Cost Reduction With Guarantees: Formal Reasoning Applied To Blockchain Technologies

Maria A Schett

A thesis submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

November 7, 2021

Abstract

Blockchain technologies are moving fast and their distributed nature as well as their high-stake (financial) applications make it crucial to “get things right”. Moreover, blockchain technologies often come with a high cost for maintaining blockchain infrastructure and for running applications. In this thesis formal reasoning is used for guaranteeing correctness while reducing the cost of *(i)* maintaining the infrastructure by optimising *blockchain protocols*, and *(ii)* running applications by optimising *blockchain programs*—so called *smart contracts*. Both have a clear cost measure: for protocols the amount of exchanged messages, and for smart contracts the monetary cost of execution. In the first result for *blockchain protocols* starting from a proof of correctness for an abstract blockchain consensus protocol using infinitely many messages and infinite state, a refinement proof transfers correctness to a concrete implementation of the protocol reducing the cost to finite resources. In the second result I move from a blockchain to a *block graph*. This block graph embeds the run of a deterministic byzantine fault tolerant protocol, thereby getting parallelism “for free” and reducing the exchanged messages to the point of omission. For *blockchain programs*, I optimise programs executed on the Ethereum blockchain. As a first result, I use superoptimisation and encode the search for cheaper, but observationally equivalent, program as a search problem for an automated theorem prover. Since solving this search problem is in itself expensive, my second result is an efficient encoding of the search problem. Finally for reusing found optimisations, my third results gives a framework to generate peephole optimisation rules for a smart contract compiler.

Impact Statement

My work guarantees correctness while reducing two fundamental costs in blockchain technologies, and more generally in distributed systems: communication cost and the cost of computation. I use formal reasoning, not only for reasoning about correctness, but also for reasoning about optimisations of blockchain protocols and blockchain programs reducing the cost of protocol messages and the cost of executing programs. For the latter, as execution costs money, we even have quantifiable savings. So my work gives evidence that formal reasoning is "worth it". I believe my work can inform future implementations of blockchains and distributed systems to reduce their resource consumption.

My thesis lives between academia and the blockchain world. In the first part of my work—*reasoning about blockchain protocols*—I am inspired by ideas from the blockchain community from openly accessible reports and give rigorous, formal proofs. In the second part of my work—*reasoning about blockchain programs*—I take ideas from the formal methods community and give optimised real-world programs written for the blockchain.

For *reasoning about blockchain protocols*, I started from the byzantine consensus protocol Stellar and from graph-based blockchain protocols, in particular the Blockmania protocol. Their correctness, and optimisations, were formally proved. We presented the findings in two conference publications: the *Conference on Principles of Distributed Systems* (OPODIS'19), and the *ACM Symposium on Principles of Distributed Computing* (PODC'21). For *reasoning about blockchain programs* I take the work on superoptimisation to programs written for the Ethereum blockchain gaining a large-scale data set for evaluation, but also allows quantifying monetary savings. The findings appeared in the *Preproceedings of the Symposium on Logic-based Program Synthesis and Transformation* (LOPSTR'19), the *Proceedings of the International Conference on Computer-Aided Verification* (CAV'20), and the *Proceedings of the Workshop on Formal Methods for Blockchains* (FMBC'20).

Three freely available prototype implementations were published under the Apache License 2.0 available on github: *(i)* ebso, the EVM bytecode superoptimiser, *(ii)* the backend of syrup, a SYnthesiseR of sUPer-optimised smart contracts, and *(iii)* ppltr, a populator for a peephole optimiser of a compiler.

The work on superoptimisation has partly inspired the research proposal[1] *GASOL: Gas Analysis and Optimization Toolkit* funded by the Ethereum foundation. I also generated benchmarks from our blockchain superoptimiser for the SMT community `smtlib.org`. Finally, I applied our work on generating peephole optimisation rules to the peephole optimiser of a fully verified compiler to Ethereum bytecode.

[1] *cf.* blog.ethereum.org/2021/07/01/esp-allocation-update-q1-2021

Acknowledgements

First and foremost I want to thank my supervisor George Danezis and my second supervisor David Pym. I also thank my previous supervisor Ilya Sergey. I would like to express my gratitude to Diego Marmsoler and Mehrnoosh Sadrzadeh—the examiners of my thesis for their time, care, and interest. This also extends to the examiners in my first-year and upgrade viva: Stefano Vissicchio and Marie Vasek. Next, I am incredibly grateful to my collaborators: Álvaro García-Pérez, Julian Nagele, Elvira Albert, Albert Rubio, Pablo Gordillo, Alejandro Hernández-Cerezo, and Vilhelm Sjöberg. I want to thank everyone in the UCL InfoSec and PPLV groups, but in particular the following (former) members: Alexandra Silva, Carsten Fuhs, James Brotherston, Simon Parkin, Nissy Sombatruang, Diana Costa, Sonia Marin, Estibaliz Fraca, Kareem Khazem, Simon Docherty, Thomas Sibut-Pinote, Thomas Cattermole, Mary Maller, Haaroon Yousaf, Tobias Kappé, Louis Parlant, Gerco van Heerdt, Tao Gu, Stefan Zetzsche, Luke Geeson, and Tiago Ferreira.

A special thank you to my internship hosts at Google: Jakub and Yana from the Android Google Search App team, and Ilya and Mitko from the Youtube team, as well as to my mentor Silviana. I am glad to have had Clément Pit-Claudel as such a great co-chair for the CAV Artifact Evaluation. I want to give my thanks to the SIGPLAN-M Mentorship programme and especially to my mentor Lindsey Kuper. I am grateful for the opportunity to teach at Queen Mary University London and in particular to Nikos Tzevelekos for his support. I also want to thank my master student Julius Rakenov. I give my thanks to University College London, and in particular in the administration to

Sarah Bentley, Dawn Bailey, Wendy Richards, and Julia Savage. My gratitude also goes to the institutions hosting me during research visits, the IMDEA Software Institute in Madrid and the Complutense University of Madrid, and for the grants for attending research events and summer schools, I received: the DeepSpec Summer School 2018, the Verified Software Workshop 2019, the Ninth Summer School on Formal Techniques 2019, and the Summer School Marktoberdorf 2019. Finally, I want to thank my friends and family.

Contents

Chapter 1

Introduction

Blockchain technologies are a young and fast moving field. With a lack of maturity come inefficiencies, as for example witnessed by under-optimised code executed on the blockchain [26, 19], or the use of consensus instead of less expensive primitives such as broadcast algorithms for cryptocurrencies [49]. The goal of this thesis is to remove inefficiencies and reduce the cost of running blockchain technologies. However, reducing cost usually comes at a price. First and foremost, engineers need to be certain that an optimisation is safe and does not introduce unwanted behaviour, *i.e.*, optimisations need to be sound. To maintain this high-assurance I rely on *formal reasoning*: transferring the correctness argument from the original to the optimisation. On the other hand, cost comes from the fact that engineers need to find opportunities for optimisations in the first place. To alleviate this I aim to make optimisations reusable and, if possible, find them automatically.

1.1 Research Hypothesis

My research hypothesis is:

By applying formal reasoning to blockchain technologies we can reduce execution costs while guaranteeing correctness [H].

I focus on reducing the cost in two settings, giving rise to the two parts of my thesis: maintaining the blockchain through *(I) blockchain protocols*, and the execution of *(II) blockchain programs*, *i.e.*, *smart contracts*.

In the first setting, *blockchain protocols*, participants communicate according to protocols sending and receiving messages to arrive at a shared state of the blockchain. My first goal is to reduce the number of messages which need to be exchanged—a bottleneck in a network—while preserving the correctness guarantees of the original protocol. Thus, my first sub-hypothesis is:

By applying formal reasoning to communication protocols we can reduce the number of exchanged messages while guaranteeing correctness $[H_a]$.

Here, I look at two specific instances: we define and show correctness of an abstract version of the Stellar [75] protocol relying on infinitely many messages and refine it to an implementation exchanging only finitely many messages, but maintaining correctness, and we develop a framework to embed the run of a deterministic protocol in a block graph thereby reducing the messages exchanged. For the second setting, *blockchain programs*, I look at executing a smart contract, which is subject to monetary fees as an incentive against wasting resources. These fees also give a clear cost model and thus a clear optimisation target. This yields my second sub-hypothesis:

By applying formal reasoning to smart contracts we can reduce the monetary fees of their execution while guaranteeing correctness $[H_b]$.

Here, I superoptimise [74] smart contracts employing a a constraint solver to automatically find cheaper, but observationally equivalent programs. I present three approaches, which are evaluated on the Ethereum blockchain with Ethereum's virtual machine (EVM): superoptimising EVM bytecode with the constraint solver Z3 [32] implementing basic and unbounded superoptimisation [55], synthesising superoptimised EVM bytecode with Max-SMT solvers, and automatically generating peephole optimisation rules for improving a smart contract compiler.

1.2 Summary of Chapters

Next I briefly summarize the goals and results of my five main results:

Chapter 3: Correctness of a Federated Consensus Protocol. In this chapter we prove the Stellar Consensus Protocol (SCP) [75] correct. We propose an abstract version of SCP that uses Stellar's federated voting primitive as a black box. This abstract protocol, however, maintains infinite state possibly sending infinitely many messages. By establishing a refinement between the abstract protocol and a concrete implementation of SCP that uses only finite state, we are carrying over the result about the correctness while reducing the needed resources.

The contents of this chapter appeared in the *Proceedings of the 23rd International Conference on Principles of Distributed Systems (OPODIS'19)* [44] and an extended version is available on arXiv [45].

Chapter 4: Embedding A Protocol in a Block DAG. In this chapter we give a generic framework to embed a deterministic byzantine fault tolerant protocol $\mathcal{P}$ into a block DAG, and employ the block DAG to locally replay interactions between servers. We prove that our embedding maintains all safety and liveness properties of $\mathcal{P}$, while efficiently compressing messages by utilising the determinism of $\mathcal{P}$ and running many parallel instances of $\mathcal{P}$ essentially "for free". Our main insight is that a block DAG is essentially a reliable point-to-point channel encoding Lamport's happened-before relation [62].

The contents of this chapter appeared in the *ACM Symposium on Principles of Distributed Computing (PODC'21)* [98] and is available on arXiv [99].

Chapter 6: Blockchain Superoptimiser. In this chapter we optimise the bytecode of the EVM through superoptimisation by encoding the operational semantics of EVM instructions as SMT formulas and leveraging the constraint solver Z3 to automatically find a cheaper bytecode. We present the EVM bytecode superoptimiser ebso[1] implementing basic and unbounded superoptimisation [55] and evaluate ebso on smart contracts from a programming competition aimed at producing the cheapest EVM bytecode. Even in this already highly optimised data set ebso still finds 19 optimisations and proves that around

[1]available at github.com/juliannagele/ebso

17 % of the analysed instruction sequences are already optimal. Furthermore we evaluate ebso on the 2500 most called smart contracts from the Ethereum blockchain and find that, in our setting, unbounded superoptimisation outperforms basic superoptimisation.

The contents of this chapter have been accepted for presentation and appeared in the pre-proceedings of the *29th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'19)* [84].

Chapter 7: Synthesising Optimisations. We aim to reduce the gas cost of a smart contract by synthesizing optimised basic blocks with an efficient Max-SMT encoding. Our approach and prototype implementation syrup[2] outperforms ebso by two orders of magnitude in gas savings and reduces the time-outs from more than 90 % to less than 9 %. In this approach, we first extract a stack functional specification from the basic blocks of a smart contract, which is simplified using rules that capture the semantics of the arithmetic and bit-wise operations. From this we synthesize optimised blocks by an efficient Max-SMT encoding. The efficiency is gained by avoiding the encoding of semantics of the arithmetic and bit-wise operations and consequently an expensive ∃∀-quantification, and expressing the problem as a Max-SMT instead of an SMT problem.

The contents of this chapter appeared in the *Proceedings of the 32nd International Conference on Computer-Aided Verification (CAV'20)* [5].

Chapter 8: Populating a Peephole Optimiser. In this chapter we find optimisation rules for a peephole optimiser of a smart contract compiler. These rules are normally constructed using human expertise, which is time-consuming and far from systematic in exploring opportunities for optimisation. We propose a pipeline to automatically generate peephole optimisation rules and generate nearly $1k$ optimisation rules for the bytecode of the EVM. Our rules reduce nearly $150k$ instructions from the 1000 most called contracts on the Ethereum blockchain. Assuming that 10 % of the bytecode of a contract is executed

[2]backend available at github.com/mariaschett/syrup-backend

per call and that savings are uniformly distributed for an average gas price of 27.6 gwei and an average ETH-USD course of \$200.62 this translates to savings of more then \$55 000. The superfluous instructions also currently waste 4.5 % of storage space. Our proposed pipeline applies superoptimisation to an existing code base to obtain optimisations from which we generate peephole optimisation rules by extracting their underlying patterns. We perform a case study for the Ethereum blockchain and provide a prototype implementation with the tool ppltr[3], which combines the superoptimiser ebso and the rule generator sorg[4].

The contents of this chapter appeared in the *Proceedings of the 2nd Workshop on Formal Methods for Blockchains (FMBC'20)* [100].

1.3 Collaboration

Chapter 3 is a collaboration with *Álvaro García-Pérez* then at the IMDEA Software Institute. My main contributions are towards *(i)* developing the pseudo-code of Algorithm 3 and Algorithm 5, and the *(ii)* proof of refinement in Section 3.3. In [44] we also prove weak validity and termination of Algorithm 3, as well as implications of servers lying about their initial position. As I have made no major contribution to this part of the paper, I have omitted these in my thesis. Chapter 4 is the joint work with my supervisor *George Danezis* at University College London. I was the leading author of the paper. Chapter 6 and Chapter 8 are the joint work with *Julian Nagele* then at Queen Mary University of London. We both contributed equally to the papers and prototypes. Chapter 7 is the joint work with *Elvira Albert* and *Albert Rubio* from the Instituto de Tecnología del Conocimiento, and the Complutense University of Madrid, and *Pablo Gordillo* from Complutense University of Madrid. My main contributions are towards *(i)* the SMT encoding in Section 7.3 and the corresponding implementation of the syrup backend, and *(ii)* the experiments in Section 7.4.

---

[3] available at github.com/mariaschett/ppltr

[4] available at github.com/mariaschett/sorg

1.4 Complementary Activities

In this final section I will briefly present activities complementary to the work in this thesis: my *teaching experience*, *research presentations*, *research events*, *community service*, and *industrial experience*.

I gained *teaching experience* by delivering the course *ECS652U Compilers* as a teaching fellow at Queen Mary University of London in the summer term 2020 based on Alex Aiken's Cool programming language [1], supervising the master thesis of *Julius Rakenov* based on Chapter 6, accomplished with distinction, titled *"Superoptimising WebAssembly"* in 2019 at UCL, and being a teaching assistant at UCL for *COMP0003: Theory of Computation*, *COMP0017: Computability and Complexity Theory*, and *ENGS102P: Design and Professional Skills: CS Scenario* based on [92]. I gave *research presentations* of the work in Chapter 4 at the *ACM Symposium on Principles of Distributed Computing (PODC'21)*[5], and the work in Chapter 8 at the *Second Workshop on Formal Methods for Blockchains (FMBC'20)*[6]. I presented the work of Chapter 6 at the IMDEA *Software Institute* in Madrid, the *29th International Symposium on Logic-Based Program Synthesis and Transformation* in Porto, and the *Complutense University of Madrid*. Furthermore I presented a research abstract *Blockmania QED* at the *Doctoral Symposium* at *Formal Methods 2019* in Porto, and a summary of my research at the *Research Spotlight Competition* of the *London Hopper Colloquium 2020*. I was fortunate to have been part of several *research events*: I was invited to the *Google Compiler and Programming Language Summit 2019* in Munich where I presented my poster *BFT Protocols through a Joint Block DAG*, and I received a travel grant to visit the *Verified Software Workshop 2019* in Cambridge, where I presented my poster *Formal Methods & The Blockchain*. I received a grant to attend the *DeepSpec Summer School 2018* in Princeton and I also attended the *Ninth Summer School on Formal Techniques 2019* in Menlo College in California, and the *Summer School Marktoberdorf 2019* in Germany. For *community service*, I co-

[5] recording available at youtu.be/zO1ENRsOViQ
[6] recording available at youtu.be/uXJKcf68vZs

chaired the *CAV Artifact Evaluation Committee 2021* together with Clément Pit-Claudel. I gained experience in reviewing as a sub-reviewer for the *39th IEEE International Conference on Distributed Computing Systems 2019*. In two summer internship at Google on the *Android Google Search App* and at *Youtube*, I gained software development and *industrial experience*.

Part I

Blockchain Protocols

Chapter 2

Background: Protocols

In this chapter I first give the necessary background on blockchains. Next I state the system model as the basis for Chapter 4 and Chapter 3, and similarly cover relevant protocols in the next section.

2.1 Blockchains

As noted by Narayanan *et al.* [87, p.20], the term *blockchain* "has no standard technical definition but is a loose umbrella term [*referring to*] systems that bear varying levels of resemblance to Bitcoin and its ledger". Neither will I attempt a technical definition, but I will introduce common aspects most relevant for my thesis: blockchains as *data structures*, as mechanisms to govern shared state, and with respect to their *history and applications*.

Data Structures. One way to look at a blockchain is as a data structure replicated in a distributed system. A blockchain is simply a *chain* of *blocks*. Or, extended to a directed, acyclic graph: a *block DAG*. In either case the imposed order is crucial: every block, except for the starting blocks—the *genesis blocks*—contains a reference to one or more predecessor blocks similar to hash chains or Merkle trees [80]. The reference is a *cryptographic hash*, *cf.* Definition 2.2.1, which renders it computationally infeasible to tamper with predecessor blocks making them effectively append-only or *immutable*. Every block contains a set of *transactions*, or even *programs*, changing the state of the blockchain.

Governance of Shared State. The goal of the replicated blockchain is to maintain shared state, *i.e.*, to implement state machine replication [102]. For example, for cryptocurrencies this shared state is keeping track of assets, *e.g.*, of bitcoins in Bitcoin [85]. Servers maintain the blockchain and clients issue transactions. However clients, and servers, may be malicious and act adversarial. A famous example is the *double-spend* attack, where a client tries to spend their assets twice. To counteract this the non-malicious servers have to agree on a chronological order of transactions: they have to find consensus. However, a majority-based consensus through voting is impossible without a one-to-one correspondence between servers and identities: an adversarial server may vote with more than one identity, *i.e.*, "they" launch a Sybil attack [33]. One way to validate identities is to pose resource-demanding challenges, *e.g.*, participants to give a *proof-of-work* [35, 33]. Then the blockchain has *open membership* or is *permissionless* as no central authority has to guarantee for the one-to-one correspondence between servers and identities as in *permissioned blockchains* [24]. Proof-of-work combined with an incentive system to reward good behaviour enables consensus [87]. Servers get paid to maintain the blockchain—but the payment is forfeited when they misbehave. Clients, on the other hand, pay this fee—which will be important in Chapter 6–Chapter 8, where this payment is a clear optimisation target. Proof-of-work is, by its nature, expensive—and thus several alternatives such as proof-of-stake, or more generally proof-of-X, have been suggested, see *e.g.*, Bano *et al.* [11] for a review and classification of consensus algorithms. Broadly, they can be classified as *(i)* "Nakamoto" or "blockchain" consensus, and *(ii)* "classical" consensus drawing from the distributed systems literature.

History and Applications. Next I present blockchain projects which are most prominent or relevant for this thesis; for an overview see *e.g.* [87]. The first popular blockchain project in 2008 is Bitcoin, published by Satoshi Nakamoto [85] promising a fully decentralized medium of exchange through proof-of-work. In 2012 and 2014, Ripple [103] and Stellar [75] (*cf.* Chapter 3)

1. *Reliable delivery.* If a correct server sends a message m to a correct server s, then s eventually delivers m.
2. *No duplication.* No message is delivered by a correct server more than once.
3. *Authenticity.* If some correct server s_2 delivers a message m with sender s_1 and s_1 is correct, then m was previously sent to s_2 by s_1.

Figure 2.1: Authenticated perfect point-to-point link after [23, Module 2.5].

proposed substituting the expensive proof-of-work consensus by a consensus mechanism based on trust between participants. Recently, also blockchains based on block DAGs have evolved—with impressive efficiency gains. Most important for my thesis are Hashgraph [10], Blockmania [30], Aleph [41], and Flare [96]. An overview of block DAG systems can be found in the SoK of Wang *et al.* [110].

2.2 System Model

This section introduces the model for the protocols in Chapter 4 and Chapter 3. A *distributed system* has a set of independent *servers* Srvrs which are connected by a *network* communicating via message passing according to a prescribed *protocol* to handle *requests* from *clients*. The network may be *asynchronous*, with no upper-bound on message transmission delays, *synchronous*, with a known upper-bound, or *partially synchronous* [34], that is synchronous after a global stabilization time. In Chapter 3 we assume a partially synchronous upper bound, in Chapter 4 the synchronicity assumptions depend on the protocol to be interpreted. Synchronicity assumptions are needed to circumvent known impossibilities such as the *FLP theorem* [37] and the *CAP theorem* [40, 46].

Between servers we assume an authenticated perfect point-to-point links after Cachin *et al.* [23, Module 2.5, page 42].

For secure communication, we assume the following cryptographic primitives: *secure cryptographic hash functions* [79, p.332], and a *digital signatures* *scheme.*

Definition 2.2.1 *Let* $\# : A \to A'$ *be a secure cryptographic hash function. We write* $\#(x)$ *for the hash of* $x \in A$*, and* $\#(A)$ *for the image of* A *under* $\#$*. By definition for any* $\#$ *it is computationally infeasible* (1) *to find any preimage* m *such that* $\#(m) = x$ *when given any* x *for which a corresponding input is not known* (preimage-resistance)*,* (2) *given* m *to find a second preimage* $m' \neq m$ *such that* $\#(m) = \#(m')$ (second-preimage resistance)*, and* (3) *to find any two distinct inputs* m*,* m' *such that* $\#(m) = \#(m')$ (collision resistance)*.*

Every server s, and every client, can prove with their *public/private key* pair *(1)* that they signed a message m by invoking $\mathsf{sign}(s, m) = \sigma$ on a message m and obtaining a *signature* σ, then *(2)* any server or client can verify with $\mathsf{verifysign}(s, m, \sigma)$ and s's public key, that s signed m. We tacitly assume that keys are generated and distributed to all servers and clients.

In in Chapter 4 and Chapter 3 our *threat model* is a *byzantine failure model*[1]. We distinguish: a *correct* server or client follows the protocol, including not stopping indefinitely. All other servers are *faulty*: a *fail-stop* server or client may stop indefinitely, and a *byzantine* server or client can deviate arbitrarily from the protocol, including adversarial coordinating with other byzantine servers and clients. Not only the servers and clients, but also the *network may fail*: messages might arrive at different times at different servers or clients, out of order, may be lost, or arrive multiple times. Finally, we assume that any *attacker* has limited computational power, cannot break cryptographic assumptions, and has limited (monetary) funds.

2.3 Protocols

We focus on protocols particularly relevant in the context of blockchains: *broadcast protocols*, where a server broadcasts a value to every other server, and *consensus protocols*, where the servers agree on a value. However, before

[1] The term “byzantine” was coined by Lamport [64] with an allegory on byzantine generals planning to attack a city—with undetected traitors among them.

1. *Validity.* If a correct server s broadcasts a value v, then every correct server eventually delivers v.
2. *No duplication.* Every correct server delivers at most one value.
3. *Integrity.* If some correct server delivers a value v with sender s and server s is correct, then v was previously broadcast by s.
4. *Consistency.* If some correct servers delivers a value v and another correct server delivers a value v', then $v = v'$.
5. *Totality.* If some value is delivered by any correct server, every correct server eventually delivers a value.

Figure 2.2: Byzantine reliable broadcast abstraction after [23, Module 3.12].

giving details on these protocols, we first investigate what it means for the implementation of a protocol to be "correct": an implementation of a protocol is correct, if it guarantees the properties defined by the interface of the protocol, e.g., in Figure 2.2 for byzantine reliable broadcast. These properties can be classified as *safety* and *liveness* properties. A protocol is safe when "no bad things happen", and live when "good things happen eventually" [7, 23]. For example, in a safe protocol, servers will not decide on two different values, and in a live protocol, servers will eventually decide on a value.

Recall that our protocols have to be correct even in the presence of byzantine servers. One way for protocols to be resilient to byzantine servers is through *quorums* in byzantine quorum systems [72, 109]. We require *(i) quorum intersection*: for any two quorums Q_1 and Q_2, their intersection $Q_1 \cap Q_2$ contains at least one correct server, and *(ii) quorum availability*: ensuring at least one quorum with only correct servers. For *threshold quorums* with f faulty servers we need at least $2f + 1$ servers to tolerate fail-stop servers, and $3f + 1$ servers to tolerate byzantine servers [64]. To give an intuition for fail-stop servers: if $f + 1$ servers are in quorums Q_1 and Q_2, every intersection of quorums $Q_1 \cap Q_2$ contains at least one correct server. The idea underlying Stellar trust-based quorums are byzantine quorum systems. Moreover Stellar's *"federated voting"* abstraction implements byzantine reliable broadcast [42].

Algorithm 1: Authenticated double-echo broadcast.

```
1  process broadcast(s ∈ Srvrs)
2      echoed, readied, delivered := false ∈ Bool
3      broadcast(v ∈ Vals)
4          echoed := true
5          send ECHO v to every s′ ∈ Srvrs
6      when received ECHO v and not echoed
7          echoed := true
8          send ECHO v to every s′ ∈ Srvrs
9      when received ECHO v from 2f + 1 different s′ ∈ Srvrs and not
        readied
10         readied := true
11         send READY v to every s′ ∈ Srvrs
12     when received READY v from f + 1 different s′ ∈ Srvrs and not
        readied
13         readied := true
14         send READY v to every s′ ∈ Srvrs
15     when received READY v from 2f + 1 different s′ ∈ Srvrs and
        not delivered
16         delivered := true
17         deliver(r)
```

Reliable Broadcast. An algorithm implements byzantine reliable broadcast, if it implements the byzantine reliable broadcast abstraction in Figure 2.2. In Algorithm 1 we show an implementation: authenticated double-echo broadcast [23, Algorithm 3.18] originated from Bracha [17]. The proof that it implements the abstraction can be found in [23]. Next we give a brief overview on the mechanisms of 1, and introduce the pseudo-code notation. The notation is based on Cachin *et al.* [23] and used in the algorithms in Chapter 3 and Chapter 4.

To execute the protocol every server s in Srvrs needs to start a **process** instance (line 1). Every protocol holds some internal state (line 2) as well as several handlers which s executes whenever the corresponding condition holds (*e.g.*, line 9–11). The *interface* to the client is the *request* broadcast(v), called by the client, for a value v at any one $s \in$ Srvrs (line 3), and the *indication* deliver(v) when s broadcasted v (line 17). We write requests and indications, as well as internal state in sans-serif. The servers communicate relying on authenticated point-to-point channels [23] to **send** messages and update their state based on **received** messages. Keywords in our language are written in **bold-face**. The messages, written in `true-text`, are either `ECHO` v or `READY` v.

The state echoed, readied, delivered (line 2), and the received messages—here left implicit—to determine *e.g.* **received** `ECHO` v from $2f + 1$ different $s' \in$ Srvrs (line 6). The protocol has two communication rounds. In the first round every correct server echoes v to every server (lines 6–8). In the second round: every correct server either received `ECHO` v from a quorum, *i.e.* $2f + 1$ servers, and can thus send `READY` v to every server (lines 9–11), or after receiving `READY` v from at least one correct node, *i.e.* $f + 1$, sends `READY` v to every server (lines 12–14). Through this case, a correct server, which cannot get a quorum of `ECHO` v, can still (safely) send `READY` v. After receiving `READY` v from a quorum the protocol ends a correct server by invoking deliver(v).

Byzantine Consensus. For a blockchain—in spite of byzantine participants—every honest participant should eventually "agree on the state", *i.e.*, they reach *consensus*. Now while byzantine consensus is overkill for money transfer, it is not clear if the same is true for executing smart contracts [49, 9]—the results rely on the protocol transferring money, but smart contracts change the state in more ways. With the rise of blockchain technologies many consensus algorithms have been proposed with slightly different formulations and notions of consensus. Most agree that consensus requires some form of *(i)* agreement *(ii)* non-triviality, and *(iii)* termination [105]. In the context of this thesis I rely on a notion from classical consensus by Cachin *et al.* [23]: a protocol implements consensus in a byzantine environment, if it implements the consensus abstraction in Figure 2.3. To give an intuition on the implementation of a consensus protocol: for consensus in the *fail-stop* scenario there is Paxos [63] or view-stamped replication [88]. On a very high level they have a sequence of views with a leader responsible for progress in deciding on a value; if the

1. *Termination.* Every correct server eventually decides on some value.
2. *Integrity.* No correct server decides twice.
3. *Agreement.* No two correct servers decide differently.
4. *Weak validity.* If all servers are correct and propose the same value v, then no correct server decides a value different from v; furthermore, if all servers are correct and some server decides v, then v was proposed by some server.

Figure 2.3: Weak byzantine consensus after [23, Module 5.10].

others suspect the leader failed, they trigger a view change and safely replace the current leader. For the *byzantine* scenario the most famous protocol is the Practical Byzantine Fault Tolerance (PBFT) [25] protocol. It is an extension of the previous view change protocol, requiring an additional communication round.

Chapter 3

Correctness of a Federated Consensus Protocol

In this chapter we initially define an abstract version of the Stellar consensus protocol [75] with a high cost as a trade-off for simplicity. This abstract protocol is infeasible, but warrants an easier proof of correctness. With this proof of correctness, we then show a refinement to a more complex, but lower cost solution: the *concrete protocol*. We then establish that the concrete protocol refines the abstract one, which reduces the number of exchanged messages while guaranteeing correctness [H_a].

Blockchain proposals, such as Stellar and Ripple [103], allow for open membership using quorum-like structures typical for classical byzantine consensus with closed membership. This is achieved by constructing quorums in a decentralised way: each server independently chooses whom to trust, and quorums arise from these individual decisions. In particular, in Stellar trust assumptions are specified using a *federated byzantine quorum system* (FBQS), where each server chooses a set of servers such that each of the chosen servers would convince the choosing server to accept the validity of a given statement. Consensus is then implemented by a fairly intricate protocol whose key component is *federated voting*—a protocol similar to Bracha's protocol in Algorithm 1 for reliable byzantine broadcast [17, 23]. In this chapter we rigorously define the Stellar Consensus Protocol (SCP) and prove it correct. Our proof to gives insights into its structure and its use of lower-level abstractions. Stellar's unique and novel way of constructing quorums—together with Stellar's role as a widely used protocol in practice—the original white paper [75] inspired several works developed alongside our work investigating Stellar's mechanisms, giving correctness arguments, and linking Stellar to established concepts in literature [42, 69, 66, 68].

3.1 Federated Voting

We consider a system consisting of a finite set of servers Srvrs with byzantine servers, and any two servers can communicate over an authenticated perfect point-to-point link and a partially synchronous network. A *federated byzantine quorum system* (FBQS) [75, 42] is a function $\mathcal{F} : \mathsf{Srvrs} \to 2^{2^{\mathsf{Srvrs}}} \setminus \{\emptyset\}$ that specifies a non-empty set of *quorum slices* for each server. We require that a server is part of every one of its own quorum slices q: $\forall s \in \mathsf{Srvrs}\ \forall q \in \mathcal{F}(s), s \in q$. Quorum slices reflect the trust choices of each server. A non-empty set of servers $U \subseteq \mathsf{Srvrs}$ is a *quorum* in an FBQS $\mathcal{F}$ iff U contains a slice for each member, *i.e.*, $\forall s \in U\ \exists q \in \mathcal{F}(s), q \subseteq U$. In this chapter we assume for simplicity that servers do not equivocate about their quorum slices, so that all the servers share the same FBQS. However, a more realistic *subjective* FBQS [42] is considered in our conference proceedings [44], where byzantine servers may lie about their quorum slices and different servers have different views on the FBQS.

A consensus protocol that runs on top of an FBQS may not guarantee global agreement, because when servers choose slices independently, only a subset of the servers may take part in a subsystem in which every two quorums intersect in at least one correct server—a basic requirement of a byzantine quorum system [72] to ensure agreement. To formalise which parts of the system may reach agreement internally, we borrow the notions of *intertwined servers* and *intact set* from [76]. Two servers s_1 and s_2 are *intertwined* iff they are correct and every quorum containing s_1 intersects every quorum containing s_2 in at least one correct server. Consider an FBQS $\mathcal{F}$ and a set of servers I. The *projection* $\mathcal{F}|_I$ of $\mathcal{F}$ to I is the FBQS over set I given by $\mathcal{F}|_I(s) = \{q \cap I \mid q \in \mathcal{F}(s)\}$. For a given set of faulty servers, a set I is an *intact set* iff I is a quorum in $\mathcal{F}$ and every member of I is intertwined with each other in the

Algorithm 2: Federated voting with quorums $\mathcal{Q}$.

process federated-voting($s \in \mathsf{Srvrs}, t \in \mathsf{Tag}$)
  voted, ready, delivered $:=$ *false* $\in$ Bool
  vote($a \in \mathsf{A}$)
    if not voted **then**
      voted $:=$ *true*
      send `VOTE`(t, a) **to every** $s' \in \mathsf{Srvrs}$;

  when received `VOTE`(t, a) **from every** $u \in U$ **for some** $U \in \mathcal{Q}$ **such that** $s \in U$ **and not** ready
    ready $:=$ *true*
    send `READY`(t, a) **to every** $s' \in \mathsf{Srvrs}$;

  when received `READY`(t, a) **from every** $u \in B$ **for some** s**-blocking** B **and not** ready
    ready $:=$ *true*
    send `READY`(t, a) **to every** $s' \in \mathsf{Srvrs}$;

  when received `READY`(t, a) **from every** $u \in U$ **for some** $U \in \mathcal{Q}$ **such that** $s \in U$ **and not** delivered
    delivered $:=$ *true*
    deliver(a);

projected FBQS $\mathcal{F}|_I$. The intact sets characterise those sets of servers that can reach consensus. Crucial for showing consensus are quorums intersection and disjoint maximal intact sets in the following lemma. The proof of the lemma can be found in [45, Lemma 3 and 4].

Lemma 3.1.1 *For intact sets I, I_1, and I_2 in an FBQS $\mathcal{F}$ holds* (i) *if any two quorums U_1 and U_2 in $\mathcal{F}$ such that $U_1 \cap I \neq \emptyset$ and $U_2 \cap I \neq \emptyset$ then the intersection $U_1 \cap U_2$ contains some server in I, and* (ii) *if $I_1 \cap I_2 \neq \emptyset$ then $I_1 \cup I_2$ is an intact set in $\mathcal{F}$.*

One of the core components of the abstract consensus protocol in is *federating voting* (*FV*) [75, 76] shown in Algorithm 2 corresponding to *Stellar broadcast* in [42]. For FV we have a set of *voting values* A. FV allows each correct server to vote for some $a \in \mathsf{A}$ through an invocation vote(a), and each server may deliver some $a' \in \mathsf{A}$ through an indication deliver(a'). Our consensus protocol uses multiple instances of FV independently from each other.

Given a maximal intact set I.

1. *Validity for intact sets.* If all servers in I vote for a, then all servers in I eventually deliver a.
2. *No duplication.* Every correct server delivers at most one voting value.
3. *Consistency for intertwined servers.* If two intertwined servers s and s' deliver a and a' respectively, then $a = a'$.
4. *Totality for intact sets.* If a server in I delivers a voting value, then every server in I eventually delivers a voting value.

Figure 3.1: Reliable byzantine voting for intact sets.

Each instance of FV is identified by a *tag* t from a set of tags Tag. Each server s runs a process federated-voting(s, t) for each tag t and also the exchanged messages are tagged with t. FV adapts Algorithm 1 to the federated setting of an FBQS: first, in lines 7–9, two servers in the same intact set I cannot send `READY` messages with two different voting values, because this would require two quorums of `VOTE` messages and these quorums would intersect in a correct server in I after Lemma 3.1.1 *(i)*. Second, lines 10–12 allow a server to send a `READY` message even if it previously voted and uses the notion of *s-blocking set* [75] for liveness guarantees. Given a server s, a set B is *s-blocking* iff B overlaps each of s's slices, *i.e.*, $\forall q \in \mathcal{F}(s),\ q \cap B \neq \emptyset$.

Lemma 3.1.2 *For an intact set I in an FBQS $\mathcal{F}$ and $s \in I$ holds no s-blocking set B exists such that $B \cap I = \emptyset$.*

Proof 3.1.1 *Since I is a quorum in $\mathcal{F}$ and by the definition of quorum, for every server $s \in I$ there exists one slice of s that lies within I.*

If s is in an intact set I, the following lemma guarantees that if s sends a `READY` message it has received `VOTE` from a quorum to which s belongs. The lemma is analogous to [43, Lemma 36].

