proof-of-concept is optimized by itself, the code is written in Go and accessing this from C/C++ requires the entire Go runtime to be included in the binary (which includes a garbage collector), as well as several adapter interfaces that make memory allocations that could otherwise be avoided. Goal P-3 can therefore be seen as partially satisfied; latency is reduced but there is room for further optimization.

### 7.3 Future work

The work presented in this thesis has introduced a number of questions we have not yet been able to answer, and demand further research. In addition to that, there are some optimization techniques we have not explored yet.

Firstly, it remains an open question whether is possible to reduce the number of participating replicas in view changes when fewer than $2f + 1$ replicas have a TEE. Should this turn out to be the case, it will impact goal $F - 1$ (we need to ensure that liveness still holds), as well as F-2 since the number of tolerated faults will go up to possibly even $\lceil \frac{n-1}{2} \rceil$. It is also possible that it will negatively affect F-4, since it might require more from the TEE than just a trusted counter.

While right now the added value of SACBFT is primarily as a bridge from no TEEs to a full TEE-enabled system, increasing the number of tolerated faults would increase the potential for SACBFT to actually replace
existing TEE-based approaches in some situations, since it would be able
to provide the same benefits with a smaller investment.

A second point that warrants more research is the impact of different
network topologies on the performance of SACBFT, something we already
briefly touched upon in Section 7.2. While the topology is not something
that can be chosen for every system, it is worthwhile to know what the
impact is of SACBFT for a given topology. This can then lead to more in-
formed choices on whether it pays off to apply SACBFT to some situation.

The performance impact of choosing different multisignature schemes
is another aspect that we want to look at in future work. In this thesis, we
use the BLS scheme, which requires only one round-trip for a vote, but is
computationally more expensive than some other algorithms that require
more round trips. It is interesting to see in which cases the added latency
of an extra round trip is preferable to the added latency of computation.
Other aggregation techniques can be investigated as well.

As mentioned in Chapter 6, PBFT outperforms SACPBFT when the
number of replicas is very small. This points to the possibility of increas-
ing performance in these cases by using the all-to-all broadcast voting
method instead. The point where it becomes more efficient to use the ag-
gregation method is very likely dependent on network properties, and
defining a method for determining this threshold given a set of network
properties is an interesting topic for future work.

The SACPBFT proof-of-concept implementation itself also has room
for improvement. For example, it might be made faster by relying on a
pure C/C++ implementation of the BLS scheme instead of the current Go
implementation. the proof-of-concept also currently creates a new thread
for each transaction. While threads for completed transactions are termi-
nated, the overhead of managing threads is not necessary, as a thread
pool could be used to manage transactions, or it can be avoided by using
strands instead of threads.
Chapter 8
Related Work

Since Lamport’s formalization of the Byzantine general’s problem [36], there has been much research on Byzantine agreement problems. Before the introduction of PBFT [12], many solutions tended to rely on a synchronous model (e.g. [21]), or attempted to circumvent the FLP impossibility [20] by relying on randomization [14, 5, 10], atomic broadcasts [45] (a mechanism allowing processors to reliably broadcast messages), or unreliable fault detectors [13, 43] (modules local to each machine that attempt to establish which of the other machines might have crashed).

PBFT was the first protocol to use the weak synchrony assumption, which for the first time put Byzantine consensus within reach of practical applications. Since its introduction, there have been many protocols building upon PBFT to improve its efficiency, both by reducing transaction latency and by lowering resource requirements.

One such example is ByzCoin [34], which uses multisignatures as a means to achieve message aggregation for its voting rounds, bringing down communication complexity. Multisignatures are not the only way to achieve message aggregation however. For example, FastBFT [37] uses shared secrets to aggregate messages, requiring a TEE but only relying on the XOR operation for shared secrets, resulting in extremely low computation overhead. SACBFT builds upon the multisignature message aggregation introduced by ByzCoin, to achieve similar reductions in communication complexity.

