b
UNIVERSITÄT
BERN

Evaluating the B3-Condition in Asymmetric Quorum Systems

Bachelor Thesis

Sabine Brunner

from
Unterkulm AG, Switzerland

Faculty of Science, University of Bern

March 1st, 2020

Prof. Christian Cachin
Orestis Alpos
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Asymmetric Byzantine quorum systems are a novel idea to define trust assumptions in blockchain protocols. They allow each process of a system to define their own subjective trust specification, instead of forcing everyone to use a system-wide trust assumption. For the system to work properly with those subjective assumptions, the B^3-condition needs to hold. Since all processes are completely free in formulating their assumptions, a way of verifying this condition is needed. This thesis introduces the design and implementation of two algorithms which verify the B^3-condition for small example systems in order to facilitate research into asymmetric Byzantine quorum systems.

Acknowledgments

First and foremost, I would like to thank my supervisor Orestis Alpos for his support, countless brainstorming sessions and his contributions to the source code. I wish to thank Prof. Christian Cachin for providing his expertise and first-hand insight into the topic and for interesting discussions. Last but not least, I want to thank my friends, family and coworkers for their patience and emotional support and especially Andrea Engel, for proofreading this thesis.

Contents

1 Introduction **1**

2 Background **3**
2.1 Trust in Blockchain Systems . . . 3
2.2 Byzantine Fault Tolerance and Quorum Systems . . . 3
2.3 Subjective Trust in Blockchain . . . 3

3 Preliminaries and System Model **5**
3.1 Symmetric Byzantine Quorum Systems . . . 5
3.2 Asymmetric Byzantine Quorum Systems . . . 6
3.3 Examples . . . 7

4 Design **9**
4.1 Expression of Asymmetric Trust Assumptions . . . 9
4.2 Verification of the B3-condition . . . 10

5 Implementation **13**
5.1 Technology . . . 13
5.2 Config File and Data Structure . . . 13
5.3 System Parser . . . 14
5.4 Implementation of an Optimized Brute-Force Algorithm . . . 15
5.5 Implementation of a Bounded Search-Tree Algorithm . . . 15

6 Evaluation **17**
6.1 Running-Time Analysis . . . 17
6.2 Datasets . . . 18
6.3 Comparison of the two Algorithms . . . 18

7 Conclusion **22**
7.1 Discussion . . . 22
7.2 Future Work . . . 22

A Command Line Tool Manual **23**
A.1 Installation . . . 23
A.2 Input File . . . 23
A.3 Usage of B3checker . . . 23
A.4 Optional Arguments . . . 24

B Performance Tests of PyRoaringBitMap vs. Python Set Data Structure **25**

1 Introduction

Due to the emergence of decentralized digital currencies, blockchain technology has gained popularity in recent years. It is becoming more important to find efficient and quantifiable algorithms for these applications and more and more research is being done in finding consensus mechanisms which ensure consistency in these public transactions. Some of the most popular cryptocurrencies like Bitcoin and Ethereum use proof-of-work to reach consensus, which relies heavily on physical computing power and therefore struggles to maintain safety and low transaction times as they scale.

More recent research has looked into applications of Byzantine quorum systems to Blockain technology. This concept has been used for years in distributed computing but has yet to be fully adapted to be used in a blockchain environment. Byzantine quorum systems define one or several lists of trusted network members, so-called quorums, to reach systemwide consensus. The most general way to define trust is accepting a threshold of failure for each execution, which is usually smaller than one third of the whole universe. However, such a definition of failure acceptance is symmetric, which implies that all participants of the system trust other network members in the same way. Since trust is inherently subjective, especially in permissionless systems where anyone may join and some members might not even be aware of each other, any member may have their own ideas about who they trust and more importantly do not trust. This leads to a new implementation of Byzantine Quorum Systems with asymmetric trust, where every member of the system defines their own subjective quorums. The intersection between all of these quorums then guarantees the safety of the system as a whole. Blockchain systems such as Ripple (XRP) and Stellar (federated Byzantine quorum systems FBQS) have already implemented protocols which model a kind of subjective trust. Cachin and Tackmann [3] introduce a new model called asymmetric Byzantine quorum system (ABQS), which, as opposed to both XRP and FBQS, strictly generalizes the classic notion of Byzantine quorum systems.

This model of asymmetric trust guarantees the safety of the system in compliance with the B^3-condition. This condition is a generalization of the classic $n > 3f$ condition of Byzantine Fault Tolerance, where f is the maximal number of faulty nodes tolerated in a universe of n processes. It says that no two quorums of any two processes may overlap only in processes whose faultiness is tolerated by both processes. Checking this condition is crucial for the protocols based on ABQS to verify the consistency and availability of transaction results. Since asymmetric fail-prone systems are a rather novel approach to trust in blockchains and have not been used in practice before, no way of computationally verifying this condition has been implemented yet. Doing this is the very aim of this project. We implement an algorithm that verifies the B^3-condition for sample systems and provide an intuitive way to build new systems with completely subjective fail-prone systems for each participant.

To verify this condition for a system, all sets of all fail-prone systems need to be compared to each other, which, due to the large number of combinations, proves to be very time-consuming. In order to

be able to test and grasp the extent and possibilities of asymmetric Byzantine quorum systems we need a way to verify the B^3-condition for test systems of different sizes and complexities. This thesis looks into the practical implementation of a B^3-verifier. Since we are working with fully enumerated sets, we cannot achieve a non-exponential runtime, and instead are looking to optimize and adapt exponential-time algorithms to this specific use case.

The two main contributions of this thesis are:

- A practical definition of asymmetric Byzantine quorum systems.
- Two algorithms to evaluate the B^3-condition and their running-time evaluation.

In Chapter 2 we talk about ways Byzantine quorum systems have been studied so far, give an insight into asymmetric trust and look at existing protocols which already use a kind of subjective trust protocol. We then describe asymmetric Byzantine quorum systems as formalized by Cachin and Tackmann [3] and give some examples in Chapter 3. Following this, Chapter 4 looks at the design of our program, from the input to the B^3-verifier and its evaluation of the system. Chapter 5 then gives a technical insight into the implementation of the program, especially the two algorithms we designed and implemented to solve the B^3-verification. Finally, in Chapter 6 we evaluate and compare the runtime of both algorithms with generated and manually defined example data.

2 Background

2.1 Trust in Blockchain Systems

In decentralized systems such as permissionless blockchains, where any party might join without verification or specific affiliations, trust becomes an important factor to guarantee both liveness and safety of a system. Many blockchain protocols rely on trustless verification such as proof-of-work for Bitcoin. However, in addition to the high energy consumption induced by the mining work needed for this kind of method, research has shown that proof-of-work protocols are also susceptible to 51% attacks, meaning that if one party contracts large enough computing power, safety can no longer be guaranteed for that system.

2.2 Byzantine Fault Tolerance and Quorum Systems

To forgo protocols like proof-of-work, there has been research into alternative architectures for permissionless blockchain protocols based on Byzantine Fault Tolerance. Threshold trust is the most general way of ensuring safety of a distributed system, where usually $n > 3f$ processes are needed to tolerate f faulty processes, as proven through the Byzantine Generals' Problem [16]. However, threshold trust is very generic in its definition: All participants trust equally and all participants are trusted equally. Especially the latter shows that this kind of trust is not smart to use in permissionless blockchains, since some participants may be very trustworthy and others not at all. To divide trust equally in this situation may be fatal for the system.

Malkhi and Reiter [13] introduce Byzantine quorum systems, where a system of processes may define several lists of processes whose agreement may suffice for the whole system to reach consensus. These lists are called *quorums*. They generalize the notion of thresholds in that some nodes may be trusted less than others, but all members of the systems still trust in the same way, meaning all quorums are identical for all processes. However, trust is inherently subjective and different members of the system might trust different processes. In a system with open membership, some processes might trust members which they know or are geographically close more than others, or they might not even be aware of all participants. In this case the traditional symmetric trust that Byzantine quorum systems offer may not be enough.

2.3 Subjective Trust in Blockchain

The blockchain networks Ripple and Stellar have both adapted Byzantine quorum systems to allow their members subjective trust assumptions. In Ripple (`https://ripple.com`) each member defines a

unique node list (UNL) of processes it trusts and whose messages it will listen to in any given procedure. System-wide consensus is reached by requesting all UNLs to overlap in a certain amount of processes. This notion of subjective trust allows for open membership of the system, similar to proof-of-work blockchains. The quorum overlap was originally set at $> 20\%$ [15], however, a close analysis of the ledger by Chase and MacBrough [4] shows that the overlap needs to be at least 90%. Additionally, even with an overlap of 99%, there can be situations where the system gets stuck without manual intervention, meaning the liveness property of the system cannot be guaranteed. MacBrough has, along with this analysis of the Ripple ledger, introduced Cobalt [12], which promises to reduce the needed UNL overlap to $> 60\%$ and guarantee liveness for this amount of overlap.