Lemma 3.1.3 *For an FBQS $\mathcal{F}$, a tag t, and an intact set I in $\mathcal{F}$, consider an execution of the instance for t of FV over $\mathcal{F}$. The first server $s \in I$ that* *sends a* $\texttt{READY}(t, a)$ *message first needs to receive a* $\texttt{VOTE}(t, a)$ *message from every member of a quorum* U *to which* s *belongs.*

Proof 3.1.2 *Let* s *be any server in* I*. By Lemma 3.1.2 no* s*-blocking set* B *exists such that* $B \cap I = \emptyset$*. Therefore, the first server* $s \in I$ *that sends a* $\texttt{READY}(t, a)$ *message does it through lines 7–9 of Algorithm 2, which means that* s *received* $\texttt{VOTE}(t, a)$ *messages from every member of a quorum* U *to which* s *belongs.*

FV satisfies the specification of reliable byzantine voting for intact sets in Figure 3.1. Again, the specification is similar to Figure 2.2. The full proof can be found in [45].

Theorem 3.1.1 *Let* $\mathcal{F}$ *be an FBQS. The instance FV over* $\mathcal{F}$ *satisfies the specification of reliable byzantine voting for intact sets in Figure 3.1.*

3.2 Abstract Stellar Consensus Protocol

Assume a set Val of *consensus values.* Each correct server proposes some $x \in \mathsf{Val}$ through an invocation $\mathsf{propose}(x)$, and each server may decide some $x' \in \mathsf{Val}$ through an indication $\mathsf{decide}(x')$. We consider a variant of the *weak byzantine consensus* specification in Figure 2.3 after [23] that we call *non-blocking byzantine consensus for intact sets*, which is defined as in Figure 3.2.

First we introduce the *abstract SCP* (ASCP), which concisely specifies the mechanism of SCP [75, 76] and highlights the modular structure present in it[1]. Like Paxos [63], ASCP uses *ballots*—pairs $\langle n, x\rangle$, where $n \in \mathbb{N}^+$ a natural positive *round number* and $x \in \mathsf{Val}$ a *consensus value.* We assume that Val is totally ordered, and we consider a special *null ballot* $\langle 0, \perp\rangle$, where $\perp \notin \mathsf{Val}$. Let $\mathsf{Ballot} = (\mathbb{N}^+ \times \mathsf{Val}) \cup \{\langle 0, \perp\rangle\}$ be the set of ballots. We write $b.n$ and $b.x$ respectively for the round and consensus value of ballot b. The set Ballot is totally ordered, where we let $b < b'$ iff either $b.n < b'.n$, or $b.n = b'.n$

[1] More precisely, in this paper we focus on Stellar's core *ballot* protocol, which aims to achieve consensus. We abstract from Stellar's *nomination* protocol—which tries to converge (best-effort) on a value to propose—by assuming arbitrary proposals to consensus.

Given a maximal intact set I.

1. *Integrity.* No correct server decides twice.
2. *Agreement for intact sets.* No two servers in I decide differently.
3. *Weak validity for intact sets.* If all servers are honest and every server proposes x, then no server in I decides a consensus value different from x; furthermore, if all servers are honest and some server in I decides x, then x was proposed by some server.
4. *Non-blocking for intact sets.* If a server s in I has not yet decided in some run of the protocol, then for every continuation of that run in which all the malicious servers stop, server s eventually decides some consensus value.

Figure 3.2: Weak byzantine consensus for intact sets.

and $b.x < b'.x$. We define the *below-and-incompatible-than* relation on ballots. We say ballots b and b' are *compatible* (written $b \sim b'$) iff $b.x = b'.x$, and *incompatible* (written $b \not\sim b'$) otherwise, where we let $\bot \neq x$ for any $x \in \mathsf{Val}$. We say ballot b is *below and incompatible than* ballot b' (written $b \lesssim\!\!\!\!\not\;\; b'$) iff $b < b'$ and $b \not\sim b'$.

To better convey SCP's mechanism, we let the abstract protocol in Algorithm 3 use FV as a black box where servers may hold a binary vote on each of the ballots: the voting values A are Booleans and Tag is the set of ballots, *i.e.*, the protocol considers a separate instance of FV for each ballot. A server voting for a Boolean a for a ballot b that carries the consensus value $b.x$ encodes the aim to either *abort* the ballot (when $a = \mathit{false}$) or to *commit* it (when $a = \mathit{true}$) thus deciding the consensus value $b.x$. We have dubbed ASCP 'abstract' because, although it specifies the protocol concisely, it is unsuited for realistic implementations. On the one hand, each server s maintains infinite state, because it stores a process $\mathsf{federated\text{-}voting}(s, b)$ for each of the infinitely many ballots b in the array $\mathsf{ballots}$ (line 2 of Algorithm 3). On the other hand, each server s may need to send or receive an infinite number of messages in order to progress (lines 6, 8, 15 and 21 of Algorithm 3, which are explained in the detailed description of ASCP below). This is done by assuming a *batched* *network semantics* (BNS) in which the network exchanges *batches*, which are (possibly infinite) sequences of messages, instead of exchanging individual messages: the sequence of messages to be sent by a server when processing an event is batched per recipient, and each batch is sent at once after the atomic processing of the event; once a batch is received, the recipient server atomically processes all the messages in the batch in sequential order. By convention, we let the statement **forall** in lines 7 and 21 of Algorithm 3 consider the ballots b' in ascending ballot order. In Section 3.3 we introduce a 'concrete' version of SCP that is amenable to implementation, since servers in it maintain finite state and exchange a finite number of messages; however, this version does not use FV as a black box.

In a nutshell, ASCP works as follows: each server uses FV to *prepare* a ballot b which carries the candidate value $b.x$, this is, it aborts every ballot $b' \lesssim\!\!\!\!\!\not\,\,\, b$, which prevents any attempt to decide a value different from $b.x$ at a round smaller than $b.n$; once b is prepared, the server uses FV again to commit ballot b, thus deciding the candidate value $b.x$. For Algorithm 3, we assume that each server$s$ creates a process $\mathsf{federated\text{-}voting}(s, b)$ for each ballot b, which is stored in the infinite array $\mathsf{ballots}[b]$ (line 2). The server keeps fields $\mathsf{candidate}$ and $\mathsf{prepared}$, which respectively contain the ballot that s is trying to commit and the highest ballot prepared so far. Both $\mathsf{candidate}$ and $\mathsf{prepared}$ are initialised to the null ballot (line 3). The server also keeps a field round that contains the current round, initialised to 0 (line 4). Once s proposes a value x, the server assigns the ballot $\langle 1, x\rangle$ to $\mathsf{candidate}$ and tries to prepare it by invoking FV's primitive $\mathsf{vote}(\mathit{false})$ for each ballot below and incompatible than $\mathsf{candidate}$ (lines 5–7). This may involve sending an infinite number of messages, which by BNS requires sending finitely many batches. Once s prepares some ballot b by receiving FV's indication $\mathsf{deliver}(\mathit{false})$ for every ballot below and incompatible than b, and if b exceeds $\mathsf{prepared}$, the server updates $\mathsf{prepared}$ to b (lines 8–9). The condition in line 8 may concern an infinite number of ballots, but it may hold after receiving a finite number

Algorithm 3: Abstract SCP with quorums $\mathcal{Q}$.

1 **process** abstract-consensus($s \in$ Srvrs)
2   ballots $:=$ [**new process** federated-voting$(s, b)]^{b \in \mathsf{Ballot}}$
3   candidate, prepared $:= \langle 0, \bot \rangle \in$ Ballot
4   round $:= 0 \in \mathbb{N}^+ \cup \{0\}$
5   propose(x)
6   candidate $:= \langle 1, x \rangle$
7   **for all** $b' \lesssim$ candidate **do** ballots$[b']$.vote(false)
8   **when** ballots$[b']$.deliver(false) **for every** $b' \lesssim b$ **and** prepared $< b$
9   prepared $:= b$
10   **if** candidate $\leqslant$ prepared **then**
11   candidate $:=$ prepared
12   ballots[candidate].vote(true)
13   **when** ballots$[b]$.deliver(true)
14   decide$(b.x)$;
15   **when exists** $U \in \mathcal{Q}$ **such that** $s \in U$ **and for each** $u \in U$ **exist** $\mathtt{M}_u \in \{\mathtt{VOTE}, \mathtt{READY}\}$ **and** $b_u \in$ Ballot **such that** round $< b_u.n$ **and either received** $\mathtt{M}_u(b_u, \mathit{true})$ **from** u **or received** $\mathtt{M}_u(b', \mathit{false})$ **from** u **for every** $b' \in [z_u, b_u)$ **with** $z_u < b_u$
16   round $:= \min\{b_u.n \mid u \in U\}$
17   start-timer$(F(\mathsf{round}))$
18   **when** timeout
19   **if** prepared $= \langle 0, \bot \rangle$ **then** candidate $:= \langle$round $+ 1$, candidate.$x\rangle$;
20   **else** candidate $:= \langle$round $+ 1$, prepared.$x\rangle$;
21   **for all** $b' \lesssim$ candidate **do** ballots$[b']$.vote(false)

of batches by BNS. If prepared reaches or exceeds candidate, then the server updates candidate to prepared, and tries to commit it by voting *true* for that ballot (lines 10–12). Once s commits some ballot b by receiving FV's indication deliver(*true*) for ballot b, the server decides the value $b.x$ (lines 13–14) and stops execution.

If the candidate ballot of a server s can no longer be aborted nor committed, then s resorts to a time-out mechanism that we describe next. The primitive start-timer(Δ) starts the server's local timer, such that a timeout event will be triggered once the specified delay Δ has expired. Invoking start-timer(Δ') while the timer is already running has the effect of restarting the timer with the new delay Δ'. In order to start the timer, a server s needs to receive, from each member of a quorum that contains s itself, messages that endorse either committing or preparing ballots with rounds bigger than round (line 15 of Algorithm 3). Since the domain of values can be infinite, the condition in line 15 requires that for each server u in some quorum U that contains s itself, there exists a ballot b_u with round $b_u.n > \mathsf{round}$, and either s receives from u a message endorsing to commit b_u, or otherwise s receives from u messages endorsing to abort every ballot in some non-empty, right-open interval $[z_u, b_u)$, whose upper bound is b_u. This condition may require receiving an infinite number of ballots, but it may hold after receiving a finite number of batches by BNS. Once the condition in line 15 holds, the server updates round to the smallest n such that every member of the quorum endorses to either commit or prepare some ballot with round bigger or equal than n, and (re-)starts the timer with delay $F(\mathsf{round})$, where F is an unbound function that doubles its value with each increment of n (lines 16–17). If the candidate ballot can no longer be aborted or committed, then $\mathsf{timeout}$ will be eventually triggered (line 18) and the server considers a new candidate ballot with the current round increased by one, and with the value $\mathsf{candidate}.x$ if the server never prepared any ballot yet (line 19) or the value $\mathsf{prepared}.x$ otherwise (line 20). Then s tries to prepare the new candidate ballot by voting *false* for each ballot below and incompatible than it (line 21). This may involve sending an infinite number of messages, which by BNS requires sending finitely many batches. The condition for starting the timer in line 15 does not strictly use FV as a black box. However, this use is warranted because line 15 only 'reads' the state of the network. ASCP makes every other change to the network through FV's primitives. We demonstrate Algorithm 3 interacting with Algorithm 2 on a concrete, small example in Appendix A.1.

ASCP guarantees the safety properties of *non-blocking byzantine consensus* in Figure 3.2. The full proof is in [45], in this thesis we show *Agreement for intact sets*. The requirement in lines 8–12 of Algorithm 3 that a server prepares the candidate ballot before voting for committing it, enforces that if a voting for committing some ballot within the servers of an intact set I succeeds, then some server in I previously prepared that ballot:

Lemma 3.2.1 *Consider an execution of ASCP with an intact set I. If a server $s_1 \in I$ commits a ballot b, then some server $s_2 \in I$ prepared b.*

Proof 3.2.1 *Assume that $s_1 \in I$ commits ballot b. By line 7 of Algorithm 2, s_1 received* $\texttt{READY}(b, \mathit{true})$ *from every member of a quorum to which s_1 belongs. By Lemma 3.1.3 the first server to do so received* $\texttt{VOTE}(b, \mathit{true})$ *messages from every member of a quorum U to which s_1 belongs. Since s_1 is intertwined with every other server in I, there exists a correct server s_2 in the intersection $U \cap I$ that sent* $\texttt{VOTE}(b, \mathit{true})$*. The server s_2 can send* $\texttt{VOTE}(b, \mathit{true})$ *only through line 6 of Algorithm 2, which means that s_2 triggers* $\mathsf{brs}[b].\mathsf{vote}(\mathit{true})$ *in line 12 of Algorithm 3. By line 8 of the same figure, this is only possible after s_2 has aborted every $b' \lessapprox b$, and the lemma holds.*

Aborting every ballot below and incompatible to the candidate prevents that one server in an intact set I prepares a ballot b_1, and concurrently another server in I sends $\texttt{READY}(b_2, \mathit{true})$ with b_2 below and incompatible than b_1:

Lemma 3.2.2 *Consider an execution of ASCP with an intact set I with $s_1, s_2 \in I$, and b_1 and b_2 be ballots such that $b_2 \lessapprox b_1$. The following two things cannot both happen: server s_1 prepares b_1 and server s_2 sends* $\texttt{READY}(b_2, \mathit{true})$*.*

Proof 3.2.2 *Assume towards a contradiction that s_1 prepares b_1, and that s_2 sends* $\texttt{READY}(b_2, \mathit{true})$*. By definition of prepare, server s_1 aborted every ballot $b \lessapprox b_1$. By line 7 of Algorithm 2, server s_1 received* $\texttt{READY}(b, \mathit{false})$ *from every member of a quorum U_b for each ballot $b \lessapprox b_1$. By assumptions, $b_2 \lessapprox b_1$, and therefore s_2 received* $\texttt{READY}(b_2, \mathit{false})$ *from every member of the quorum U_{b_2}. By Lemma 3.1.3, the first server $u_1 \in I$ that sent* $\texttt{READY}(b_2, \mathit{false})$ *received* $\texttt{VOTE}(b_2, \mathit{false})$ *from a quorum U_1 to which u_1 belongs. Since s_2 sent* $\texttt{READY}(b_2, \mathit{true})$ *and by Lemma 3.1.3, the first server $u_2 \in I$ that* *sent* $\texttt{READY}(b_2, true)$ *received* $\texttt{VOTE}(b_2, true)$ *from a quorum* U_2 *to which* u_2 *belongs. Since* u_1 *and* u_2 *are intertwined, the intersection* $U_1 \cap U_2$ *contains some correct server* s*, which sent both* $\texttt{VOTE}(b_2, false)$ *and* $\texttt{VOTE}(b_2, true)$ *messages. By the use of the Boolean* voted *in line 3 of Algorithm 2 this results in a contradiction and we are done.*

Lemma 3.2.3 *Consider an execution of ASCP with an intact set* I*. If a server* $s_1 \in I$ *commits a ballot* b_1*, then the largest ballot* b_2 *prepared by any server* $s_2 \in I$ *before* s_1 *commits* b_1 *is such that* $b_1 \sim b_2$*.*

Proof 3.2.3 *Assume server* s_1 *commits ballot* b_1*. By the guard in line 13 of Algorithm 2, server* s_1 *received the message* $\texttt{READY}(b_1, true)$ *from every member of a quorum to which* s_1 *belongs, which entails that server* s_1 *received* $\texttt{READY}(b_1, true)$ *from itself. By Lemma 3.1.3, the first server* $u \in I$ *that send* $\texttt{READY}(b_1, true)$ *needs to receive a* $\texttt{VOTE}(b_1, true)$ *message from every member of some quorum to which* u *belongs. Thus,* u *itself triggered* $\mathsf{brs}[b_1].\mathsf{vote}(true)$*, which by lines 7 and 21 of Algorithm 3 means that* u *prepared ballot* b_1*. Hence, the largest ballot* b_2 *such that there exists a server* $s_2 \in I$ *that triggers* $\mathsf{brs}[b_2].\mathsf{vote}(true)$ *before* s_1 *commits* b_1*, is bigger or equal than* b_1*. If* $b_2 = b_1$*, then* $b_2.x = b_1.x$ *and by lines 8–12 of Algorithm 3, server* s_2 *prepares* b_2 *before it triggers* $\mathsf{brs}[b_2].\mathsf{vote}(true)$ *and the lemma holds.*

If $b_2 > b_1$*, then we assume towards a contradiction that* $b_2.x \neq b_1.x$*. By lines 8–12 of Algorithm 3, server* s_2 *prepared* b_2*, but this results in a contradiction by Lemma 3.2.2, because* s_1 *and* s_2 *are intertwined and* s_1 *sent* $\texttt{READY}(b_1, true)$*, but* $b_1 \lnsim b_2$*. Therefore* $b_2.x = b_1.x$*, and by lines 8–12 of Algorithm 3, server* s_2 *prepares* b_2 *before it triggers* $\mathsf{brs}[b_2].\mathsf{vote}(true)$*.*

Agreement for intact sets holds as follows: assume towards a contradiction that two servers in I respectively commit ballots b_1 and b_2 with different values. A server in I prepared the bigger of the two ballots by Lemma 3.2.1, which results in a contradiction by Lemma 3.2.2. Finally, correctness of ASCP is captured by Theorem 3.2.1 below. The full proof can be found in [45].

Algorithm 4: Part 1/2: Bunched voting with quorums $\mathcal{Q}$.

process bunched-voting($s \in$ Srvrs)
    max-vt-prep, max-rd-prep, max-dl-prep $:= \langle 0, \bot \rangle \in$ Ballot
    Bllts-vt-cmt, Bllts-rd-cmt, Bllts-dl-cmt $:= \emptyset \in 2^{\mathsf{Ballot}}$
    prepare(b)
        if max-vt-prep $< b$ **then**
            max-vt-prep $:= b$
            send `VOTE`(PREP max-vt-prep) **to every** $s' \in$ Srvrs
        end
    end
    when exists maximum b **such that** max-vt-prep $< b$ **and exists** $U \in \mathcal{Q}$ **such that** $v \in U$ **and for every** $u \in U$ **received** `VOTE`(PREP b_u) **where** $b' \lesssim b_u$ **for every** $b' \lesssim b$
        max-rd-prep $:= b$
        send `READY`(PREP max-rd-prep) **to every** $s' \in$ Srvrs
    end
    when exists maximum b **such that** max-rd-prep $< b$ **and exists** s**-blocking** B **such that for every** $u \in B$ **received** `READY`(PREP b_u) **where** $b' \lesssim b_u$ **for every** $b' \lesssim b$
        max-rd-prep $:= b$
        send `READY`(PREP max-rd-prep) **to every** $s' \in$ Srvrs
    end
    when exists maximum b **such that** max-dl-prep $< b$ **and exists** $U \in \mathcal{Q}$ **such that** $v \in U$ **and for every** $u \in U$ **received** `READY`(PREP b_u) **where** $b' \lesssim b_u$ **for every** $b' \lesssim b$
        max-dl-prep $:= b$
        prepared(max-dl-prep)
    end

Theorem 3.2.1 *The ASCP protocol over $\mathcal{F}$ satisfies the specification of byzantine consensus for intact sets in Figure 3.2.*

3.3 Concrete Stellar Consensus Protocol

Next we introduce the *concrete SCP* (CSCP) which is amenable to implementation because each server s maintains finite state and only needs to send and receive a finite number of messages in order to progress. CSCP relies on *bunched voting* (BV) shown in Algorithm 4, which generalises FV and embodies all of FV's instances for each of the ballots.

Algorithm 4: Part 2/2: Bunched voting with quorums $\mathcal{Q}$.

22
23 commit(b)
24   **if** $b \notin$ Bllts-vt-cmt **and** max-vt-prep $= b$ **then**
25   Bllts-vt-cmt := Bllts-vt-cmt $\cup \{b\}$ **send** VOTE(CMT b) **to every** $s' \in$ Srvrs
26 **when received** VOTE(CMT b) **from every** $u \in U$ **for some** $U \in \mathcal{Q}$ **such that** $s \in U$ **and** $b \notin$ Bllts-rd-cmt
27   Bllts-rd-cmt := Bllts-rd-cmt $\cup \{b\}$
28   **send** READY(CMT b) **to every** $s' \in$ Srvrs
29 **when received** READY(CMT b) **from every** $u \in B$ **for some** s**-blocking** B **and** $b \notin$ Bllts-rd-cmt
30   Bllts-rd-cmt := Bllts-rd-cmt $\cup \{b\}$
31   **send** READY(CMT b) **to every** $s' \in$ Srvrs
32 **when received** READY(CMT b) **from every** $u \in U$ **for some** $U \in \mathcal{Q}$ **such that** $v \in U$ **and** $b \notin$ Bllts-dl-cmt
33   Bllts-dl-cmt := Bllts-dl-cmt $\cup \{b\}$
34   committed(b)

CSCP considers a single instance of BV, and thus each server s keeps a single process bunched-voting(s). In BV, servers exchange messages that contain two kinds of statements: a *prepare statement* PREP b encodes the aim to abort the possibly infinite range of ballots that are lower and incompatible than b; and a *commit statement* CMT b encodes the aim to commit ballot b. Algorithm 4 depicts BV. A server s stores the highest ballot for which s has respectively voted, readied, or delivered a prepare statement in fields max-vt-prep, max-rd-prep, and max-dl-prep (line 2). It also stores the set of ballots for which s has respectively voted, readied, or delivered a commit statement in fields Bllts-vt-cmt, Bllts-rd-cmt, and Bllts-dl-cmt (line 3). All these fields are finite and thus s maintains only finite state. When a server s invokes prepare(b), if b exceeds the highest ballot for which s has voted a prepare, then the server updates max-vt-prep to b and sends VOTE(PREP b) to every other server (lines 4–7). The protocol then proceeds with the usual stages of FV, with the caveat that at each stage of the protocol only the maximum ballot is considered for which the server can send a message—or deliver an indication—with a prepare statement. In particular, when there exists a ballot b that exceeds max-rd-prep and such that s received a message VOTE(PREP b_u) from each member u of some quorum to which s belongs, then the server proceeds as follows: it checks that each b' lower and incompatible than b_u is also lower and incompatible than b (line 10). If b is the maximum ballot passing the previous check for every member u of the quorum, then the server updates the field max-rd-prep to b and sends the message READY(PREP b) to every other server (lines 11–12). The server s checks similar conditions for the case when it receives messages READY(PREP b_u) from each member u of a s-blocking set, and proceeds similarly by updating max-rd-prep to b and sending READY(PREP b) to every other server (lines 14–16). The server will update max-dl-prep and trigger the indication prepared(b) when the same conditions are met after receiving messages READY(PREP b_u) from each member u of a quorum to which s belongs (lines 18–20). When a server s invokes commit(b) then the protocol proceeds with the usual stages of FV with two minor differences (lines 23–34). First, a server s only votes commit for the highest ballot for which s has voted a prepare statement (condition max-vt-prep $= b$ in line 24). Second, the protocol uses the sets of ballots Bllts-vt-cmt, Bllts-rd-cmt and Bllts-dl-cmt in order to keep track of the stage of the protocol for each ballot. The structure of CSCP in Algorithm 5 directly relates to ASCP in Algorithm 3. A server proposes a value x in line 5. A server tries to prepare a ballot b by invoking prepare(b) in line 7, and receives the indication prepared(b) in line 8. A server tries to commit a ballot b by invoking commit(b) in line 12, and receives the indication committed(b) in line 13. A server decides a value x in line 14. Time-outs are set in lines 15–17 and triggered in line 18. Again, we demonstrate Algorithm 5 interacting with Algorithm 4 on the same concrete, small example in Appendix A.2.

Next we establish a correspondence between CSCP in and ASCP: the concrete protocol observationally refines the abstract one, which means that any externally observable behaviour of the former can also be produced by the

Algorithm 5: Concrete SCP with quorums $\mathcal{Q}$.

```
process concrete-consensus(s ∈ Srvrs)
    brs := new process bunched-voting()
    candidate, prepared := ⟨0, ⊥⟩ ∈ Ballot
    round := 0 ∈ ℕ⁺ ∪ {0}
    propose(x)
        candidate := ⟨1, x⟩
        brs.prepare(candidate)
    when brs.prepared(b) and prepared < b
        prepared := b
        if candidate ⩽ prepared then
            candidate := prepared
            brs.commit(candidate)
    when brs.committed(b)
        decide(b.x)
    when exists U ∈ Q such that v ∈ U and for each u ∈ U exist
      M_u ∈ {VOTE, READY} and b_u ∈ Ballot such that round < b_u.n and
      received M_u(STMT_u b_u) from u with STMT_u ∈ {CMT, PREP}
        round := min{b_u.n | u ∈ U}
        start-timer(F(round))
    when timeout
        if prepared = ⟨0, ⊥⟩ then candidate := ⟨round + 1, candidate.x⟩;
        else candidate := ⟨round + 1, prepared.x⟩;
        brs.prepare(candidate)
```

latter [36]. Informally, the refinement shows that for every execution of CSCP there exists an execution of ASCP (with some behaviour of faulty servers) such that each server in the intact set I decides the same value in both of the executions. The refinement result allows us to carry over the correctness of ASCP established in Theorem 3.2.1 to CSCP. We first define several notions required to formalise our refinement result. A *history* is a sequence of the events s.propose(x) and s.decide(x), where s is a correct server and x a value. The specification of consensus assumes that s triggers an event s.propose(x), thus a history will have s.propose(x) for every correct server s. A *concrete trace* τ is a sequence of events that subsumes histories, and contains events s.prepare(b), s.commit(b), s.prepared(b), s.committed(b), s.start-timer(n),

s.timeout, s.send(m, s'), and s.receive(m, s'), where s is a correct server and s' is any server, b is a ballot, m is a message in $\{\texttt{VOTE}(stmt), \texttt{READY}(stmt)\}$ with *stmt* a statement in $\{\text{PREP } b, \text{CMT } b\}$, and n is a round. An *abstract trace* τ is a sequence of events that subsumes histories, and contains events s.start-timer(n), s.timeout, and batched events s.vote-batch$([b_i], a)$, s.deliver-batch$([b_i], a)$, s.send-batch$([m_i], s')$, and s.receive-batch$([m_i], s')$,where s is a correct server and s' is any server, n is a round, $[b_i]$ is a sequence of ballots, a is a Boolean, and $[m_i]$ is a sequence of messages in $\{\texttt{VOTE}(b, a), \texttt{READY}(b, a)\}$. The sequences of ballots and messages above, which represent a possibly infinite number of 'batched' events, ensure that the length of any abstract trace is bounded by ω. Given a trace τ, a history $H(\tau)$ can be uniquely obtained from τ by removing every event in τ different from s.propose(x) or s.decide(x). An execution of CSCP (respectively, ASCP) *entails* a *concrete trace* (respectively, *abstract trace*) τ iff for every invocation and indication as well as for every send or receive primitive in an execution of the protocol in Algorithm 5 (respectively, for every invocation, indication and primitive in an execution of the protocol in Algorithm 3, where the vote, deliver, send and receive events are batched together), τ contains corresponding events in the same order.

We are interested in traces that are relative to some intact set I. Given a trace τ, the *I-projected* trace $\tau|_I$ is obtained by removing the events $s.\mathsf{ev} \in \tau$ such that $s \notin I$.

Next, we define a simulation function which maps a trace of events in the concrete protocol of Algorithm 5 to a trace of events in the abstract protocol of Algorithm 3: every event in the concrete trace is mapped to an event in the abstract trace. The key idea is that every single event in the concrete trace is mapped to a batched event in the abstract trace unfolding *e.g.*, for preparing a ballot into a batch of triggering the event vote with *false* (3.2) for every ballot below and incompatible, or one single event of voting for a prepare message into a batch of voting false for every ballot below and incompatible, but not yet voted for (3.6).

Definition 3.3.1 *We inductively define a* simulation function σ *from concrete to abstract traces.*

$$\sigma([\,]) = [\,] \tag{3.1}$$

$$\sigma(\tau \cdot [s.\mathsf{prepare}(b)]) = \sigma(\tau) \cdot s.\mathsf{vote\text{-}batch}([b', b' \lesssim b], \mathit{false}) \tag{3.2}$$

$$\sigma(\tau \cdot [s.\mathsf{commit}(b)]) = \sigma(\tau) \cdot s.\mathsf{vote\text{-}batch}([b', \phi(\sigma(\tau)) < b' \leqslant b], \mathit{true}) \tag{3.3}$$

$$\sigma(\tau \cdot [s.\mathsf{prepared}(b)]) = \sigma(\tau) \cdot s.\mathsf{deliver\text{-}batch}([b', b' \lesssim b \wedge \forall s.\mathsf{deliver\text{-}batch}(bs) \in \sigma(\tau).\, (b', \mathit{false}) \notin bs], \mathit{false}) \tag{3.4}$$

$$\sigma(\tau \cdot [s.\mathsf{committed}(b)]) = \sigma(\tau) \cdot \mathsf{deliver\text{-}batch}([b], \mathit{true}) \tag{3.5}$$

$$\sigma(\tau \cdot [s.\mathsf{op}(\mathtt{VOTE}(\textsc{prep}\ b), u)]) = \sigma(\tau) \cdot s.\mathsf{op\text{-}batch}([\mathtt{M}(b', \mathit{false}), b' \lesssim b \wedge \forall a \in \mathsf{Bool}.\, \forall s.\mathsf{op\text{-}batch}(ms, u) \in \sigma(\tau).\, \mathtt{M}(b', a) \notin ms], u) \tag{3.6}$$

$$\sigma(\tau \cdot [s.\mathsf{op}(\mathtt{READY}(\textsc{prep}\ b), u)]) = \sigma(\tau) \cdot s.\mathsf{op\text{-}batch}([\mathtt{M}(b', \mathit{false}), b' \lesssim b \wedge \forall s.\mathsf{op\text{-}batch}(ms, u) \in \sigma(\tau).\, \mathtt{M}(b', \mathit{false}) \notin ms], u) \tag{3.7}$$

$$\sigma(\tau \cdot [s.\mathsf{op}(\mathtt{VOTE}(\textsc{cmt}\ b), u)]) = \sigma(\tau) \cdot s.\mathsf{op\text{-}batch}([\mathtt{VOTE}(b', \mathit{true}), \phi(\sigma(\tau)) < b' \leqslant b], u) \tag{3.8}$$

$$\sigma(\tau \cdot [s.\mathsf{op}(\mathtt{READY}(\textsc{cmt}\ b), u)]) = \sigma(\tau) \cdot s.\mathsf{op\text{-}batch}([\mathtt{READY}(b, \mathit{true})], u) \tag{3.9}$$

$$\sigma(\tau \cdot [e]) = \sigma(\tau) \cdot [e] \qquad \textit{otherwise} \tag{3.10}$$

Here, $\phi(\tau) = max\{b \mid \forall b' \lesssim b.\ s.b'.\mathsf{deliver}(\mathit{false}) \in \tau\}$ *and let* $\mathsf{op} \in \{\mathsf{send}, \mathsf{receive}\}$ *and* $\mathtt{M} \in \{\mathtt{VOTE}, \mathtt{READY}\}$.

The proof of the following Lemma A.2.6 is shown in the Appendix A. The statement follows from a case analysis on the definition of σ by induction on τ. For every case, we trace the execution of CSCP and by applying σ show that the execution yields an execution in ASCP.

Lemma 3.3.1 *Let I be some intact set and τ be a trace entailed by an execution of CSCP. For every finite prefix τ' of the projected trace $\tau|_I$, the simulated $\rho' = \sigma(\tau')$ is the prefix of a trace entailed by an execution of ASCP.*

Theorem 3.3.1 can be established by showing that, for every finite prefix τ

of a trace entailed by CSCP, the simulation $\sigma(\tau)$ is a prefix of a trace entailed by ASCP.

Theorem 3.3.1 *For an intact set I and for every execution of CSCP with trace τ, there exists an execution of ASCP with trace ρ and $H(\tau|_I) = H(\rho|_I)$.*

Proof 3.3.1 *Let τ be the trace entailed by an execution of CSCP. We prove that there exists a trace ρ entailed by an execution of ASCP such that $H(\tau|_I) = H(\rho|_I)$. Assume towards a contradiction that for all traces ρ, if $H(\tau|_I) = H(\rho|_I)$ then ρ is not a trace entailed by an execution of ASCP. Fix the trace ρ to be $\sigma(\tau|_I)$, which entails that $H(\tau|_I) = H(\rho|_I)$ by definition of σ and H. Since the number of events in a trace entailed by ASCP is bounded by ω, we denote the ith event of ρ as e_i, with i a natural number. Since ρ is not a trace entailed by an execution of ASCP by assumptions, there exists $i \geq 0$ such that the prefix $[e_i, \ldots, e_i]$ of ρ is a trace entailed by ASCP, but the prefix $[e_0, \ldots, e_{i+1}]$ of ρ is not a trace entailed by ASCP. Since σ maps one event of a concrete trace into one event of an abstract trace, there exists a finite prefix τ' of $\tau|_I$ such that $\sigma(\tau') = [e_0, \ldots, e_{i+1}]$, but this leads to a contradiction because $\sigma(\tau')$ is a trace of an execution of ASCP by Lemma A.2.6. Therefore, there exists a trace ρ entailed by an execution of ASCP such that $H(\tau|_I) = H(\rho|_I)$.*

By Theorem 3.2.1 every execution of ASCP enjoys the properties of weak byzantine consensus, and so does every execution of CSCP by refinement.

Corollary 3.3.1 *The CSCP protocol satisfies the specification of byzantine consensus for intact sets in Figure 3.2*

Proof 3.3.2 *Let τ be the trace entailed by an execution of CSCP. Assume towards a contradiction that the execution does not satisfy some of the properties of* Integrity, Agreement for intact sets, Weak validity for intact sets, *or* Non-blocking for intact sets. *By Theorem 3.3.1, there exists a trace ρ entailed by an execution of ASCP over $\mathcal{F}$ and such that $H(\tau|_I) = H(\rho|_I)$. By definition of history, $H(\tau|_I)$ and $H(\rho|_I)$ coincide in their respective propose and decide events. Since $\rho|_I$ is entailed by an execution of ASCP, this execution fails to* *satisfy some of the properties of* Integrity*,* Agreement for intact sets*,* Weak validity for intact sets*, or* Non-blocking for intact sets*, which contradicts Theorem 3.2.1.*

Chapter 4

Embedding a Protocol in a Block DAG

Novel designs generalise a blockchain to a more generic directed acyclic graph between blocks [110]: a *block DAG*. In this chapter we leverage block DAGs and show that any deterministic byzantine fault tolerant protocol $\mathcal{P}$ can be embedded in a block DAG while maintaining $\mathcal{P}$'s safety and liveness properties. Because $\mathcal{P}$ is deterministic, and a block DAG essentially embodies Lamport's happened-before relations [62], every sever can locally replay $\mathcal{P}$—as a black-box—for every other server, inferring messages without explicitly receiving them. This allows to omit the sending of every message, which can be determined from the protocol thereby reducing the number of exchanged messages. We guarantee correctness by showing that a block DAG is a *reliable point-to-point link* $[H_a]$.

Block DAGs are now underlying several implementations of consensus protocols: Hashgraph [10] used by the Hedera network, as well as Aleph [41], Blockmania [30], and Flare [96]. They report impressive performance results compared to traditional protocols that materialise point-to-point messages as direct network messages—especially as maintaining a joint block DAG is simple and scalable and can leverage widely-available distributed key-value stores. However, their arguments are inherently tied to their specific applications and requirements, but both specification and formal arguments of Hashgraph, Aleph, Blockmania, and Flare are structured around two phases: *(i)* building a block DAG, and *(ii)* running a protocol on top of the block DAG. When implemented, their specification, and arguments for correctness, safety and liveness are far

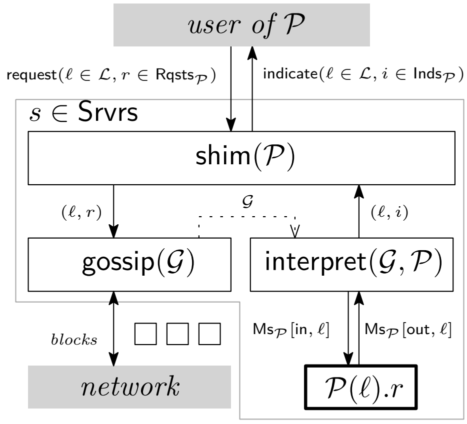

Figure 4.1: Components and interfaces.

from simple. The goal of our work is to give a clear separation of the high-level protocol $\mathcal{P}$ and the underlying block DAG to allow for easy re-usability and to strengthen the foundations and persuasiveness of systems based on block DAGs.