In an attempt to reduce the amount of resources used, Yin et. al.[52] have shown that by separating agreement from execution, there only need to be $2f + 1$ executing processes (although it still requires $3f + 1$ agreement processes). TEE-enabled protocols improve upon this by lowering the number of processes to $2f + 1$ altogether [18, 50, 16, 37, 32, 44]. Two of the earliest TEE-based protocols [16, 18], depend on trusted hardware to
implement attested append-only memory and atomic multicasts respectively. Veronese et al. [50] remarked that simplicity is a key concern for trusted components, a requirement not satisfied by those algorithms. This moved them to specify the USIG trusted counter service, forming the core of MinBFT. Going even further, CheapBFT [32] combines the efforts of MinBFT and ReBFT [19] to lower the lower the number of replicas participating in agreement to $f + 1$ when there are no faults, and only involving the remaining $f$ replicas when faults are detected. FastBFT draws inspiration from ByzCoin and CheapBFT to reach $O(n)$ communication complexity with $f + 1$ active replicas.

When we consider the complexity of TEE-enabled BFT protocols in more detail, we see that this varies a lot. We based the trusted counter used in SACBFT on MinBFT's USIG service. This is one of the simplest forms of a TEE. On the other end of the spectrum we find protocols such as [44] that rely on trusted hypervisors and virtual machines to achieve similar benefits. The complexity of TEEs used by other protocols lie somewhere between a trusted counter and a complete VM infrastructure.

ReBFT is like SACBFT in that it is a framework that can be applied to other protocols. The authors demonstrate this by applying it to PBFT and MinBFT. SACBFT and ReBFT can compliment each other to produce a protocol that has the benefits of both frameworks, as we showed with SACRePBFT in Section 4.4.2.

In the Wormhole model [49], existing TEE-enabled BFT protocols can be described with the TEEs being wormholes. For all of these systems, $N_w = N_p$, where $N_w$ is the number of wormholes and $N_p$ is the number of processes. The Wormhole model allows for the case where $N_w < N_p$, and this exactly what SACBFT explores. By requiring fewer wormholes, it brings the hybrid fault model closer to classical BFT algorithms.

There has been some research investigating the lower bound on required replicas when the system contains partial reliable broadcast channels [31]. This work can provide a basis for determining the lower bound for systems where a subset of replicas has a TEE.

With the introduction of Bitcoin [38], there have been many efforts to improve upon its proof-of-work Nakamoto consensus algorithm. Algorithms targeting open blockchains differ from the traditional BFT algorithms in that they seek to achieve consensus in a permissionless setting; participants are not known before the protocol starts, and there is no pre-existing trust (as required for traditional algorithms during system setup). Research to alternatives includes proof-of-stake protocols [33, 7], proof-of-space [3, 1], and proof-ofelapsed-time [30].
Chapter 9

Conclusion

In this work we introduced SACBFT, a framework that provides transformations for existing BFT protocols to utilize available trusted hardware and increase performance. It is the result of investigating the previously unexplored assumption that a subset of all machines in a BFT system has access to trusted hardware. Through our framework, we enable existing protocols to tap into the potential of these trusted components. Additionally, SACBFT uses modern techniques to reduce the communication complexity.

The framework consists of two transformations. The first transformation requires at least one replica to have access to a trusted counter, but eliminates the need for a voting round during the \textsc{Prepare} phase of existing protocols. The second transformation implements message aggregation through multisignatures to reduce the communication complexity of all remaining voting rounds to $O(n)$. Together, these transformations can significantly increase the performance of a classical protocol, as shown by the results produced by our proof-of-concept implementation on our cluster setup.

We also discussed when a transformed protocol might need to fall back to the original protocol, and showed that with $f+1$ TEEs, this is no longer necessary.

While SACBFT is already capable of providing a performance boost to existing protocols, there remain some open questions which demand future research, as described in Section 7.3. Notably, we have not yet established whether it is possible to reduce the number of replicas required to successfully complete a view change when the system contains fewer than $2f+1$ TEEs. This could lead to a major improvement in the resilience of SACBFT.
Bibliography


[40] K Ohta and Takuya Okamoto. “Multisignature schemes secure against active insider attacks”. In: 82 (January 1999).