The Stellar consensus protocol (`https://stellar.org`) [14] was originally forked from Ripple's protocol and introduces federated Byzantine quorum systems (FBQS), generalizing Ripple's trust definitions. Instead of one list, every node may define a collection of lists of trusted processes called quorum slices. Each one of these quorum slices suffices to convince said node of agreement. A quorum in Stellar is defined as a set of nodes which contains one quorum slice for each of its members. The whole quorum will then be able to reach agreement together. As with the UNLs in the Ripple protocol, to guarantee system-wide consensus, these quorums need to overlap in at least one process. As proven by Lachowski [10], verifying this overlap for all quorums in a system is NP-complete. In recent years the decentralization of Stellar has been questioned and Kim et al. [9] showed that if only two of the validator nodes are removed from the system, it would stop working, as most participating nodes are highly dependent on the same few specific nodes.

Both Ripple and Stellar implement subjective trust in their protocols, however, neither of them strictly generalizes Byzantine quorum systems. The FBQS does not reduce to the known symmetric Byzantine quorum systems used in distributed computing, if symmetric trust choices are made for every participating node. Cachin and Tackmann [3], subsequently, introduce asymmetric Byzantine quorum systems (ABQS), which model subjective trust that strictly generalizes classic symmetric Byzantine quorum systems according to Malkhi and Reiter [13]. They base their model on previous work done by Damgård et al. [7], who formalized subjective quorum systems that satisfy a system-wide consistency condition and a local availability condition. Cachin and Tackmann [3] also prove the B^3-condition must be satisfied in order for an ABQS to exist.

3
Preliminaries and System Model

A quorum system for a universe of processes is a collection of subsets of all the processes, called quorums, which intersect in at least one node. Each quorum can then act on behalf of the universe, improving availability and efficiency for the whole universe. Consensus between all quorums can be guaranteed through the intersection property. In the BQS-protocol the universe defines a collection of subsets of possibly faulty processes, also known as Byzantine processes, which make up a system of so-called fail-prone sets. All members of a fail-prone set may fail simultaneously in any execution of the protocol. Malkhi and Reiter [13] adapt classic quorum systems to systems which need to tolerate Byzantine faults. Cachin and Tackmann [3] then use their definition of Byzantine dissemination quorum systems as a basis for their model of asymmetric trust. In this chapter we describe both the symmetric Byzantine quorum systems as well as the asymmetric model based on these systems and show a couple of examples of asymmetric fail-prone systems and how they relate to the B^3-condition.

3.1 Symmetric Byzantine Quorum Systems

Symmetric Byzantine quorum systems are introduced by Malkhi and Reiter [13] as follows: We consider a *universe* $\mathcal{P} = \{p_1, p_2, ...p_n\}$ of n processes. A *quorum system* $\mathcal{Q} \subseteq 2^{\mathcal{P}}$ is a collection of subsets of $\mathcal{P}$, each $Q \in \mathcal{Q}$ is called a *quorum*, such that no quorum is a subset of another and each pair intersects in at least one process. A *fail-prone system* $\mathcal{F} \subseteq 2^{\mathcal{P}}$ is a collection of subsets of $\mathcal{P}$, each $F \in \mathcal{F}$ is called a *fail-prone set*, such that no fail-prone set is contained in another. All processes in a fail-prone set may fail together during any execution of the protocol. The fail-prone system generalizes the usual threshold failure definitions and allows the system to define more specific failure assumptions.

Definition 1 (Byzantine dissemination quorum system [13])**.** A *Byzantine dissemination quorum system* for a fail-prone system $\mathcal{F}$ is a collection of sets of processes $\mathcal{Q} \subseteq 2^{\mathcal{P}}$, where each $Q \in \mathcal{Q}$ is called a *quorum*, such that no quorum is a subset of another quorum and the following properties hold:

Consistency: The intersection of any two quorums contains not only processes which may fail together, i.e.,

$$\forall Q_1, Q_2 \in \mathcal{Q}, \forall F \in \mathcal{F} : Q_1 \cap Q_2 \not\subseteq F.$$

Availability: For any fail-prone set in $\mathcal{F}$, there exists a disjoint quorum in $\mathcal{Q}$, i.e.,

$$\forall F \in \mathcal{F} : \exists Q \in \mathcal{Q} : F \cap Q = \emptyset.$$

In order to guarantee safety and liveness of the system even when accepting processes to be malicious or fail during execution, Byzantine quorum systems need to guarantee the availability and consistency properties. These properties allow for protocols to work despite faulty or malicious behavior of some processes in the system. To ensure the availability property we introduce the canonical quorum system, which is the *bijective complement* of a set $\mathcal{S}$ defined as $\overline{\mathcal{S}} = \{\mathcal{P} \setminus S | S \in \mathcal{S}\}$. For any fail-prone system defined for a universe of processes, its canonical quorum system will satisfy the availability condition by definition. To satisfy the consistency condition the so called Q^3-condition has to hold.

Definition 2 (Q^3-condition [13])**.** A fail-prone system $\mathcal{F}$ satisfies the *Q^3-condition* ($Q^3(\mathcal{F})$), when it holds

$$\forall F_1, F_2, F_3 \in \mathcal{F} : \mathcal{P} \not\subseteq F_1 \cup F_2 \cup F_3.$$

The Q^3-condition generalizes the common notion that Byzantine faults are tolerated if the faulty nodes constitute less than one third of the whole system.

3.2 Asymmetric Byzantine Quorum Systems

As in the symmetric case, we look at a universe $\mathcal{P} = \{p_1, ..., p_n\}$ of n processes. An *asymmetric fail-prone system* $\mathbb{F} = [\mathcal{F}_1, ..., \mathcal{F}_n]$ is a list of fail-prone systems, where $\mathcal{F}_i \subseteq 2^{\mathcal{P}}$ describes the subjective fail-prone system of p_i. An *asymmetric quorum system* $\mathbb{Q} = [\mathcal{Q}_1, \ldots, \mathcal{Q}_n]$ is a list of quorum systems, where $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ describes the subjective quorum system of p_i. Each process only accepts their individually defined fail-prone sets to fail during an execution of the protocol. In order to guarantee consensus among all members of the universe, Damgård et al. [7] introduce a new system-wide consistency property.

In the following definition and from here on out, $\mathcal{A}^*$ denotes the collection of all subsets of the sets in $\mathcal{A}$, that is, $\mathcal{A}^* = \{A' | A' \subseteq A, A \in \mathcal{A}\}$ [3].

Definition 3 (Asymmetric Byzantine quorum system [3])**.** An *asymmetric Byzantine quorum system* for $\mathbb{F}$ is a list of collections of sets $\mathbb{Q} = [\mathcal{Q}_1, \ldots, \mathcal{Q}_n]$, where $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ for $i \in [1, n]$. The set $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ is called the *quorum system of* p_i and any set $Q_i \in \mathcal{Q}_i$ is called a *quorum for* p_i. The following properties hold:

Consistency: The intersection of two quorums for any two processes does not contain only processes that can fail together according to both processes, i.e., for all $i, j \in [1, n]$

$$\forall Q_i \in \mathcal{Q}_i, \forall Q_j \in \mathcal{Q}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : Q_i \cap Q_j \not\subseteq F_{ij}.$$

Availability: For any process p_i and any set of processes that may fail together according to p_i, there exists a disjoint quorum for p_i in $\mathcal{Q}_i$, i.e., for all $i \in [1, n]$

$$\forall F_i \in \mathcal{F}_i : \exists Q_i \in \mathcal{Q}_i : F_i \cap Q_i = \emptyset.$$

In this definition every process can define its own personal trust which means that every member may trust differently and may be trusted differently. As in the symmetric case, to ensure availability of the system we define a process' quorum system as its fail-prone system's canonical quorum system. To ensure the consistency property, we need the B^3-condition, a generalization of the Q^3-condition for the asymmetric case.

Definition 4 (B^3-condition [3])**.** An asymmetric fail-prone system $\mathbb{F}$ satisfies the *B^3-condition* ($B^3(\mathbb{F})$), when it holds for all $i, j \in [1, n]$ that

$$\forall F_i \in \mathcal{F}_i, \forall F_j \in \mathcal{F}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : \mathcal{P} \not\subseteq F_i \cup F_j \cup F_{ij}$$

This condition needs to be satisfied to guarantee that no two asymmetric quorum systems include quorums which overlap only in processes that are accepted to be faulty by both parties.