Figure 4.1 shows the interfaces and components of our proposed block DAG framework parametric by a deterministic BFT protocol $\mathcal{P}$. At the top, we have a *user* seeking to run one or multiple instances of $\mathcal{P}$ on servers Srvrs. First, to distinguish between multiple *protocol instances* the user assigns them a *label* ℓ from a set of *labels* $\mathcal{L}$. Now, for $\mathcal{P}$ there is a set of possible *requests* Rqsts$_\mathcal{P}$. However, instead of requesting $r \in$ Rqsts$_\mathcal{P}$ from $s_i \in$ Srvrs running $\mathcal{P}$ for protocol instance ℓ, the user calls the high-level interface of our block DAG framework: request(ℓ, r) in shim$(\mathcal{P})$. Internally, s_i passes (ℓ, r) on to gossip$(\mathcal{G})$—which continuously builds s_i's *block DAG* $\mathcal{G}$ by receiving and disseminating *blocks*. The passed (ℓ, r) is included into the next block s_i disseminates, and s_i also includes references to other received blocks, where cryptographic primitives prevent byzantine servers from adding cycles between blocks [73]. These blocks are continuously exchanged by the servers utilizing the low-level interface to the *network* to exchange blocks. Independently, indicated by the dotted line, s_i interprets $\mathcal{P}$ by *reading* $\mathcal{G}$ and running interpret$(\mathcal{G}, \mathcal{P})$. To do so, s_i locally simulates every protocol instance $\mathcal{P}$ with label ℓ by simulating one process instance of $\mathcal{P}(\ell)$ for every server $s \in$ Srvrs. To drive the simulation, s_i

passes the request r read from a block in $\mathcal{G}$ to $\mathcal{P}$, and then s_i simulates the message exchange between any two servers based on the structure of the block DAG and the deterministic protocol $\mathcal{P}$. Therefore s_i moves messages between in- and out-buffers $\mathsf{Ms}_{\mathcal{P}}[\mathsf{in}, \ell]$ and $\mathsf{Ms}_{\mathcal{P}}[\mathsf{out}, \ell]$. Eventually, the simulation $\mathcal{P}(\ell)$ of the server s_i will indicate i from the set of possible *indications* $\mathsf{Inds}_{\mathcal{P}}$. We show how the block DAG essentially acts as a reliable point-to-point link and describe how any deterministic BFT protocol $\mathcal{P}$ can be interpreted on a block DAG. Finally, after $\mathsf{interpret}$ indicated i, $\mathsf{shim}(\mathcal{P})$ can indicate i for ℓ to the user of $\mathcal{P}$. From the user's perspective, the embedding of $\mathcal{P}$ acted as $\mathcal{P}$, *i.e.*, $\mathsf{shim}(\mathcal{P})$ *maintained* $\mathcal{P}$*'s interfaces and properties.* We prove this and illustrate the block DAG framework for $\mathcal{P}$ instantiated with byzantine reliable broadcast protocol.

4.1 Building a Block DAG

The networking component of the block DAG protocol is very simple: it has one core message type, namely a *block*, which is constantly disseminated. A block contains authentication for references to previous blocks, requests associated to instances of protocol $\mathcal{P}$, meta-data and a signature. Servers only exchange and validate blocks. From these blocks with their references to previous blocks, servers build their block DAGs. Although servers build their block DAGs locally, eventually correct servers have a joint block DAG $\mathcal{G}$. As we show in the next Section 4.2, the servers can then *independently* interpret $\mathcal{G}$ as multiple instances of $\mathcal{P}$.

We assume a fixed and finite set of servers Srvrs known by every $s' \in \mathsf{Srvrs}$ and we assume $3f+1$ servers to tolerate at most f byzantine servers. The exact requirements on the network synchronicity depend on the protocol $\mathcal{P}$, that we want to embed, *e.g.*, we may require partial synchrony [34] to avoid FLP [37]. The only network assumption we impose for building block DAGs is the following:

Assumption 4.1.1 (Reliable Delivery) *For two correct servers s_1 and s_2, if s_1 sends a block B to s_2, then eventually s_2 receives B.*

A *directed graph* $\mathcal{G}$ is a pair of *vertices* V and *edges* $\mathsf{E} \subseteq \mathsf{V} \times \mathsf{V}$. We write $\varnothing$ for the *empty graph*. If there is an edge from v to v', that is $(v, v') \in \mathsf{E}$, we write $v \rightharpoonup v'$. If v' is *reachable* from v, then (v, v') is in the transitive closure of $\rightharpoonup$, and we write $\rightharpoonup^+$. We write $\rightharpoonup^*$ for the reflexive and transitive closure, and $v \rightharpoonup^n v'$ for $n \geqslant 0$ if v' is reachable from v in n steps. A graph $\mathcal{G}$ is *acyclic*, if $v \rightharpoonup^+ v'$ implies $v \neq v'$ for all nodes $v, v' \in \mathcal{G}$. We abbreviate $v \in \mathcal{G}$ if $v \in \mathsf{V}_{\mathcal{G}}$, and $V \subseteq \mathcal{G}$ if $v \in \mathcal{G}$ for all $v \in V$. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be directed graphs. We define $\mathcal{G}_1 \cup \mathcal{G}_2$ as $(\mathsf{V}_{\mathcal{G}_1} \cup \mathsf{V}_{\mathcal{G}_2}, \mathsf{E}_{\mathcal{G}_1} \cup \mathsf{E}_{\mathcal{G}_2})$, and $\mathcal{G}_1 \leqslant \mathcal{G}_2$ holds if $\mathsf{V}_{\mathcal{G}_1} \subseteq \mathsf{V}_{\mathcal{G}_2}$ and $\mathsf{E}_{\mathcal{G}_1} = \mathsf{E}_{\mathcal{G}_2} \cap (\mathsf{V}_{\mathcal{G}_1} \times \mathsf{V}_{\mathcal{G}_1})$. Note, for $\leqslant$ we not only require $\mathsf{E}_{\mathcal{G}_1} \subseteq \mathsf{E}_{\mathcal{G}_2}$, but additionally $\mathsf{E}_{\mathcal{G}_1}$ must already contain all edges from $\mathsf{E}_{\mathcal{G}_2}$ between vertices in $\mathcal{G}_1$. The following definition for inserting a new vertex v is restrictive: it permits to extend $\mathcal{G}$ only by a vertex v and edges to this v.

Definition 4.1.1 *Let $\mathcal{G}$ be a directed graph, v be a vertex, and a E be a set of edges of the form $\{(v_i, v) \mid v_i \in V \subseteq \mathcal{G}\}$. We define* $\mathsf{insert}(\mathcal{G}, v, E) = (\mathsf{V}_{\mathcal{G}} \cup \{v\}, \mathsf{E}_{\mathcal{G}} \cup E)$.

This unconventional definition of inserting a vertex is sufficient for building a block DAG—and helps to establish useful properties of the block DAG in the next lemma: *(i)* inserting a vertex is idempotent, *(ii)* the original graph is a subgraph of the graph with a newly inserted vertex, and *(iii)* a block DAG is acyclic by construction.

Lemma 4.1.1 *For a directed graph $\mathcal{G}$, a vertex v, and a set of edges $E = \{(v_i, v) \mid v_i \in V \subseteq \mathcal{G}\}$, the following properties of* $\mathsf{insert}(\mathcal{G}, v, E)$ *hold:* (i) *if $v \in \mathcal{G}$ and $E \subseteq \mathsf{E}_{\mathcal{G}}$, then* $\mathsf{insert}(\mathcal{G}, v, E) = \mathcal{G}$*;* (ii) *if $E = \{(v_i, v) \mid v_i \in V \subseteq \mathcal{G}\}$ and $v \notin \mathcal{G}$, then* $\mathcal{G} \leqslant \mathsf{insert}(\mathcal{G}, v, E)$*; and* (iii) *if $\mathcal{G}$ is acyclic, $v \notin \mathcal{G}$, then* $\mathsf{insert}(\mathcal{G}, v, E)$ *is acyclic.*

Proof 4.1.1 *By definition of $\mathcal{G}$ and* insert *holds* (i)*. For* (ii)*, let $\mathcal{G}' =$* $\mathsf{insert}(\mathcal{G}, v, E)$*. By definition of* insert*, $\mathsf{V}_{\mathcal{G}} \subseteq \mathsf{V}_{\mathcal{G}'}$. Assume $v \notin \mathcal{G}$. As E contains only edges such that (v_i, v) where $v \notin \mathcal{G}$, $\mathsf{E}_{\mathcal{G}} = \mathsf{E}_{\mathcal{G}'} \cap (\mathsf{V}_{\mathcal{G}} \times \mathsf{V}_{\mathcal{G}})$ holds. For* (ii)*, let $\mathcal{G}' =$* $\mathsf{insert}(\mathcal{G}, v, E)$*. By definition of* insert*, $\mathsf{V}_{\mathcal{G}} \subseteq \mathsf{V}_{\mathcal{G}'}$. Assume $v \notin \mathcal{G}$.*

As E contains only edges such that (v_i, v) where $v \notin \mathcal{G}$, $\mathsf{E}_{\mathcal{G}} = \mathsf{E}_{\mathcal{G}'} \cap (\mathsf{V}_{\mathcal{G}} \times \mathsf{V}_{\mathcal{G}})$ holds.

To give some intuitions, for Lemma 4.1.1 *(ii)*, if $v \in \mathcal{G}$ and $\mathcal{G}' = \mathsf{insert}(\mathcal{G}, v, E)$, then $\mathsf{E}_{\mathcal{G}'} \cap (\mathsf{V}_{\mathcal{G}} \times \mathsf{V}_{\mathcal{G}}) = \mathsf{E}_{\mathcal{G}}$ may not hold. For example, let $\mathcal{G}$ have vertices v_1 and v_2 with $\mathsf{E}_{\mathcal{G}} = \varnothing$, and $\mathcal{G}' = \mathsf{insert}(\mathcal{G}, v_2, \{(v_1, v_2)\})$ with $\mathsf{E}_{\mathcal{G}'} = \{(v_1, v_2)\}$. Then we have $\mathsf{E}_{\mathcal{G}} \neq \mathsf{E}_{\mathcal{G}'} \cap (\mathsf{V}_{\mathcal{G}} \times \mathsf{V}_{\mathcal{G}})$. For Lemma 4.1.1 *(iii)*, if $v \in \mathcal{G}$, then $\mathsf{insert}(\mathcal{G}, v, E)$ may add a cycle. For example, take $\mathcal{G}$ with vertices $\{v_1, v_2\}$ and $\mathsf{E}_{\mathcal{G}} = \{(v_1, v_2)\}$ then $\mathsf{insert}(\mathcal{G}, v_1, \{(v_2, v_1)\})$ contains a cycle.

Based on this definition of graphs, we next define block DAGs. We start with blocks.

Definition 4.1.2 *A* block $B \in \mathsf{Blks}$ *has* (i) *an* identifier n *of the server s which built B,* (ii) *a* sequence number $\mathsf{k} \in \mathbb{N}_0$*,* (iii) *a finite list of hashes of* predecessor *blocks* $\mathsf{preds} = [\mathsf{ref}(B_1), \ldots, \mathsf{ref}(B_k)]$*,* (iv) *a finite list of labels and requests* $\mathsf{rs} \in 2^{\mathcal{L} \times \mathsf{Rqsts}}$*, and* (v) *a* signature $\sigma = \mathsf{sign}(\mathsf{n}, \mathsf{ref}(B))$*. Here,* ref *is a secure cryptographic hash function computed from* n*,* k*,* preds*, and* rs*, but not σ. By not depending on σ,* $\mathsf{sign}(B.\mathsf{n}, \mathsf{ref}(B))$ *is well defined.*

We use B and $\mathsf{ref}(B)$ interchangeably, which is justified by *collision resistance* of ref (Definition 2.2.1*(3)*). We use register notation, *e.g.*, $B.\mathsf{n}$ or $B.\sigma$, to refer to elements of a block B, and abbreviate $B' \in \{B' \mid \mathsf{ref}(B') \in B.\mathsf{preds}\}$ with $B' \in B.\mathsf{preds}$. Given blocks B and B' with $B.\mathsf{n} = B'.\mathsf{n}$ and $B'.\mathsf{k} = B.\mathsf{k} + 1$. If $B \in B'.\mathsf{preds}$ then we call B a *parent* of B' and write $B'.\mathsf{parent} = B$. We require that every block has at most one parent. Otherwise, we consider B as not well formed, *i.e.*, not valid. We call B a *genesis block* if $B.\mathsf{k} = 0$. A genesis block B cannot have a parent block, because $B.\mathsf{k} = 0$ and 0 is minimal in $\mathbb{N}_0$.

Lemma 4.1.2 *For blocks B_1 and B_2, if $B_1 \in B_2.\mathsf{preds}$ then $B_2 \notin B_1.\mathsf{preds}$.*

Proof 4.1.2 *Let $x_1 = \mathsf{ref}(B_1)$ and $x_2 = \mathsf{ref}(B_2)$. By assumption, $x_1 \in B_2.\mathsf{preds}$. Assume towards a contradiction that $x_2 \in B_1.\mathsf{preds}$. Then, to com-*

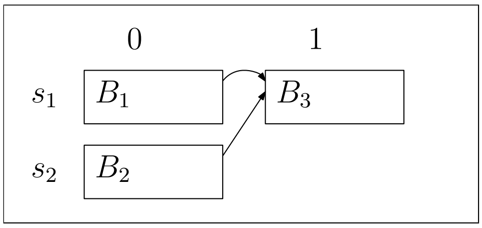

Figure 4.2: A block DAG with 3 blocks B_1, B_2, and B_3.

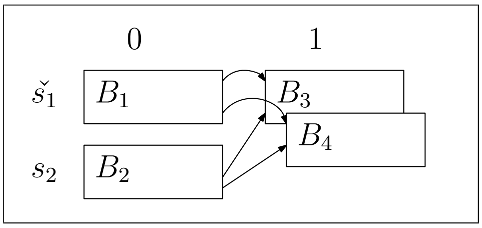

Figure 4.3: A block DAG, where $\check{s}_1$ is equivocating on the blocks B_3 and B_4.

pute x_1 we need to know $x_2 = \mathsf{ref}(B_2)$, but this contradicts preimage-resistance of ref*.*

Lemma 4.1.2 prevents a byzantine server $\check{s}$ to include a cyclic reference between $\check{B}$ and B by *(i)* waiting for—or building itself—a block B with $\mathsf{ref}(\check{B}) \in B.\mathsf{preds}$, and then *(ii)* building a block $\check{B}$ such that $\mathsf{ref}(\check{B}) \in B$. As with secure time-lines [73], Lemma 4.1.2 gives a temporal ordering on B and $\check{B}$. This is a static, cryptographic property, based on the security of hash functions, and not dependent on the order in which blocks are received on a network. While this prevents byzantine servers from introducing cycles, they can still build "faulty" blocks. So next we define three checks for a server to ascertain that a block is well-formed. If a block passes these checks, the block is *valid* from this server's point of view and the server *validated* the block.

Definition 4.1.3 *A server s considers a block B* valid*, written* $\mathsf{valid}(s, B)$*, if* (i) *s confirms* $\mathsf{verify}_\sigma(B.\mathsf{n}, B.\sigma)$, i.e.*, that $B.\mathsf{n}$ built B,* (ii) *either* (a) *B is a genesis block, or* (b) *B has exactly one parent, and* (iii) *s considers all blocks $B' \in B.\mathsf{preds}$ valid.*

Especially, *(ii)* deserves our attention: a server $\check{s}_1$ may still build two different blocks having the same parent. However, $\check{s}_1$ will not be able to create a further block to 'join' these two blocks with a different parent—their successors will remain split. Essentially, this forces a linear history from every block. So both, B_3 and B_4 in Figure 4.3 are valid. However, a block B_5 created by $\check{s}_1$ with $\mathsf{preds} = [\mathsf{ref}(B_4), \mathsf{ref}(B_3)$ would not be valid, as it has more than one parent: B_3 and B_4.

We assume, that if a correct server s considers a block B valid, then s can forward any block $B' \in B.\mathsf{preds}$. That is, s has received the full content of B'—not only $\mathsf{ref}(B')$—and persistently stores B'. From valid blocks and their predecessors, a correct server builds a *block DAG*:

Definition 4.1.4 *For a server s, a* block DAG $\mathcal{G} \in \mathsf{Dags}$ *is a directed acyclic graph with vertices $\mathsf{V}_\mathcal{G} \subseteq \mathsf{Blks}$, where* (i) $\mathsf{valid}(s, B)$ *holds for all $B \in \mathsf{V}_\mathcal{G}$, and* (ii) *if $B \in B'.\mathsf{preds}$ then $B \in \mathsf{V}_\mathcal{G}$ and $(B, B') \in \mathsf{E}_\mathcal{G}$ holds for all $B' \in \mathsf{V}_\mathcal{G}$. Let B' be a block such that $\mathsf{valid}(s, B')$ holds and $B \in \mathcal{G}$ for all $B \in B'.\mathsf{preds}$. Then s* inserts *B' into $\mathcal{G}$ by $\mathsf{insert}(\mathcal{G}, B', \{(B, B') \mid B \in B'.\mathsf{preds}\})$ after Definition 4.1.1 and we write $\mathcal{G}.\mathsf{insert}(B)$.*

The preconditions guarantee that $\mathcal{G}.\mathsf{insert}(B')$ is a block DAG, as shown by the following two lemmas.

Lemma 4.1.3 *For a block DAG $\mathcal{G}$ and a block $B \in \mathcal{G}$ holds $\mathcal{G} = \mathcal{G}.\mathsf{insert}(B)$,* i.e.*,* insert *is idempotent.*

Proof 4.1.3 *By definition of* insert *on block DAGs E is fixed to $\{(B, B') \mid B \in B'.\mathsf{preds}\}$. Since $B \in \mathcal{G}$ also $\{(B, B') \mid B \in B'.\mathsf{preds}\} \subseteq \mathsf{E}_\mathcal{G}$ by definition of block DAG. Thus, $\mathcal{G}.\mathsf{insert}(B) = \mathcal{G}$ by Lemma 4.1.1* (i)*.*

Lemma 4.1.4 *Let $\mathcal{G}$ be a block DAG for a server s and let B' be a block such that $\mathsf{valid}(s, B')$ holds and for all $B \in B'.\mathsf{preds}$ holds $B \in \mathcal{G}$. Let $\mathcal{G}' = \mathcal{G}.\mathsf{insert}(B')$. Then $\mathcal{G}'$ is a block DAG for s.*

Proof 4.1.4 *To show $\mathcal{G}'$ is a block DAG we need to show that $\mathcal{G}'$ adheres to Definition 4.1.4. For condition* (i) *we have to show that s considers all blocks in $\mathcal{G}'$ valid. By definition of* insert *holds $\mathsf{V}_{\mathcal{G}'} = \mathsf{V}_\mathcal{G} \cup \{B'\}$. As $\mathcal{G}$ is a block DAG for s, $\mathsf{valid}(s, B)$ holds for all $B \in \mathsf{V}_\mathcal{G}$ and $\mathsf{valid}(s, B')$ follows from the assumption of the lemma. For condition* (ii) *we have to show that for every backwards reference to B from the block B', the block DAG $\mathcal{G}'$ contains B and an edge from B to B'. The former—for all $B \in B'.\mathsf{preds}$ we have $B \in \mathcal{G}$—holds by*

Algorithm 6: Building the block DAG $\mathcal{G}$ and block $\mathcal{B}$.

```
module gossip(s ∈ Srvrs, 𝒢 ∈ Dags, rqsts ∈ 2^(ℒ×Rqsts))
    ℬ := {n : s, k : 0, preds : [ ], rs : [ ], σ : null} ∈ Blks
    blks := ∅ ∈ 2^Blks
    when received B ∈ Blks and B ∉ 𝒢
        blks := blks ∪ {B}
    when valid(s, B′) for some B′ ∈ blks
        𝒢.insert(B′)
        ℬ.preds := ℬ.preds · [ref(B′)]
        blks := blks \ {B′}
    when B′ ∈ blks and B ∈ B′.preds where B ∉ blks and B ∉ 𝒢
        send FWD ref(B) to B′.n
    when received FWD ref(B) from s′ and B ∈ 𝒢
        send B to s′
    when disseminate()
        ℬ := {ℬ with rs : rqsts.get(), σ : sign(s, ℬ)}
        𝒢.insert(ℬ)
        send ℬ to every s′ ∈ Srvrs
        ℬ := {n : s, k : ℬ.k + 1, preds : [ref(ℬ)], rs : [ ], σ : null}
```

assumption of the lemma. The latter—$(B, B') \in \mathsf{E}_{\mathcal{G}'}$ for $B \in B'.\mathsf{preds}$— holds by definition of insert*. As $\mathcal{G}$ is a block DAG, condition* (ii) *holds for every block in $\mathcal{G}$. It remains to show, that $\mathcal{G}'$ is acyclic. If $B' \in \mathcal{G}$ then by Lemma 4.1.3, $\mathcal{G}' = \mathcal{G}$ and $\mathcal{G}$ is acyclic. If $B' \notin \mathcal{G}$ then $\mathcal{G}'$ is acyclic by Lemma 4.1.1* (iii)*.*

Example 4.1.1 *In Figure 4.2 we show a block DAG with three blocks B_1, B_2, and B_3, where $B_1 = \{\mathsf{n} = s_1, \mathsf{k} = 0, \mathsf{preds} = [\,]\}$, $B_2 = \{\mathsf{n} = s_2, \mathsf{k} = 0, \mathsf{preds} = [\,]\}$, and $B_3 = \{\mathsf{n} = s_1, \mathsf{k} = 1, \mathsf{preds} = [\mathsf{ref}(B_1), \mathsf{ref}(B_2)]\}$. Here, $\mathsf{parent}(B_3) = B_1$. Consider now Figure 4.3 adding the block: $B_4 = \{\mathsf{n} = s_1, \mathsf{k} = 1, \mathsf{preds} = [\mathsf{ref}(B_1), \mathsf{ref}(B_2)]\}$. While all blocks in Figure 4.3 are valid, with block B_4, $\check{s}_1$ is* equivocating *on the block B_3—and vice versa. We omitted* rs *in the block DAGs. However, to give a small example: a possible request could be* broadcast(42)*.*

To build a block DAG and blocks every correct server follows the gossip protocol in Algorithm 6. By building a block DAG every correct server will eventually have a joint view on the system. By building a block, every server can inject messages into the system: either explicit messages from the high-level protocol by directly writing those into the block, or implicit messages by adding references to other blocks. In Algorithm 6, a server s builds *(i)* its block DAG $\mathcal{G}$ in lines 4–13, and *(ii)* its current block $\mathcal{B}$ by including requests and references to other blocks in lines 14–18. The servers communicate by exchanging blocks. Assumption 4.1.1 guarantees, that a correct s will eventually receive a block from another correct server. Moreover, every correct server s will regularly request $\mathsf{disseminate}()$ in line 14 and will eventually send their own block $\mathcal{B}$ in line 17. This is guaranteed by the high-level protocol (*cf.* Section 4.3).

Every server s operates on four data structures. The two data structures which are shared with Algorithm 7 are given as arguments in line 1: *(i)* the block DAG $\mathcal{G}$, which Algorithm 7 will only read, and *(ii)* a buffer rqsts, where Algorithm 7 inserts pairs of labels and requests. On the other hand, s also keeps *(iii)* the block $\mathcal{B}$ which s currently builds (line 2), and *(iv)* a buffer blks of received blocks (line 3). To build its block DAG, s inserts blocks into $\mathcal{G}$ in line 7 and line 16. It is guaranteed that by inserting those blocks $\mathcal{G}$ remains a block DAG by as shown by the following lemmas:

Lemma 4.1.5 *For every correct server s executing* gossip *of Algorithm 6, whenever the execution reaches line 16 then* $\mathsf{valid}(s, \mathcal{B})$ *holds.*

Proof 4.1.5 *We need to show, that once the execution reaches line 16 Definition 4.1.3* (i)–(iii) *holds. As s is correct and signs $\mathcal{B}$ in line 15* (i) $\mathsf{verify}_\sigma(s, \mathcal{B}.\sigma)$ *holds. We prove* (ii) *and* (iii) *by induction on the times n the execution reaches line 16. For the base case, $\mathcal{B}$ is* (a) *a genesis block with $\mathcal{B}.\mathsf{k} = 0$ as initialized in line 2. Moreover $\mathcal{B}$ has no parent. As s is correct and only inserts B' in $\mathcal{B}.\mathsf{preds}$ in line 8 whenever s considers B' valid in line 6, s considers all $B' \in \mathcal{B}.\mathsf{preds}$ valid. In the step case, $\mathcal{B}_{n+1}$ is updated in line 18. We show that* (b) *$\mathcal{B}_{n+1}$ has exactly one parent $\mathcal{B}_n$. By line 18, $\mathcal{B}_{n+1}.\mathsf{n} = \mathcal{B}_n.\mathsf{n}$ and $\mathcal{B}_{n+1}.\mathsf{k} = \mathcal{B}_n.\mathsf{k} + 1$. As $\mathcal{B}_n$ is inserted in $\mathcal{B}_{n+1}.\mathsf{preds}$ in line 18, by definition $\mathcal{B}_{n+1}.\mathsf{parent} = \mathcal{B}_n$. By induction hypothesis, s considers $\mathcal{B}_n$ valid, and again, as s is correct and only inserts B' in $\mathcal{B}.\mathsf{preds}$ in line 8 whenever s considers B' valid in line 6,* (iii) *s*

considers all $B' \in \mathcal{B}$.preds *valid.*

Lemma 4.1.6 *For every correct server s executing* gossip *of Algorithm 6 $\mathcal{G}$ is a block* DAG.

Proof 4.1.6 *We prove the lemma by induction on the times n the execution reaches line 7 or line 16 of Algorithm 6. As $\mathcal{G}$ is initialized to the empty block* DAG *in Algorithm 8 in line 3, $\mathcal{G}$ is a block* DAG *for the base case $n = 0$. In the step case, by induction hypothesis, $\mathcal{G}$ is a block* DAG*. By Lemma 4.1.4* $\mathcal{G}$.insert(B') *is a block* DAG *if* (i) valid(s, B') *holds, and* (ii) *for all $B \in B'$*.preds *holds $B' \in \mathcal{G}$. The former* (i)*,* valid(s, B')*, holds either by line 6 or by Lemma 4.1.5. As s inserts any block B which s has received and considers valid by lines 6–8, for the latter* (ii) *it suffices to show that s considers all $B \in B'$*.preds *valid. As s considers B' valid, by Definition 4.1.3* (ii)*, s considers all $B \in B'$*.preds *valid.*

To insert a block, s keeps track of its received blocks as candidate blocks in the buffer blks (line 4–5). Whenever s considers a $B' \in$ blks valid (line 6), s inserts B' in $\mathcal{G}$ (line 7). However, to consider a block B' valid, s has to consider all its predecessors valid—and s may not have yet received every $B \in B'$.preds. That is, $B' \in$ blks but $B \notin$ blks and $B \notin \mathcal{G}$ (*cmp.* line 10). Now, s can request forwarding of B from the server that built B', *i.e.* from s' where B'.n $= s'$, by sending `FWD` B to s' (lines 10–11). To prevent s from flooding s' an implementation would guard lines 10–11, *e.g.* by a timer $\Delta_{B'}$. That is, we implicitly assume that for every block B' a correct server waits a reasonable amount of time before (re-)issuing a forward request. The wait time should be informed by the estimated round-trip time and can be adapted for repeating forwarding requests.

On the other hand, s also answers to forwarding requests for a block B from s', where $B \in B'$.preds of some block B' disseminated by s (lines 12–13). It is not necessary to request forwarding from servers other than s'. We only require that correct servers will eventually share the same blocks. This mechanism, together with Assumption 4.1.1 and s's eventual dissemination of $\mathcal{B}$, allows us to establish the following lemma:

Lemma 4.1.7 *For a correct server s executing* gossip*, if s receives a block B, which s considers valid, then* (i) *every correct server will eventually receive B, and* (ii) *every correct server will eventually consider B valid.*

Proof 4.1.7 *For* (i)*, by assumption s considers B valid, and hence by lines 6–8 adds a reference to B to $\mathcal{B}$. As s is correct, s eventually will* $\mathsf{disseminate}()$*, and then s disseminates $\mathcal{B}$ in line 17. We refer to this disseminated $\mathcal{B}$ as B'. By Assumption 4.1.1, every correct server will eventually receive B'. Assume a correct server s', which has received B', but has not received B. As s' has not received B, by Definition 4.1.3* (iii)*, s' does not consider B' valid. After time $\Delta_{B'}$ by lines 10–11 s' will request B from s by sending* `FWD` *B. Again by Assumption 4.1.1, after s receives* `FWD` *B from s' by lines 12–13, s will send B to s', which will eventually arrive, and s' receives B.*

For (ii)*, we have to show, that* $\mathsf{valid}(s', B)$ *eventually holds for all correct servers s'. For Definition 4.1.3* (i)*, as s considers B valid and s is correct, B has a valid signature. This can be checked by every s'. We show Definition 4.1.3* (ii) (a) *and* (iii) *by induction on the sum of the length of the paths from genesis blocks to B. For the base case, B does not have predecessors. As s considers B valid, then B is a genesis block, and s' will consider B a genesis block, so Definition 4.1.3* (ii) (a) *and* (iii) *hold. For the step case, let $B' \in B.\mathsf{preds}$. By Lemma 4.1.7 ((i)), every correct server s' will eventually receive B'. By induction hypothesis, s' will eventually consider B' valid. The same reasoning holds for every $B' \in B.\mathsf{preds}$. It remains to show that B has exactly one parent or is a genesis block. Again, this follows by s considering B valid. As $B.\mathsf{parent} \in B.\mathsf{preds}$ s' also considers $B.\mathsf{parent}$ valid.*

In parallel to building $\mathcal{G}$, s builds its current block $\mathcal{B}$ by *(i)* continuously adding a reference to any block B', which s receives and considers valid in line 8 (adding at most one reference to B' by Lemma 4.1.8), and *(ii)* eventually sending $\mathcal{B}$ to every server in line 17.

Lemma 4.1.8 *For every block B every correct server s executing* gossip *of Algorithm 6 inserts* $\mathsf{ref}(B)$ *at most once in any block B' with $B'.\mathsf{n} = s$.*

Proof 4.1.8 *By line 4 of Algorithm 6, a correct server adds a block B to* blks *only if $B \notin \mathcal{G}$, and as* blks *is a set, B appears at most once in* blks*. Either B remains in* blks*, or by lines 6–8, for any block B' with $B'.\mathsf{n} = s$, after* $\mathsf{ref}(B)$ *is inserted in B', $B \in \mathcal{G}$ holds. Thus, for no future execution $B \notin \mathcal{G}$ holds and therefore $B \notin$* blks*. As s is correct, it will not enter lines 6–8 again for B.*

Just before s sends $\mathcal{B}$, s injects literal inscriptions of $(\ell_i, r_i) \in \mathsf{rqsts}$ into $\mathcal{B}$ in line 15. Now rs holds requests r_i for the protocol instances $\mathcal{P}$ with label ℓ_i. These requests will eventually be read in Algorithm 7. Finally, s signs $\mathcal{B}$ in line 15, sends $\mathcal{B}$ to every server, and starts building its next $\mathcal{B}$ in line 18 by incrementing the sequence number k, initializing preds with the parent block, as well as clearing rs and σ.

Example 4.1.2 *Recall the block DAG from Example 4.1.1, Figure 4.2. Assume s_2 holds this block DAG as $\mathcal{G}$ in Algorithm 6. Now, assume receives $B_5 = \{\mathsf{n} = s_1, \mathsf{k} = 3, \mathsf{preds} = [\mathsf{ref}(B_4)]\}$. Immediately, s_2 stores B_5 in* blks *(lines 3–4). Now, as* $\mathsf{valid}(s_2, B_5)$ *does not hold, s_2 sends* FWD $\mathsf{ref}(B_4)$ *to $B_5.\mathsf{n}$ (lines 10–11). When s_1 receives the message s_1 will send B_4 to s_2 (lines 12–13). Once s_2 receives $B_4 = \{\mathsf{n} = s_1, \mathsf{k} = 2, \mathsf{preds} = [\mathsf{ref}(B_3)]\}$, and as* $\mathsf{valid}(s_2, B_4)$ *holds, s_2 inserts B_4 in $\mathcal{G}$ (lines 6–9). If s_2 now receives B_4 again, B_4 will not be stored in* blks *(line 4). Finally,* $\mathsf{valid}(s_2, B_5)$ *holds and s_2 inserts B_5 in $\mathcal{G}$.*

So far we established, how s builds its own block DAG. Next we want to establish the concept of a *joint block DAG* between two correct servers s and s'. Let $\mathcal{G}_s$ and $\mathcal{G}_{s'}$ be the block DAG of s and s'. We define their *joint block DAG* $\mathcal{G}'$ as a block DAG $\mathcal{G}' \geqslant \mathcal{G}_s \cup \mathcal{G}_{s'}$. This joint block DAG is a block DAG for s and for s':

Lemma 4.1.9 *Let s and s' be correct servers with block DAGs $\mathcal{G}_s$ and $\mathcal{G}_{s'}$. Then their joint block DAG $\mathcal{G} \geqslant \mathcal{G}_s \cup \mathcal{G}_{s'}$ is a block DAG for s.*

Proof 4.1.9 *Let $bs = B_1, \ldots, B_{k-1}$ be blocks such that $B_i \in \mathcal{G}_{s'}$ but $B_i \notin \mathcal{G}_s$ for $1 \leqslant i < k$. We show the statement by induction on $|bs|$. As $\mathcal{G}_s$ is a block DAG for s, the statement holds for the base case. For the step case we pick a $B_i \in bs$ such that $B_i.\mathsf{preds} \cap bs = \varnothing$. Such a B_i exists, as in the worst case, $\mathcal{G}_s$ and $\mathcal{G}_{s'}$ are completely disjoint and B_i is a genesis block in $\mathcal{G}_s$. It remains to show that s considers B_i valid and all $B_i.\mathsf{preds}$ are in $\mathcal{G}_s$. Then by Lemma 4.1.4 $\mathcal{G}_s.\mathsf{insert}(B_i)$ is a block DAG and by induction hypothesis the statement holds. For all $B' \in B_i.\mathsf{preds}$ holds $B' \in \mathcal{G}_s$ by definition of bs. Moreover, as $\mathcal{G}_s$ is the block DAG of s, s considers every B' valid. Then by* (iii) *of Definition 4.1.3, together with the fact that s' is correct therefore* (i) *and* (ii) *hold for s, s considers B_i valid.*

Intuitively, we want any two correct servers to be able to 'gossip some more' and arrive at their joint block DAG $\mathcal{G}'$.

Lemma 4.1.10 *Let s and s' be correct servers with block DAGs $\mathcal{G}_s$ and $\mathcal{G}_{s'}$. By executing* gossip *in Algorithm 6, eventually s has a block DAG $\mathcal{G}'_s$ such that $\mathcal{G}'_s \geqslant \mathcal{G}_s \cup \mathcal{G}_{s'}$.*

Proof 4.1.10 *By Lemma 4.1.6 any block DAG $\mathcal{G}'$ obtained through* gossip *is a block DAG, and by Lemma 4.1.9 $\mathcal{G}'$ is a block DAG for s. It remains to show that by executing* gossip*, eventually $\mathcal{G}'$ will be the block DAG for s. As s' received and considers all $B \in \mathcal{G}_{s'}$ valid, by Lemma 4.1.7* (ii*) s will eventually consider every B valid. By executing* gossip*, s will eventually insert every B in its block DAG and $\mathcal{G}'$ will contain all $B \in \mathcal{G}_{s'}$.*

Lemma 4.1.11 *If $B_1 \in \mathcal{G}$ for the block DAG $\mathcal{G}$ of a correct server s, then eventually for a block DAG $\mathcal{G}'$ of s where $\mathcal{G}' \geqslant \mathcal{G}$ holds $B_2 \in \mathcal{G}'$ and $B_2.\mathsf{n} = s$ and $B_1 \rightharpoonup B_2$.*

Proof 4.1.11 *For a correct server s it holds that $B_1 \in \mathcal{G}$ only after s inserted B_1 either in line 7 or in line 16. Then by either line 8 or 18, respectively, $B_1 \in \mathcal{B}.\mathsf{preds}$ for $\mathcal{B}.\mathsf{n} = s$. As s is correct s will eventually call* disseminate() *and s will reach line 16 for $\mathcal{B}$ and insert $\mathcal{B}$ to $\mathcal{G}$ for some $\mathcal{G}' \geqslant \mathcal{G}$.*

In the next section, we will show how s and s' can independently interpret a deterministic protocol $\mathcal{P}$ on this joint block DAG.

4.2 Interpreting a Protocol

Every server s interprets the protocol $\mathcal{P}$ embedded in its local block DAG $\mathcal{G}$. This interpretation is completely *decoupled* from building the block DAG in Algorithm 6. To interpret one *protocol instance* of $\mathcal{P}$ tagged with label ℓ, server s locally runs one *process instance* of $\mathcal{P}$ with label ℓ for every other server $s_i \in \mathsf{Srvrs}$. Thereby, s treats $\mathcal{P}$ as a black-box which *(i)* takes a request or a message, and *(ii)* returns messages or an indication. A server s can fully simulate the protocol instance $\mathcal{P}$ for any other server because their requests and messages have been embedded in the block DAG $\mathcal{G}$ by Algorithm 6. User requests r_j to $\mathcal{P}$ are embedded in a block $B \in \mathcal{G}$ in $B.\mathsf{rs}$ and s reads these requests from the block and passes them on to the simulation of $\mathcal{P}$. Since $\mathcal{P}$ is deterministic, s can—after the initial request r_j for $\mathcal{P}$—compute all subsequent messages which would have been sent in $\mathcal{P}$ by interpreting edges between blocks, such as $B_1 \rightharpoonup B_2$, as messages sent from $B_1.\mathsf{n}$ to $B_2.\mathsf{n}$. There is no need for explicitly sending these messages. Indeed, our goal is to show that the interpretation of a deterministic protocol $\mathcal{P}$ embedded in a block DAG implements a reliable point-to-point link.

We fix the following notation: the set of all messages in a protocol $\mathcal{P}$ is $\mathsf{M}_{\mathcal{P}}$. Every message $m \in \mathsf{M}_{\mathcal{P}}$ has a $m.\mathsf{sender}$ and a $m.\mathsf{receiver}$. We assume an arbitrary, but fixed, total order on messages: $<_{\mathsf{M}}$. A protocol $\mathcal{P}$ is *deterministic* if a state q and a sequence of messages $m \in \mathsf{M}_{\mathcal{P}}$ determine state q' and outgoing messages $M \subseteq 2^{\mathsf{M}_{\mathcal{P}}}$. In particular, deterministic protocols do not rely on random behaviour such as coin-flips.

To treat $\mathcal{P}$ as a black-box, we assume the following high-level interface: *(i)* an interface to *request* $r \in \mathsf{Rqsts}_{\mathcal{P}}$, and *(ii)* an interface where $\mathcal{P}$ *indicates* $i \in \mathsf{Inds}_{\mathcal{P}}$. When a request r reaches a process instance, we assume that it immediately returns messages $m_1, \ldots, m_k$ triggered by r. This is justified, as s runs all process instances locally. As requests do not depend on the state of

Algorithm 7: Interpreting protocol $\mathcal{P}$ on the block DAG $\mathcal{G}$.

1 **module** $\mathsf{interpret}(\mathcal{G} \in \mathsf{Dags}, \mathcal{P} \in \textbf{module})$
2 $\quad \mathcal{I}[B \in \mathsf{Blks}] \coloneqq \mathit{false} \in \mathsf{Bool}$
3 $\quad$ **when** $B \in \mathcal{G}$ **where** $\mathsf{eligible}(B)$
4 $\quad\quad B.\mathsf{Pls} \coloneqq$ **copy** $B.\mathsf{parent}.\mathsf{Pls}$
5 $\quad\quad$ **for every** $(\ell_j \in \mathcal{L}, r_j \in \mathsf{Rqsts}) \in B.\mathsf{rs}$
6 $\quad\quad\quad B.\mathsf{Ms}[\mathsf{out}, \ell_j] \coloneqq B.\mathsf{Pls}[\ell_j].r_j$
7 $\quad\quad$ **for every** $\ell_j \in \{\ell_j \mid (\ell_j, r_j) \in B_j.\mathsf{rs} \wedge B_j \in \mathcal{G} \wedge B_j \rightharpoonup^+ B\}$
8 $\quad\quad\quad$ **for every** $B_i \in B.\mathsf{preds}$
9 $\quad\quad\quad\quad B.\mathsf{Ms}[\mathsf{in}, \ell_j] \coloneqq B.\mathsf{Ms}[\mathsf{in}, \ell_j] \cup \{m \mid m \in B_i.\mathsf{Ms}[\mathsf{out}, \ell_j]$ and $m.\mathsf{receiver} = B.\mathsf{n}\}$
10 $\quad\quad\quad$ **for every** $m \in B.\mathsf{Ms}[\mathsf{in}, \ell_j]$ **ordered by** $<_{\mathsf{M}}$
11 $\quad\quad\quad\quad B.\mathsf{Ms}[\mathsf{out}, \ell_j] \coloneqq B.\mathsf{Ms}[\mathsf{out}, \ell_j] \cup B.\mathsf{Pls}[\ell_j].\mathsf{receive}(m)$
12 $\quad\quad \mathcal{I}[B] = \mathit{true}$
13 $\quad$ **when** $B.\mathsf{Pls}[\ell_j].i$
14 $\quad\quad \mathsf{indicate}(\ell_j, i, B.\mathsf{n})$

the process instance, also these messages do not depend on the current state of process instance. We also assume a low-level interface for $\mathcal{P}$ to *receive* a message m. Again, we assume that when m reaches a process instance, it immediately returns the messages $m_1, \ldots, m_k$ triggered by m.

Algorithm 7 shows the protocol executed by s for interpreting a deterministic protocol $\mathcal{P}$ on a block DAG $\mathcal{G}$. The key task is to 'get messages from one block and give them to the next block'.

Therefore s traverses through every $B \in \mathcal{G}$. To keep track of which blocks in $\mathcal{G}$ it has already interpreted, s uses $\mathcal{I}$ in line 2. Note, that edges in $\mathcal{G}$ impose a partial order: s considers a block $B \in \mathcal{G}$ as $\mathsf{eligible}(B)$ for interpretation if *(i)* $\mathcal{I}[B] = \mathit{false}$, and *(ii)* for every $B_i \in B.\mathsf{preds}$, $\mathcal{I}[B_i] = \mathit{true}$ holds. While there may be more than one B eligible, every $B \in \mathcal{G}$ is interpreted eventually:

Lemma 4.2.1 *For a block $B \in \mathcal{G}$ and a correct server executing* $\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ *in Algorithm 7 every B is eventually picked in line 3.*

Proof 4.2.1 *To pick B in line 3,* $\mathsf{eligible}(B)$ *has to hold. As $\mathcal{G}$ is finite and* *acyclic, every $B \in \mathcal{G}$ is* $\mathsf{eligible}(B)$ *eventually.*

Now s picks an eligible B in line 3 and *interprets* B in lines 4–12. To interpret B, s needs to keeps track of two variables for every protocol instance ℓ_j: 1. the state of the process instance ℓ_j for a server $s_i \in \mathsf{Srvrs}$ in $\mathsf{PIs}[\ell_j]$, and 2. the state of in-going and out-going messages in $\mathsf{Ms}[\mathsf{in}, \ell_j]$ and $\mathsf{Ms}[\mathsf{out}, \ell_j]$.

Our goal is to track changes to these two variables—the process instances PIs and message buffers Ms—throughout the interpretation of $\mathcal{G}$. To do so, we assign their state to every block B. Before B is interpreted, we assume $B.\mathsf{PIs}[\ell_j]$ to be initialized with $\bot$, and $B.\mathsf{Ms}[d \in [\{\mathsf{in}, \mathsf{out}\}, \ell_j]$ with $\varnothing$. They remain so while B is $\mathsf{eligible}$:

Lemma 4.2.2 *When the execution of* $\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ *reaches line 7 of Algorithm 7 then for all $\ell_j \in \{\ell_j \mid (\ell_j, r) \in B_j.\mathsf{rs} \wedge B_j \in \mathcal{G} \wedge B_j \rightharpoonup^* B\}$ holds $B.\mathsf{PIs}[\ell_j] \neq \bot$.*

Proof 4.2.2 *We show the statement by induction on the length of the longest path from the genesis blocks to B. The base cases $n = 0$ holds by assumption, as $\mathsf{PIs}[\ell]$ is started on every genesis block. For the step case, by induction hypothesis the statement holds for $B_i \in B.\mathsf{preds}$, and as $B.\mathsf{parent} \in B.\mathsf{preds}$ by line 4 the statement holds.*

Lemma 4.2.3 *For $B \in \mathcal{G}$ if $\mathcal{I}[B] = \mathit{false}$ then $B.\mathsf{Ms}[d, \ell] = \varnothing$ and $B.\mathsf{PIs}[\ell] = \bot$ for $\ell \in \mathcal{L}$ and $d \in \{\mathsf{in}, \mathsf{out}\}$.*

Proof 4.2.3 *For every B, $\ell \in \mathcal{L}$, and $d \in \{\mathsf{in}, \mathsf{out}\}$, initially we have $B.\mathsf{Ms}[d, \ell] = \varnothing$ and $B.\mathsf{PIs}[\ell] = \bot$. Assume towards a contradiction that $B.\mathsf{Ms}[d, \ell] \neq \varnothing$ or $B.\mathsf{PIs}[\ell] \neq \bot$. As $B.\mathsf{Ms}[d, \ell]$ and $B.\mathsf{PIs}[\ell]$ are only modified in lines 4–12 after B is picked in line 3, then by line 12 $\mathcal{I}[B] = \mathit{true}$ contradicting $\mathcal{I}[B] = \mathit{false}$.*

After interpreting B, 1. $B.\mathsf{PIs}[\ell_j]$ holds the state of the process instance ℓ_j of the server s_i, which built B, *i.e.*, $s_i = B.\mathsf{n}$, and 2. $B.\mathsf{Ms}[\mathsf{in}, \ell_j]$ holds the in-going messages for s_i and $\mathsf{Ms}[\mathsf{out}, \ell_j]$ the out-going messages from s_i for process instance ℓ_j[1].

As a starting point for computing the state of $B.\mathsf{PIs}[\ell_j]$, s copies the state from the parent block of B in line 4. For the base case, *i.e.* all (genesis) blocks B without parents, we assume $B.\mathsf{PIs}[\ell_j] \coloneqq$ **new process** $\mathcal{P}(\ell_j, s_i)$ where $s_i = B.\mathsf{n}$. This is effectively a simplification: we assume a running process instance ℓ_j for every $s_i \in \mathsf{Srvrs}$. In an implementation, we would only start process instances for ℓ_j after receiving the first message or request for ℓ_j for $s_i = B.\mathsf{n}$. Now in our simplification, we start all process instances for every label at the genesis blocks and pass them on from the parent blocks. This leads us to our step case: B has a parent. As $B.\mathsf{parent} \in B.\mathsf{preds}$, $B.\mathsf{parent}$ has been interpreted and moreover $B.\mathsf{parent}.\mathsf{n} = s_i$:

Lemma 4.2.4 *For all $B.\mathsf{PIs}[\ell] \neq \perp$ holds that $B.\mathsf{PIs}[\ell]$ was started with $\mathcal{P}(\ell, B.\mathsf{n})$.*

Proof 4.2.4 *Either* (i) *B is a genesis block, and then by assumption started with $B.\mathsf{n}$ and ℓ, or* (ii) *B has a parent and by line 4, $\mathsf{PIs}[\ell]$ is copied from $B.\mathsf{parent}$ and as $B.\mathsf{parent}.\mathsf{n} = B.\mathsf{n}$, $B.\mathsf{PIs}[\ell]$ was initialized with $B.\mathsf{n}$ and ℓ (Lemma 4.2.2).*

Next, to advance the copied state on B, s processes 1. all incoming requests r_j given by $B.\mathsf{rs}$ in lines 5–6, and 2. all incoming messages from $B_i.\mathsf{n}$ to $B.\mathsf{n}$ given by $B_i \rightharpoonup B$ in lines 8–11. For the former (1), s reads the labels and requests from the field $B.\mathsf{rs}$. Here r_j is the literal transcription of the user's original request given to $\mathcal{P}$. To give an example, if $\mathcal{P}$ is reliable broadcast, then r_j could read '$\mathsf{broadcast}(42)$' (*cf.* Section 4.3). When interpreting, s requests r_j from $B.\mathsf{n}$'s simulated protocol instance: $B.\mathsf{PIs}[\ell_j].r_j$. For the latter (2), s collects *(i)* in $B.\mathsf{Ms}[\mathsf{in}, \ell]$ all messages for $B.\mathsf{n}$ from $B_i.\mathsf{Ms}[\mathsf{out}, \ell]$ where $B_i \in B.\mathsf{preds}$ in lines 8–9 and then feeds *(ii)* $m \in B.\mathsf{Ms}[\mathsf{in}, \ell]$ to $B.\mathsf{PIs}[\ell]$

[1] An equivalent representation would keep process instances $\mathsf{PIs}[B, \ell_j, B.\mathsf{n}]$ and message buffers $\mathsf{Ms}[B, d \in \{\mathsf{in}, \mathsf{out}\}, \ell_j]$ explicitly as global state. We chose this notation to accentuate the information flow throughout $\mathcal{G}$.

in lines 10–11 in order $<_{\mathsf{M}}$. This (arbitrary) order is a simple way to guarantee that every server interpreting Algorithm 7 will execute exactly the same steps. By feeding those messages and requests to $B.\mathsf{PIs}[\ell_j]$ in lines 6 and 11 s computes 1. the next state of $B.\mathsf{PIs}[\ell_j]$ and 2. the out-going messages from $B.\mathsf{n}$ in $B.\mathsf{Ms}[\mathsf{out}, \ell_j]$. By construction, $m.\mathsf{sender} = B.\mathsf{n}$ for $m \in B.\mathsf{Ms}[\mathsf{out}, \ell_j]$, *cf.* the following Lemma 4.2.5.

Lemma 4.2.5 *If* $m \in B.\mathsf{Ms}[\mathsf{out}, \ell]$ *then* $m.\mathsf{sender} = B.\mathsf{n}$.

Proof 4.2.5 *By lines 6 and 11 of Algorithm 7* $m \in B.\mathsf{Ms}[\mathsf{out}, \ell]$ *if either* $m \in B.\mathsf{PIs}[\ell].(B.\mathsf{rs})$ *or* $m \in B.\mathsf{PIs}[\ell].\mathsf{receive}(m')$ *for some* m' *of no importance. Important is, that* $B.\mathsf{PIs}[\ell]$ *was initialized by* $B.\mathsf{n}$ *by Lemma 4.2.4, and thus every out-going message* m *has* $m.\mathsf{sender} = B.\mathsf{n}$. *It remains to show that every* B *with* $B.\mathsf{n} = s$ *was build by* s, *which follows by the signature* $B.\mathsf{n}$.

Lemma 4.2.6 *If* $m \in B.\mathsf{Ms}[\mathsf{out}, \ell]$ *then there is a block* B' *such that* $(\ell, r) \in B'.\mathsf{rs}$ *and* $B' \rightharpoonup^* B$.

Proof 4.2.6 *In Algorithm 7,* $m \in B.\mathsf{Ms}[\mathsf{out}, \ell]$ *only after the execution reaches either 1. line 6, and then* $B' = B$, *or 2. line 11, end then by line 7 exists a* B_j *such that* $(\ell_j, r) \in B_j.\mathsf{rs}$ *for a label* $\ell \in \{\ell_j \mid (\ell_j, r_j) \in B_j.\mathsf{rs} \land B_j \in \mathcal{G} \land B_j \rightharpoonup^+ B\}$.

Once, s has completed this, s marks B as interpreted in line 12 and can move on to the next eligible block. After s interpreted B, the simulated process instance $B.\mathsf{PIs}[\ell_j]$ may indicate $i \in \mathsf{Inds}$. If this is the case, s indicates i for ℓ_j on behalf of $B.\mathsf{n}$ in lines 13–14. Note, that none of the steps used the fact that it was s who interpreted $B \in \mathcal{G}$. So, for every B, every $s' \in \mathsf{Srvrs}$ will come to the exact same conclusion.

Still we glossed over a detail, s actually had to take a choice—more than one B may have been eligible in line 3. This is a feature: by having this choice we can think of interpreting a $\mathcal{G}'$ with $\mathcal{G}' \geqslant \mathcal{G}$ as an 'extension' of interpreting $\mathcal{G}$. And, for two eligible B_1 and B_2 it does not matter if we pick B_1 before

B_2. Informally, this is because when we pick B_1 in line 3, only the state with respect to B_1 is modified—and this state does not depend on B_2:

Lemma 4.2.7 *For a block* $B \in \mathcal{G}$ *and an* $\ell \in \mathcal{L}$*, if* $\mathcal{I}[B]$ *holds,* (i) *then* $B.\mathsf{Ms}[d, \ell]$ *will never be modified again for every* $d \in \{\mathsf{in}, \mathsf{out}\}$*.* (ii) *then* $B.\mathsf{Pls}[\ell]$ *will never be modified again.*

Proof 4.2.7 *For part* (i)*, assume that* $B.\mathsf{Ms}[d, \ell]$ *is modified. This can only happen in lines 6, 9, and 11 and only for* B *picked in line 3, but as* $\mathcal{I}[B]$*,* B *cannot be picked in line 3, leading to a contradiction. For part* (ii) *assume that* $B.\mathsf{Pls}[d, \ell]$ *is modified. This can only happen in lines 4 and 11, and only for* B *picked in line 3, but as* $\mathcal{I}[B]$*,* B *cannot be picked in line 3, leading to a contradiction.*

Another detail we glossed over is line 7: when interpreting B, s interprets the process instances of every ℓ_j relevant on B *at the same time*. Again, because $\ell_j \neq \ell'_j$ are independent instances of the protocol with disjoint messages, *i.e.*, $B_i.\mathsf{Ms}[\mathsf{out}, \ell_j]$ in line 9 is independent of any $B_i.\mathsf{Ms}[\mathsf{out}, \ell'_j]$, they do not influence each other and the order in which we process ℓ_j does not matter.

Finally, we give some intuition on how byzantine servers can influence $\mathcal{G}$ and thus the interpretation of $\mathcal{P}$. When running gossip, a byzantine server $\check{s}$ can only manipulate the state of $\mathcal{G}$ by *(i)* sending an equivocating block, *i.e.* building a B and B' with $\check{s} = B.\mathsf{parent.n}$ and $\check{s} = B'.\mathsf{parent.n}$. When interpreting B and B', s will split the state for $\check{s}$ and have two 'versions' of $\mathsf{Pls}[\ell_j]$—$B'.\mathsf{Pls}[\ell_j]$ and $B.\mathsf{Pls}[\ell_j]$—sending conflicting messages for ℓ_j to servers referencing B and B'. However, as $\mathcal{P}$ is a BFT protocol, the servers s_i simulating $\mathcal{P}$ (run by s) can deal with equivocation. Then $\check{s}$ could *(ii)* reference a block multiple times, or *(iii)* never reference a block. Again as $\mathcal{P}$ is a BFT protocol, the servers s_i simulating $\mathcal{P}$ can deal with duplicate messages and with silent servers.

Going back to Algorithm 7, the key task of s interpreting $\mathcal{G}$ is to get messages from one block to the next block. So we can see this interpretation

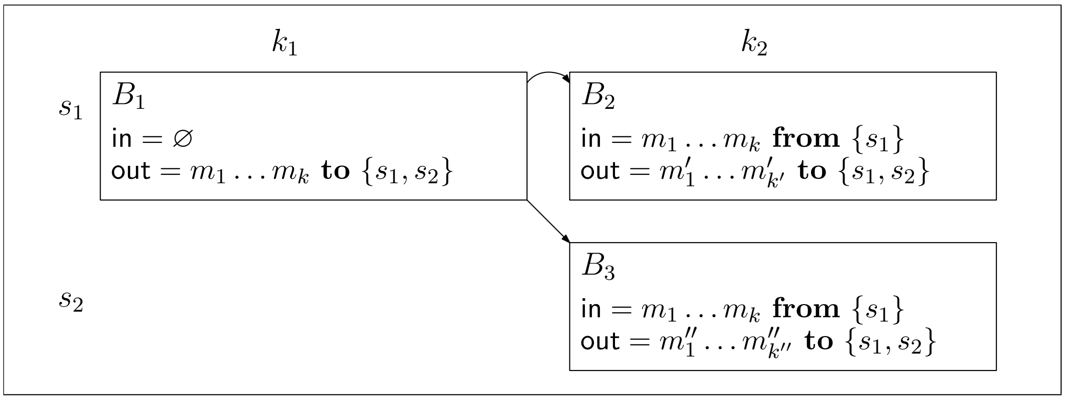

Figure 4.4: Interpretation of a block DAG.

of a block DAG as an implementation of a *communication channel.* That is, for a correct server s executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ *(i)* a server s_1 *sends* messages $m_1, \dots, m_k$ for a protocol instance ℓ_j in either line 6 or line 11 of Algorithm 7, and *(ii)* a server s_2 *receives* a message m for a protocol instance ℓ_j in line 11 of Algorithm 7.

Example 4.2.1 *Figure 4.4 shows* $\mathsf{Ms}[\mathsf{in}, \ell]$ *and* $\mathsf{Ms}[\mathsf{out}, \ell]$ *for some label* ℓ*. The messages* $m_1, \dots, m_k$ *in* B_1 *were not triggered by any input as* $\mathsf{in} = \varnothing$*, so they stem from a request to* s_1*. Next, in the interpretation the* $m_1, \dots, m_k$ *are moved to* $\mathsf{Ms}[\mathsf{in}, \ell]$ *of the corresponding successor blocks* B_2 *and* B_3*. There,* $m_1, \dots, m_{k'}$ *trigger messages* $m'_1, \dots, m'_k$ *and* $m''_1, \dots, m''_{k''}$ *in* $\mathsf{Ms}[\mathsf{out}, \ell]$*, respectively.*

The next lemma relates the sent and received messages with the message buffers Ms and follows from tracing changes to the variables in Algorithm 7:

Lemma 4.2.8 *For a correct server* s *executing* $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$

(i) *a server* s_1 sends m *for a protocol instance* ℓ' *iff there is a* $B_1 \in \mathcal{G}$ *with* $B_1.\mathsf{n} = s_1$ *such that* $m \in B_1.\mathsf{Ms}[\mathsf{out}, \ell']$ *for a* $B' \in \mathcal{G}$ *with* $(\ell', r) \in B'.\mathsf{rs}$ *and* $B' \rightharpoonup^* B_1$*.*

(ii) *a server* s_2 receives *a message* m *for protocol instance* ℓ' *iff there are some* $B_1, B_2 \in \mathcal{G}$ *with* $B_1 \rightharpoonup B_2$ *and* $B_2.\mathsf{n} = s_2$ *and* $m \in B_2.\mathsf{Ms}[\mathsf{in}, \ell']$ *for a* $B' \in \mathcal{G}$ *such that* $(\ell', r) \in B'.\mathsf{rs}$ *and* $B' \rightharpoonup^* B_1$*.*

Proof 4.2.8 *By definition s_1 sends m for some protocol instance ℓ' if s reaches in Algorithm 7 either line 6 with $B.\mathsf{rs}$, or line 11 with $B.\mathsf{PIs}[\ell'].\mathsf{receive}(m)$ for some B picked in line 3. By Lemma 4.2.2 $B.\mathsf{PIs}[\ell'] \neq \bot$ and $B.\mathsf{PIs}[\ell'].\mathsf{n} = s_1$ by assumption, by Lemma 4.2.4 $B.\mathsf{n} = s_1$. B will be our witness for B_1. Now $m \in B.\mathsf{Ms}[\mathsf{out}, \ell']$, by the assignment in either line 6 with $(\ell', r) \in B.\mathsf{rs}$ (by line 5), or in line 11 with $(\ell', r) \in B_j.\mathsf{rs}$ for some $B_j \rightharpoonup^+ B$ (by line 7). B_j is our witness for $B' \neq B_1$. For the other direction, we have $B_1 \in \mathcal{G}$ with $B_1.\mathsf{n} = s_1$ such that $m \in B_1.\mathsf{Ms}[\mathsf{out}, \ell']$ for a $B' \in \mathcal{G}$ with $(\ell', r) \in B'.\mathsf{rs}$ and $B' \rightharpoonup^* B_1$. By Lemma 4.2.1, eventually B_1 is picked in Algorithm 7 line 3. By assumption, $m \in B_1.\mathsf{Ms}[\mathsf{out}, \ell']$ through either* (i) *line 6, or* (ii) *as $B' \rightharpoonup^+ B_1$ and thus $\ell' \in \{\ell_j \mid (\ell_j, r) \in B_j.\mathsf{rs} \land B_j \in \mathcal{G} \land B_j \rightharpoonup^+ B\}$ from line 11. Then, by definition, s_1 sends m for protocol instance ℓ'.*

The following lemma shows our key observation from before: interpreting a block DAG is independent from the server doing the interpretation. That is, s and s' will arrive at the same state when interpreting $B \in \mathcal{G}$.

Lemma 4.2.9 *If $\mathcal{G} \leqslant \mathcal{G}'$ then for every $B \in \mathcal{G}$, a deterministic protocol $\mathcal{P}$ and correct servers s and s' executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ and $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ it holds that $B.\mathsf{PIs}[\ell_j] = B.\mathsf{PIs}'[\ell_j]$ and $B.\mathsf{Ms}[\mathsf{out}, \ell_j] = B.\mathsf{Ms}'[\mathsf{out}, \ell_j]$ for $(\ell_j, r) \in B_j.\mathsf{rs}$ with $B_j \rightharpoonup^n B$ for $n \geqslant 0$.*

Proof 4.2.9 *In this proof, when executing $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ we write Ms' and PIs' to distinguish from Ms and PIs when executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$. We show $B_1.\mathsf{Ms}[\mathsf{out}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{out}, \ell_j]$ and $B_1.\mathsf{PIs}[\ell_j] = B_1.\mathsf{PIs}'[\ell_j]$ by induction on n—the length of the path from B_j to B_1 in $\mathcal{G}$ and $\mathcal{G}'$. For the base case we have $B_1 = B_j$ and $\ell_j \in \{\ell_j \mid (\ell_j, r_j) \in B_1.\mathsf{rs}\}$. By Lemma 4.2.1, B_1 is picked eventually in line 3 of Algorithm 7 when executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$. Then, by line 6 $B_1.\mathsf{Ms}[\mathsf{out}, \ell]$ is $B_1.\mathsf{PIs}[\ell_j].(B_1.\mathsf{rs})$. By the same reasoning, when executing $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$, $B_1.\mathsf{Ms}'[\mathsf{out}, \ell] = B_1.\mathsf{PIs}[\ell_j].(B_1.\mathsf{rs})$. As $B_1.\mathsf{PIs}[\ell_j].(B_1.\mathsf{rs})$ are deterministic and depend only on B_1, ℓ_j, and $\mathcal{P}$, we know that $B_1.\mathsf{PIs}[\ell] = B_1.\mathsf{PIs}'[\ell]$ and $B_1.\mathsf{PIs}[\ell] = B_1.\mathsf{PIs}'[\ell]$, and conclude the base case. For the* *step case by induction hypothesis for* $B_i \in B_1.\mathsf{preds}$ *with* $B_j \rightharpoonup^{n-1} B_i$ *holds* (i) $B_i.\mathsf{Ms}[\mathsf{out}, \ell_j] = B_i.\mathsf{Ms}'[\mathsf{out}, \ell_j]$*, and* (ii) $B_i.\mathsf{PIs}[\ell_j] = B_i.\mathsf{PIs}'[\ell_j]$*. Again by Lemma 4.2.1,* B_1 *is picked eventually in line 3 of Algorithm 7 when executing* $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ *and* $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$*. In line 4 and as* $B_1.\mathsf{parent} \in B_1.\mathsf{preds}$ *and* (ii)*, now* $B_1.\mathsf{PIs}[\ell_j] = B_1.\mathsf{PIs}'[\ell_j]$*. Now, as* $\mathcal{P}$ *is deterministic, we only need to establish that* $B_1.\mathsf{Ms}[\mathsf{in}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{in}, \ell_j]$ *to conclude that* $B_1.\mathsf{PIs}[\ell_j] = B_1.\mathsf{PIs}'[\ell_j]$ *and* $B_1.\mathsf{Ms}[\mathsf{out}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{out}, \ell_j]$*, which as* $(\ell_j, r) \notin B_1.\mathsf{rs}$*, is only modified in this line 11. By Lemma 4.2.3 below, we know for both executions that* $B_1.\mathsf{Ms}[\mathsf{in}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{in}, \ell_j] = \varnothing$*, before* B_1 *is picked. Now, by* (i) *and line 9* $B_1.\mathsf{Ms}[\mathsf{in}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{in}, \ell_j]$*, and we conclude the proof.*

A straightforward consequence of Lemma 4.2.9 is, that when in the interpretation of s, a server s_1 sends a message m for ℓ_j, then s_1 sends m in the interpretation of s':

Lemma 4.2.10 *For a correct server* s *executing* $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ *if a server* s_1 *sends a message* m *for a protocol instance* ℓ_j*, then* s_1 *sends* m *for a correct server* s' *executing* $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ *for a block* DAG $\mathcal{G}' \geqslant \mathcal{G}$*.*

Proof 4.2.10 *Again, in the following proof, we write* Ms' *and* PIs' *when executing* $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ *to distinguish from* Ms *and* PIs *when executing* $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$*. As* s_1 *sends a message* m *for a protocol instance* ℓ_j*, by Lemma 4.2.8* (i) *there is a* $B_1 \in \mathcal{G}$ *with* $B_1.\mathsf{n} = s_1$ *such that* $m \in B_1.\mathsf{Ms}[\mathsf{out}, \ell_j]$ *for a* $B_j \in \mathcal{G}$ *with* $(\ell_j, r) \in B_j.\mathsf{rs}$ *and* $B_j \rightharpoonup^n B_1$ *for* $n \geqslant 0$*. By* $\mathcal{G}' \geqslant \mathcal{G}$*,* $B_1 \in \mathcal{G}$*,* $B_j \in \mathcal{G}$*, and the path* $B_j \rightharpoonup^n B_1$ *are in* $\mathcal{G}'$*. By Lemma 4.2.9* $m \in B_1.\mathsf{Ms}'[\mathsf{out}, \ell_j]$*, and then by Lemma 4.2.8* (i)*,* s_1 *sends s* m *for a correct server* s' *executing* $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$*.*

Curiously, s_1 does not have to be correct: we know s_1 sent a block B in $\mathcal{G}$, that corresponds to a message m in the interpretation of s. Now this block will be interpreted by s' and the same message will be interpreted—and for that the server s_1 does not need to be correct. By Lemma 4.2.11 $\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ has the properties of an authenticated perfect point-to-point link after [23, Module 2.5, p. 42] in Figure 2.1.

Lemma 4.2.11 *For a block DAG $\mathcal{G}$ and a correct server s executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ holds*

(i) *if a correct server s_1 sends a message m for a protocol instance ℓ to a correct server s_2, then s_2 eventually receives m for protocol instance ℓ for a correct server s' executing $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ and a block DAG $\mathcal{G}' \geqslant \mathcal{G}$ (*reliable delivery*).*

(ii) *for a protocol instance ℓ no message is received by a correct server s_2 more than once (*no duplication*).*

(iii) *if some correct server s_2 receives a message m for protocol instance ℓ with sender s_1 and s_1 is correct, then the message m for protocol instance ℓ was previously sent to s_2 by s_1 (*authenticity*).*

We first give a proof sketch: for *(i)*, we observe that every message m sent in $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ will be sent in $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ for $\mathcal{G}' \geqslant \mathcal{G}$ by Lemma 4.2.10. Now by Lemma 4.1.10, s' will eventually have some $\mathcal{G}' \geqslant \mathcal{G}$. By Lemma 4.2.8 *(i)* we have witnesses $B_1, B_2 \in \mathcal{G}'$ with $B_1 \rightharpoonup B_2$, and by Lemma 4.2.8 *(ii)* we found a witness B_2 to receive the message on when executing $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$. For *(ii)*, we observe, that duplicate messages are only possible if s_2 inserted the block B_1, which gives rise to the message m, in two different blocks built by s_2, but this contradicts the correctness of s_2 by Lemma 4.1.8. For *(iii)*, we observe that only s_1 can build and sign any block B_1 with $s_1 = B.\mathsf{n}$, which gives rise to m.

Proof 4.2.11 *For* (i) reliable delivery*, by assumption s_1 sends a message m to a correct server s_2 for a correct server s executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$. By Lemma 4.1.10 s' will eventually have some $\mathcal{G}_1 \geqslant \mathcal{G}$. Then by Lemma 4.2.10, s_1 sends m in $s'.\mathsf{interpret}(\mathcal{G}_1, \mathcal{P})$ for $\mathcal{G}_1 \geqslant \mathcal{G}$. Then by Lemma 4.2.8*(i) *there is a $B_1 \in \mathcal{G}_1$ with $B_1.\mathsf{n} = s_1$ such that $m \in B_1.\mathsf{Ms}[\mathsf{out}, \ell_j]$ for $B_j \in \mathcal{G}_1$ with $(\ell_j, r) \in B_j.\mathsf{rs}$ and $B_j \rightharpoonup^* B_1$. With B_1 we found our first witness. By Lemma 4.1.11, there is $\mathcal{G}_2 \geqslant \mathcal{G}_1$ such that $B_2 \in \mathcal{G}_2$ and $B_2.\mathsf{n} = s_2$ and $B_1 \rightharpoonup B_2$. Then by Lemma 4.1.10 eventually s' will have some $\mathcal{G}' \geqslant \mathcal{G}_2$. By $m \in B_1.\mathsf{Ms}[\mathsf{out}, \ell_j]$,*

$B_1 \rightharpoonup B_2$ and $m.\mathsf{receiver} = s_2$ by assumption, by lines 9–10 of Algorithm 7 we have $B.m \in \mathsf{Ms}[\mathsf{in}, \ell_j]$. Now we have found our second witness B_2. By Lemma 4.2.8 (ii)*, s_2 receives m in $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$*

For (ii) no duplication*, we assume towards a contradiction, that s_2 received m more than once. Then by Lemma 4.2.8*(ii) *there are some $B_1, B_2 \in \mathcal{G}$ with $B_1 \rightharpoonup B_2$, $B_2.\mathsf{n} = s_2$ and $m \in B_2.\mathsf{Ms}[\mathsf{in}, \ell]$,* and *$B_1' \rightharpoonup B_2'$, $B_2'.\mathsf{n} = s_2$ and $m \in B_2'.\mathsf{Ms}[\mathsf{in}, \ell]$ for a $B_j \in \mathcal{G}$ such that $(\ell, r) \in B_j.\mathsf{rs}$ and $B_j \rightharpoonup^* B_1$, but $B_2 \neq B_2'$. That s_2 received the exact same message m twice is only possible, if $B_1 = B_1'$. That is, s_2 built $B_2' \neq B_2$ and inserted B_1 in both, which contradicts Lemma 4.1.8 as s_2 is correct.*

For (iii) authenticity*, by Lemma 4.2.8* (ii) *there are some $B_1, B_2 \in \mathcal{G}$ with $B_1 \rightharpoonup B_2$ and $B_2.\mathsf{n} = s_2$ and $m \in B_2.\mathsf{Ms}[\mathsf{in}, \ell]$ for a $B \in \mathcal{G}$ such that $(\ell, r) \in B_j.\mathsf{rs}$ and $B_j \rightharpoonup^* B_1$. Then by line 9 of Algorithm 7 exists an $B_i \in B_2.\mathsf{preds}$ such that $m \in B_i.\mathsf{Ms}[\mathsf{out}, \ell]$. As $m \in B_i.\mathsf{Ms}[\mathsf{out}, \ell]$ by Lemma 4.2.5 $B_i.\mathsf{n} = m.\mathsf{sender}$ and as $m.\mathsf{sender} = s_1$, $B_i.\mathsf{n} = s_1$. B_i will be our witness for B_1. As $m \in B_i.\mathsf{Ms}[\mathsf{out}, \ell]$ by Lemma 4.2.6 there is a B' such that $(\ell, r) \in B'.\mathsf{rs}$ and $B' \rightharpoonup^* B_i$. B' is our witness for B_j. Hence there is a $B_1 \in \mathcal{G}$ with $B_1.\mathsf{n} = s_1$ such that $m \in B_1.\mathsf{Ms}[\mathsf{out}, \ell]$ for a $B_1 \in \mathcal{G}$ with $(\ell, r) \in B_j.\mathsf{rs}$ and $B_j \rightharpoonup^* B_1$ and by Lemma 4.2.8* (i) *s_1 m was sent by s_1.*

Before we compose gossip and $\mathsf{interpret}$ in the next section under a shim, we highlight the key benefits of using $\mathsf{interpret}$ in Algorithm 7. By leveraging the block DAG structure together with $\mathcal{P}$'s determinism, we can *compress messages* to the point of omitting some of them. When looking at line 11 of Algorithm 7, the messages in the buffers $\mathsf{Ms}[\mathsf{out}, \ell]$ and $\mathsf{Ms}[\mathsf{in}, \ell]$ have never been sent over the network. They are locally computed, functional results of the calls $\mathsf{receive}(m)$. The only 'messages' actually sent over the network are the requests r_i read from $B.\mathsf{rs}$ in line 6. To determine the messages following from these request, the server s simulates an instance of protocol $\mathcal{P}$ for every $s_i \in \mathsf{Srvrs}$—simply by simulating the steps in the deterministic protocol. However, not every step can be simulated: as s does not know s_i's private key, s cannot sign a message on

Algorithm 8: Interfacing between gossip, interpret and user of $\mathcal{P}$.