3.3 Examples

Example 1. In the following example, taken directly from Cachin and Tackmann [3], we consider a universe of processes $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$ and the following asymmetric fail-prone system $\mathbb{F}_A$:

$$\mathbb{F}_A\text{:} \quad \begin{aligned} \mathcal{F}_1 &= \Theta_1^4(\{p_2, p_3, p_4, p_5\}) \\ \mathcal{F}_2 &= \Theta_1^4(\{p_1, p_3, p_4, p_5\}) \\ \mathcal{F}_3 &= \Theta_1^2(\{p_1, p_2\}) * \Theta_1^2(\{p_4, p_5\}) \\ \mathcal{F}_4 &= \Theta_1^4(\{p_1, p_2, p_3, p_5\}) \\ \mathcal{F}_5 &= \{\{p_2, p_4\}\} \end{aligned}$$

The notation $\Theta_k^n(\mathcal{S})$ for a set $\mathcal{S}$ with n elements denotes the threshold combination operator and enumerates all subsets of $\mathcal{S}$ of cardinality k. The operator $*$ for two threshold sets defines the union of their possible subsets: $\mathcal{A} * \mathcal{B} = \{A \cup B | A \in \mathcal{A}, B \in \mathcal{B}\}$.

After we enumerate all fail-prone systems in $\mathbb{F}_A$ according to the above notions, it looks like this:

$$\mathbb{F}_A\text{:} \quad \begin{aligned} \mathcal{F}_1 &= \{\{p_2\}, \{p_3\}, \{p_4\}, \{p_5\}\} \\ \mathcal{F}_2 &= \{\{p_1\}, \{p_3\}, \{p_4\}, \{p_5\}\} \\ \mathcal{F}_3 &= \{\{p_1, p_4\}, \{p_1, p_5\}, \{p_2, p_4\}, \{p_2, p_5\}\} \\ \mathcal{F}_4 &= \{\{p_1\}, \{p_2\}, \{p_3\}, \{p_5\}\} \\ \mathcal{F}_5 &= \{\{p_2, p_4\}\} \end{aligned}$$

Looking at all enumerated systems and their subsets' length we can immediately see that the only pair of candidates to possibly violate the B^3-condition are $\mathcal{F}_3$ and $\mathcal{F}_5$, since all other combinations cannot possibly reach the needed size of $|\mathcal{P}| = 5$. It can quickly be deducted that the condition holds for this pair as well, which means $B^3(\mathbb{F}_A)$ and therefore $\mathbb{Q}_A = \overline{\mathbb{F}_A}$ is an asymmetric quorum system for $\mathcal{P}$.

Example 2. This example is an adapted version of Cachin and Tackmann's second example [3]. This time we have a universe of six processes $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ and the following asymmetric fail-prone system $\mathbb{F}_B$:

$$\mathbb{F}_B\text{:} \quad \begin{aligned} \mathcal{F}_1 &= \Theta_2^3(\{p_2, p_4, p_5\}) * \{\{p_6\}\} \\ \mathcal{F}_2 &= \Theta_2^3(\{p_3, p_4, p_5\}) * \{\{p_6\}\} \\ \mathcal{F}_3 &= \Theta_2^3(\{p_1, p_4, p_5\}) * \{\{p_6\}\} \\ \mathcal{F}_4 &= \Theta_1^4(\{p_1, p_2, p_3, p_5\}) * \{\{p_6\}\} \\ \mathcal{F}_5 &= \Theta_1^4(\{p_1, p_2, p_3, p_4\}) * \{\{p_6\}\} \\ \mathcal{F}_6 &= \{\{p_1\}\} * \Theta_1^4(\{p_2, p_3, p_4, p_5\}) \end{aligned}$$

Let us take a closer look at the enumerated FPS of processes 1 and 6:

$$\begin{aligned} \mathcal{F}_1 &= \{S_1 = \{p_2, p_4, p_6\}, S_2 = \{p_2, p_5, p_6\}, S_3 = \{p_4, p_5, p_6\}\} \\ \mathcal{Q}_1 &= \{Q_1 = \{p_1, p_3, p_5\}, Q_2 = \{p_1, p_3, p_4\}, Q_3 = \{p_1, p_2, p_3\}\} \\ \mathcal{F}_6 &= \{T_1 = \{p_1, p_2\}, T_2 = \{p_1, p_3\}, T_3 = \{p_1, p_4\}, T_4 = \{p_1, p_5\}\} \\ \mathcal{Q}_6 &= \{R_1 = \{p_3, p_4, p_5, p_6\}, R_2 = \{p_2, p_4, p_5, p_6\}, R_3 = \{p_2, p_3, p_5, p_6\}, R_4 = \{p_2, p_3, p_4, p_6\}\} \end{aligned}$$

Here, $S_1 \cup T_2 \cup (S_2 \cap T_4) = \mathcal{P}$ and therefore the B^3-condition does not hold for this example system. The two processes cannot reach consensus in quorums Q_1 and R_2 since their only overlap is accepted by both to be faulty. This means that $\mathbb{Q}_A = \overline{\mathbb{F}_A}$ is not an asymmetric quorum system for $\mathcal{P}$ and in case of faulty processes, the system cannot reach a consensus in this configuration of $\mathbb{F}_B$.

4 Design

In order to verify whether a given system has a valid asymmetric quorum system, we need an algorithm that checks the B^3-condition for all possible fail-prone sets of any two processes. Since the size of a process' fail-prone system can be up to $2^{|\mathcal{P}|}$, the running-time of verifying B^3 for all sets of the fail-prone systems will grow exponentially to the number of processes in a universe. The verification of the B^3-condition is in NP, and comparable to the set cover problem, which has been proven to be NP-complete [8] and can only be approximated in polynomial running-time through greedy algorithms. However, since our set cover problem is parameterized, meaning we need to find a combination of at most three sets, a greedy set cover algorithm does not make sense in our scenario. Instead, we will try to optimize certain steps of the algorithm to improve performance of the exponential-time algorithm. First, we optimize a straightforward brute-force algorithm to omit the time-consuming creation and checking of the $\mathcal{F}_i{}^* \cap \mathcal{F}_j{}^*$ intersection and instead use a linear time verification for checking F_{ij}. In a second step, we use a bounded search-tree algorithm to try and optimize the number of steps we go through in each comparison of two systems, by removing sets that do not need to be checked as early as possible. Since we have a limit of 3 sets which need to be checked against each other, our tree will have a fixed height. However, its width can be very large. This means this algorithm may only work for rather small fail-prone sets, since in that case more sets can be removed at each level of the tree.

Apart from the B^3-checker itself, we need a fast and intuitive way to create new systems and test them against the checker. We design an input data structure which lets a user add any number of processes to their system and gives them complete freedom in defining their respective fail-prone systems. This input is then parsed into a list of enumerated fail-prone systems and verified by the B^3-checker.

4.1 Expression of Asymmetric Trust Assumptions

Since the program will mainly be used for testing purposes, the input information for a system is assumed to be entered by one user, instead of many single nodes by multiple users. The input interface thereby needs to be intuitive and very adaptable without many configuration changes. To grant the user a complete overview over the test system and offer fast adaptability, we define the input as one file with a JSON data structure.

In the examples shown in Section 3.3 we use the threshold combination operator Θ to forgo defining all enumerated fail-prone sets one by one and instead grant the possibility to define different thresholds of subsets of the universe. We also introduce the operator $*$ which grants us to "unify" two thresholds in that it allows any combination between two threshold definitions to be part of the fail-prone system.

We adopt these specifiers for our input file and translate them into two JSON keys, `select` and

`out-of`, which define the threshold and the list of processes respectively. To allow complete freedom in the fail-prone systems they decide to input for each process, the user may choose between defining the systems either as a list of enumerated sets, a dictionary of nested threshold combinations or a mixture between the two. This means the user is free to test specific predefined systems or check different combinations of simple and nested thresholds. While nesting threshold combinations deeper than three levels will not be intuitive either for the writing or reading of the input, the possibility is still available to the user, to guarantee truly individual trust assumptions.

Considering these requirements, we define the JSON input file with three levels as follows: The first level is a list of all processes in the universe. Each process is a dictionary containing the name of the process, other metadata the user might find important in order to distinguish or mark the process, and the fail-prone system. This system is itself either a dictionary again or a simple list. If it is a list, all included sets must be enumerated, meaning all sets are explicitly written and no threshold combinations can be used. If the user wishes to add threshold combinations to their fail-prone system, it has to be written as a dictionary. Within the dictionary, the user states the threshold (select) and the list of processes (out-of) the threshold is to be used on. This can be a simple list of enumerated processes, a list of nested thresholds or a mixture between the two.

This design of the input has the following advantages: Firstly, by making the system a list, processes may be added and removed easily. Secondly, the naming of the processes helps the user find problematic fail-prone systems faster, should the B^3-checker fail. And lastly, all systems can be changed and adapted by working on a single file, without needing to change between different processes or members of the system. The dictionary structure also allows the user to add any additional information they might need for testing and adapting, such as notes or test results, to the same file. These additional keys will not be read by the input parser and can therefore be added and deleted without compromising the workflow of the system. Additionally, a future advantage of this input file is that each process in a system is independent of any other processes. This means in a future implementation of the asymmetric protocol the configuration file can be adapted to be created and changed through multiple inputs by different members of the system.

4.2 Verification of the B3-condition

The B^3-verifier is the main part of the program. It takes a parsed list of asymmetric fail-prone systems as an input and outputs whether the given system fulfills the B^3-condition. To design an efficient algorithm for this condition, we first need to analyze its complexity and similar problems that might help us find a solution.

A straightforward brute-force algorithm for verifying the B^3-condition is depicted in Algorithm 1. For every pair of asymmetric fail-prone systems $\mathcal{F}_i, \mathcal{F}_j$ the union between every two sets $F_i \in \mathcal{F}_i, F_j \in \mathcal{F}_j$ is made. Each of those unions is then again unified with one of $F_{ij} \in \mathcal{F}_i \cap \mathcal{F}_j$ and compared to the whole universe. If this union contains all processes of the universe, the B^3-condition does not hold, otherwise we move on to the next pair of systems. It should be noted that instead of using the collection of all subsets of the sets, denoted by * as depicted in Definition 4, we can use $\mathcal{F}_i$ and $\mathcal{F}_j$ without all subsets of their respective sets, since our goal is to cover the whole universe with their union, and therefore all subsets of larger sets do not need to be checked by our algorithm.

The complexity of the B^3-check is dependent on the following values:

- The number of processes that need to be compared to each other is $\binom{|\mathcal{P}|}{2}$.
- The number of fail-prone sets to be checked in a single asymmetric fail-prone system can be up to $2^{\mathcal{P}}$.

The verification of the B^3-condition can be seen as a special case of the unweighted set cover problem, which is one of Karp's 21 NP-complete problems [8] and is not fixed-parameter-tractable [5].

Algorithm 1 Brute-Force

```
1: procedure CHECKB3(universe, failProneSystems)
2:     for system_i, system_j ∈ failProneSystems do
3:         intersection ← {F_i ∩ F_j | F_i ∈ F_i, F_j ∈ F_j}
4:         union ← {F_i ∪ F_j | F_i ∈ F_i, F_j ∈ F_j}
5:         for F_ij ∈ union do
6:             for F_k ∈ intersection do
7:                 threeUnion ← F_ij ∪ F_k
8:                 if universe ⊆ threeUnion then return false
9:     return true
```

Standard set cover problem: Given a universe X of n items and a collection S of m subsets of X : $S = S_1, S_2, ..., S_m$. We assume that $\bigcup S = X$. The aim is to find a subcollection of sets in S, *of minimum size*, that covers all of X.

The verification of the B^3-condition between two processes can be described in a similar way:

Set cover for the B^3-condition: Given a universe $\mathcal{P}$ of n items and two collections $\mathcal{F}_i$ of m subsets of $\mathcal{P}$ and $\mathcal{F}_j$ of p subsets of $\mathcal{P}$. We assume that the union of all the sets in $\mathcal{F}_i$ and $\mathcal{F}_j$ is U, $\bigcup \mathcal{F}_i \cup \bigcup \mathcal{F}_j = U$. The aim is to find a subcollection of *three sets* that covers all of $\mathcal{P}$ with the following constraints:

- One set has to come from $\mathcal{F}_i$
- One set has to come from $\mathcal{F}_j$
- One set has to be a subset of both a set in $\mathcal{F}_i$ and a set in $\mathcal{F}_j$

What makes our case difficult is that we cannot test any three-set combination of one system. Since we have essentially three systems to choose from ($\mathcal{F}_i, \mathcal{F}_j$ and $\mathcal{F}_i \cap \mathcal{F}_j$) and the constraint that every set needs to come from a different system, we have to test any three-set combination between all three systems.

The set cover algorithm can usually be approximated in polynomial time through a greedy algorithm [6], assuming that it can be solved by the input. The greedy algorithm will find the first best combination of subsets that cover the whole universe, but not necessarily the minimal number of subsets. Since our case is parameterized, meaning we accept at most three sets to cover the whole universe, this heuristic does not work for the B^3-check. Additionally, unlike different set cover applications, we need to assume the universe cannot be covered by the subsets of the two systems, as the opposite case would be the undesirable scenario in our algorithm. Since we aim for a positive outcome, we should presume that all individual combinations need to be checked.

A similar problem to the set cover is the vertex cover, another one of Karp's 21 NP-complete problems [8].

Vertex Cover: Given a graph $G = (V; E)$, find the minimal number of vertices $v \in V$ which connect all edges $e \in E$ of the graph.

As opposed to set cover, however, vertex cover is fixed-parameter-tractable, which means if given only a fixed number of nodes, there exists a polynomial-time algorithm to solve the vertex cover problem.

Fixed-parameter-tractable vertex cover algorithm: Given $G = (V; E)$ and a fixed parameter k

1. Pick an edge.
2. One of its endpoints must be covered.
3. Try both of its endpoints and check unconnected edges recursively until you have tested k vertices.

This algorithm uses a bounded search-tree to cover all vertices. A parameterized set cover problem can be tackled in the same way and allows a fixed tree height bounded by the parameter. Due to the fact that any fail-prone system can have up to $2^{|\mathcal{P}|}$ sets, even though we do have a fixed parameter of three sets to cover our universe, the width of our bounded search-tree is still exponential, and therefore a polynomial time algorithm is not possible in our case. If we assume that the fail-prone sets of a system are rather small, however, this algorithm can still save time as we can rule out certain sets from the beginning if we start the algorithm by choosing items in the universe in a certain order.

Other possible optimizations of an exponential-time algorithm include special heuristics tailored to the problem at hand.

Heuristics: The following heuristics could be used to remove sets from the systems that do not have to be checked.

- Do not test sets $F_i \in \mathcal{F}_i$, which are too small to potentially fill the whole universe $\mathcal{P}$ if it was unified with the largest set of the respective other two systems.

$$F_i : |F_i| + |max(\mathcal{F}_j)| + |max(\mathcal{F}_i \cap \mathcal{F}_j|) < |\mathcal{P}|$$

- Do not test pairs of systems which do not cover the whole universe $\mathcal{P}$ when unified completely.

However, heuristics can only be justified when tested against large input data. This is currently not possible to do, since no system using asymmetric fail-prone systems has been implemented in practice. Therefore, we forgo the inclusion of these heuristics for this first implementation of the verifier.

Considering these points, we specify two different algorithms to verify the B^3-condition, each of which takes a different starting point. Our first algorithm is based on the brute-force algorithm in Algorithm 1, with an optimized solution for one of its most time consuming steps: the testing of each pair of sets against all $F_{ij} \in \mathcal{F}_i \cap \mathcal{F}_j$. The second algorithm is implemented in a similar way to the fixed-parameter-tractable algorithm for the vertex cover problem, starting with the universe of processes and checking all sets for its inclusion. We take a closer look at the exact implementation of both algorithms in Chapter 5.

5 Implementation

To allow the B^3-verification of any system defined by a user we implemented a program in form of a command line tool which takes a universe of processes with their respective asymmetric fail-prone systems as an input and outputs whether or not the B^3-condition holds for this universe. In the following, we introduce the key parts of this implementation. We start with the overall technology used and then dive into the technical implementations of the input, the parser of the input into enumerated systems, and the two verifier algorithms.

5.1 Technology

The system input file is a simple JSON file. The program is written in Python, which offers many efficient set operation functions. Python's set data structure is built on its dictionary data structure and is therefore very fast [1]. Since for bigger systems, fail-prone sets will grow accordingly, we also had a look at Roaring Bitmaps [11] and its corresponding Python wrapper PyRoaringBitMaps [2], which offers very fast set operations by using compressed bitmap data structures. Some performance tests with example sets of sizes between 10 and 300 showed that Python's own set structure performs better for smaller sets than the PyRoaringBitMaps. Roaring will improve set operation performances from a size of around 100 (see Figure B.3, Figure B.2, Figure B.1, Figure B.4) and upwards; the time of checking whether or not an element is included in a set is not dependent on set size and is around the same for both data structures (Figure B.5). For sets smaller than 100, Python's common set data structure is faster. Since our asymmetric fail-prone systems at the moment are not expected to be larger than 100 processes, we use the set data structure as a default, with the possibility to be changed to bitmaps by the user.

5.2 Config File and Data Structure

For the configuration a simple JSON file is used through which one or multiple systems can be entered into the program. Every system is a list and consists of multiple processes, each process being defined as a dictionary. Every process definition must include a `PubKey`, which is its identifier, and the `FailProneSystem`. The processes may have other metadata that are ignored by the program but might be needed by the user for identification etc.