1 **module** shim($s \in$ Srvrs, $\mathcal{P} \in$ **module**)
2 rqsts $:= \varnothing \in 2^{\mathcal{L} \times \mathsf{Rqsts}}$
3 $\mathcal{G} := \varnothing \in$ Dags
4 *gssp* $:=$ **new process** gossip($s, \mathcal{G}$, rqsts)
5 *intprt* $:=$ **new process** interpret($\mathcal{G}, \mathcal{P}$)
6 **when** request($\ell \in \mathcal{L}, r \in$ Rqsts)
7   rqsts.put(ℓ, r)
8 **when** *intprt*.indicate(ℓ, i, s') **where** $s' = s$
9   indicate(ℓ, i)
10 **repeatedly**
11   *gssp*.disseminate()

s_i's behalf. However, this is not necessary, because s can derive the authenticity of the message triggered by a block B from the signature of B, *i.e.*, $B.\sigma$. So instead of signing individual messages, s_i can give a *batch signature* $B.\sigma$ for authenticating every message materialized through B. Finally, s interprets protocol instances with labels ℓ_j *in parallel* in line 7 of Algorithm 7. While traversing the block DAG, s uses the structure of the block DAG to interpret requests and messages for every ℓ_j. Now, the same block giving rise to a request in process instance ℓ_j may materialize a message in process instance ℓ'_j. The (small) price to pay is the increase of block size by references to predecessor blocks, *i.e.*, B.preds.

4.3 Using the Framework

The protocol shim($\mathcal{P}$) in Algorithm 8 is responsible for the choreography of the external user of $\mathcal{P}$, the gossip protocol in Algorithm 6, and the interpret protocol in Algorithm 7. Therefore, the server s executing shim($\mathcal{P}$) in Algorithm 8 keeps track of two synchronized data structures *(i)* a buffer of labels and requests rqsts in line 2, and *(ii)* and the block DAG $\mathcal{G}$ in line 3. By calling rqsts.put(ℓ, r), s inserts (ℓ, r) in rqsts, and by calling rqsts.get(), s gets *and* removes a suitable number of requests $(\ell_1, r_1), \ldots, (\ell_n, r_n)$ from rqsts. To insert a block B in $\mathcal{G}$, s calls $\mathcal{G}$.insert(B) from Definition 4.1.4. We tacitly assume these operations are atomic. When starting an instance of gossip and $\mathsf{interpret}$ in line 4 and 5, s passes in references to theses shared data structures. When the external user of protocol $\mathcal{P}$ requests $r \in \mathsf{Rqsts}$ for $\ell \in \mathcal{L}$ from s via the request $\mathsf{request}(\ell, r)$ to $\mathsf{shim}(\mathcal{P})$ then s inserts (ℓ, r) in rqsts in lines 6–7. By executing gossip, s writes (ℓ, r) in $\mathcal{B}$ in Algorithm 6 line 15, and as eventually $\mathcal{B} \in \mathcal{G}$, r will be requested from protocol instance $\mathsf{PIs}[\ell]$ when s executes line 6 in Algorithm 7:

Lemma 4.3.1 *For a correct server s executing* $\mathsf{shim}(\mathcal{P})$*, if some* $\mathsf{request}(r, \ell)$ *is requested from s, then r is requested in $\mathcal{P}$.*

Proof 4.3.1 *By executing* $\mathsf{shim}(\mathcal{P})$*, a correct server s inserts (ℓ, r) in* rqsts *in line 6–7 of Algorithm 8. Then executing* $\mathsf{gossip}(s, \mathcal{G}, \mathsf{rqsts})$*, s will eventually disseminate a block B with $B.\mathsf{n} = s$ and $(\ell, r) \in B.\mathsf{rs}$ in line 15 of Algorithm 6 and $B \in \mathcal{G}$ after triggering* $\mathsf{disseminate}$ *in lines 10–11 of Algorithm 8. Now, executing* $\mathsf{interpret}(\mathcal{G}, \mathcal{P})$*, s for $B \in \mathcal{G}$ will call $B.\mathsf{PIs}[\ell].\mathsf{rs}$ in line 6 in Algorithm 7.*

On the other hand, when $\mathsf{interpret}$ indicates $i \in \mathsf{Inds}$, for the interpretation of $\mathcal{P}$ *for itself*, *i.e.*, $s = s'$, then s indicates to the user of $\mathcal{P}$ in line 8–9 of Algorithm 8:

Lemma 4.3.2 *For a correct server s executing* $\mathsf{shim}(\mathcal{P})$*, if $\mathcal{P}$ indicates* $i \in \mathsf{Inds}_{\mathcal{P}}$ *for s, then* $\mathsf{shim}(\mathcal{P})$ *triggers* $\mathsf{indicate}(\ell, i)$*.*

Proof 4.3.2 *By assumption a correct s indicates i for ℓ and hence indicates in* $\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ *lines 13–14 of Algorithm 7. Then, by executing* $\mathsf{shim}(\mathcal{P})$*, as* $s = s'$ $\mathsf{indicate}(\ell, i \in \mathsf{Inds}_{\mathcal{P}})$ *is triggered in lines 8–9 of Algorithm 8.*

For s to only indicate when $s = s'$ might be an over-approximation: s trusts s's interpretation of $\mathcal{P}$ as s is correct for s. We believe this restriction can be lifted. Finally, as promised in Section 4.1, in lines 10–11 s repeatedly requests $\mathsf{disseminate}$ from gossip to disseminate $\mathcal{B}$. Within the control of s, the time between calls to $\mathsf{disseminate}$ can be adapted to meet the network assumptions of $\mathcal{P}$ and can be enforced *e.g.*, by an internal timer, the block's payload, or when s falls n blocks behind. For our proofs we only need to guarantee that a correct s will eventually request $\mathsf{disseminate}$.

Example 4.3.1 *Assume a server s running Algorithm 8 instantiated with $\mathcal{P}$ as* byzantine reliable broadcast *in Algorithm 1 and a client which requests $r = \mathsf{broadcast}(42)$ from s. Now, s passes r to* gossip *(lines 6–7) and once the interpretation of $\mathcal{G}$ indicates for s, s indicates to the client (lines 8–9).*

Following [23], a protocol $\mathcal{P}$ implements an interface $\mathbb{I}$ and has properties $\mathbb{P}$, which are shown to hold for $\mathcal{P}$. For any property, which holds for a protocol $\mathcal{P}$ and where the proof of the property relies on the reliable point-to-point abstraction in Lemma 4.2.11, $\mathbb{P}$ holds for $\mathsf{shim}(\mathcal{P})$. Again following [23], these are the properties of any algorithm that *uses* the reliable point-to-point link abstraction.

Taking together what we have established for gossip in Section 4.1, *i.e.* that correct servers will eventually share a joint block DAG, and that $\mathsf{interpret}$ gives a point-to-point link between them in Section 4.2, for $\mathsf{shim}(\mathcal{P})$ the following holds:

Theorem 4.3.1 *For a correct server s and a deterministic protocol $\mathcal{P}$, if $\mathcal{P}$ is an implementation of* (i) *an interface $\mathbb{I}$ with requests $\mathsf{Rqsts}_{\mathcal{P}}$ and indications $\mathsf{Inds}_{\mathcal{P}}$ using the reliable point-to-point link abstraction such that* (ii) *a property $\mathbb{P}$ holds, then $\mathsf{shim}(\mathcal{P})$ in Algorithm 8 implements* (i) *$\mathbb{I}$ such that* (ii) *$\mathbb{P}$ holds.*

Proof 4.3.3 *By Lemma 4.3.1 and Lemma 4.3.2,* (i) $\mathsf{shim}(\mathcal{P})$ *implements the interface $\mathbb{I}$ of $\mathsf{Rqsts}_{\mathcal{P}}$ and $\mathsf{Inds}_{\mathcal{P}}$. For* (ii)*, by assumption $\mathbb{P}$ holds for $\mathcal{P}$ using a reliable point-to-point link abstraction. By Lemma 4.2.11 $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ implements a reliable point-to-point link. As Algorithm 7 treats $\mathcal{P}$ as a black-box every $B.\mathsf{Pls}[\ell]$ holds an execution of $\mathcal{P}$. Assume this execution violates $\mathbb{P}$, but then an execution of $\mathcal{P}$ violates $\mathbb{P}$ which contradicts the assumption that $\mathbb{P}$ holds for $\mathcal{P}$.*

Our proof relies on a point-to-point link between two correct servers and thus we can translate the argument of all safety and liveness properties, for which their reasoning relies on the point-to-point link abstraction, to our block DAG framework. Because we provide an abstraction, we cannot directly translate implementation-level properties measuring performance such as latency or throughput. They rely on the concrete implementation. Also, as discussed in Section 4.2, properties related to signatures do not directly translate, because blocks—not messages—are (batch-)signed. Finally, we note that in our setting the complexity measure of calls to the reliable point-to-point link abstraction is slightly misleading, because we are optimising the messages transmitted by the point-to-point link abstraction.

In the remainder of this chapter, we will sketch how a user may use the block DAG framework. Our example for $\mathcal{P}$ is *byzantine reliable broadcast* (BRB). Given an implementation of byzantine reliable broadcast after [23, Module 3.12, p. 117], *e.g.*, Algorithm 1: this is the $\mathcal{P}$, which the user passes to $\mathsf{shim}(\mathcal{P})$, *i.e.*, in the block DAG framework $\mathcal{P}$ is fixed to an implementation of BRB. The request in BRB is $\mathsf{broadcast}(v)$ for a value $v \in \mathsf{Vals}$, so $\mathsf{Rqsts}_{\mathcal{P}} = \{\mathsf{broadcast}(v) \mid v \in \mathsf{Vals}\}$. For simplicity and generality, we assume that $\mathcal{P}$—not $\mathsf{shim}(\mathcal{P})$—authenticates requests, *i.e.*, requests are self-contained and can be authenticated while simulating $\mathcal{P}$ (*e.g.*, Algorithm 1 line 3). However, in an implementation $\mathsf{shim}(\mathcal{P})$ may be employed to authenticate requests. On the other hand, BRB indicates with $\mathsf{deliver}(v)$, so $\mathsf{Inds}_{\mathcal{P}} = \{\mathsf{deliver}(v) \mid v \in \mathsf{Vals}\}$. The messages sent in BRB are $\mathsf{M}_{\mathcal{P}} = \{\mathtt{ECHO}\ v, \mathtt{READY}\ v \mid v \in \mathsf{Vals}\}$ where sender and $\mathsf{receiver}$ are the $s \in \mathsf{Srvrs}$ running $\mathsf{shim}(\mathcal{P})$. When executing line 9 of $\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ in Algorithm 7, then $\mathsf{receive}(\mathtt{ECHO}\ 42)$ is triggered, and **received** ECHO 42 holds (*e.g.*, in Algorithm 1 in line 6). As we assume $\mathcal{P}$ returns messages immediately, *e.g.*, when the simulation reaches **send** ECHO 42, then ECHO 42 is returned immediately *e.g.*, in line 8 of Algorithm 1. The interface $\mathbb{I}$ is $\mathsf{Rqsts} = \{\mathsf{broadcast}(v) \mid v \in \mathsf{Vals}\}$ and $\mathsf{Inds} = \{\mathsf{deliver}(v) \mid v \in \mathsf{Vals}\}$ The properties $\mathbb{P}$ of BRB—validity, no duplication, integrity, consistency, and

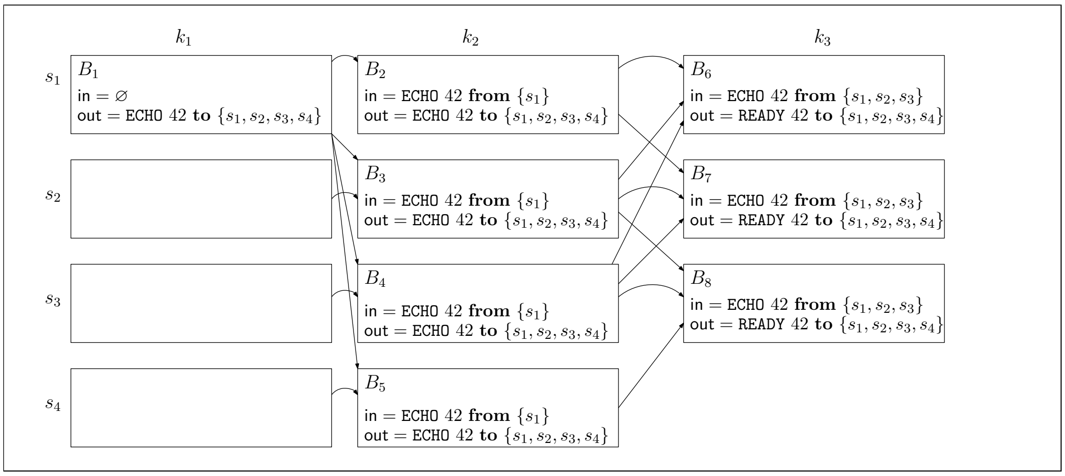

Figure 4.5: The message buffers for $(\ell_1, \mathsf{broadcast}(42)) \in B_1.\mathsf{rs}$.

totality—are preserved.

Figure 4.5 shows a block DAG for an execution of $\mathsf{shim}(\mathcal{P})$ using byzantine reliable broadcast. It further explicitly shows the in- and out-going messages from $\mathsf{Ms}[\mathsf{in}, \ell_1]$ and $\mathsf{Ms}[\mathsf{out}, \ell_1]$ for a protocol instance ℓ_1 and the request $\mathsf{broadcast}(42)$ at block B_1. None of these messages are ever actually sent over the network—every server interpreting this block DAG can use interpret in Algorithm 7 to replay an implementation of BRB and get the same picture. Figure 4.5 shows only the (unsent) messages for ℓ_1 and $\mathsf{broadcast}(42)$ in $B_1.\mathsf{rs}$, but $B_1.\mathsf{rs}$ may hold more requests such as $\mathsf{broadcast}(21)$ for ℓ_2, and all the messages of all these requests could be materialized in the same manner—without any messages, or even additional blocks, sent. Moreover, not only B_1 holds such requests—also B_3 does. For example, $B_3.\mathsf{rs}$ may contain $\mathsf{broadcast}(25)$ for ℓ_3. Then, for ℓ_3 on B_3 materializes out = ECHO 25 to s_1, s_2, s_3, and again, without sending any messages, for ℓ_3 on B_6, B_7, and B_8 materializes in = ECHO 25 from s_2. This is, of course, the same for every B_i.

To recap, what makes interpreting $\mathcal{P}$ on a block DAG so attractive: sending blocks instead of messages in a deterministic $\mathcal{P}$ results in a compression of messages—up to their omission. And not only do these messages not have to be sent, they also do not have to be signed. It suffices, that every server signs their blocks. Finally, a single block sent is interpreted as messages for a large number of parallel protocol instances.

Part II

Blockchain Programs

Chapter 5

Background: Programs

In this chapter I first give the necessary background on smart contracts and in the next section focus on the Ethereum virtual machine. Then I give a brief introduction into SMT solvers and superoptimisation.

5.1 Smart Contracts

Bitcoin has a restricted scripting language to manipulate the state of the blockchain. In 2013 the restriction on the language to manipulate state was lifted: the Ethereum blockchain[1] [22] introduced a (quasi) Turing-complete programming language to write programs to own, transfer, or even destroy cryptocurrency—so called *smart contracts*. Now most blockchains come with a smart contract language: Facebook's diem blockchain [77] with the Move [107] language from 2019, or the Tezos blockchain [47] with Michelson [61].

Programs deployed and executed on the blockchain are called *smart contracts*. Most of the terminology used here is with respect to the Ethereum blockchain, but concepts are similar for other blockchains. The source code of a smart contract resides on the blockchain and is thus public and immutable. When a smart contract is called, all servers execute the smart contract usually on a Virtual Machine, such as the Ethereum virtual machine (EVM) specified in the "yellow paper" [111, 112]. Several implementations of the EVM are available,[2] *e.g.*, a Go implementation geth[3] by the Ethereum foundation.

[1] *cf.* ethereum.org

[2] `https://eth.wiki/concepts/evm/implementations`

[3] `https://geth.ethereum.org/`

The servers execute the smart contract for a fee, usually called *gas*, which fuels the execution and depends on: *(i)* the program; every instruction comes with a *gas cost*[4], *(ii)* the current state, and *(iii)* the arguments to the call. The price of gas varies depending *e.g.*, on the utilisation of the network, but is paid up-front. Unused gas is refunded, but if the caller has not provided enough gas, the state is reverted and the money is lost. Usually, smart contract languages are quasi Turing-complete programming languages. We say *quasi* Turing-complete because paying for execution circumvents the halting problem [108]: every execution terminates.

Smart contracts are often written in a high-level language and compiled to low-level bytecode which gets deployed on the blockchain. In Ethereum, smart contracts could be written *e.g.*, in the object-oriented Solidity[5] and with solc compiled to EVM bytecode (see Figure 7.1 for example code).

Other blockchains come with their own languages. The designated language for the diem blockchain is the Move programming language compiling to Move bytecode [107]. Move is a formally specified, typed language. The Move virtual machine is also stack-based, but unlike the EVM it comes with typed locals to move elements on the stack. The Tezos blockchain supports the Michelson [61] bytecode language—again a typed language with a formal specification. Also the Michelson virtual machine relies on a stack—but a typed stack with integers, strings, bytes, and tags.

5.2 Ethereum Virtual Machine

The EVM as basis for Chapters 6–8 is specified in the Byzantium Version e94ebda [111] of the yellow paper[6]. The main components of the *state* of the EVM are: a *stack*, which holds *words*, *i.e.*, bit vectors of size 256. The maximal *stack size* is 2^{10}. A stack can over- and underflow. Both lead the EVM to enter an *exceptional halting* state. The EVM further has a volatile *memory*, which is a word-addressed byte array, and a persistent, key-value *storage*

[4]With the caveat that pricing of instructions is hard [114] leading to a possible attack [90].
[5]`https://soliditylang.org/`
[6]A newer version Petersburg Version 3e2c089 [112] is now available.

storing word-addressed words on the Ethereum blockchain. EVM *bytecode* directly corresponds to more human-friendly *instructions*, *e.g.* the EVM bytecode `6029600101` encodes the following sequence of instructions: `PUSH` 41 `PUSH` 1 `ADD`. We call a finite sequence of instructions a *program* p and define the *size* $|p|$ of a program as the number of its instructions. Instructions manipulate the state. For example, every instruction ι, takes $\delta(\iota)$ words from the stack and adds $\alpha(\iota)$ words to the stack. Instructions can be classified into different categories. Next we will give all instructions relevant in Chapter 6–8. From the *Stop and Arithmetic Operations* we consider: bit-vector addition (`ADD`), multiplication (`MUL`), subtraction (`SUB`), (signed) division (`DIV`/`SDIV`), (signed) modulo (`MOD` /`SMOD`), a modulo addition and multiplication operation (`ADDMOD`/`MULMOD`). We cannot fully consider `EXP` and `SIGNEXTEND` as we are lacking the support from the SMT solver. However, we consider them as *uninterpreted instructions*—leveraging the fact that they will always return the same result for the same arguments. The `STOP` instruction, which halts the execution, changes the control flow and serves as a instruction to determine the boundaries of a basic block. From the *Comparison and Bitwise Logic Operations* we consider: bit-vector comparison (signed) less-than (`LT`/`SLT`), (signed) greater-than (`GT`/`SGT`), equality (`EQ`), a check for zero (`ISZERO`), and bitwise `AND`, `OR`, `XOR`, `NOT`. Again, we do not encode `BYTE`, but consider it as uninterpreted instruction, similar to *SHA3* (`SHA3`).

For instructions holding *Environmental Information*, which depend on the runtime, we can encode some as uninterpreted instructions: `ADDRESS`, which returns the address of the executing account, `BALANCE`, which returns the balance, `ORIGIN`, which returns the origination address, `CALLER`, which returns the caller address, and `CALLVALUE`, `CALLDATALOAD` as well as `CALLDATASIZE`, which return information about the input data, `CODESIZE`, which gives the size of the code, `GASPRICE`, which gives the price of gas, `EXTCODESIZE`, which gives the size of an account's code, and `RETURNDATASIZE` giving the size of output data.

We cannot encode some of these instructions, as they have an outside effect on a state of the EVM we do not model. Hence we need to honour the order of these instructions. These are copy instructions: CALLDATACOPY, CODECOPY, EXTCODECOPY, RETURNDATACOPY.

For instructions holding *Block Information*, again we can encode all of them as uninterpreted instructions: BLOCKHASH, which gives the hash of a block, COINBASE, which gives an address, the TIMESTAMP of a block and its NUMBER, DIFFICULTY, and GASLIMIT.

The instructions for *Stack, Memory, Storage, and Flow Operations* we encode are: POP, which pops an element from stack. For the instructions to encode memory we treat the loading of a value (MLOAD) as an uninterpreted instruction, but we do not encode instructions to store values (MSTORE). For storage we can encode both: loading (SLOAD) and storing (SSTORE). Finally, the jump instructions to alter control flow (JUMP, JUMPI, JUMPDEST) are used to determine basic blocks. We also encode the program counter (PC), the amount of available gas (GAS) and the size of memory (MSIZE) as uninterpreted instructions.

The instructions performing *Push Operations* for pushing 1 to 32 bit-words (PUSH) on the stack are all encoded in our work, as well as the *Duplication Operation* for duplicating the first (DUP1) up to the 16^{th} element (DUP16) of the stack, and the *Exchange Operations* to swap the first and the second (SWAP1) up to the first and the 17^{th} element (SWAP16).

Finally, as the *Logging Operations* to log the state (LOG1 to LOG4), as well as all the *System operations* have an outside effect, we cannot encode them. These system operations can create a new account with code (CREATE), call an account (CALL), possibly with another account's code (CALLCODE, DELEGATECALL and STATICCALL), may return data (RETURN), reverting the execution (REVERT), be invalid (INVALID), or self-destruct (SELFDESTRUCT). We leave all these instructions at their original position in the code.

Note that for Chapter 7 we are exclusively focusing on the instructions concerning the stack: PUSH, POP, DUP, and SWAP.

5.3 Satisfiability Modulo Theories

If a problem can be expressed as a first-order logic formula with equality—possibly in combination with theories, preferably *decidable* theories [60]—then this problem could be solved by a Satisfiability Modulo Theories (SMT) solver. Theories relevant for Chapter 6 to 8 are the theory of bit-vectors, theory of linear integer arithmetic, and the theory of uninterpreted functions. Once the problem is suitably expressed, an off-the-shelf SMT solver as a black-box can find a solution (given enough time if decidable). There are many SMT solvers[7]; two prominent open-source one's are: Microsoft research's Z3 [32], and CVC4 [15]. Moreover, a strong SMT community has formed with a yearly competition[8] [13] on collected benchmarks, and defined SMT-LIB standards (current version: 2.6 [14]). Finally, consider expressing the problem not as a satisfiability problem—but as an *optimisation* problem, trying to satisfy as many clauses as possible. Here again, a solution can be found with an off-the-shelf SMT solver such as the one's we leverage in Chapter 7: Z3, MathSAT [28], or Barcelogic [16].

5.4 Superoptimisation

Superoptimisation is the "look for the smallest program" [74]: given a *source* program p superoptimisation tries to generate a *target* program p'—possibly in a different language—such that *(i)* p' is equivalent to p, and *(ii)* the cost of p' is minimal with respect to a given cost function C.

Basic Superoptimisation. A standard approach to superoptimisation [74, 50, 101, 106] is shown in Algorithm 9. We call this approach *basic superoptimisation* (BasicSO). The input is a program p to superoptimise and a cost function C. Here, we search through all possible *candidate instruction* sequences in increasing cost (line 15 and 21). With a constraint solver, *e.g.*, an SMT solver, we check whether a candidate correctly implements the source program: we encode this as a request χ to the solver in line 16. If the solver

[7] A list is maintained at `http://smtlib.cs.uiowa.edu/solvers.shtml`.
[8] The 15^{th} SMT-COMP ran in 2020.

Algorithm 9: Basic superoptimisation.

```
12 Function BasicSO(p, C) is
13     n := 0
14     while true do
15         forall p′ ∈ {p′ | C(p′) = n} do
16             χ := EncodeBso(p, p′)
17             if Satisfiable(χ) then
18                 m := GetModel(χ)
19                 p′ := DecodeBso(m)
20                 return p′
21         n := n + 1
```

Algorithm 10: Unbounded superoptimisation.

```
22 Function UnboundedSO(p, C) is
23     p′ := p
24     χ := EncodeUso(p′) ∧ Bound(p′, C)
25     while Satisfiable(χ) do
26         m := GetModel(χ)
27         p′ := DecodeUso(m)
28         χ := χ ∧ Bound(p′, C)
29     return p′
```

returns yes, *i.e.*, our request to the solver is Satisfiable(χ) in line 17, then the candidate program correctly implements the source program and we return this candidate as solution in line 20. However, with increasing cost of the candidate programs, the search space dramatically increases. Consider for example an instruction loading an immediate argument: we have to check all possible immediate arguments, *i.e.*, for 32 bit-vector we have to check 2^{32} possibilities. To deal with this explosion one idea is to move some of the search to the solver by using *templates* [50, 106]. Templates leave holes in the candidate program, that the solver must then fill. Thus, if our encoding is satisfiable, we obtain a model in line 18, indicating how we can fill the holes by decoding the provided model to obtain the final target program in line 19.

Unbounded Superoptimisation. The idea of templates is pushed further in *unbounded superoptimisation* [56, 55]. Instead of searching through candidate programs and calling the SMT solver on them, it shifts the search into the solver, *i.e.*, the encoding expresses all candidate instruction sequences of any length that correctly implement the source program. This approach (UnboundedSO) is shown in Algorithm 10. From BasicSO the request to our solver changes to: is there a program implementing the source program p *within a bound* given by the original program (line 24)? If the solver returns yes, *i.e.*, our request to the solver is Satisfiable(χ) in line 25, then there is an instruction sequence that correctly implements the source program. Again, this target program is reconstructed from the model in lines 26 and 27. Now p' holds a correct, but possibly non-optimal, solution. Thus, to eventually obtain the optimal solution, we add the new found bound to the encoding χ in line 28 and iterate until the solver cannot find a program with a smaller bound any more and the solver returns no for Satisfiable(χ).

Chapter 6

Blockchain Superoptimiser

In this chapter we leverage formal reasoning about smart contracts to reduce the monetary fees of their execution while still guaranteeing correct execution [H_b].

Example 6.0.1 *Consider the expression* $3+(0-x)$ *in Figure 6.1, which corresponds to the program* `PUSH` *0* `SUB` `PUSH` *3* `ADD`*. This program takes an argument* x *from the stack to compute the expression above. However, clearly one can save the* `ADD` *instruction and instead compute* $3-x$*,* i.e.*, optimise the program to* `PUSH` *3* `SUB`*. The first program costs* $12\,\mathsf{g}$ *to execute on the* EVM*, while the second costs only* $6\,\mathsf{g}$*.*

We built a tool that automatically finds this optimisation and similar others that are missed by state-of-the-art smart contract compilers: the **E**VM **b**ytecode **s**uper**o**ptimiser ebso. To find these optimisations, ebso implements superoptimisation. Superoptimisation is often considered too slow to use during software development except for special circumstances. We argue that compiling smart contracts is such a circumstance. Since bytecode, once it has been deployed to the blockchain, cannot change again, spending extra time optimising a program that may be called many times, might well be worth it: the clear cost model of gas makes it easy to define optimality.[1]

[1]Of course setting the gas price of individual instructions, such that it accurately reflects the computational cost is hard, and has been a problem in the past see *e.g.* news.ycombinator.com/item?id=12557372.

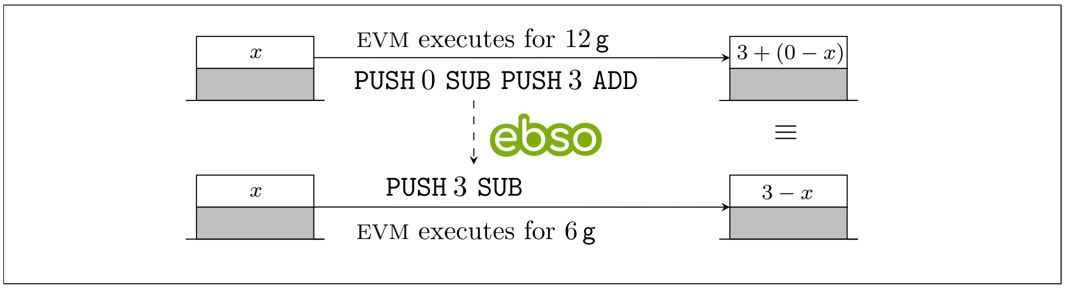

Figure 6.1: Overview over ebso.

6.1 Encoding

The main ingredients of superoptimisation in Algorithm 9 and 10 are EncodeBso/Uso producing the SMT encoding, and DecodeBso/Uso reconstructing the target program from a model. We present our encodings for the semantics of EVM bytecode and start by encoding three parts of the EVM execution state: *(i)* the stack, *(ii)* gas consumption, and *(iii)* whether the execution is in an exceptional halting state. We model the stack as an uninterpreted function together with a counter, which points to the next free position on the stack.

Definition 6.1.1 *A state* $\sigma = \langle \mathcal{S}, \mathsf{c}, \mathsf{hlt}, \mathsf{g} \rangle$ *consists of*

(i) *a function* $\mathcal{S}(\vec{x}, j, n)$ *that, after the program has executed* j *instructions on input variables from* $\vec{x}$ *returns the word from position* n *in the stack,*

(ii) *a function* $\mathsf{c}(j)$ *that returns the number of words on the stack after executing* j *instructions. Hence* $\mathcal{S}(\vec{x}, j, \mathsf{c}(j) - 1)$ *returns the top of the stack.*

(iii) *a function* $\mathsf{hlt}(j)$ *that returns true* ($\top$) *if exceptional halting has occurred after executing* j *instructions, and false* ($\bot$) *otherwise.*

(iv) *a function* $\mathsf{g}(\vec{x}, j)$ *that returns the amount of gas consumed after executing* j *instructions.*

Here the functions in σ represent *all* execution states of a program, indexed by variable j.

Example 6.1.1 *Symbolically executing the program* `PUSH` *41* `PUSH` *1* `ADD` *using our representation above we have*

$$\begin{array}{cccc} \mathsf{g}(0) = 0 & \mathsf{g}(1) = 3 & \mathsf{g}(2) = 6 & \mathsf{g}(3) = 9 \\ \mathsf{c}(0) = 0 & \mathsf{c}(1) = 1 & \mathsf{c}(2) = 2 & \mathsf{c}(3) = 1 \\ \mathcal{S}(1,0) = 41 & \mathcal{S}(2,0) = 41 & \mathcal{S}(2,1) = 1 & \mathcal{S}(3,0) = 42 \end{array}$$

and $\mathsf{hlt}(0) = \mathsf{hlt}(1) = \mathsf{hlt}(2) = \mathsf{hlt}(3) = \bot$.

Note that this program does not consume any words that were already on the stack. This is not the case in general. For instance we might be dealing with the body of a function, which takes its arguments from the stack. Hence we need to ensure that at the beginning of the execution sufficiently many words are on the stack. To this end we first compute the *depth* $\hat{\delta}(p)$ of the program p, *i.e.*, the number of words a program p consumes. Then we take variables $x_0, \ldots, x_{\hat{\delta}(p)-1}$ that represent the input to the program and initialize our functions accordingly.

Definition 6.1.2 *For a program with* $\hat{\delta}(p) = d$ *we initialize the state* σ *using*

$$\mathsf{g}_\sigma(0) = 0 \wedge \mathsf{hlt}_\sigma(0) = \bot \wedge \mathsf{c}_\sigma(0) = d \wedge \bigwedge_{0 \leqslant \ell < d} \mathcal{S}_\sigma(\vec{x}, 0, \ell) = x_\ell$$

For instance, for the program consisting of the single instruction `ADD` we set $\mathsf{c}(0) = 2$, and $\mathcal{S}(\{x_0, x_1\}, 0, 0) = x_0$ and $\mathcal{S}(\{x_0, x_1\}, 0, 1) = x_1$. We then have $\mathcal{S}(\{x_0, x_1\}, 1, 0) = x_1 + x_2$.

To encode the effect of EVM instructions we build SMT formulas to capture their operational semantics. That is, for an instruction ι and a state σ we give a formula $\tau(\iota, \sigma, j)$ that defines the effect on state σ if ι is the j-th instruction that is executed. Since large parts of these formulas are similar for every instruction and only depend on δ and α we build them from smaller building blocks.

Definition 6.1.3 *For an instruction ι and state σ we define:*

$$\begin{aligned}
\tau_{\mathsf{g}}(\iota,\sigma,j) &\equiv \mathsf{g}_\sigma(\vec{x}, j+1) = \mathsf{g}_\sigma(\vec{x}, j) + C(\sigma, j, \iota) \\
\tau_{\mathsf{c}}(\iota,\sigma,j) &\equiv \mathsf{c}_\sigma(j+1) = \mathsf{c}_\sigma(j) + \alpha(\iota) - \delta(\iota) \\
\tau_{\mathsf{pres}}(\iota,\sigma,j) &\equiv \forall\, n. n < \mathsf{c}_\sigma(j) - \delta(\iota) \rightarrow \mathcal{S}_\sigma(\vec{x}, j+1, n) = \mathcal{S}_\sigma(\vec{x}, j, n) \\
\tau_{\mathsf{hlt}}(\iota,\sigma,j) &\equiv \mathsf{hlt}_\sigma(j+1) = \mathsf{hlt}_\sigma(j) \vee \mathsf{c}_\sigma(j) - \delta(\iota) < 0 \vee \mathsf{c}_\sigma(j) - \delta(\iota) + \alpha(\iota) > 2^{10}
\end{aligned}$$

Here $C(\sigma, j, \iota)$ is the gas cost of executing instruction ι on state σ after j steps.

The formula τ_{g} adds the cost of ι to the gas cost incurred so far. The formula τ_{c} updates the counter for the number of words on the stack according to δ and α. The formula τ_{pres} expresses that all words on the stack below $\mathsf{c}_\sigma(j) - \delta(\iota)$ are preserved. Finally, τ_{hlt} captures that exceptions relevant to the stack can occur through either an underflow or an overflow, and that once it has occurred an exceptional halt state persists. For now the only other component we need is how the instructions affect the stack $\mathcal{S}$, *i.e.*, a formula $\tau_{\mathcal{S}}(\iota, \sigma, j)$. Here we only give an example and refer to our implementation or the yellow paper [111] for details. We have

$$\begin{aligned}
\tau_{\mathcal{S}}(\mathtt{ADD}, \sigma, j) &\equiv \mathcal{S}_\sigma(\vec{x}, j+1, \mathsf{c}_\sigma(j+1) - 1) \\
&= \mathcal{S}_\sigma(\vec{x}, j, \mathsf{c}_\sigma(j) - 1) + \mathcal{S}_\sigma(\vec{x}, j, \mathsf{c}_\sigma(j) - 2)
\end{aligned}$$

Finally these formulas yield an encoding for the semantics of an instruction.

Definition 6.1.4 *For an instruction ι and state σ we define*

$$\tau(\iota,\sigma,j) \equiv \tau_{\mathcal{S}}(\iota,\sigma,j) \wedge \tau_{\mathsf{c}}(\iota,\sigma,j) \wedge \tau_{\mathsf{g}}(\iota,\sigma,j) \wedge \tau_{\mathsf{hlt}}(\iota,\sigma,j) \wedge \tau_{\mathsf{pres}}(\iota,\sigma,j)$$

Then to encode the semantics of a program p all we need to do is to apply τ to the instructions of p.

Definition 6.1.5 *For a program $p = \iota_0 \cdots \iota_n$ we set $\tau(p, \sigma) \equiv \bigwedge_{0 \leqslant j \leqslant n} \tau(\iota_j, \sigma, j)$.*

Before building an encoding for superoptimisation we consider another aspect of the EVM for our state representation: storage and memory. The gas cost for storing words depends on the words that are currently stored. Similarly, the cost for using memory depends on the number of bytes currently used. This is why the cost of an instruction $C(\sigma, j, \iota)$ depends on the state and the function g_σ accumulating gas cost depends on $\vec{x}$.

To add support for storage and memory to our encoding there are two natural choices: the theory of arrays or an Ackermann encoding. However, since we have not used arrays so far, they would require the solver to deal with an additional theory. For an Ackermann encoding we only need uninterpreted functions, which we have used already. Hence, to represent storage in our encoding we extend states with an uninterpreted function $\mathsf{str}(\vec{x}, j, k)$, which returns the word at key k after the program has executed j instructions. Similarly to how we set up the initial stack we need to deal with the values held by the storage before the program is executed. Thus, to initialize str we introduce fresh variables to represent the initial contents of the storage. More precisely, for all `SLOAD` and `SSTORE` instructions occurring at positions $j_1, \ldots, j_\ell$ in the source program, we introduce fresh variables $s_1, \ldots, s_\ell$ and add them to $\vec{x}$. Then for a state σ we initialize str_σ by adding the following conjunct to the initialization constraint from Definition 6.1.2:

$$\forall w.\ \mathsf{str}_\sigma(\vec{x}, 0, w) = \mathit{ite}(w = a_{j_1}, s_1, \mathit{ite}(w = a_{j_2}, s_2, \ldots, \mathit{ite}(w = a_{j_\ell}, s_\ell, w_\perp)))$$

where $a_j = \mathcal{S}_\sigma(\vec{x}, j, \mathsf{c}(j) - 1)$ and $w_\perp$ is the default value for words in the storage. The effect of the two storage instructions `SLOAD` and `SSTORE` can then be encoded as follows:

$$\begin{aligned}
\tau_{\mathcal{S}}(\mathtt{SLOAD}, \sigma, j) &\equiv \mathcal{S}_\sigma(\vec{x}, j+1, \mathsf{c}_\sigma(j+1) - 1) = \mathsf{str}(\vec{x}, j, \mathcal{S}_\sigma(\vec{x}, j, \mathsf{c}_\sigma(j) - 1)) \\
\tau_{\mathsf{str}}(\mathtt{SSTORE}, \sigma, j) &\equiv \forall w.\ \mathsf{str}_\sigma(\vec{x}, j+1, w) = \\
&\qquad \mathit{ite}(w = \mathcal{S}_\sigma(\vec{x}, j, \mathsf{c}_\sigma(j) - 1), \mathcal{S}_\sigma(\vec{x}, j, \mathsf{c}_\sigma(j) - 2), \mathsf{str}_\sigma(\vec{x}, j, w))
\end{aligned}$$

Moreover all instructions except `SSTORE` preserve the storage, that is, for $\iota \neq$ `SSTORE` we add the following conjunct to $\tau_{\mathsf{pres}}(\iota, \sigma, j)$: $\forall w.\ \mathsf{str}_\sigma(\vec{x}, j+1, w) = \mathsf{str}_\sigma(\vec{x}, j, w)$.