The asymmetric fail-prone system of a process is either a list or a dictionary. If it is a list, every entry of the list must be a flat list, which means all fail-prone sets must be enumerated. If the user wants to use any non-enumerated sets, the system needs to be structured like a nested dictionary in the following way:

```
[
  system_name: [
    { PubKey:"processX", FailProneSystem:
      {select:1, out-of:
        [
          {select:2, out-of:
            ["processX", "processY", "processZ"]
          },
          {select:2, out-of:
            [
              {select:1, out-of:["processA", "processB"]},
              "processC"
            ]
          }
        ]
      }
    },
    { PubKey:"processY", FailProneSystem:
      [
        ["processX", "processY"],
        ["processZ", "processY"],
        ["processX", "processZ"]
      ]
    }
  ]
]
```

Figure 5.1. Configuration of an asymmetric fail-prone system.

Any threshold is defined through the two keys `select` and `out-of`, where `select` defines the threshold and `out-of` the list of processes or nested threshold definitions. All `select` values may be any number from 1 to the length of the chosen list of processes in `out-of`. The `out-of` value is always a list and may contain another threshold dictionary, a list of processes or a combination of the two. The nesting of thresholds is not limited to any level. The $*$ operator depicted in the two examples in Section 3.3 can be expressed through this syntax as well: Since the threshold statements can be nested, a union between two thresholds or sets of processes can be defined as follows:

$$\{select : 2, out\text{-}of : [\{select : 1, out\text{-}of : [p_1, p_2]\}, \{select : 1, out\text{-}of : [p_4, p_5]\}]\}.$$

This definition is the same as the one of $\mathcal{F}_3$ from example 1 in Section 3.3: $\Theta_1^2(\{p_1, p_2\}) * \Theta_1^2(\{p_4, p_5\})$.

Figure 5.1 shows a nested threshold for `processX`, and an explicitly enumerated fail-prone system for `processY`.

Since some systems, like Stellar, prefer to define their personal quorum systems instead of the fail-prone systems as assumed by Cachin and Tackmann [3], we allow the key `QuorumSystem` in addition to `FailProneSystem`, and let the user define the quorum systems for a process. The quorum system will then be translated into the fail-prone system by using the canonical quorum system property, and then tested against the B^3-checker.

5.3 System Parser

The input file is intuitive to understand for the user, reduces space by defining the fail-prone systems as threshold dictionaries, and can contain a lot of information about the system in the same place. However,

once the user wants to check the B^3-condition, the input needs to be translated in a way the program can read and evaluate the condition on fully enumerated sets. This is done by a parser, which receives the user's input file through a file path and translates it into two different objects. The first object is a key dictionary which collects all processes by their identifiers (`PubKey`) and gives each one an integer ID replacing the String type names given by the user, which will then be used as a reference in the enumerated fail-prone systems. The second object is a dictionary of all processes identified by their Integer IDs and their respective fail-prone systems, enumerated into a list of sets. Each fail-prone system is enumerated by recursively calculating the cartesian product of each threshold combination defined in the input file. If the input for a process does not contain any threshold sets, all the parser does is add the already enumerated fail-prone system to the list of processes.

5.4 Implementation of an Optimized Brute-Force Algorithm

For the first algorithm, we optimized a straightforward brute-force algorithm. The algorithm takes the universe of processes U and a list of fail-prone systems FPS as the input and compares every two processes, including a process with itself. The enumerated fail-prone systems $\mathcal{F}_i, \mathcal{F}_j \in FPS$ are iterated, and their union $\mathcal{F}_{ij} : \{F_i \cup F_j | F_i \in \mathcal{F}_i, F_j \in \mathcal{F}_j\}$ is computed. For each $F_{ij} \in \mathcal{F}_{ij}$, its difference to the list of all processes, $D = \mathcal{P} \setminus F_{ij}$, is computed. The algorithm then searches both fail-prone systems for D. If it can be found in both sets, the B^3-condition is violated and the algorithm stops. If the B^3-condition holds for every two processes $\mathcal{F}_i$ and $\mathcal{F}_j$ in the system, it holds for the full system.

This algorithm is optimized from the brute-force algorithm in that we do not need to calculate the intersections of the two fail-prone systems $\mathcal{F}_i \cap \mathcal{F}_j$, which saves computing time. Additionally, checking only $\mathcal{F}_i$ and $\mathcal{F}_j$ for every $F_i \cup F_j$ is much less time consuming than checking all their intersections.

Algorithm 2 Optimized Brute-Force

```
1:  procedure CHECKB3OPTIMIZED(U, FPS)
2:      for system_i, system_j ∈ FPS do
3:          union ← {F_i ∪ F_j | F_i ∈ F_i, F_j ∈ F_j}
4:          for F_ij ∈ union do
5:              D ← U \ F_ij
6:              for F_i ∈ system_i do
7:                  if D ⊆ F_i then
8:                      for F_j ∈ system_j do
9:                          if D ⊆ F_j then return false
10:     return true
```

5.5 Implementation of a Bounded Search-Tree Algorithm

The second algorithm is inspired by a bounded search-tree. Although the set cover problem can provably not be solved in polynomial time if $P \neq NP$ [8] and is not fixed-parameter-tractable, the time complexity of the comparison of all sets with each other can still be improved in certain cases, by adapting the fixed-parameter-tractable algorithm for the vertex cover problem to our specific set cover problem case. Since our upper bound of sets that need to be compared to each other is three in every case, we can use a search-tree with a height of three to check the B^3-condition.

As opposed to the first algorithm, where we started with the fail-prone sets and checked for the differences to the list of processes in the system, in this algorithm we start with the list of processes, choose one of them and select all fail-prone sets of a system that contain the chosen process. This way, we can eliminate all fail-prone sets which do not contain the chosen process and, depending on how the sets were chosen, might end up with fewer sets to compare. However, the chosen process may be part

of any one of the three following systems: the fail-prone system of the first process (1), the fail-prone system of the second process (2), or the intersection between both systems (3). This means that we need to check all three systems for this one chosen process and continue from there.

We start by choosing a process from the universe and selecting all sets of all systems according to (1), (2) and (3), which include said process. Next, for every chosen set, its difference to the universe is calculated, a new process from the remaining universe is chosen, and we recursively match the sets to the chosen processes until we reach a tree height of 3. It has to be noted that for the second step, only the two systems the set chosen in step one is *not* part of need to be checked. For the third step, only the last remaining system needs to be checked. For example, if the first set was chosen from system 1, for the second step we need to check systems 2 and 3. If then we chose a set from system 2, in the third step all we have to check are the sets in system 3.

This shows that as opposed to the vertex cover problem, our search-tree is not bounded in its width. While the tree may be fixed in its depth, it can be very wide for large fail-prone systems.

Algorithm 3 Bounded Search-Tree

procedure INIT($universe, failProneSystems$)
  for $system_i, system_j$ in $failProneSystems$ **do**
    $intersection \leftarrow \{F_i \cap F_j | F_i \in \mathcal{F}_i, F_j \in \mathcal{F}_j\}$
    if not HOLDSB3($universe, system_i, system_j, intersection$) **then return** $false$
  return $true$

procedure HOLDSB3($universe, \mathcal{X}, \mathcal{Y}, \mathcal{Z}$) ▷ $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$ are systems of sets
  select $node \in universe$
  if $\mathcal{X} \neq \emptyset$ **then**
    select subsystem $sub\mathcal{X} \subseteq \mathcal{X}$ s.t. $\forall X \in sub\mathcal{X} : node \in X$
    for $set \in sub\mathcal{X}$ **do**
      $difference \leftarrow universe \setminus set$
      if $difference = \emptyset$ **then return** $false$
      SEARCHTREE($difference, \emptyset, \mathcal{Y}, \mathcal{Z}$)
  if $\mathcal{Y} \neq \emptyset$ **then**
    select subsystem $sub\mathcal{Y} \subseteq \mathcal{Y}$ s.t. $\forall Y \in sub\mathcal{Y} : node \in Y$
    for $set \in sub\mathcal{Y}$ **do**
      $difference \leftarrow universe \setminus set$
      if $difference = \emptyset$ **then return** $false$
      SEARCHTREE($difference, \mathcal{X}, \emptyset, \mathcal{Z}$)
  if $\mathcal{Z} \neq \emptyset$ **then**
    select subsystem $sub\mathcal{Z} \subseteq \mathcal{Z}$ s.t. $\forall Z \in sub\mathcal{Z} : node \in Z$
    for $set \in sub\mathcal{Z}$ **do**
      $difference \leftarrow universe \setminus set$
      if $difference = \emptyset$ **then return** $false$
      SEARCHTREE($difference, \mathcal{X}, \mathcal{Y}, \emptyset$)
  return $true$

6 Evaluation

To decide which of the two algorithms is more suited for the efficient verification of the B^3-condition, we need to evaluate their running-time for different universe sizes and definitions of asymmetric fail-prone systems. In this chapter, we take a look at the running-time of both algorithms, define synthetic datasets of varying universe sizes and fail-prone system definitions, and evaluate the running-time of both algorithms according to these datasets.

6.1 Running-Time Analysis

Since the definition of asymmetric quorums (Definition 3) does not allow quorums that are subsets of other quorums, the largest size a fail-prone system can have is

$$\binom{|\mathcal{P}|}{\lfloor \frac{|\mathcal{P}|}{2} \rfloor}$$

according to Sperner's theorem [17]. The Sperner family defines a family F of subsets of a finite set $\mathcal{P}$ in which none of the sets contains another.

Running-time of the optimized brute-force algorithm: Given two processes $\mathcal{F}_i$ and $\mathcal{F}_j$ where in the worst-case $|\mathcal{F}_i| = |\mathcal{F}_j| = \binom{|\mathcal{P}|}{\frac{|\mathcal{P}|}{2}}$, the running-time is

$$(|\mathcal{F}_i \cup \mathcal{F}_j|) \cdot (|\mathcal{F}_i| + |\mathcal{F}_j|) = O(2^{|\mathcal{P}|})$$

where the computation of $\mathcal{F}_i \cup \mathcal{F}_j$ has a runtime of $|\mathcal{F}_i| \cdot |\mathcal{F}_j|$

Running-time of the bounded search-tree algorithm: Given two processes $\mathcal{F}_i$ and $\mathcal{F}_j$ where in the worst-case $|\mathcal{F}_i| = |\mathcal{F}_j| = \binom{|\mathcal{P}|}{\frac{|\mathcal{P}|}{2}}$, the running-time is

$$(|\mathcal{F}_i| + |\mathcal{F}_j| + |\mathcal{F}_i \cap \mathcal{F}_j|) \cdot (|\mathcal{F}_i| + |\mathcal{F}_j|) \cdot (|\mathcal{F}_i \cap \mathcal{F}_j|) = O(2^{|\mathcal{P}|})$$

where the computation of $\mathcal{F}_i \cap \mathcal{F}_j$ has a runtime of $|\mathcal{F}_i| \cdot |\mathcal{F}_j|$

The running-time analysis of both algorithms shows the bounded search-tree algorithm will perform slower in the worst-case. However, this analysis alone cannot tell how they will perform in practice, since the very point of asymmetric fail-prone systems is that they are defined subjectively and can therefore have many different sizes for each process.

The following points may influence the performance of the algorithms:

1. The size of the universe
2. The size of the fail-prone systems (number of sets)
3. The size of the fail-prone sets (number of processes in a set)

Since we assume most asymmetric fail-prone systems will be defined by thresholds, naturally, as the universe grows, so will the fail-prone systems for each process. For every additional process in a universe, each member can potentially grow their thresholds by one process and thereby grow their fail-prone systems exponentially.

The running-time of the bounded search-tree algorithm depends on both the size of the asymmetric fail-prone systems and the size of the fail-prone sets, since the larger a set becomes, the higher the chance that a specific item is included in it. The running-time of the optimized brute-force algorithm is mostly dependent on the size of the fail-prone systems, as it needs to iterate through all sets of both systems.

6.2 Datasets

In order to compare the two algorithms to each other, we test them against synthetic sample data.[1] Since the running-time of the B^3-verifier is exponential in the number of processes in a universe, to evaluate our algorithms we need to test them against systems of different sizes. Due to the asymmetric property, there is a large number of different fail-prone sets any process can have, and depending on the system each process chooses, the dataset might be biased towards one specific algorithm. To make the testing as fair as possible, we define three different datasets, each of which has a specific property we need to take into account for future – truly subjective – fail-prone systems.

Our first dataset implements completely symmetric trust, meaning all processes accept failures of a threshold of $\frac{n}{3}$ of the n processes in the system. This is the classic threshold trust used for Byzantine Fault Tolerance, and therefore the largest system of sets that still guarantees the B^3-condition will hold. Testing this dataset will show how the two algorithms perform in worst-case situations, where the fail-prone systems and their respective fail-prone sets are rather large.

The second, random dataset is built as follows: For every process in the universe with size n we select a random size for the asymmetric fail-prone system between 1 and $\binom{n}{\frac{n}{2}}$, and for every set we select its size randomly between 1 and $\lfloor \frac{n}{3} \rfloor$. Testing this dataset will help us understand how the algorithms will perform for very subjective fail-prone systems, with very different sizes of both the systems and their sets.

For our third dataset, we choose explicitly defined symmetrically rotating fail-prone systems. Every process has similarly structured trust assumptions, but about different subsets of the universe. Every process trusts itself never fails. It then defines two separate trust assumptions for the other processes in the system: One is a simple threshold trust for one subset of the system, the other is an $*$-union of two smaller thresholds of the rest of the system. An example for this can be seen in Figure 6.1, for one process in a universe of seven. In this example process a believes itself will never fail, two processes out of $\{d, e, f\}$ may fail together and one out of $\{g, h\}$ may fail together with one out of $\{b, c\}$. This assumption can be written as $\mathcal{F}_a = \{\Theta_2^3(\{d, e, f\}), \Theta_1^2(\{g, h\}) * \Theta_1^2(\{b, c\})\}$. Systems of larger sizes are designed in a similar way. Testing this dataset will show how the two algorithms perform when working with specific manually defined asymmetric systems.

6.3 Comparison of the two Algorithms

For our first dataset we only analyze the running-time of one pair of processes, since every comparison takes the same amount of time for symmetric threshold trust. The other two datasets implement a more asymmetric trust, and we analyze the running-time of verifying B^3 for the whole universe.

[1] We executed all tests with the Python `timeit` function on a machine with an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 199GHz Processor and 16.0 GB RAM

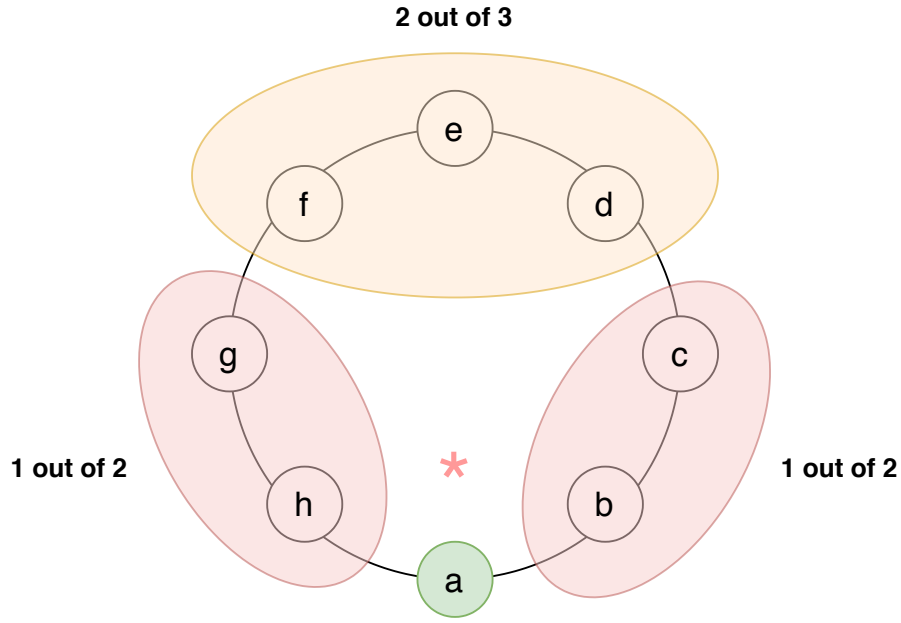


Figure 6.1. Symmetrically rotating asymmetric fail-prone system example for process a.

Figure 6.2 shows that for a symmetric threshold of $n > 3f$, the optimized brute-force algorithm works much faster for systems larger than 10 processes. This is because in this case, the bounded search-tree has a width of $\binom{n}{\frac{n}{3}}$ and all processes are distributed equally across the fail-prone sets, meaning for each processes in the list, one third of all sets need to be checked. However, for the dataset with randomly defined sets, the bounded search-tree performs better the larger the universe becomes, as shown in Figure 6.3. As in the random case, in our third dataset the bounded search-tree algorithm shows a significant running-time improvement as the universe becomes larger (Figure 6.4).

Unfortunately, our evaluation ends with these generated datasets, since no protocols based on Cachin and Tackmann's [3] model of asymmetric Byzantine quorum systems have been implemented in practice so far, and therefore no practical data on which we could test our algorithms exists. Although Stellar uses subjectively defined quorums for their system, it works with a different protocol that allows system members to define very small quorum slices (usually with a threshold of around 9 processes out of the currently 88 active nodes [2]), and therefore very large canonical fail-prone sets, leading to an almost guaranteed failure of the B^3-verification.