To encode memory a similar strategy is an obvious way to go. However, we first want to evaluate the solver's performance on the encodings obtained when using stack and storage. Since the solver already struggled, due to the size of the programs and the number of universally quantified variables, see Section 6.3, we have not added an encoding of memory.

Finally, to use our encoding for superoptimisation we need an encoding of equality for two states after a certain number of instructions. Either to ensure that two programs are equivalent (they start and end in equal states) or different (they start in equal states, but end in different ones). The following formula captures this constraint.

Definition 6.1.6 *For states σ_1 and σ_2 and program locations j_1 and j_2 we define*

$$\begin{aligned}\epsilon(\sigma_1, \sigma_2, j_1, j_2) \equiv\ & \mathsf{c}_{\sigma_1}(j_1) = \mathsf{c}_{\sigma_2}(j_2) \wedge \mathsf{hlt}_{\sigma_1}(j_1) = \mathsf{hlt}_{\sigma_2}(j_2) \\ & \wedge \forall\, n.n < \mathsf{c}_{\sigma_1}(j_1) \rightarrow \mathcal{S}_{\sigma_1}(\vec{x}, j_1, n) = \mathcal{S}_{\sigma_2}(\vec{x}, j_2, n) \\ & \wedge \forall\, w.\mathsf{str}_{\sigma_1}(\vec{x}, j_1, w) = \mathsf{str}_{\sigma_2}(\vec{x}, j_2, w)\end{aligned}$$

Since we aim to improve gas consumption, we do not demand equality for g.

We now have all ingredients needed to implement basic superoptimisation: simply enumerate all possible programs ordered by gas cost and use the encodings to check equivalence. However, since already for one `PUSH` there are 2^{256} possible arguments, this will not produce results in a reasonable amount of time. Hence we use templates as described in Section 5.4. We introduce an uninterpreted function $\mathsf{a}(j)$ that maps a program location j to a word, which will be the argument of `PUSH`. The solver then fills these templates and we can get the values from the model. This is a step forward, but since we have 80 encoded instructions, enumerating all permutations still yields too large a search space. Hence we use an encoding similar to the CEGIS algorithm [50]. Given a collection of instructions, we formulate a constraint representing all possible permutations of these instructions. It is satisfiable if there is a way to connect the instructions into a target program that is equivalent to the source program. The order of the instructions can again be reconstructed from the model provided by the solver. More precisely given a source program p and a list of candidate instructions $\iota_1, \ldots, \iota_n$, EncodeBso from Algorithm 9 takes variables $j_1, \ldots, j_n$ and two states σ and σ' and builds the following formula

$$\begin{aligned}&\forall \vec{x}.\, \epsilon(\sigma, \sigma', 0, 0) \wedge \epsilon(\sigma, \sigma', |p|, n) \wedge \tau(p, \sigma)\\ &\quad \wedge \bigwedge_{1 \leqslant \ell \leqslant n} \tau(\iota_\ell, \sigma', j_\ell) \wedge \bigwedge_{1 \leqslant \ell < k \leqslant n} j_\ell \neq j_k \wedge \bigwedge_{1 \leqslant \ell \leqslant n} j_\ell \geqslant 0 \wedge j_\ell < n\end{aligned}$$

Here the first line encodes the source program, and says that the start and final states of the two programs are equivalent. The second line encodes the effect of the candidate instructions and enforces that they are all used in some order. If this formula is satisfiable we can simply get the j_i from the model and reorder the candidate instructions accordingly to obtain the target program.

Unbounded superoptimisation shifts even more of the search into the solver, encoding the search space of all possible programs. To this end we take a variable n, which represents the number of instructions in the target program and an uninterpreted function $\mathsf{instr}(j)$, which acts as a template, returning the instruction to be used at location j. Then, given a set of candidate instructions the formula to encode the search can be built as follows:

Definition 6.1.7 *Given a set of instructions* CI *the formula* $\rho(\sigma, n)$ *is*

$$\forall j.\, j \geqslant 0 \wedge j < n \rightarrow \bigwedge_{\iota \in \mathsf{CI}} \mathsf{instr}(j) = \iota \rightarrow \tau(\iota, \sigma, j) \wedge \bigvee_{\iota \in \mathsf{CI}} \mathsf{instr}(j) = \iota$$

Finally, the constraint produced by EncodeUso *from Algorithm 10 is*

$$\forall \vec{x}.\, \tau(p, \sigma) \wedge \rho(\sigma', n) \wedge \epsilon(\sigma, \sigma', 0, 0) \wedge \epsilon(\sigma, \sigma', |p|, n) \wedge \mathsf{g}_\sigma(\vec{x}, |p|) > \mathsf{g}_{\sigma'}(\vec{x}, n).$$

During our experiments we observed that the solver struggles to show that the formula is unsatisfiable when p is already optimal. To help in these cases we additionally add a bound on n: since the cheapest EVM instruction has gas cost 1, the target program cannot use more instructions than the gas cost of p, *i.e.*, we add $n \leqslant \mathsf{g}_\sigma(\vec{x}, |p|)$.

In our application domain there are many instructions that fetch information from the outside world. For instance, `ADDRESS` gets the Ethereum address of the account currently executing the bytecode of this smart contract. Since it is not possible to know these values at compile time we cannot encode their full semantics. However, we would still like to take advantage of structural optimisations where these instructions are involved, *e.g.*, via `DUP` and `SWAP`.

Example 6.1.2 *Consider the program* `ADDRESS DUP1`*. The same effect can be achieved by simply calling* `ADDRESS ADDRESS`*. Duplicating words on the stack, if they are used multiple times, is an intuitive approach. However, because executing* `ADDRESS` *costs* $2\,\mathsf{g}$ *and* `DUP1` *costs* $3\,\mathsf{g}$*, perhaps unexpectedly, the second program is cheaper.*

To find such optimisations we need a way to encode `ADDRESS` and similar instructions. For our purposes, these instructions have in common that they put arbitrary but fixed words onto the stack. Analogous to uninterpreted functions, we call them *uninterpreted instructions* and collect them in the set UI. To represent their output we use universally quantified variables—similar to input variables. To encode the effect uninterpreted instructions have on the stack, *i.e.*, $\tau_\mathcal{S}$, we distinguish between *constant* and *non-constant* uninterpreted instructions.

Let $\mathsf{ui}_c(p)$ be the set of *constant uninterpreted instructions* in p, *i.e.* $\mathsf{ui}_c(p) = \{\iota \in p \mid \iota \in \mathsf{UI} \wedge \delta(\iota) = 0\}$. Then for $\mathsf{ui}_c(p) = \{\iota_1, \ldots, \iota_k\}$ we take variables $u_{\iota_1}, \ldots, u_{\iota_k}$ and add them to $\vec{x}$, and thus to the arguments of the state function $\mathcal{S}$. The formula $\tau_\mathcal{S}$ can then use these variables to represent the unknown word produced by the uninterpreted instruction, *i.e.*, for $\iota \in \mathsf{ui}_c(p)$ with the corresponding variable u_ι in $\vec{x}$, we set $\tau_\mathcal{S}(\iota, \sigma, j) \equiv \mathcal{S}_\sigma(\vec{x}, j+1, \mathsf{c}_\sigma(j)) = u_\iota$.

For a *non-constant instruction* ι, such as `BLOCKHASH` or `BALANCE`, the word put onto the stack by ι depends on the top $\delta(\iota)$ words of the stack. We again model this dependency using an uninterpreted function. That is, for every non-constant uninterpreted instruction ι in the source program p, $\mathsf{ui}_n(p) = \{\iota \in p \mid \iota \in \mathsf{UI} \wedge \delta(\iota) > 0\}$, we use an uninterpreted function f_ι. Conceptually, we can think of f_ι as a read-only memory initialized with the values that the calls to ι produce.

Example 6.1.3 *The instruction* `BLOCKHASH` *gets the hash of a given block b. Optimising the program* `PUSH` b_1 `BLOCKHASH` `PUSH` b_2 `BLOCKHASH` *depends on the values b_1 and b_2. If $b_1 = b_2$ then the cheaper program* `PUSH` b_1 `BLOCKHASH` `DUP1` *yields the same state as the original program.*

To capture this behaviour, we need to associate the arguments b_1 and b_2 of `BLOCKHASH` with the two different results they may produce. As with constant uninterpreted instructions, to model arbitrary but fixed results, we add fresh variables to $\vec{x}$. However, to account for different results produced by ℓ invocations of ι in p we have to add ℓ variables. Let p be a program and $\iota \in \mathsf{ui}_n(p)$ a unary instruction which appears ℓ times at positions $j_1, \ldots, j_\ell$ in p. For variables $u_1, \ldots, u_\ell$, we initialize f_ι as follows:

$$\forall w.\, f_\iota(\vec{x}, w) = ite(w = a_{j_1}, u_1, ite(w = a_{j_2}, u_2, \ldots, ite(w = a_{j_\ell}, u_\ell, w_\perp)))$$

where a_j is the word on the stack after j instructions in p, that is $a_j = \mathcal{S}_\sigma(\vec{x}, j, \mathsf{c}(j) - 1)$, and $w_\perp$ is a default word. This approach straightforwardly extends to instructions with more than one argument. Here we assume that uninterpreted instructions put exactly one word onto the stack, *i.e.*, $\alpha(\iota) = 1$ for all $\iota \in \mathsf{UI}$. This assumption is easily verified for the EVM: the only instructions with $\alpha(\iota) > 1$ are `DUP` and `SWAP`. Finally we set the effect a non-constant uninterpreted instruction ι with associated function f_ι has on the stack:

$$\tau_{\mathcal{S}}(\iota, \sigma, j) \equiv \mathcal{S}_\sigma(\vec{x}, j+1, \mathsf{c}_\sigma(j+1) - 1) = f_\iota(\vec{x}, \mathcal{S}_\sigma(\vec{x}, j, \mathsf{c}_\sigma(j) - 1))$$

For some uninterpreted instructions there might a be way to partially encode their semantics. The instruction `BLOCKHASH` returns 0 if it is called for a block number greater than the current block number. While the current block number is not known at compile time, the instruction `NUMBER` does return it. Encoding this interplay between `BLOCKHASH` and `NUMBER` could potentially be exploited for finding optimisations.

6.2 Implementation

We implemented basic and unbounded superoptimisation in our tool ebso available under the Apache-2.0 license: `github.com/juliannagele/ebso`. The encoding employed by ebso uses several background theories: *(i)* uninterpreted functions (UF) for encoding the state of the EVM, for templates, and for encoding uninterpreted instructions, *(ii)* bit vector arithmetic (BV) for operations on words, *(iii)* quantifiers for initial words on the stack and in the storage, and the results of uninterpreted instructions, and *(iv)* linear integer arithmetic (LIA) for the instruction counter. Hence following the SMT-LIB classification[2] ebso's constraints fall under the logic UFBVLIA. As SMT solver we chose Z3 [32], version 4.7.1 which we call with default configurations. In particular, Z3 performed well for the theory of quantified bit vectors and uninterpreted functions in the last SMT competition (albeit non-competing).[3]

The aim of our implementation is to provide a prototype without relying on heavy engineering and optimisations such as exploiting parallelism or tweaking Z3 strategies. However, without any optimisation, for the full word size of the EVM—256 bit—ebso did not handle the simple program `PUSH 0 ADD POP` within a reasonable amount of time. Thus we need techniques to make ebso viable. By investigating the models generated by Z3 run with the default configuration, we believe that the problem lies with the leading universally quantified variables. And we have plenty of them: for the input on the stack, for the storage, and for uninterpreted instructions. By reducing the word size

[2] `smtlib.cs.uiowa.edu/logics.shtml`
[3] `smt-comp.github.io/2019/results/ufbv-single-query`

to a small k, we can reduce the search space for universally quantified variables from 2^{256} to some significantly smaller 2^k. Still then we need to check any target program found with a smaller word size. To give an example: the program `PUSH 0 SUB PUSH 3 ADD` from Example 6.0.1 optimises to `NOT` for word size 2 bit, because then the binary representation of 3 is all ones. When using word size 256 bit this optimisation is not correct. To ensure that the target program has the same semantics for word size 256 bit, we use *translation validation*: we ask the solver to find inputs, which distinguish the source and target programs, *i.e.*, where both programs start in equivalent states, but their final state is different. Using our existing machinery this formula is easy to build:[4]

Definition 6.2.1 *Two programs p and p' are equivalent if $\nu(p, p', \sigma, \sigma') \equiv \exists\vec{x}, \tau(p, \sigma) \wedge \tau(p', \sigma') \wedge \epsilon(\sigma, \sigma', 0, 0) \wedge \neg\epsilon(\sigma, \sigma', |p|, |p'|)$ is unsatisfiable. Otherwise, p and p' are different, and the values for the variables in $\vec{x}$ from the model are a corresponding witness.*

A subtle problem remains: how can we represent the program `PUSH 224981` with only k bit? Our solution is to replace arguments $a_1, \ldots, a_m$ of `PUSH` where $a_i \geqslant 2^k$ with fresh, universally quantified variables $c_1, \ldots, c_m$. If a target program is found, we replace c_i by the original value a_i, and check with translation validation whether this target program is correct. A drawback of this approach is that we might lose potential optimisations.

Example 6.2.1 *The program* `PUSH` *0b111...111* `AND` *optimises to the empty program. However, abstracting the argument of* `PUSH` *translates the program to* `PUSH` c_i `AND`*, which does not allow the same optimisation.*

Like many compiler optimisations, ebso optimises basic blocks. Therefore we split EVM bytecode along instructions that change the control flow, *e.g.*, `JUMPI`, or `SELFDESTRUCT`. Similarly we further split basic blocks into (ebso) blocks so that they contain only encoded instructions. Instructions, which

[4]This approach also allows for other over-approximations. For instance, we tried using integers instead of bit vectors, which performed worse.

are not encoded, or encodable, include instructions that write to memory, *e.g.* `MSTORE`, or the log instructions `LOG`.

Next we give a conjecture which would need to be proved to assure correctness. To formally proof this conjecture, we would need to introduce and model the whole state of the EVM and cannot restrict ourselves to modelling only the stack and storage as well as only a subset of the instructions. This work is exploratory to determine the value of the approach—before spending the effort of formalisation. Several formal models of the EVM could serve as basis for this proof: Hirai [54] used the meta-tool Lem [83] to formalise the semantics of the EVM. This formalisation was extended by Amani *et al.* [8] by a program logic using the interactive proof assistant Isabelle/HOL to provide an approach to the verification of Ethereum smart contracts. Another formalisation of the EVM semantics by Hildenbrandt *et al.* [53] uses the K-framework [95], a rewriting-based framework for defining programming language design and semantics.

Any of these formalisations could be used as the basis of the proof of:

Conjecture 6.2.1 *If program p superoptimises to program t then in any program we can replace p by t without changing the semantic of the program.*

Proof Idea 6.2.1 *We would show the statement by induction on the program context (c_1, c_2) of the program $c_1 p c_2$. By assumption, the statement holds for the base case $([\,], [\,])$. For the step case $(\iota c_1, c_2)$, we observe that every instruction ι is deterministic,* i.e.*, executing ι starting from a state σ leads to a deterministic state σ'. By induction hypothesis, executing $c_1 p c_2$ and $c_1 t c_2$ from a state σ' leads to the same state σ'', and therefore we can replace $\iota c_1 p c_2$ by $\iota c_1 t c_2$. We can reason analogously for $(c_1, c_2 \iota)$.*

The proof idea gives two key insights for the proof. For one, we require every instruction to be deterministic—which is the case for the EVM. For two, we require that the context of the part of the state which is not modelled, as well as the remaining code, *i.e.* through reflection, does not change through the instruction and no side-effects occur. We assure this by restricting to instructions which only touch the modelled state according to the yellow paper [111] (*cf.* Section 5.2, *e.g.*, the stack with `SWAP` or `ADD`, and the storage with `SSTORE`) and respect the order of instructions touching the context (*e.g.*, `LOG`). However, without introducing the whole state, we cannot formally show this non-interference.

6.3 Evaluation

We evaluated ebso on two real-word data sets: *(i)* optimising an already highly optimised data set in Section 6.3, and *(ii)* a large-scale data set from the Ethereum blockchain to compare basic and unbounded superoptimisation in Section 6.3. We use ebso to extract ebso blocks from our data sets. From the extracted blocks *(i)* we remove duplicate blocks, and *(ii)* we remove blocks which are only different in the arguments of `PUSH` by abstracting to word size 4 bit. We run both evaluations on a cluster [58] consisting of nodes running Intel Xeon E5645 processors at 2.40 GHz, with one core and 1 GiB of memory per instance.

We successfully validated all optimisations found by ebso by running a reference implementation of the EVM on pseudo-random input. Therefore, we run the bytecode of the original input block and the optimised bytecode to observe that both produce the same final state. The EVM implementation we use is `go-ethereum`[5] version `1.8.23`.

Optimise the Optimised. This evaluation tests ebso against human intelligence. Underlying our data set are 200 Solidity contracts (GG_{raw}) we collected from the *1st Gas Golfing Contest*. We did not join the contest, but we used the contracts written by the winners to see whether we can still find optimisations in these highly optimised contracts. [6] In that contest competitors had to write the most gas-efficient Solidity code for five given challenges: *(i)* integer sorting, *(ii)* implementing an interpreter, *(iii)* hex decoding, *(iv)* string searching, and *(v)* removing duplicate elements. Every challenge had two categories: *stan-*

[5] github.com/ethereum/go-ethereum
[6] g.solidity.cc

	#	%
optimised (optimal)	19 (10)	0.69 % (0.36 %)
proved optimal	481	17.54 %
time-out (trans. val. failed)	2243 (196)	81.77 % (7.15 %)

Table 6.1: Aggregated results of running ebso on GG.

dard and *wild*. For wild, any Solidity feature is allowed—even inlining EVM bytecode. The winner of each track received 1 Ether. The Gas Golfing Contest provides a very high-quality data set: the EVM bytecode was not only optimised by the solc compiler, but also by humans leveraging these compiler optimisations and writing inline code themselves. To collect our data set GG, we first compiled the Solidity contracts in GG_{raw} with the same set-up as in the contest.[7] One contract in the wild category failed to compile and was thus excluded from GG_{raw}. From the generated `.bin-runtime` files, we extracted our final data set GG of 2743 distinct blocks.

For this evaluation, we run ebso in its default mode: unbounded superoptimisation. We run unbounded superoptimisation because, as can be seen in Section 6.3, in our context unbounded superoptimisation outperformed basic superoptimisation. As time-out for this challenging data set, we estimated 1 h as reasonable. Table 6.1 shows the aggregated results of running ebso on GG. In total, ebso optimises 19 blocks out of 2743, 10 of which are shown to be optimal. Moreover, ebso can prove for more than 17 % of blocks in GG that they are already optimal. It is encouraging that ebso even finds optimisations in this already highly optimised data set. The quality of the data set is supported by the high percentage of blocks being proved as optimal by ebso. Next we examine three found optimisations more closely. Our favourite optimisation `POP PUSH` 1 `SWAP1 POP PUSH` 0 to `SLT DUP1 EQ PUSH` 0 witnesses that superoptimisation can find unexpected results, and that unbounded superoptimisation can stop with non-optimal results: `SLT DUP1 EQ` is, in fact, a round-about and op-

[7]Namely, `$ solc --optimise --bin-runtime --optimise-runs 200` with solc compiler version `0.4.24` available at github.com/ethereum/solidity/tree/v0.4.24.

	uso		bso	
	#	%	#	%
optimised (optimal)	943 (393)	1.54 % (0.64 %)	184	0.3 %
proved optimal	3882	6.34 %	348	0.57 %
time-out (trans. val. failed)	56 392 (1467)	92.12 % (2.4 %)	60 685	99.13 %

Table 6.2: Aggregated results of running ebso with uso and bso on EthBC.

timisable way to pop two words from the stack and push 1 on the stack. Some optimisations follow clear patterns. The optimisations `CALLVALUE DUP1 ISZERO PUSH 81` to `CALLVALUE CALLVALUE ISZERO PUSH 81` and `CALLVALUE DUP1 ISZERO PUSH 364` to `CALLVALUE CALLVALUE ISZERO PUSH 364` are both based on the fact that `CALLVALUE` is cheaper than `DUP1`. Finding such patterns and generalizing them into peephole optimisation rules is the goal of Chapter 8. Unfortunately, ebso hit a time-out in nearly 82 % of all cases, where we count a failed translation validation as part of the time-outs, since in that case ebso continues to search for optimisations after increasing the word size.

Unbounded *vs.* Basic Superoptimisation. We compare unbounded and basic superoptimisation, which we will abbreviate with uso and bso, respectively, with a considerably larger data set. Fortunately, there is a rich source of EVM bytecode accessible: contracts deployed on the Ethereum blockchain. Assuming that contracts that are called more often are well constructed, we queried the 2500 most called contracts[8] using Google BigQuery.[9] From them we extract our data set EthBC of 61 217 distinct blocks. We estimated a cut-off point of 15 min as reasonable. Due to the high volume, we only run the full evaluation once.

Table 6.2 shows the aggregated results of running ebso on EthBC. Out of 61 217 blocks in EthBC, ebso finds 943 optimisations using uso out of which it proves 393 to be optimal. Using bso 184 optimisations are found. Some blocks were shown to be optimal by both approaches. Also, both approaches time

[8] up to block number 7 300 000 deployed on Mar-04-2019 01:22:15 AM +UTC
[9] cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

out in a majority of the cases: uso in more than 92 %, and bso in more than 99 %. Over all 61 217 blocks the total amount of gas saved for uso is 17 871 and 6903 for bso. For all blocks where an optimisation is found, the average gas saving per block in uso is 29.63 %, and 46.1 % for bso. The higher average for bso can be explained by *(i)* bso's bias for smaller blocks, where relative savings are naturally higher, and *(ii)* bso only providing optimal results, whereas uso may find intermediate, non-optimal results. The optimisation with the largest gain, is one which we did not necessarily expect to find in a deployed contract: a redundant storage access. Storage is expensive, hence optimised for in deployed contracts, but uso and bso both found `PUSH` 0 `PUSH` 4 `SLOAD` `SUB` `PUSH` 4 `DUP2` `SWAP1` `SSTORE` `POP` which optimises to the empty program—because the program basically loads the value from key 4 only to store it back to that same key. This optimisation saves at least 5220 g, but up to 20 220 g.

From Table 6.2 we see that on EthBC, uso outperforms bso by roughly a factor of five on found optimisations; more than ten times as many blocks are proved optimal by uso than by bso. As we expected, most optimisations found by bso were also found by uso, but surprisingly, bso found 21 optimisations, on which uso failed. We found that nearly all of the 21 source programs are fairly complicated, but have a short optimisation of two or three instructions. To pick an example, the block `PUSH` 0 `PUSH` 12 `SLOAD` `LT` `ISZERO` `ISZERO` `ISZERO` `PUSH` 12250 is optimised to the relatively simple `PUSH` 1 `PUSH` 12250—a candidate block, which will be tried early on in bso. Moreover, all 21 blocks are cheap: costing less than 10 g. We believe unfortunate, non-deterministic choices within the solver to be the reason they have not been found by uso.

Chapter 7

Synthesis using Max-SMT

In this chapter we expand on how we can leverage formal reasoning about smart contracts to reduce the monetary fees of their execution [H_b] by addressing the main shortfall of Chapter 6—lack of performance—by an improved encoding and a shift to Max-SMT solvers. The experimental results of Chapter 6 confirm the extreme computational demands of the technique: ebso times out in 92 % of the blocks used in the evaluation. This is a severe limitation for the use of the technique, and the problem of finding the optimal code for an EVM block still remains very challenging. The complexity stems mainly from three sources: first, the problem is expressed in the theory of bit-vector arithmetic with bit-width size of 256, which is a challenging width size for most SMT solvers. Second, expressing the problem involves an $\exists\forall$-quantification, since we want to find an assignment of instructions that works for all values in the initial stack. Third, since we look for the gas-optimal code, the problem is not a satisfaction problem but rather an optimisation problem. We propose a novel method for gas optimisation which is based on synthesising optimised EVM blocks using Max-SMT. We implemented our approach in syrup, and evaluated it on the same data set used for evaluating ebso in Chapter 6. Our results are very promising: while ebso timed out in 92.12 % of the blocks, we only time out in 8.64 % and obtain gains that are two orders of magnitude larger than ebso. These results show that we have found the right balance between what is optimised by means of symbolic execution and symbolic simplification using rules and what is encoded as a Max-SMT problem.

```
pragma solidity ^0.4.25;
contract addExp{
  function ae(uint x3, uint x2, uint x1,
    uint x0) returns (uint){
      uint x = x3+x2;
      uint y = x1+x0;
      return x**y;   //EXP operation
  }
}
```

```
JUMPDEST
PUSH1 0x00
DUP1
PUSH1 0x00
DUP6
DUP8
ADD
SWAP2
POP
DUP4
DUP6
ADD
SWAP1
POP
DUP1
DUP3
EXP
SWAP3
POP
POP
POP
SWAP5
SWAP4
POP
POP
POP
POP
JUMP
```

Figure 7.1: Solidity code and under-optimised EVM bytecode using solc (right).

7.1 Optimal Bytecode as a Synthesis problem

We provide a general overview of our method for synthesising superoptimised smart contracts from given EVM bytecode. We use the motivating example in Figure 7.1 whose Solidity source code contract appears to the left and the EVM bytecode generated by the solc compiler appears to the right. The gas consumed by the bytecode in Figure 7.1 (excluding the `JUMPDEST` and `JUMP` opcodes that cannot be optimised and are thus not accounted in the examples) is 76. Our approach is based on optimising the operations that modify the stack as we have a great coverage of all potential bytecode optimisations while we still remain scalable, *i.e.*, we do not optimise instructions whose effects are not reflected in the stack, *e.g.*, `MSTORE`, `SSTORE`, `LOG1` or `EXTCODECOPY`.

Extracting Stack Functional Specifications. Our method takes as input the set of blocks that make up the control flow graph (CFG) of the bytecode. The first step is, for each of the blocks, to extract from it a *stack functional specification* (SFS) from which the superoptimised bytecode will be synthesised. The SFS is a functional description of the initial stack when entering the block and the final stack after executing the block, which instead of using bytecode instructions to determine how the final stack is computed, is defined by means of *symbolic first-order terms* over the initial stack elements. The SFS for our running example is shown in Figure 7.2. As can be observed, it consists of an initial stack shown at the left which simply determines what the size of

$$\begin{array}{|c|}\hline x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline x_4 \\ \hline \end{array} \implies \begin{array}{|c|}\hline x_4 \\ \hline exp(x_2 + x_3, x_0 + x_1) \\ \hline \end{array}$$

Figure 7.2: Initial and final stack.

the input stack to the block is and assigns a symbolic variable as identifier to each stack position, *e.g.*, the initial stack contains five elements named $x_0, \ldots, x_4$, while the output stack contains two elements: x_4 at the top, and the symbolic term $\exp(x_2 + x_3, x_0 + x_1)$ at the bottom. The output stack is obtained by symbolic execution of the bytecodes that operate on the stack, as it will be formalized in Section 7.2. The resulting expressions are then optimised by means of simplification rules based on the semantics of the non-stack operations, *e.g.*, the neutral elements, double negations or idempotent operations are removed, operations on constants performed. This captures a relevant part of the semantics of the non-stack operators.

The Synthesis Problem. This section hints on how the generated bytecode will be, and on that the synthesis of optimal bytecode from the specification is challenging.

Example 7.1.1 *From Figure 7.2, we know that we have to compute $x_0 + x_1$ and $x_2 + x_3$, but we have to decide which summation we compute first. We show two possible computations in Figure 7.3 (both can be synthesised as we will show in Section 7.3). On the left, we have the best bytecode (together with the stack evolution) when we first compute $x_2 + x_3$ and on the right when we first compute $x_0 + x_1$. Computing first one sub-expression or the other has an impact on the consumed gas, since the bytecode on the left has a gas cost of* 31 *and the bytecode on the right has a gas cost of* 25*, which is indeed the optimum.*

Both solutions are far better than the original generated bytecode whose gas cost was 76*. Besides, note that the cost of the two additions and the*

`SWAP3`	$[x_3, x_1, x_2, x_0, x_4]$		
`SWAP1`	$[x_1, x_3, x_2, x_0, x_4]$	`ADD`	$[x_0 + x_1, x_2, x_3, x_4]$
`SWAP2`	$[x_2, x_3, x_1, x_0, x_4]$	`SWAP2`	$[x_3, x_2, x_0 + x_1, x_4]$
`ADD`	$[x_2 + x_3, x_1, x_0, x_4]$	`ADD`	$[x_3 + x_2, x_0 + x_1, x_4]$
`SWAP2`	$[x_0, x_1, x_2 + x_3, x_4]$	`SWAP1`	$[x_0 + x_1, x_3 + x_2, x_4]$
`ADD`	$[x_0 + x_1, x_2 + x_3, x_4]$	`EXP`	$[(x_0 + x_1) ** (x_2 + x_3), x_4]$
`EXP`	$[(x_0 + x_1) ** (x_2 + x_3), x_4]$	`SWAP1`	$[x_4, (x_0 + x_1) ** (x_2 + x_3)]$
`SWAP1`	$[x_4, (x_0 + x_1) ** (x_2 + x_3)]$		

Figure 7.3: Bytecode for SFS in Figure 7.2 and optimised bytecode (right).

exponentiation is in total 16 *(that necessarily has to remain), which means that the optimal code has used only* 9 *units of gas for the rest while the original code needed* 60 *units.*

The next example shows that the optimal code is obtained when the sub-terms of the exponential are computed in the other order (compared to the previous example). Hence, an exhaustive search of all possibilities (with its associated computational demands) must be carried out to find the optimum.

Example 7.1.2 *Let us now in Figure 7.4 consider a slight variation of the previous example which the functional specification is* $[x_0, x_1, x_2, x_3]$ *to* $[x_3, (x_0 + x_1) ** (x_0 + x_2)]$. *Now, on the left-hand of Figure 7.4 side we have the best bytecode (together with the stack evolution) when we compute first* $x_0 + x_2$ *and on the right-hand side we have the best bytecode when we compute first* $x_0 + x_1$*. In this case the bytecode on the left has a gas cost of* 28*, which is indeed the optimum, and the bytecode on the right has a gas cost of* 31*. The original bytecode has gas cost* 74*, so again the improvement is huge.*

Both examples show that, in principle, even if we have the functional specification that guides the search, we have to exhaustively try all possible ways to obtain it, if we want to ensure that we have found the optimal bytecode.

Characteristics of our SMT Encoding. Our approach to superoptimise blocks is based on restricting the problem in such a way that we have both a

DUP1	$[x_0, x_0, x_1, x_2, x_3]$	DUP1	$[x_0, x_0, x_1, x_2, x_3]$
SWAP3	$[x_2, x_0, x_1, x_0, x_3]$	SWAP2	$[x_1, x_0, x_0, x_2, x_3]$
ADD	$[x_2 + x_0, x_1, x_0, x_3]$	ADD	$[x_1 + x_0, x_0, x_2, x_3]$
SWAP2	$[x_0, x_1, x_2 + x_0, x_3]$	SWAP2	$[x_2, x_0, x_1 + x_0, x_3]$
ADD	$[x_0 + x_1, x_2 + x_0, x_3]$	ADD	$[x_2 + x_0, x_1 + x_0, x_3]$
EXP	$[(x_0 + x_1) ** (x_2 + x_0), x_3]$	SWAP1	$[x_1 + x_0, x_2 + x_0, x_3]$
SWAP1	$[x_3, (x_0 + x_1) ** (x_2 + x_0)]$	EXP	$[(x_1 + x_0) ** (x_2 + x_0), x_3]$
		SWAP1	$[x_3, (x_1 + x_0) ** (x_2 + x_0)]$

Figure 7.4: Bytecode for SFS $[x_0, x_1, x_2, x_3]$ to $[x_3, (x_0 + x_1) ** (x_0 + x_2)]$.

great coverage of most EVM code optimisations and we can propose an encoding in a simple theory where an SMT solver can perform efficiently. To this end, the key point is to handle all non-stack operations, like `ADD`, `SUB`, `AND`, `OR`, `LT`, as *uninterpreted bytecodes.* This allows us to simplify the encoding in two directions. First, by considering them as uninterpreted bytecodes we can avoid reasoning on the theory of bit-vectors with width 256. Second, and even more important, this allows us to express the problem in the existentially quantified fragment, avoiding the $\exists/\forall$ alternation: We start from the SFS by introducing fresh variables abstracting out all terms built with uninterpreted functions, in such a way that every fresh variable represents a term $f(a_1, \ldots, a_n)$, where every a_i is either a (256 bit) numeric value, a fresh variable, or an initial stack variable. We also have sharing by having a single variable for every term, *e.g.*, $(x_0 + 1) ** (x_0 + 1)$, where x_0 is the top of the initial stack, is abstracted into $y_0 = \mathtt{EXP_U}(y_1, y_1)$ and $y_1 = \mathtt{ADD_U}(x_0, 1)$, where y_0 and y_1 are fresh variables and $\mathtt{EXP_U}$ and $\mathtt{ADD_U}$ are the uninterpreted bytecodes for exponentiation and addition, respectively. Now, in order to avoid universal quantification, we take advantage of the fact that only values from 0 to $2^{256} - 1$ can be introduced in the stack by a `PUSH` opcode and hence only this range can appear in the SFS. Therefore, if we assign values from 2^{256} on to fresh variables and initial stack variables we avoid the confusion between themselves and all other values in the problem.

After these two key observations have been made, we fix the maximal number n of opcodes and highest size h of the stack that is allowed in a solution. This can be bound by analysing the original code generated by the compiler. From this, we roughly encode the problem using variables $o_0, \ldots, o_{n-1}$ to express the operations of our code (together with variables $p_0, \ldots, p_{n-1}$ that encode the value $0 \leqslant p_i \leqslant 2^{256} - 1$ added to the stack when o_i is a `PUSH`), variables $s_0^i, \ldots, s_{h-1}^i$ to encode the contents of the stack before executing the operation o_i, where s_0^i is the top of the stack (we also use some Boolean variables to express the active part of the stack). Using this, we can encode the behaviour of all stack operations: `POP`, `PUSH`, `DUP`, `SWAP` for all its versions (like `DUP1`, `DUP2`, ...). For the uninterpreted bytecodes f_u, we basically add for every abstraction $y = f_u(a_1, \ldots, a_m)$ assertions stating that if we have $a_1, \ldots, a_m$ at the top of the stack at step i (*i.e.*, $s_0^i, \ldots, s_{m-1}^i$) and we take the operation f in o_i then in step $i+1$ we have $y, s_m^i, \ldots$ on the top of the stack. Again, as all fresh variables and initial stack variables have been replaced by values form 2^{256} on, there is no confusion with all other values.

As a final remark, we have also encoded the commutativity property of uninterpreted bytecodes representing the `ADD`, `MUL`, `AND`, `OR`, *etc.* This can be easily made by considering that the arguments can occur at the top of the stack in the two possible orders. Other properties like associativity are more difficult to encode.

Optimal Synthesis Using Max-SMT. The last key element is how we encode the optimisation problem of finding the bytecode with minimal gas cost. First, let us describe which notion of optimality we are considering. Our problem is defined as, given an SFS in which all occurring bytecodes there are considered uninterpreted and maybe commutative, we have to provide the bytecode with minimal gas cost whose SFS is equal modulo commutativity to the given one. From the encoding we have described in the previous section, we know that every solution to the SMT problem will have the same SFS as the given one. Hence, we only need to find the solution with minimal gas cost. In Chapter 6, this was made by implementing a loop on top of the SMT solving process which was calling the solver asking every time for a better solution in terms of gas, which was also encoded in the SMT problem. Such an approach cannot be easily implemented in an incremental way using the SMT solver as a black box without the corresponding performance penalty. Alternatively, we propose to encode the problem as a Max-SMT problem and hence, we can easily use any Max-SMT optimiser, like Z3 [32], Barcelogic [16], or (Opti)MathSAT [28], as a black box with an important gain in efficiency. The Max-SMT encoding adds to the previously defined SMT encoding some soft constraints, indicating which is the cost associated to choosing every family of operators. Then, choosing an operator from the *base* family has cost 2, from the *verylow* 3, and so on and the optimal solution is the solution that minimizes this cost, which can be obtained with a Max-SMT optimiser.