[2]This can be deducted from the threshold sizes defined for each node in the Stellar Public API: `https://api.stellarbeat.io/v1/nodes` (Information retrieved in December 2019)

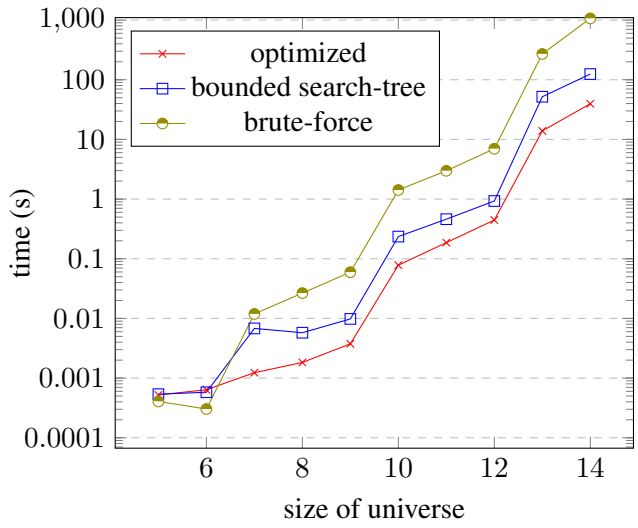


Figure 6.2. Comparison of the algorithms with universes of different sizes. The fail-prone system of each process contains $\binom{n}{\frac{n}{3}}$ sets.

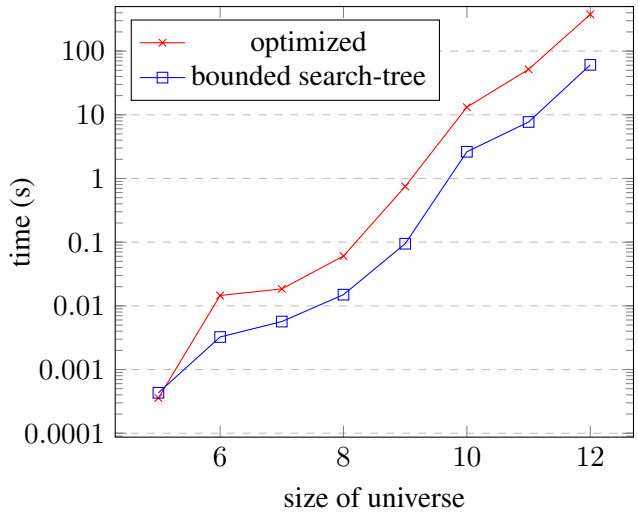


Figure 6.3. Comparison for random amounts and sizes of fail-prone-sets

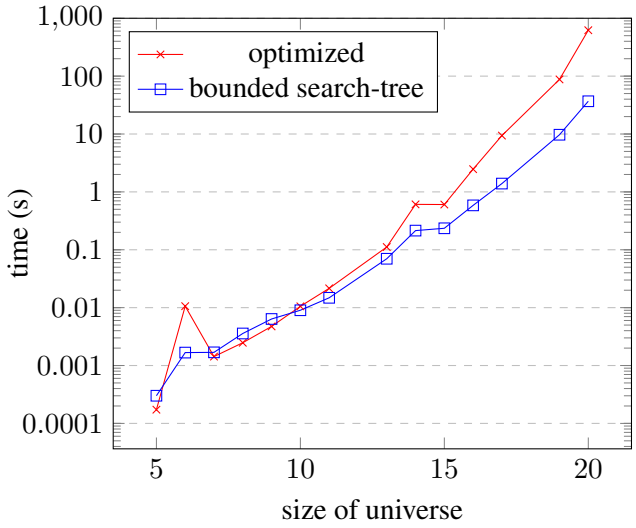


Figure 6.4. Comparison for predefined rotated asymmetric fail-prone systems. (Note: The systems with 12 and 18 processes have been removed from this dataset, since the B^3-condition did not hold for them).

7 Conclusion

7.1 Discussion

Cachin and Tackmann's [3] new asymmetric trust protocols offer subjective trust definitions for every member of a blockchain system. For the protocol to be able to guarantee safety and liveness of the system, the B^3-condition must hold. At this point in time, no program that can verify this condition for any asymmetric fail-prone system exists. The purpose of this project was to design and implement such an algorithm and optimize it to a level where it can be used to test examples and smaller test systems to gain insight into how such an asymmetric trust protocol could work in practice. We implemented two algorithms to verify the B^3-condition, both of which show a considerable improvement of performance in comparison to a brute-force algorithm. However, the running-times of both algorithms still grow exponentially with the number of processes in the universe. As a system grows, so will the fail-prone systems of each process, which means neither of our algorithms is feasible to use for larger universes and therefore they will not be practical when working with an actual running system.

The process of parsing the input into enumerated sets has the advantage that we can divide the parsing and the evaluation of the B^3-condition into two steps, and can test different set cover algorithms on simple integer sets, instead of having to think about how to evaluate our specific definition of threshold quorums in addition to testing different algorithms. Enumerating the system into sets of integers also gives us the possibility to use bitmaps for evaluating set operations faster. This, however, only makes sense when working with larger systems, as discussed in Section 5.1.

7.2 Future Work

In order to be able to check B^3 for bigger systems, we need to find a new way to check the condition, which does not include enumerating all sets, but instead uses the configuration file in a different way to check the systems against each other. It will be interesting to find a way to compare the systems without parsing them and enumerating the sets. Additionally, once the way members of a system choose their fail-prone systems can be studied in more detail, heuristics as discussed in Chapter 4 may be applied to the algorithm.

Our program currently lets a user define a system of processes with their respective fail-prone systems and test those systems for the B^3-condition. While the output for a failed test will return the processes for which it failed, it does not yet give information on how to fix the system.

A Command Line Tool Manual

A.1 Installation

In order to install the B3checker you need the following installed on your machine:

- Python 3.7.0
- PyRoaringBitmap
- requests

You can download the code from
`https://gitlab.inf.unibe.ch/crypto-students/2019.bsc.sabine.brunner.git`.

If you have not yet installed `PyRoaringBitMap` and `requests`, run

```
pip install pyroaring
pip install requests
```

Finished! You can now use the B3checker.

A.2 Input File

To define your systems you can either use the predefined `system-config.json` file in the project directory, or use a another file from you computer. If you create your own file, it must be a `.json` file and have the same data structure as we define in `system-config.json`. `system-config.json` already contains a number of example systems that can help in understanding the data structure of asymmetric fail-prone systems. The structure of an example system can be seen in Figure 5.1.

A.3 Usage of B3checker

If you wish to check a system from a file on your computer, run:

```
python b3checker.py "system_name" --file_path "file_path"
```

If you wish to check a system you defined in the `system-config.json` file, run:

```
python b3checker.py "system_name"
```

The system will output whether or not the B^3-condition holds for the tested system. In case it does not hold, it will print the two processes and the three fail-prone sets for which it failed., e.g.:
The `PubKey` and set of the first processes:

```
1: ['2', '4', '6']
```

The `PubKey` and set of the second process:

```
6: ['1', '3']
```

The intersection set of both fail-prone systems:

```
['5']
```

A.4 Optional Arguments

If you want to test only part of the system you can use the following optional commands:

- `--one_process`

  Test B^3 for only one process against the whole system.
  Takes one argument:

  - PubKey of the process

- `--two_processes`

  Test B^3 for only two processes.
  Takes two arguments:

  - PubKey of the first process
  - PubKey of the second process

Apart from the required `sys_name` argument, there are several other optional commands one can use to configure the B^3-checker:

`--parse_quorum_system`

Use the `QuorumSystem` defined in the input, instead of the default `FailProneSystem`. The `QuorumSystem` will be parsed into its canonical fail-prone system and then be checked.

`--bitmap`

Use bitmap data structure instead of default Python sets.

`--use_opt`

Use optimized brute-force algorithm instead of default bounded search-tree algorithm.

`--show_all_faults`

If B^3-check fails, show all combinations where it fails. When using this command, the b3checker will write a file `all_failures.json` which includes all combinations of processes, for which the B^3-check failed.

You can run

```
python b3checker.py -h
```

anytime to see how to use these optional arguments.

B

Performance Tests of PyRoaringBitMap vs. Python Set Data Structure

The following are performance tests we did for PyRoaringBitMap and the common Python set data structure to determine for which set sizes Roaring starts to show performance improvements.

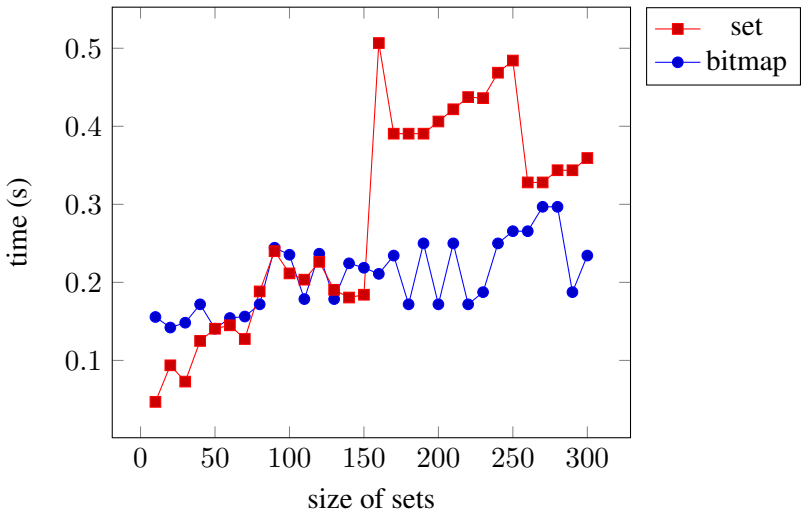


Figure B.1. Comparison of the `union`-operation

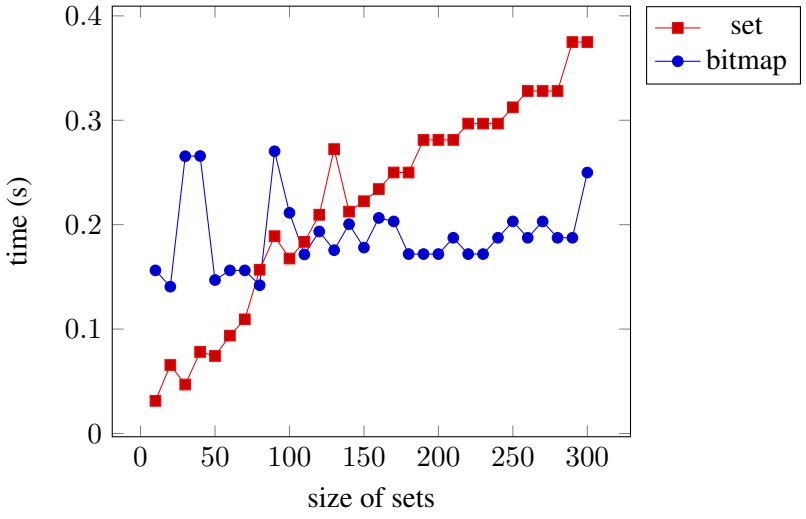


Figure B.2. Comparison of the `intersection`-operation

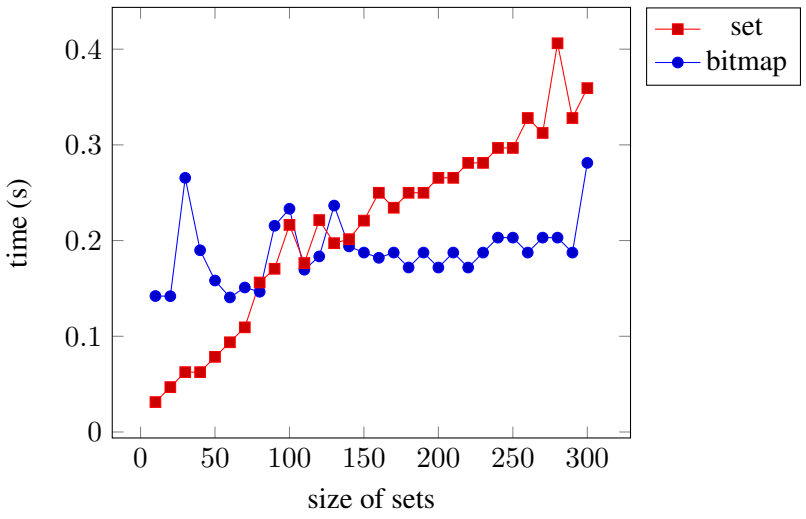


Figure B.3. Comparison of the `difference`-operation

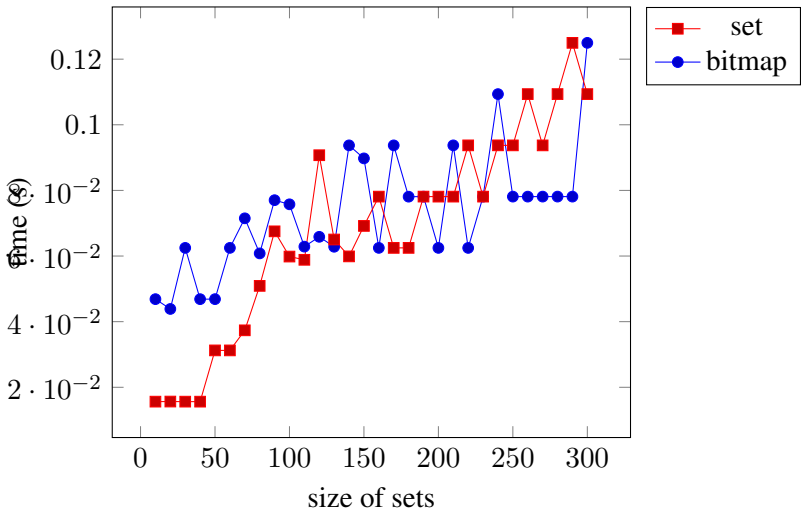


Figure B.4. Comparison of the `subset`-operation

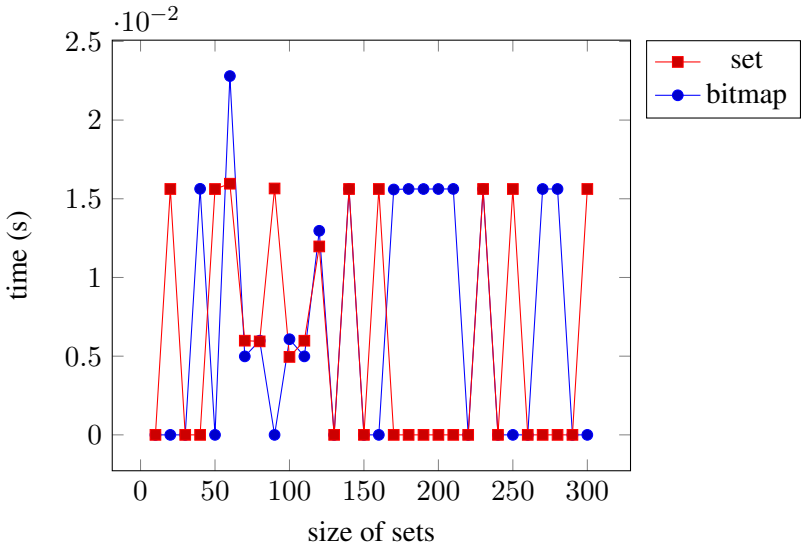


Figure B.5. Comparison of the `includes`-operation

Bibliography

[1] TimeComplexity. `https://wiki.python.org/moin/TimeComplexity`, 2017. Accessed: 2020-02-04.

[2] PyRoaringBitMap. `https://github.com/Ezibenroc/PyRoaringBitMap`, 2020. Accessed: 2020-02-04.

[3] Christian Cachin and Björn Tackmann. Asymmetric distributed trust. *arXiv preprint arXiv:1906.09314*, 2019.

[4] Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol. *arXiv preprint arXiv:1802.07242*, 2018.

[5] Rajesh Chitnis, Mohammad Taghi Hajiaghayi, and Guy Kortsarz. Fixed-parameter and approximation algorithms: A new look. In *International Symposium on Parameterized and Exact Computation*, pages 110–122. Springer, 2013.

[6] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[7] Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 357–375. Springer, 2007.

[8] Richard M Karp et al. Complexity of computer computations. *Reducibility among combinatorial problems*, 23(1):85–103, 1972.

[9] Minjeong Kim, Yujin Kwon, and Yongdae Kim. Is Stellar as secure as you think? In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 377–385. IEEE, 2019.

[10] Łukasz Lachowski. Complexity of the quorum intersection property of the federated Byzantine agreement system. *arXiv preprint arXiv:1902.06493*, 2019.

[11] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with Roaring. *Software: Practice and Experience*, 46(11):1547–1569, 2016.

[12] Ethan MacBrough. Cobalt: BFT governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

[13] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[14] David Mazieres. The Stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32, 2015.

[15] David Schwartz, Noah Youngs, Arthur Britto, et al. The Ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8), 2014.

[16] Robert Shostak, Marshall Pease, and L Lamport. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[17] Emanuel Sperner. Ein Satz über Untermengen einer endlichen Menge. *Mathematische Zeitschrift*, 27(1):544–548, 1928.

Erklärung

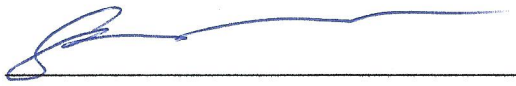
Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern, 1. März 2020

Ort/Datum



Unterschrift