7.2 SFS from EVM Bytecode

The starting point of our work is the CFG of the EVM bytecode to be optimised. There are a number of tools, *e.g.*, Ethir [4], Madmax [48], Mythril [81] or Rattle [94]) that are able to compute the CFG and we do not need to formalise, neither to implement, this initial CFG generation step. Since there are bytecode instructions that we do not optimise, for each of the blocks of the provided CFG, we first perform a further block-partitioning that splits a basic block into the sub-blocks that will be optimised by our method as defined below. A basic block is defined as a sequence of EVM instructions without any `JUMP` bytecode.

Definition 7.2.1 (block-partitioning) *Given a basic block* $B = [b_0, b_1, ..., b_n]$*, we define its block-partitioning* $\mathsf{blocks}(B)$ *as the longest blocks* $b_i, \ldots, b_j$ *for which*

$$\mathsf{blocks}(B) = \left\{ b_i, \ldots, b_j \;\middle|\; \begin{array}{l} (\forall k. i < k < j, b_k \notin \mathsf{Jump} \cup \mathsf{Terminal} \cup \{\texttt{JUMPDEST}\}) \;\wedge \\ (i{=}0 \vee b_{i-1} \in \mathsf{Split} \cup \{\texttt{JUMPDEST}\}) \;\wedge \\ (j{=}n \vee b_{j+1} \in \mathsf{Jump} \cup \mathsf{Split} \cup \mathsf{Terminal}) \end{array} \right\}$$

where $\mathsf{Jump} = \{\texttt{JUMP}, \texttt{JUMPI}\}$*,* $\mathsf{Terminal} = \{\texttt{RETURN}, \texttt{REVERT}, \texttt{INVALID}, \texttt{STOP}\}$*,*

```
1  SSTORE
2  SWAP1
3  DUP5
4  SWAP1
5  MLOAD
6  SWAP1
7  DUP2
8  MSTORE
9  PUSH1 0x20
10 ADD
11 PUSH1 0x40
12 MLOAD
13 DUP1
14 SWAP2
15 SUB
16 SWAP1
17 LOG2
18 POP
19 PUSH1 0x01
20 SWAP2
21 SWAP1
22 POP
23 JUMP
```

Block 1	Block 2	Block 3
2 SWAP1 3 DUP5 4 SWAP1 5 MLOAD 6 SWAP1 7 DUP2	9 PUSH1 0x20 10 ADD 11 PUSH1 0x40 12 MLOAD 13 DUP1 14 SWAP2 15 SUB 16 SWAP1	19 POP 20 PUSH1 0x01 21 SWAP2 22 SWAP1 23 POP

Figure 7.5: CFG block of a real smart contract (top), and blocks generated to build the functional description of the EVM bytecode (bottom).

and Split = {SSTORE, MSTORE, LOG, CALLDATACOPY, CODECOPY, EXTCODECOPY, RETURNDATACOPY}.

The bytecodes whose effects are not reflected on the stack induce the partitioning and are omitted in the fragmented sub-blocks. These include the bytecodes that modify the memory, the storage or record a log, that belong to the Split set. Figure 7.5 shows a CFG block at the top and the blocks generated to build the functional description at the bottom. The original CFG block contains the bytecodes SSTORE, MSTORE, and LOG2. Thus, it is split into three different blocks that do not contain these bytecodes.

Once we have the partitioned blocks from the CFG, we obtain a functional description of the output stack (*i.e.*, the stack after executing the sequence of bytecodes in the block) using symbolic execution for each of the partitioned blocks. As the stack is empty before executing a transaction and the number of elements that each EVM bytecode consumes and produces is known, the size of the stack at the beginning of each block can be inferred statically. We can thus assume that the initial stack size is given within the CFG. A symbolic stack $\mathcal{S}$ is a list of size k that represents the state of the stack where the list position 0 corresponds to the top of the stack and $k-1$ is the index of the bottom of

$$\begin{array}{llcl}
\textit{(i).} & \tau(\mathcal{S}, \texttt{PUSH } x) & = & [x \mid \mathcal{S}] \\
\textit{(ii).} & \tau(\mathcal{S}, \texttt{DUP}i) & = & [\mathcal{S}[i-1] \mid \mathcal{S}] \\
\textit{(iii).} & \tau(\mathcal{S}, \texttt{SWAP}i) & = & \textit{temp} = \mathcal{S}[0], \mathcal{S}[0] = \mathcal{S}[i], \mathcal{S}[i] = \textit{temp} \\
\textit{(iv).} & \tau(\mathcal{S}, \texttt{POP}) & = & \mathcal{S}.\textit{remove}(0) \\
\textit{(v)} & \tau(\mathcal{S}, \texttt{OP}) & = & [\texttt{OP}(\mathcal{S}[0], ..., \mathcal{S}[\delta-1]) \mid \mathcal{S}[\delta : \textit{len}(\mathcal{S})]\ ]
\end{array}$$

Figure 7.6: Symbolic execution of the instructions that operate on the stack.

the stack, such that $\mathcal{S}[i]$ is the symbolic value stored at the position i of the stack. Initially, the input stack maps each index to a symbolic variable s_i.

The symbolic execution of each bytecode is defined using the transfer function τ which takes a stack $\mathcal{S}$ and a bytecode and transforms the stack. Here, $\mathcal{S}$ is represented as a list with operations to concatenate an element ($|$), index an element at position i ($\mathcal{S}[i]$) or a range from i to j ($\mathcal{S}[i : j]$), remove an element at position i ($\textit{remove}(i)$), and get the length of the list ($\textit{len}(\mathcal{S})$). The bytecodes transform the $\mathcal{S}$ as described in Figure 7.6: *(i)* the `PUSH` bytecode adds the value x to the top of the stack, *(ii)* `DUP`i duplicates the element at position $i-1$ to the top of the stack, *(iii)* `SWAP`i exchanges the value at the top of the stack with the one stored at position i using a temporary variable *temp*, *(iv)* `POP` deletes the value stored in the top of the stack, *(v)* `OP` represents all other EVM bytecodes that operate with the stack (arithmetic and bit-wise operations among others). In that case, τ creates a symbolic expression that is a functor with the same name as the original EVM bytecode and as arguments the symbolic expressions stored in the stack elements that it consumes. Here, δ stands for the number of elements that the EVM bytecode `OP` gets from the stack.

Now, the SFS can be defined using the function τ as follows.

Definition 7.2.2 *Given a block B with an initial size of the stack k, the initial state of the stack $\mathcal{S}_0$ stores at each position $i \in \{0, ..., k-1\}$ a symbolic variable s_i. Then, the transfer function τ is extended to the block B, denoted by $\tau(B)$, as: $[s_0, \ldots, s_{k-1}]$ if B is empty; and $\tau(\tau(B'), o)$ if B has o as last operation*

pp2:	$\tau(\mathcal{S}, \texttt{PUSH1}\ 0x00)$	$=$	$[0, s_0, s_1, s_2, s_3, s_4]$
pp3:	$\tau(\mathcal{S}, \texttt{DUP1})$	$=$	$[0, 0, s_0, s_1, s_2, s_3, s_4]$
pp5:	$\tau(\mathcal{S}, \texttt{DUP6})$	$=$	$[s_2, 0, 0, 0, s_0, s_1, s_2, s_3, s_4]$
pp6:	$\tau(\mathcal{S}, \texttt{DUP8})$	$=$	$[s_3, s_2, 0, 0, 0, s_0, s_1, s_2, s_3, s_4]$
pp7:	$\tau(\mathcal{S}, \texttt{ADD})$	$=$	$[\texttt{ADD}(s_3, s_2), 0, 0, 0, s_0, s_1, s_2, s_3, s_4]$
pp8:	$\tau(\mathcal{S}, \texttt{SWAP2})$	$=$	$[0, 0, \texttt{ADD}(s_3, s_2), 0, s_0, s_1, s_2, s_3, s_4]$
pp9:	$\tau(\mathcal{S}, \texttt{POP})$	$=$	$[0, \texttt{ADD}(s_3, s_2), 0, s_0, s_1, s_2, s_3, s_4]$
pp15:	$\tau(\mathcal{S}, \texttt{DUP1})$	$=$	$[\texttt{ADD}(s_1, s_0), \texttt{ADD}(s_1, s_0), \texttt{ADD}(s_3, s_2),$ $0, s_0, s_1, s_2, s_3, s_4]$
pp16:	$\tau(\mathcal{S}, \texttt{DUP3})$	$=$	$[\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0), \texttt{ADD}(s_1, s_0),$ $\texttt{ADD}(s_3, s_2), 0, s_0, s_1, s_2, s_3, s_4]$
pp17:	$\tau(\mathcal{S}, \texttt{EXP})$	$=$	$[\texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0)), \texttt{ADD}(s_1, s_0),$ $\texttt{ADD}(s_3, s_2), s_0, s_1, s_2, s_3, s_4]$
pp27:	$\tau(\mathcal{S}, \texttt{POP})$	$=$	$[s_4, \texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0))]$

Figure 7.7: Selected results after program points from Figure 7.1.

and B' is the resulting block without o. The SFS of B is $\mathcal{S}_0 \Longrightarrow \mathcal{S} = \tau(B)$.

Example 7.2.1 *Consider the block formed by the EVM bytecode shown in Figure 7.1, starting with the bytecode at program point 2 (pp2 for short) and finishing with the bytecode at pp27. Before executing the block symbolically, the initial stack is $\mathcal{S}_0 = [s_0, s_1, s_2, s_3, s_4]$ and $k = 5$. Figure 7.7 shows results at the next program points after applying the transfer function τ for selected examples. Altogether, the output stack of the SFS given by τ for the block in Figure 7.1 is $\mathcal{S} = [s_4, \texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0))]$. For example, we can see that τ updates the stack inserting a 0 in the top of the stack at pp2. At pp8, it swaps the element in the top of the stack ($\texttt{ADD}(s_3, s_2)$) with the element stored at position 2 (0). It generates a symbolic expression to represent the addition at pp7 with the values stored in the position of the stack that it consumes. At pp17 it generates a new symbolic expression $\texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0))$ to represent the exponentiation of the two elements stored in the top of the stack. Note that in this case these elements are also symbolic expressions of the two previous additions symbolically executed before.*

Finally, we capture optimisations based on the semantics of the arithmetic and bit-wise operations, by applying simplification rules on the SFS of the block before we proceed to generate the optimised code. This simplification besides reducing the number of operations includes other notions of simplification as well. The easiest examples are the application of simplification rules like with the units of every operation, or with the idempotence of bit-wise Boolean operators.

7.3 Optimal Synthesis using Max-SMT

We describe our Max-SMT encoding and start by pre-processing the SFS into an abstract form that is convenient for the encoding. The SFS and the encoding generated for the example shown in Figure 7.1 are available at github.com/mariaschett/syrup-backend/tree/master/examples/cav2020.

Abstracting Uninterpreted Functions. Before we apply our encoding, we need to abstract all sub-expressions occurring in the SFS, by introducing new fresh variables $s_k, s_{k+1}, \ldots$ that start after the last stack variable in the initial stack $[s_0, \ldots, s_{k-1}]$ of size k. In this process we have a mapping from fresh variables to shallow expressions of depth one, *i.e.*, built with a function symbol and variables or constants as arguments. Here we introduce the *minimal* number of fresh variables that allow us to describe the SFS using only shallow expressions. By minimal, we mean that we use the same variable if some sub-term occurs more than once. We also take into account commutativity properties to avoid creating unnecessary fresh variables. Finally if an uninterpreted function occurs more than once, we add a subscript from 0 on to distinguish them. As a result we have that the *abstracted SFS* is defined by a stack S containing only stack variables, fresh variables or constants in $\{0, \ldots, 2^{256} - 1\}$ and a map M from fresh variables to shallow terms formed by an uninterpreted function applied to stack variables, fresh variables or constants (in $\{0, \ldots, 2^{256} - 1\}$). Trivially, all positions in the stack in the SFS and the abstracted SFS are equal when the map is fully applied to remove all fresh variables and the subscripts are removed. Moreover, we have that every uninterpreted function of the SFS has a fresh variable assigned in the map and all function symbols in the map are different.

Example 7.3.1 *The abstraction of the SFS* $[s_4, \texttt{EXP}(\texttt{ADD}(s_3, s_2), \texttt{ADD}(s_1, s_0))]$ *shown in Example 7.2.1 needs three fresh variables* s_5*,* s_6 *and* s_7*. Then, the abstracted SFS is the stack* $S = [s_4, s_7]$ *and the mapping* M *is defined as* $\{s_5 \mapsto \texttt{ADD}_0(s_3, s_2), s_6 \mapsto \texttt{ADD}_1(s_1, s_0), s_7 \mapsto \texttt{EXP}(s_5, s_6)\}$.

Modelling the Stack. A key element in our encoding is the representation of the stack and the elements it contains. As mentioned in Section 7.1, a first observation is that in our approach we will only have in the stack constants in the domain $\{0, \ldots, 2^{256} - 1\}$—we do not care if they represent a negative number or not, as they are handled simply as 256-bit words—initial stack variables $s_0, \ldots, s_{k-1}$ and fresh variables $s_k, \ldots, s_v$. In order to distinguish between constants and the variables s_i, we assign to every variable s_i, with $i \in \{0, \ldots, v\}$, the constant $2^{256} + i$. Now, for instance, we can establish that a `PUSH` operation can only introduce a constant in $\{0, \ldots, 2^{256} - 1\}$ and that fresh variables s_i can only be introduced by uninterpreted functions if the appropriate arguments are in the stack:

$$S_V \equiv \bigwedge_{0 \leqslant i < v} s_i = 2^{256} + i$$

The rest of stack operations, like `DUP` or `SWAP`, just duplicate or move whatever is in the stack. Since in our encoding we will use the variables $s_0, \ldots, s_v$, as they are part of the SFS, we have a first constraint assigning the constant values to all these variables.

Let us now show how we model the stack along the execution of the instructions. First, we have to fix a bound on the number of operations b_o and the size of the stack b_s. We can apply different heuristics to this end though considering the initial number of operations and the maximum number of stack elements involved in the block are sound bounds. We have to express a stack of b_s positions after executing j operations with $j \in \{0, \ldots, b_o\}$. To this end, on the one hand, we use existentially quantified variables $x_{i,j} \in \mathbb{Z}$ with

$i \in \{0, \ldots, b_s - 1\}$ and $j \in \{0, \ldots, b_o\}$ to express the word at position i of the stack after executing the first j operations of the code, where $x_{0,j}$ encodes the word on the top of the stack. On the other hand to complete the modelling we introduce propositional variables $u_{i,j}$ with $i \in \{0, \ldots, b_s - 1\}$ and $j \in \{0, \ldots, b_o\}$, to denote the *utilisation* of the stack, *i.e.*, the words that the stack currently holds. Here, $u_{i,j}$ indicates that the word at position i of the stack after executing the first j operations exists or not.

Additionally, to simplify the next definitions we have the following parametrised constraint that, given an instruction step j with $0 < j \leqslant b_o$, two stack positions α and β and a shift amount $\delta \in \mathbb{Z}$, with $0 \leqslant \alpha$, $0 \leqslant \alpha + \delta$, $\beta < b_s$ and $\beta + \delta < b_s$, imposes that the stack after executing $j + 1$ instructions between positions α and β is the same as the stack after executing the j instruction but with a shift of δ; they are moved up if negative and moved down otherwise:

$$\mathsf{move}(j, \alpha, \beta, \delta) \equiv \bigwedge_{\alpha \leqslant i \leqslant \beta} u_{i+\delta,j+1} = u_{i,j} \ \wedge \ x_{i+\delta,j+1} = x_{i,j}$$

Encoding of Instructions. Let $\mathcal{I}$ be the set of instructions occurring in our problem. The set $\mathcal{I}$ is split in three subsets $\mathcal{I}_C \uplus \mathcal{I}_U \uplus \mathcal{I}_S$, where: $\mathcal{I}_C$ contains the commutative uninterpreted functions occurring in the map M of the abstracted SFS, $\mathcal{I}_U$ contains the non-commutative uninterpreted functions occurring in M, and $\mathcal{I}_S$ contains the stack operations: `PUSH`, that introduces an up to 32-byte item on top of the stack; `POP` that removes the top of the stack; `DUP`k, with $k \in \{1, \ldots, 16\}$ that copies the $k-1$ element of the stack on top of the stack; `SWAP`k, with $k \in \{1, \ldots, 16\}$ that swaps the top of the stack with the k element of the stack; and an extra operation `NOP` that does nothing. Note that, although in EVM there are 32 different `PUSH` instructions depending on the amount of bytes needed to express the item, in our context this distinction is unnecessary, since we can decide afterwards which `PUSH` we need by checking in the obtained solution which is the value to be pushed. Also, the operations `DUP`k in $\mathcal{I}_S$ are reduced to only those with $k < b_s$, otherwise we go beyond the maximal size of the stack. Similarly, the operations SWAPk in $\mathcal{I}_S$ are reduced to only those with $k < b_s$.

Let θ be a mapping from the set of instructions in $\mathcal{I}$ to consecutive different non-negative integers in $\{0, \ldots, m_\iota\}$, where $m_\iota + 1$ is the cardinality of $\mathcal{I}$. In order to encode the selected instructions at every step, we introduce the existentially quantified variables $t_j \in \{0, \ldots, m_\iota\}$, with $j \in \{0, \ldots, b_o - 1\}$ where for every instruction $\iota \in \mathcal{I}$, if $t_j = \theta(\iota)$ then we have that the operation executed at step j is ι. Additionally, we introduce associated existentially quantified variables $a_j \in \{0, \ldots, 2^{256} - 1\}$, with $j \in \{0, \ldots, b_o - 1\}$, to express the value pushed at the top of the stack when $t_j = \theta(\texttt{PUSH})$. Otherwise the value of a_j is meaningless.

Encoding the Stack Operations. First we show how we encode the effect of choosing in t_j one of the operations in $\mathcal{I}_S$ that does not depend on the particular (abstracted) SFS we are considering. The following parametrised constraints show this effect:

$$
\begin{aligned}
C_{\texttt{PUSH}}(j) \equiv t_j = \theta(\texttt{PUSH}) \; &\Rightarrow 0 \leqslant a_j < 2^{256} \wedge \neg u_{b_s-1,j} \wedge u_{0,j+1} \wedge \\
&\quad\; x_{0,j+1} = a_j \wedge \mathsf{move}(j, 0, b_s - 2, 1) \\
C_{\texttt{DUP}k}(j) \equiv t_j = \theta(\texttt{DUP}k) \; &\Rightarrow \neg u_{b_s-1,j} \wedge u_{k-1,j} \wedge u_{0,j+1} \wedge \\
&\quad\; x_{0,j+1} = x_{k-1,j} \wedge \mathsf{move}(j, 0, b_s - 2, 1) \\
C_{\texttt{SWAP}k}(j) \equiv t_j = \theta(\texttt{SWAP}k) \; &\Rightarrow u_{k,j} \wedge u_{0,j+1} \wedge x_{0,j+1} = x_{k,j} \wedge u_{k,j+1} \wedge \\
&\quad\; x_{k,j+1} = x_{0,j} \wedge \mathsf{move}(j, 1, k - 1, 0) \wedge \\
&\quad\; \mathsf{move}(j, k + 1, b_s - 1, 0) \\
C_{\texttt{POP}}(j) \equiv t_j = \theta(\texttt{POP}) \; &\Rightarrow u_{0,j} \wedge \neg u_{b_s-1,j+1} \wedge \mathsf{move}(j, 1, b_s - 1, -1) \\
C_{\texttt{NOP}}(j) \equiv t_j = \theta(\texttt{NOP}) \; &\Rightarrow \mathsf{move}(j, 0, b_s - 1, 0)
\end{aligned}
$$

Notice that the stack before executing the instruction t_j is given in the variables $x_{0,j}, \ldots, x_{b_s-1,j}$ and $u_{0,j}, \ldots, u_{b_s-1,j}$, while the stack after executing t_j is given in $x_{0,j+1}, \ldots, x_{b_s-1,j+1}$ and $u_{0,j+1}, \ldots, u_{b_s-1,j+1}$.

In order to avoid redundant solutions with `NOP` in intermediate steps, we have to add as well a constraint stating that once we choose `NOP` as instruction t_j we can only choose `NOP` for the following instructions $t_{j+1}, t_{j+2} \ldots$:

$$C_{\texttt{from}\mathtt{NOP}} \equiv \bigwedge_{0 \leqslant j < b_o - 1} t_j = \theta(\mathtt{NOP}) \Rightarrow t_{j+1} = \theta(\mathtt{NOP})$$

Encoding Uninterpreted Operations. The encoding of the uninterpreted operations comes from the map M of the abstracted SFS. First of all, note that, every function f occurs only once in M, since subscripts are introduced, and for every $r \mapsto f(o_0, \ldots, o_{n-1})$ in M we have that $f \in I_C \uplus \mathcal{I}_U$, r is a fresh variable, and $o_0, \ldots, o_{n-1}$ are either initial stack variables, fresh variables or constants. Note also that if $f \in \mathcal{I}_C$ then $n = 2$. Therefore, we define in the encoding the effect of choosing in t_j the uninterpreted function f with $r \mapsto f(o_0, \ldots, o_{n-1})$ in M, as an operation that takes its arguments $o_0, \ldots, o_{n-1}$ from the stack and places its result r in the stack, where o_0 must be at the top of the stack.

$$\begin{aligned} C_U(j, f) \equiv t_j = \theta(f) \Rightarrow & \bigwedge_{0 \leqslant i \leqslant n-1} (u_{i,j} \wedge x_{i,j} = o_i) \wedge u_{0,j+1} \wedge x_{0,j+1} = r \wedge \\ & \mathsf{move}(j, n, \mathsf{min}(b_s - 2 + n, b_s - 1), 1 - n) \wedge \\ & \bigwedge_{b_s - n + 1 \leqslant i \leqslant b_s - 1} \neg u_{i,j+1} \\ & \text{where } f \in \mathcal{I}_U \text{ and } r \mapsto f(o_0, \ldots, o_{n-1}) \in M \end{aligned}$$

Now for the commutative functions the only difference is that we know that $n = 2$ and that we can find the arguments in any of both orders in the stack:

$$\begin{aligned} C_C(j, f) \equiv t_j = \theta(f) \Rightarrow\ & u_{0,j} \wedge u_{1,j} \wedge \\ & ((x_{0,j} = o_0 \wedge x_{1,j} = o_1) \vee (x_{0,j} = o_1 \wedge x_{1,j} = o_0)) \wedge \\ & u_{0,j+1} \wedge x_{0,j+1} = r \wedge \mathsf{move}(j, 2, b_s - 1, -1) \wedge \neg u_{b_s - 1, j} \\ & \text{where } f \in \mathcal{I}_C \text{ and } r \mapsto f(o_0, o_1) \in M \end{aligned}$$

Finding the Target Program. We assign to every $\iota \in \mathcal{I}$ an integer. Then, $t_j \in \mathbb{Z}$ encodes the chosen instruction at position j in the target program for $0 \leqslant j < b_o$. To encode the selection of an instruction for every t_j, we have the following constraint:

$$
\begin{aligned}
C_{\mathcal{I}} \equiv C_{\texttt{fromNOP}} \wedge \bigwedge_{0 \leqslant j < b_o} 0 \leqslant t_j \leqslant m_\iota \wedge \\
C_{\texttt{PUSH}}(j) \wedge C_{\texttt{DUP}k}(j) \wedge C_{\texttt{SWAP}k}(j) \wedge C_{\texttt{POP}}(j) \wedge C_{\texttt{NOP}}(j) \wedge \\
\bigwedge_{f \in \mathcal{I}_U} C_U(j, f) \wedge \bigwedge_{f \in \mathcal{I}_C} C_C(j, f)
\end{aligned}
$$

Complete Encoding. Let us conclude our encoding by defining the formula C_{SFS} that states the whole problem of finding an EVM block for a given initial stack $[s_0, \ldots, s_{k-1}]$ and abstracted SFS with final stack $[f_0, \ldots, f_{w-1}]$ and map M. Hence, we introduce a constraint B to describe how the stack at the beginning is and a constraint E to describe how the stack at the end is and combine all the constraints defined above to express C_{SFS}.

$$
\begin{aligned}
B &\equiv \bigwedge_{0 \leqslant \alpha < k} (u_{\alpha,0} \wedge x_{\alpha,0} = s_\alpha) \wedge \bigwedge_{k \leqslant \beta \leqslant b_s - 1} \neg u_{\beta,0} \\
E &\equiv \bigwedge_{0 \leqslant \alpha < w} (u_{\alpha,b_o} \wedge x_{\alpha,b_o} = f_\alpha) \wedge \bigwedge_{w \leqslant \beta \leqslant b_s - 1} \neg u_{\beta,b_o} \\
C_{SFS} &\equiv S_V \wedge C_{\mathcal{I}} \wedge B \wedge E
\end{aligned}
$$

Finally, let us mention that the performance of the used SMT solvers greatly improves when the following (redundant) constraint, which states that all functions in $\mathcal{I}_U \uplus \mathcal{I}_C$ should be eventually used, is added:

$$
\bigwedge_{\iota \in \mathcal{I}_U \uplus \mathcal{I}_C} \bigvee_{0 \leqslant j < b_o} t_j = \theta(\iota)
$$

Empirical evidence shows, that this constraint helps the solver to establish optimality, and removing it increases the time-outs and time taken by roughly 50 %. On the other hand, adding the similar constraint that all functions in

$\mathcal{I}_U \uplus \mathcal{I}_C$ are used at most once, while also helping the solvers to show optimality for already optimal blocks, the performance for finding optimisations decreases by a similar rate. As the latter is our main motivation, we did not include the constraint.

From Models to EVM Blocks. The following definition shows how we can extract a concrete set of operations from a model for the formula C_{SFS} that computes the given SFS.

Definition 7.3.1 *Given a model σ for C_{SFS} we have that $\mathsf{block}(\sigma)$ is defined as the sequence of EVM operations $o_0, \ldots, o_f$ where f is the largest $j \in \{0, \ldots, b_o - 1\}$ such that $t_j \neq \theta(\texttt{NOP})$. Now for all $\alpha \in \{0, \ldots, f\}$ the operation o_α is taken as*

1. *$o_\alpha = \texttt{PUSH}k a_\alpha$ if $t_\alpha = \theta(\texttt{PUSH})$ and a_α can be represented with k bytes.*
2. *$o_\alpha = \iota$ if $t_\alpha = \theta(\iota)$ where $\iota \in \mathcal{I}_S \setminus \{\texttt{PUSH}\}$*
3. *$o_\alpha = \iota$ if $t_\alpha = \theta(\iota)$ where $\iota \in \mathcal{I}_U \uplus \mathcal{I}_C$ and ι has no subscript.*
4. *$o_\alpha = \iota$ if $t_\alpha = \theta(\iota_l)$ where $\iota_l \in \mathcal{I}_U \uplus \mathcal{I}_C$ and has subscript l.*

Optimisation Using Max-SMT. We want to obtain the optimal solution. Since the cost of the solution can be expressed in terms of the cost of every of the instructions we select in all t_j, we will introduce soft constraints expressing the cost of every selection. A (partial weighted) Max-SMT problem is an optimisation problem where we have an SMT formula which establishes the *hard constraints* of the problem and a set of pairs $\{[C_1, \omega_1], \ldots, [C_m, \omega_m]\}$, where each C_i is an SMT clause and ω_i is its weight, that establishes the *soft constraints*. In this context, the optimisation problem consists in finding the model that satisfies the hard constraints and minimizes the sum of the weights of the falsified soft constraints. Our approach to find the optimal code is by encoding the problem as a Max-SMT optimisation problem, where we add to the SMT formula C_{SFS} which defines our *hard constraints* a set of *soft constraints* such that sum of the weights of the falsified soft constraints coincides with the cost (in terms of gas) of the operations taken in every step. Therefore the optimal solution to the Max-SMT problem coincides with the optimal solution in terms of gas cost.

In the EVM, every operation has an associated gas cost, which in general is constant, but in some few cases may depend on the particular arguments it is applied to or on the state of the blockchain. All these operations that are non-constant are considered as uninterpreted, and hence we cannot change the operands on which they are applied. Therefore, omitting the non-constant part cannot affect which is the optimal solution. Thanks to this, we can split our set of instructions $\mathcal{I}$ in $p+1$ disjoint sets $W_0 \uplus \ldots \uplus W_p$ where all instructions in W_i have the same constant cost cost_i, and such that the costs are strictly increasing, *i.e.*, $\mathsf{cost}_0 = 0$ and $\mathsf{cost}_{i-1} < \mathsf{cost}_i$ for all $i \in \{1, \ldots, p\}$.

In the following we describe the encoding we have chosen for the weighted clauses (we have tried other slightly simpler alternatives but, in general, they behave worse). Let $w_i = \mathsf{cost}_i - \mathsf{cost}_{i-1}$ for $i \in \{1, \ldots, p\}$. Hence, we have that $w_i > 0$ and, moreover, $\mathsf{cost}_i = \Sigma_{1 \leqslant \alpha \leqslant i} w_\alpha$ for $i \in \{1, \ldots, p\}$. Then, our Max-SMT problem O_{SFS} is obtained adding to C_{SFS} the following soft constraints

$$O_{SFS} \equiv C_{SFS} \wedge \bigwedge_{0 \leqslant j < b_o} \bigwedge_{1 \leqslant i \leqslant p} [\bigvee_{\iota \in W_0 \uplus \ldots \uplus W_{i-1}} t_j = \theta(\iota), w_i]$$

Therefore, if the selected instruction at step j is ι (*i.e.*, $t_j = \theta(\iota)$) for some $\iota \in W_i$ then we accumulate the weight w_α of all soft clauses with $\alpha \in \{1, \ldots, i\}$, which as said sums cost_i, and hence we accumulate the cost of executing the instruction ι.

7.4 Evaluation

This section presents the results of our evaluation using syrup, the synthesiser of superoptimised smart contracts that implements our approach. Our tool syrup uses Ethir [4] to generate the CFGs of the analysed contracts and Z3 [32] version 4.8.7, Barcelogic [16], and MathSAT [28] version 1.6.3, namely its optimality framework (Opti)MathSAT, as SMT solvers. We refer by s-Z3, s-Bar, s-OMS, to the results of using syrup with the respective solvers. Experiments have been

	ebso	s-Z3	s-Bar	s-OMS	s-All
A	3882 (6.34 %)	20 636 (33.71 %)	20 783 (33.95 %)	20 973 (34.26 %)	20 988 (34.28 %)
O	393 (0.64 %)	25 922 (42.34 %)	26 458 (43.22 %)	28 063 (45.84 %)	28 195 (46.06 %)
B	550 (0.90 %)	6288 (10.27 %)	3051 (4.98 %)	5293 (8.65 %)	5726 (9.35 %)
N	n/a	1933 (3.16 %)	563 (0.92 %)	837 (1.37 %)	1020 (1.67 %)
T	56 392 (92.12 %)	6438 (10.52 %)	10 362 (16.93 %)	6051 (9.88 %)	5288 (8.64 %)
G	27 726	1 188 311	1 003 717	1 272 381	1 309 875
S	not avail.	13 710 904.75	13 141 046.21	12 239 980.85	10 948 011.57

Table 7.1: Result of optimising with syrup

performed on a cluster with Intel Xeon Gold 6126 CPUs at 2.60 GHz, 2 GB of memory and time-out of 15 min, running CentOS Linux 7.6. The main components of syrup are implemented in Python and OCaml. The backend of syrup generating SMT constraints from a SFS is open-source and can be found at github.com/mariaschett/syrup-backend. Our tool accepts smart contracts written in versions of Solidity up to 0.4.25 and EVM bytecode v1.8.18, namely the three new EVM bytecodes (`SHL`, `SHR` and `SAR`) introduced from the Solidity compiler version 0.5.0 are not handled yet by Ethir. We use the same data set (and the results for ebso) from Chapter 6: the blocks of the 2500 most called contracts deployed on the Ethereum blockchain[1] after removing the duplicates and the blocks which are only different in the arguments of `PUSH` by abstracting to word size 4 bit. This results in a data set of 61 217 blocks.

As seen in Definition 7.2.1, we split the 61 217 blocks on certain bytecodes that are not optimised, leading to a total of 72 450. For comparison, we merge the split blocks back together. The Table 7.1 shows the results of optimising the 61 217 blocks by ebso (first column), and by syrup for every solver (next columns). In column s-All, we use the 3 solvers as a single framework in syrup that yields the best solution returned by any of the solvers (in parenthesis we show percentages).

Row **A** shows the number of blocks that were *A*lready optimal, *i.e.*, those that cannot be optimised because they already consume the minimal amount of gas and ebso/syrup find bytecode with the same consumption. Row **O** contains

[1] up to Ethereum blockchain block number 7 300 000 until 2019-03-04 01:22:15 UTC

the number of blocks that have been optimised and the found solution has been proven to be *O*ptimal, *i.e.*, the one that consumes the minimum amount of gas needed to obtain the SFS provided. The solvers used are able to provide the best solution found until the time-out is reached. Row **B** contains the number of blocks that have been optimised into a *B*etter solution that consumes less gas but it is not shown to be the optimum. Row **N** shows the number of blocks that have *N*ot been optimised and not proven to be optimal, *i.e.*, the solution found is the original one but there may exist a better one. Row **T** contains the number of blocks for which no model could be found when the *T*ime-out was reached. Row **G** contains the accumulated *G*as savings for all optimised blocks. Importantly, the real savings would be larger if the optimised blocks are part of a loop and hence might be executed multiple times. Row **S** shows the time in *S*econds in which each setting analyses all the blocks.

Let us first compare the results by ebso and our best results when using the portfolio of solvers in s-All. It is clear from the figures that syrup significantly outperforms ebso on the number of blocks handled (while ebso times out in 92.12 % of the blocks, we only time-out in 8.64 %) and on the overall gas gains (two orders of magnitude larger). For the analysed blocks (*i.e.*, those that do not time-out), the percentages of syrup for number of optimised into better blocks, into optimal blocks, and those proven to be already optimal, are much larger than those of ebso. We now discuss how the gains for the blocks that ebso can analyse compare to the gains by syrup. In particular, if missing part of the semantics of the uninterpreted instructions and the bytecode `SSTORE` significantly affects the gains. Out of 943 examples, where ebso found an optimisation, in 46 cases syrup proved optimality *wrt.* the SFS and saved 348 gas but saved less gas than ebso (total 10 514 gas). The source of this gain is the `SSTORE` bytecode: there are two blocks where ebso saves 5000 each, because it realises that we read from a key in storage to then store the value back unchanged. Our framework naturally extends to handle this storage optimisation. However, in nearly all of 393 cases, where ebso found an optimal solution—in 378 cases—syrup saves as much as ebso amounting to 2670 gas. That is, the additional semantics did not improve savings. Furthermore, in 43 cases out of 943, the semantics did impede ebso's performance so that syrup found a better result with 597 gas versus 440 of ebso. Therefore, we can conclude that syrup is far more scalable and precise than ebso, the cases in which syrup optimises less than ebso are seldom and can be naturally handled in the future. Moreover, they are offset by the cases where syrup did find an optimisation, whereas ebso did not.

Chapter 8

Populating a Peephole Optimiser

Differently to the previous Chapter 7, we now accept the high cost of finding optimisations in Chapter 6, but focus on how to generalise them into optimisation rules to be reusable at a low cost [H_b]. We then leverage these optimisation rules in a *peephole optimiser* which uses pattern matching to optimise a small fragment of code, *i.e.*, a peephole, by applying the optimisation rules. Finding sound optimisation rules is a bottleneck as witnessed by the peephole optimiser of the Solidity compiler solc.[1] Currently, solc features fewer than 20 rules compared to LLVM's 1000+ rules. Thus we propose a pipeline to automatically populate the peephole optimiser of a smart contract compiler by combining techniques from constraint solving and rewriting as illustrated in Figure 8.1.

Smart contract languages typically have a large and accessible code base to use as a basis for finding optimisations, *e.g.*, code deployed to public blockchains or test cases. This allows us to start from an existing code base, to *(1) find optimisations* by using automated tools to synthesize observationally equivalent but cheaper instruction sequences.

To give an example, the bytecode for the Ethereum virtual machine `PUSH 0 SUB PUSH 3 ADD SHA3` computes a hash of $3 + (0 - w)$ for some word w already on the stack. As $3 + (0 - w) = 3 - w$ the bytecode corresponding to `PUSH 3 SUB SHA3`, computes the same result *and* cheaper. From such optimisations, we can *(2) generate rules.* Using concepts from rewriting we generalize "unnec-

[1] `github.com/ethereum/solidity/blob/019ec63f63bae7bbe89f5b62bb7b202ef5dadce6/libevmasm/PeepholeOptimiser.cpp`

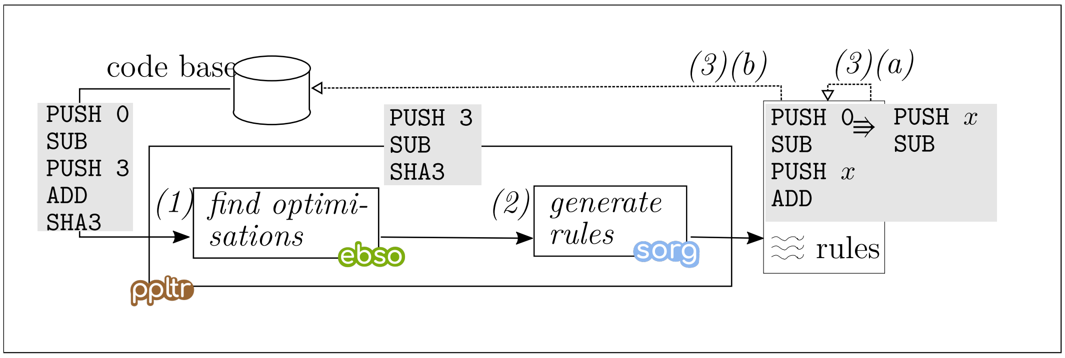

Figure 8.1: Pipeline to automatically generate peephole optimisation rules from a code base.

essarily specific" arguments and strip away "unnecessary" context to obtain optimisation rules.

For the above example, we generate the rule `PUSH` 0 `SUB` `PUSH` x `ADD` $\Rrightarrow$ `PUSH` x `SUB` by generalizing 3 to x. Finally we can *(3)* feed back and apply the generated rules to *(a)* the rules themselves, and *(b)* the code base and again start the cycle to find new optimisations. We demonstrate the applicability of our pipeline in a case study for bytecode EVM. We implemented a prototype: ppltr, a peephole otimisation rule generator. For phase *(1)*, we use our tool ebso from Chapter 6. For phase *(2)*, we use sorg, a superoptimisation based rule generator. All tools are available open-source under the Apache-2.0 license[2]. We evaluated our approach on bytecode of the 250 most called contracts of the Ethereum blockchain, where we found 2032 distinct optimisations from which we automatically generated 993 optimisation rules.

8.1 Procedure

We assume a machine model with a *state* over a set of *words* $\mathbb{W}$ with an observational equivalence relation $\equiv$ on states, which may take only parts of the state into account. States are modified based on *instructions* from a set $\mathcal{I}$, where an instruction $\iota \in \mathcal{I}$ deterministically transforms a state σ into some state σ' denoted by $\sigma \xrightarrow{\iota} \sigma'$. Some instructions act only on parts of the state,

[2]Available at `github.com/juliannagele/ebso/tree/v2.1`, `github.com/mariaschett/sorg/tree/v1.1`, and `github.com/mariaschett/ppltr/tree/v1.0`.

while others take immediate arguments from $\mathbb{W}$. We write $\iota(w_1, \ldots, w_k)$ for an instruction $\iota \in \mathcal{I}$ which takes k immediate arguments $w_1, \ldots, w_k \in \mathbb{W}$ and say that ι has arity k. For example, in a stack-based machine the instruction `PUSH 3` takes the immediate argument 3, while `SUB` has arity 0, but consumes two arguments from the stack. A *program* ρ is a sequence of instructions $\iota_0 \cdots \iota_n$. The length of ρ is its number of instructions, denoted by $|\rho|$. We write ε for the *empty* program and $\rho \cdot \tau$ for the concatenation of programs ρ and τ. A program $\rho = \iota_0 \cdots \iota_n$ transforms a family of states $\boldsymbol{\sigma} = (\sigma_j)_{j \leqslant n+1}$ by stepwise transformation, *i.e.*, $\sigma_0 \xrightarrow{\iota_0} \sigma_1 \xrightarrow{\iota_1} \ldots \xrightarrow{\iota_n} \sigma_{n+1}$, and we write $\sigma_0 \overset{\rho}{\twoheadrightarrow} \sigma_{n+1}$. Here σ_j is the state after executing j instructions, and σ_0 is the designated start state. We often write states instead of families of states, when the distinction is clear from the context.

We write $\mathsf{cost}(\iota, \sigma)$ for the cost incurred by executing instruction ι on state σ. The cost of executing a program is simply the sum of the cost of its instructions: $\mathsf{cost}(\iota_0 \cdots \iota_n, \boldsymbol{\sigma}) = \sum_{j=0}^{n} \mathsf{cost}(\iota_j, \sigma_j)$. Two programs ρ and τ are *equal*, denoted by $\rho = \tau$, if they are syntactically equal, and *equivalent*, $\rho \equiv \tau$, if they are observationally equivalent, *i.e.*, for states $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma'}$ with $\sigma_0 \equiv \sigma'_0$, $\sigma_0 \overset{\rho}{\twoheadrightarrow} \sigma_{|\rho|+1}$, and $\sigma'_0 \overset{\tau}{\twoheadrightarrow} \sigma'_{|\tau|+1}$ we have $\sigma_{|\rho|+1} \equiv \sigma'_{|\tau|+1}$.

Definition 8.1.1 *Let ρ and τ be programs with $\rho \equiv \tau$ and $\mathsf{cost}(\rho, \boldsymbol{\sigma}) > \mathsf{cost}(\tau, \boldsymbol{\sigma})$ for all states $\boldsymbol{\sigma}$. Then τ is an* optimisation *of ρ, and we write $\rho \gtreqless \tau$.*

We will show how we can obtain such optimisations—and we will use them to generate optimisation rules. To do so, we need to define what constitutes a rule. Therefore we abstract over the immediate arguments of instructions by using a countably infinite set of variables $\mathcal{V}$. We extend $\mathcal{I}$ to $\mathcal{I}^{\mathcal{V}}$ by adding instructions $\iota(x_1, \ldots, x_k)$ for all $x_1, \ldots, x_k \in \mathcal{V}$ and all $\iota \in \mathcal{I}$ of arity $k > 0$.

A program over $\mathcal{I}^{\mathcal{V}}$ is called a *program schema*. To obtain a *maximal schema* of a program schema s every $\iota(w_1, \ldots, w_k)$ in s is replaced by $\iota(x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are fresh variables from $\mathcal{V}$. All variables in a program schema s are collected in $\mathcal{V}\mathsf{ar}(s)$.

A *substitution* $\gamma : \mathcal{V} \to \mathbb{W} \cup \mathcal{V}$ maps variables to variables and words. In a *ground* substitution $\overline{\gamma}$ the range is restricted to $\mathbb{W}$, *i.e.*, $\overline{\gamma} : \mathcal{V} \to \mathbb{W}$. We *apply* γ to a schema s by replacing all variables x in s by $\gamma(x)$ and write $s\gamma$ for the result. Note that $s\overline{\gamma}$ is a program. A substitution γ is *at least as general* as a substitution γ', denoted $\gamma \leq \gamma'$, if there is a substitution γ'' such that $\gamma\gamma'' = \gamma'$. If $\gamma \leq \gamma'$ and $\gamma' \nleq \gamma$ then we say γ is *more general* than γ' and write $\gamma \lessdot \gamma'$. We call program schemas s and t *observationally equivalent*, and write $s \equiv t$, if $s\overline{\gamma} \equiv t\overline{\gamma}$ holds for all $\overline{\gamma}$ and write $\mathsf{cost}(s, \boldsymbol{\sigma}) > \mathsf{cost}(t, \boldsymbol{\sigma'})$ if $\mathsf{cost}(s\overline{\gamma}, \boldsymbol{\sigma}) > \mathsf{cost}(t\overline{\gamma}, \boldsymbol{\sigma'})$ for all $\overline{\gamma}$.

Definition 8.1.2 *Let ℓ and r be program schemas with $\ell \equiv r$ and $\mathsf{cost}(\ell, \boldsymbol{\sigma}) > \mathsf{cost}(r, \boldsymbol{\sigma})$. Then $\ell \Rrightarrow r$ is an (optimisation)* rule.

By definition, every optimisation $\rho \gtreqless \tau$ is an optimisation rule $\rho \Rrightarrow \tau$. A *context* C is a pair of program schemas (s_1, s_2). We write $C[t]$ for the program schema $s_1 \cdot t \cdot s_2$ and call s_1 a *prefix* and s_2 a *postfix* of $C[t]$. A context (s_1, s_2) is *at least as general* as a context (t_1, t_2), denoted by $(s_1, s_2) \leqslant (t_1, t_2)$, if there is a context (r_1, r_2) such that $r_1 \cdot s_1 = t_1$ and $s_2 \cdot r_2 = t_2$. If $C \leqslant C'$ and $C' \nleqslant C$ then we say C is *more general* than C' and write $C < C'$.

The following definition captures all optimisation rules that can produce a given optimisation when instantiated.

Definition 8.1.3 *The* optimisation rules *for an optimisation $\rho \gtreqless \tau$ are defined as $\mathcal{R}(\rho \gtreqless \tau) = \{\ell \Rrightarrow r \mid \rho = C[\ell\gamma]$ and $\tau = C[r\gamma]$ for some substitution γ and context $C\}$.*

For a formal proof we need to ensure that applying peephole optimisations is sound by the following conjecture. The proof idea is the same as in Conjecture 6.2.1 in Chapter 6.

Conjecture 8.1.1 *If $\rho \equiv \tau$ then $C[\rho] \equiv C[\tau]$ for all contexts C.*

Find Optimisations. As Definition 8.1.1 suggests finding an optimisation for a program ρ necessitates finding *(i)* an observationally equivalent program τ, where *(ii)* the cost of τ is less than the cost of ρ. In Chapter 6, we express the above as an SMT problem: given a source program ρ, is there a target program τ such that for all possible inputs, executing ρ and τ results in the same final state, but the cost of τ is less than the cost of ρ? However, our rule generation is robust to how we find our optimisations: given an optimisation, we can generate an optimisation rule for it.

As Definition 8.1.3 indicates generating optimisation rules from optimisations requires us *(i)* to find a substitution γ, and *(ii)* to find a context C.

Find a Substitution. In the first step we generalise the immediate arguments of instructions in an optimisation $\rho \gtreqless \tau$ by finding a substitution. We capture all possible generalisations of a rule using the following definition.

Definition 8.1.4 *The* generalised rules *of an optimisation rule* $\rho \Rrightarrow \tau$ *are defined as* $\mathsf{Gen}(\rho \Rrightarrow \tau) = \{\ell \Rrightarrow r \mid \ell\gamma = \rho$ *and* $r\gamma = \tau$ *for some substitution* $\gamma\}$.

Example 8.1.1 *Let* $\rho \equiv \tau$ *be the optimisation from the introduction,* i.e., `PUSH` *0* `SUB PUSH` *3* `ADD SHA3` $\equiv$ `PUSH` *3* `SUB SHA3`. *Then* $\mathsf{Gen}(\rho \equiv \tau)$ *consists of two rules:* `PUSH` *0* `SUB PUSH` x `ADD SHA3` $\Rrightarrow$ `PUSH` x `SUB SHA3` *and* $\rho \Rrightarrow \tau$ *itself. Note that the pair* `PUSH` y `SUB PUSH` x `ADD SHA3` *and* `PUSH` x `SUB SHA3` *is not in* $\mathsf{Gen}(\rho \equiv \tau)$. *Applying the substitution* $\gamma = \{x \mapsto 3, y \mapsto 0\}$ *would yield the original optimisation, but since* `PUSH` y `SUB PUSH` x `ADD SHA3` $\not\equiv$ `PUSH` x `SUB SHA3` *they do not constitute an optimisation rule.*

To implement Gen we can do an exhaustive search as follows: start from a maximal schema for the given optimisation and try all possibilities of mapping the variables back to the original values, checking whether the result yields a rule. The following procedure implements this approach, additionally using an order on the candidate substitutions to prune the search space.

Definition 8.1.5 *We define the function* $\mathsf{generalise}$ *in Algorithm 11.*

Using the order $\lessdot$ on substitutions to prune the search space is key for implementation. Pruning only removes rules covered by others as the following lemma shows.

Algorithm 11: Generalise the optimisation rule $\rho \Rrightarrow \tau$.

1 **function** $\mathsf{generalise}(\rho \Rrightarrow \tau)$
2 $\quad \mathcal{R} \coloneqq \varnothing$
3 $\quad \ell_0, r_0 \coloneqq$ maximal program schemas ℓ_0 and r_0 for ρ and τ with $\mathcal{V}\mathsf{ar}(\ell_0) \cap \mathcal{V}\mathsf{ar}(r_0) = \varnothing$
4 $\quad \gamma_0 \coloneqq$ the substitution γ_0 with $\rho = \ell_0\gamma_0$ and $\tau = r_0\gamma_0$
5 $\quad \Gamma \coloneqq \{\gamma \mid \gamma(x) = \gamma_0(x)$ or $\gamma(x) = y$ for $\gamma_0(x) = \gamma_0(y)$ and $x, y \in \mathcal{V}\mathsf{ar}(\ell_0) \cup \mathcal{V}\mathsf{ar}(r_0)\}$
6 $\quad$ **forall** $\gamma \in \Gamma$ **do**
7 $\quad\quad$ **if** $\ell_0\gamma \equiv r_0\gamma$ **then**
8 $\quad\quad\quad \mathcal{R} \coloneqq \mathcal{R} \cup \{\ell_0\gamma \Rrightarrow r_0\gamma\}$
9 $\quad\quad\quad \Gamma \coloneqq \Gamma \setminus \{\gamma' \mid \gamma \lessdot \gamma'\}$
10 $\quad\quad$ **else**
11 $\quad\quad\quad \Gamma \coloneqq \Gamma \setminus \{\gamma' \mid \gamma' \lessdot \gamma\}$
12 $\quad$ **return** $\mathcal{R}$

Lemma 8.1.1 *For every $\ell \Rrightarrow r \in \mathsf{Gen}(\alpha)$ of a rule α there is a $\ell' \Rrightarrow r' \in \mathsf{generalise}(\alpha)$ and a substitution γ such that $\ell'\gamma = \ell$ and $r'\gamma = r$.*

Proof 8.1.1 *We fix $\ell \Rrightarrow r \in \mathsf{Gen}(\alpha)$. Let ℓ_0 and r_0 be the maximal schemas of α. By definition of maximal schema there is a γ' such that $\ell_0\gamma' = \ell$ and $r_0\gamma' = r$. A renaming of γ' is in Γ and thus either $\mathsf{generalise}(\alpha)$ will consider it at some point, or it will be removed by either line 9 or line 11.*

If it is considered then a renaming of $\ell \Rrightarrow r$ is in $\mathsf{generalise}(\alpha)$. If it is removed by line 9, then a substitution γ with $\gamma \lessdot \gamma'$ and and $\ell_0\gamma \equiv r_0\gamma$ was considered. Thus $\ell_0\gamma \Rrightarrow r_0\gamma$ is in $\mathsf{generalise}(\alpha)$ and we have $\ell_0\gamma\gamma'' = \ell$ and $r_0\gamma\gamma'' = r$ for some γ'' by $\gamma \lessdot \gamma'$. If γ' was removed by line 11 then a substitution γ with $\gamma' \lessdot \gamma$ and and $\ell_0\gamma \not\equiv r_0\gamma$ was considered, but this contradicts the assumption $\ell \Rrightarrow r \in \mathsf{Gen}(\alpha)$, because observational equivalence is closed under substitution.

Example 8.1.2 *Take the optimisation* `PUSH` *0* `PUSH` *0* `ADD` $\geqslant \varepsilon$. *Then, ℓ_0 is* `PUSH` x_1 `PUSH` x_2 `ADD`, *r_0 is ε (line 3), and $\gamma_0 = \{x_1 \mapsto 0, x_2 \mapsto 0\}$ (line 4). The set Γ holds* $\{$ (i) $\{x_1 \mapsto 0, x_2 \mapsto 0\}$, (ii) $\{x_1 \mapsto x_1, x_2 \mapsto x_2\}$, (iii) $\{x_1 \mapsto 0, x_2 \mapsto x_2\}$, (iv) $\{x_1 \mapsto x_1, x_2 \mapsto 0\}$, (v) $\{x_1 \mapsto x_2, x_2 \mapsto x_2\}\}$. *Now, assuming we first pick* (iii) *for γ in line 6. As* `PUSH` *0* `PUSH` $x_2 = \ell_0\gamma \equiv r_0\gamma = \varepsilon$, *we add the rule* *to $\mathcal{R}$ (line 8). Then, we remove* (i) *from* Γ *in line 9. Now, because* (iii) $\lessdot$ (i) *holds,* i.e., (i) *instantiates more as* (iii) *we can also remove* (i) *from* Γ.

Assume next we pick (v) *for* γ *in line 6. Now,* `PUSH` x_1 `PUSH` x_2 $= \ell_0\gamma \equiv r_0\gamma = \varepsilon$ *does not hold we remove* (v) *from* Γ *(line 11). Additionally, as* (v) $\lessdot$ (ii)*, we can also remove* (ii)*.*

Find a Context. As a second step We strip the generalised rules of any unnecessary pre- and postfix. Again we first capture all possible stripped rules and then give an implementation.

Definition 8.1.6 *The* stripped rules *of a rule* $\rho \Rrightarrow \tau$ *are defined as* $\mathsf{Con}(\rho \Rrightarrow \tau) = \{\ell \Rrightarrow r \mid \rho = C[\ell] \text{ and } \tau = C[r]\}$.

Example 8.1.3 *Continuing Example 8.1.1, for the rule* `PUSH` *0* `SUB` `PUSH` x `ADD` `SHA3` $\Rrightarrow$ `PUSH` x `SUB` `SHA3` *the stripped rules* Con *contain the rule* `PUSH` *0* `SUB` `PUSH` x `ADD` $\Rrightarrow$ `PUSH` x `SUB`*, obtained by stripping away the context* (ϵ, `SHA3`)*, and the original rule itself, since applying the empty context* (ϵ, ϵ) *to a program yields the program itself.*

Example 8.2.2 shows further rules stripped of their context in EVM bytecode.

To implement Con we follow the same strategy as for Gen: try all possible contexts in an exhaustive search, checking whether they yield a rule and use an order contexts to prune the search space.

Definition 8.1.7 *We define the function* strip *in Algorithm 12.*

Again, the order on contexts allows us to prune the search space without loss.

Lemma 8.1.2 *For every* $\ell \Rrightarrow r \in \mathsf{Con}(\alpha)$ *of a rule* α *there is a* $\ell' \Rrightarrow r' \in \mathsf{strip}(\alpha)$ *and a context* C *such that* $C[\ell'] = \ell$ *and* $C[r'] = r$.

Proof 8.1.2 *We fix a rule* $\ell \Rrightarrow r \in \mathsf{Con}(\alpha)$*. Let* (s_0, t_0) *be the longest common prefix and the longest common postfix of* α *and be* ℓ_0, r_0 *the program schemas with* $s_0 \cdot \ell_0 \cdot t_0 \Rrightarrow s_0 \cdot r_0 \cdot t_0 = \alpha$*. A context* C' *with* $C'[\ell_0] = \ell$ *and* $C'[r_0] = r$ *is in*

Algorithm 12: Strip context from the optimisation rule $\rho \Rrightarrow \tau$.

1 **function** $\mathsf{strip}(\rho \Rrightarrow \tau)$
2 $\quad \mathcal{R} \coloneqq \varnothing$
3 $\quad (s_0, t_0) \coloneqq$ the longest common prefix s_0 and the longest common postfix t_0 of ρ and τ
4 $\quad \ell_0, r_0 \coloneqq$ the program schemas ℓ_0 and r_0 with $s_0 \cdot \ell_0 \cdot t_0 = \rho$ and $s_0 \cdot r_0 \cdot t_0 = \tau$
5 $\quad \Gamma \coloneqq \{C \mid C = (s, t)$ where $s' \cdot s = s_0$ and $t \cdot t' = t_0$ for some $s', t'\}$
6 **forall** $C \in \Gamma$ **do**
7 $\quad$ **if** $C[\ell_0] \equiv C[r_0]$ **then**
8 $\quad\quad \mathcal{R} \coloneqq \mathcal{R} \cup \{C[\ell_0] \Rrightarrow C[r_0]\}$
9 $\quad\quad \Gamma \coloneqq \Gamma \setminus \{C' \mid C < C'\}$
10 $\quad$ **else**
11 $\quad\quad \Gamma \coloneqq \Gamma \setminus \{C' \mid C' < C\}$
12 $\quad$ **return** $\mathcal{R}$

Γ *and thus either* $\mathsf{strip}(\alpha)$ *will consider it at some point, or it will be removed by either line 9 or line 11.*

If it is considered then $\ell \Rrightarrow r$ *is in* $\mathsf{strip}(\alpha)$*. If it is removed by line 9, then a context* C *with* $C < C'$ *and and* $C[\ell_0] \equiv C[r_0]$ *was considered. Thus* $C[\ell_0] \Rrightarrow C[r_0]$ *is in* $\mathsf{strip}(\alpha)$ *and we have* $C''[C[\ell_0] = \ell$ *and* $C''[C[r_0]] = r$ *for some* C'' *by* $C < C'$*. If* C' *was removed by line 11 then a context* C *with* $C' < C$ *and and* $C[\ell_0] \not\equiv C[r_0]$ *was considered. Again this contradicts the assumption* $\ell \Rrightarrow r \in \mathsf{Con}(\alpha)$*, because observational equivalence is closed under context.*

Example 8.1.4 *Take the optimisation* `CALLVALUE DUP1 ADD` $\geqslant$ `CALLVALUE CALLVALUE ADD`*. Then,* s_0 *is* `CALLVALUE`*,* t_0 *is* `ADD` *(line 3), and* ℓ_0 = `DUP1` *and* r_0 = `CALLVALUE` *(line 4). The set* Γ *holds* $\{$ (i) *(*ε*,* ε*),* (ii) *(*`CALLVALUE`*,* ε*),* (iii) *(*ε*,* `ADD`*),* (iv) *(*`CALLVALUE`*,* `ADD`*)*$\}$*. Now, assuming we first pick* (ii) *for* C *in line 6. As* `CALLVALUE DUP1` $= C[\ell_0] \equiv C[r_0] =$ `CALLVALUE CALLVALUE`*, we add the rule to* $\mathcal{R}$ *(line 8). Then, we remove* (ii) *from* Γ *in line 9. Now, because* (ii) $<$ (iv) *holds,* i.e., (iv) *is more specific than* (ii) *we can also remove* (iv) *from* Γ*. Assume next we pick* (i) *for* C *in line 6. Now,* `DUP1` $= C[\ell_0] \equiv C[r_0] =$ `CALLVALUE` *does not hold we remove* (iii) *from* Γ *(line 11). Additionally, as*

(iii) < (i)*, we can also remove* (iii)*.*

Soundness and Completeness. Finally, we combine the two functions and for an optimisation $\rho \gtreqless \tau$ define $\mathsf{sorg}(\rho \gtreqless \tau) = \{\mathsf{strip}(\ell \Rrightarrow r) \mid \ell \Rrightarrow r \in \mathsf{generalise}(\rho \Rrightarrow \tau)\}$. The rules generated by $\mathsf{sorg}(\rho \gtreqless \tau)$ are *sound*: for every $\ell \Rrightarrow r \in \mathsf{sorg}(\rho \gtreqless \tau)$ there is a substitution γ and a context C such that $C[\ell\gamma] = \rho$ and $C[r\gamma] = \tau$. This directly follows from $\mathsf{generalise}(\rho \Rrightarrow \tau) \subseteq \mathsf{Gen}(\rho \Rrightarrow \tau)$ and $\mathsf{strip}(\rho \Rrightarrow \tau) \subseteq \mathsf{Con}(\rho \Rrightarrow \tau)$. The rules generated by $\mathsf{sorg}(\rho \gtreqless \tau)$ are also *complete*: for every $\ell \Rrightarrow r \in \mathcal{R}(\rho \gtreqless \tau)$ there is a $\ell' \Rrightarrow r' \in \mathsf{sorg}(\rho \gtreqless \tau)$, a substitution γ and a context C such that $C[\ell'\gamma] = \ell$ and $C[r'\gamma] = r$. This directly follows from Lemmas 8.1.1 and 8.1.2.

8.2 Case Study

To demonstrate the applicability of our pipeline from Figure 8.1 we implement it in the context of Ethereum for EVM bytecode.

Find Optimisations with ebso. We find optimisations using our tool ebso from Chapter 6 using unbounded superoptimisation. In the best case ebso produces a cheaper, observationally equivalent ebso block.

Generate Rules with sorg. To generate rules for EVM bytecode we implemented sorg, a superoptimisation based rule generator. Like ebso, sorg is implemented in OCaml; sorg depends on ebso for the representation of EVM bytecode and SMT encoding to check observational equivalence.

The main contribution of sorg is to provide notions of program schema, substitutions, and context in order to implement the two main procedures: generalise and strip. For generalise we implement the procedure from Definition 8.1.5, keeping only the most general rules in the result.

Example 8.2.1 *In our evaluation in Section 8.3, we found the following optimisation:*

```
SWAP1 POP PUSH 0 PUSH 1 MUL PUSH 0 ⋛ SWAP1 POP PUSH 0 DUP1
```

Generalizing immediate arguments and dropping the prefix `SWAP1 POP` *sorg yields two optimisation rules:* `PUSH` x `PUSH` *1* `MUL PUSH` $x \Rrightarrow$ `PUSH` x `DUP1` *as well as* `PUSH` *0* `PUSH` x `MUL PUSH` *0* $\Rrightarrow$ `PUSH` *0* `DUP1`*.*

For strip we implement the procedure from Definition 8.1.7, keeping only the most stripped rules.

Example 8.2.2 *From the rule* `CALLVALUE DUP1 POP` $\Rrightarrow$ `CALLVALUE CALLVALUE POP` *sorg can either strip the postfix* `POP` *or the prefix* `CALLVALUE`*, obtaining the rules* `CALLVALUE DUP1` $\Rrightarrow$ `CALLVALUE CALLVALUE` *and* `DUP1 POP` $\Rrightarrow$ `CALLVALUE POP`*.*

One main ingredient of both generalise and strip is a check for observational equivalence. To determine observational equivalence in sorg we use an SMT encoding with which we already used in Chapter 6 in Definition 6.2.1: two program schemas ρ and τ, we have $\rho \equiv \tau$ if there are no inputs that distinguish them. With sorg we can now automatically generate rules, but it remains to glue the tools together and implement a feedback mechanism.

Coordinate with ppltr. To coordinate our tools ebso and sorg we implemented the tool ppltr, a populator for a peephole optimiser. As ebso and sorg, ppltr is implemented in OCaml. The tool has two main tasks. The first is to manage the interfaces, *i.e.*, to generate ebso blocks from smart contracts, generate ebso blocks for a given size k, prepare optimisations generated by ebso as input for sorg, and analyse and de-duplicate a set of rules produced by sorg. The second main task is to feed back the optimisation rules, *i.e.*, to rewrite right-hand sides of the optimisation rules themselves, and apply the optimisation rules to ebso blocks. To achieve the latter task, ppltr implements a rewrite engine.

8.3 Evaluation

We evaluate our pipeline by generating optimisation rules for EVM bytecode. We collected the 250 most called smart contracts until block 9 786 000 at Apr-01-2020 12:17:26 PM +UTC from the Ethereum blockchain using Google Big-

Table 8.1: Accumulated savings when applying the rules in $\mathcal{R}_2$ on most called contracts.

	acc. gas savings	acc. length savings	
250 most called contracts	106 811 g	35 699 instructions	3.94 %
1000 most called contracts	435 002 g	146 376 instructions	4.58 %

Query[3]. We split the 250 contracts into 106 798 ebso blocks $\mathcal{E}$. As peephole optimisation rules typically span only few instructions, we restrict the size of a block: using a sliding window we split every block larger than 6 instructions into k blocks of at most 6 instructions. To reduce the noise, we remove blocks which are only different in the arguments of `PUSH` keeping only those with words of size smaller than 5 bit. We so obtain 54 301 ebso blocks. Using ebso find 1580 optimisations from these blocks, run on a cluster with Intel Xeon Gold 6126 CPUs at 2.60 GHz, 2 GB of memory and a time-out of 15 min. From these optimisations, we generate 1525 rules with sorg, run on the same set-up. For 48 optimisations sorg timed out and could not generate rules and we removed roughly half the rules, as they were duplicates generated from different optimisations. Thus we arrive at 758 rules $\mathcal{R}_0$, which we use with the rewrite engine of ppltr to *(a)* rewrite the right-hand sides of $\mathcal{R}_0$ reducing 4 rules, and *(b)* rewrite our original ebso blocks in $\mathcal{E}$, which changed 17 255 ebso blocks.

We again use the same window-size and noise reduction to get 25 585 new ebso blocks. Going through the same procedure, we find 452 optimisations with ebso, and generate 435 rules $\mathcal{R}_1$ with sorg with 16 time-outs. Combining the results we get 993 rules $\mathcal{R}_2 = \mathcal{R}_0 \cup \mathcal{R}_1$ which are available at

`github.com/mariaschett/ppltr/blob/v1.0/eval/17-reduced-rules.csv`

We right-reduced 31 rules in $\mathcal{R}_2$ and discarded 967 replicated rules originating from different optimisations. One optimisation generated two rules (*cf.* Example 8.2.1).

To estimate gas and size saving on a contract level we apply the rules in $\mathcal{R}_2$

[3] `cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-\smart-contract-analytics`.

to 1. our original 250 most called smart contracts, and 2. extend the data set to the 1000 most called contracts. Table 8.1 shows our results. The first column shows the accumulated gas savings over all contracts, and the second column shows the accumulated length savings. Note that results depend on the order in which the rules are applied. First, we can observe that the rules translate well from 250 to 1000 contracts, achieving roughly 4 times higher savings, which demonstrates that $\mathcal{R}_2$ also extends beyond the original data set, from which it was generated. Now let us consider the gas savings. In Table 8.1 we accumulate the cost of all the removed instructions for each contract. How much is actually saved, however, depends on how often the contract is called and which parts are executed. Unfortunately we lack the resources to replay all the transactions to determine the exact savings. Taking into account how often a contract was called, we save 7.41×10^{10} `g` for the former and 1.02×10^{11} `g` for the latter. Assuming that about 10 % of a contract is executed per call and that savings are uniformly distributed, this translates to 41 049.33 \$ and 56 505.15 \$ for a gas price of 27.6 gwei and an ETH-USD course of 200.62 \$, which are averages from `etherscan.io/charts`.

While the cost of executing a cheap instruction like `ADD` or `POP` may be negligible, the cost of storing that instruction may not be so. Therefore, we also look at the savings in length: the overall storage space of the bytecode reduces by more than 4.5 %. The contract with the highest length saving was reduced by 19.94 %, removing 345 from originally 1730 instructions.

We also analyse which rules are applied to the contracts. Applying rules may lead to the applicability of other rules, but exploring all rewrite sequences is intractable, and we assume that initial applicability on a contract is a reasonable proxy. Figure 8.2 groups rules in $\mathcal{R}_2$ by their applicability to the 1000 most called contracts. We can observe a long tail: more than half of the nearly $1k$ rules are applicable only 10 times or less, whereas the top 50 rules are applicable more than 500 times. This suggests that, if a smaller set of rules is desired, this analysis can guide which rules to discard.

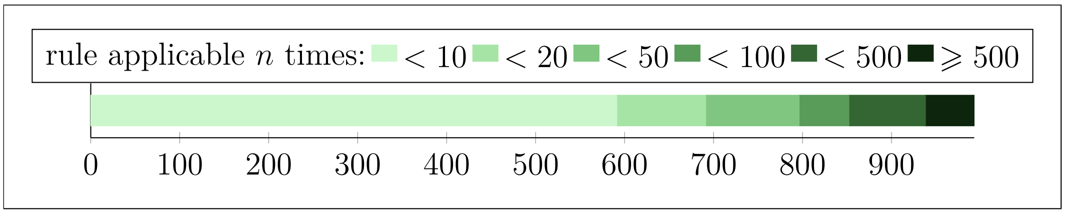

Figure 8.2: Applicability of rules in $\mathcal{R}_2$ to 1000 most called contracts.

1. `SWAP1 POP POP` $\Rrightarrow$ `POP POP` (×8926)
2. `ISZERO ISZERO ISZERO` $\Rrightarrow$ `ISZERO` (×7893)
3. `PUSH` y `PUSH` x `SWAP1` $\Rrightarrow$ `PUSH` x `PUSH` y (×7742)
4. `CALLVALUE DUP1` $\Rrightarrow$ `CALLVALUE CALLVALUE` (×7740)
5. `SWAP1 SLOAD SWAP1 PUSH` x `EXP SWAP1` $\Rrightarrow$ `PUSH` x `EXP SWAP1 SLOAD` (×5625)

Figure 8.3: Rules most applied to the 1000 most called contracts.

Next we inspect the rules within $\mathcal{R}_2$. The five most applied rules for the 1000 most called contracts are listed in Figure 8.3. Most of these rules are relatively simple and should clearly be applied exhaustively. The fourth rule is perhaps a bit unexpected and may have been missed on manual inspection, but it is cheaper to execute `CALLVALUE` twice than duplicating its result. The last rule hints at a specific compiler produced anti-pattern. Our approach could also be leveraged to detect those.

Figure 8.4 shows the six rules with the highest gas savings, 17 `g` and 15 `g`. We consider two of these rules in more detail. The rule `PUSH` 1 `MUL PUSH` 0 `NOT AND` $\Rrightarrow \varepsilon$ combines two observations—that 1 and `PUSH` 0 `NOT` are neutral elements for multiplication and `AND` respectively. Depending on the implementation of the peephole optimiser it may be desirable to split this rule which could be achieved by left-reducing the rules. Key to the rule `PUSH` 0 `DUP6 DUP5 SUB LT ISZERO` $\Rrightarrow$ `PUSH` 1 is the less-than comparison `LT` with the smallest element 0 always evaluating to false. The rule does not depend on the result of `DUP6 DUP5 SUB`, and indeed this is replaced by `DUP2 PUSH` x `AND` in the otherwise identical rule in the last line. Generalising those two rules would require the use of higher-order patterns.

1. PUSH 1 MUL DUP3 PUSH 0 NOT AND $\Rrightarrow$ DUP3
   PUSH 1 MUL PUSH 0 NOT AND $\Rrightarrow$ ε
2. PUSH 0 DUP6 DUP5 SUB LT ISZERO $\Rrightarrow$ PUSH 1
   PUSH 0 NOT AND EQ ISZERO ISZERO $\Rrightarrow$ EQ
   SWAP1 PUSH 0 NOT AND SWAP1 $\Rrightarrow$ ε
   PUSH 0 DUP2 PUSH x AND LT ISZERO $\Rrightarrow$ PUSH 1

Figure 8.4: Rules saving most gas.

Table 8.2: Added (+) and removed (−) instructions by group.

	arith.	comp.	`ISZERO`	bitwise	`DUP`i	`SWAP`i	`PUSH`	`POP`	env./mem.
(+)	10	27	24	12	47	28	134	14	29
(−)	80	92	108	83	345	952	182	173	18

Rules may not only save gas, but also reduce the length of the produced code. These often coincide, and indeed the top 14 length-reducing rules, removing 5 instructions each, subsume the above gas-saving rules. On the other end, there are also rules which save gas but do not reduce the length such as `CALLVALUE DUP1` $\Rrightarrow$ `CALLVALUE CALLVALUE` saving 1 g. In Table 8.2, we analyse the right-hand sides of $\mathcal{R}_2$. We investigated which instructions were *added* (+), *i.e.*, do not appear on the left-hand side, and *removed* (−), *i.e.*, appear on the left- but not the right-hand side of the rule. We group instructions for arithmetic, comparison, bitwise operations, and environment/memory. Unsurprisingly, many more instructions were removed than added, which is expected, because removing instructions always saves gas. The majority of removed instructions is concerned with the stack layout. Surprisingly, also `ISZERO` is often redundant—as also observed in the second rule in Figure 8.3. Still, instructions are also synthesized on the right-hand side giving rise to optimisations taking the semantic of an instructions into account—potentially also interacting with stack manipulation, for example the rule `SWAP1 LT` $\Rrightarrow$ `GT`.

Finally, we also successfully validated all rules $\mathcal{R}_2$ by running a reference implementation of the EVM, `go-ethereum` version `1.9.14` on pseudo-random input.[4] Therefore, we run the bytecode of every block in $\mathcal{E}$ and the bytecode obtained by applying the rewrite rules to observe that both produce the same final state.

[4] `github.com/ethereum/go-ethereum`

Chapter 9

Related Work

In this chapter I relate Part I and Part II of this thesis to other results related to blockchain protocols and programs.

9.1 Blockchain Protocols

For Part I we investigate the genealogy of our results, related protocols, works related to the Stellar consensus protocol, and compare threshold logical clocks with block DAGs.

Genealogy. The basis for Chapter 3 is the Stellar protocol from the white paper by Mazières [75]. Another building block is García-Pérez *et al.* [42] investigating Stellar's federated voting and its relationship to Bracha's broadcast over classical Byzantine quorum systems. However, they did not address the full Stellar consensus protocol. The basis for Chapter 4—apart from the block DAG works Hashgraph [10], Blockmania [30], Aleph [41], and Flare [96]—is the idea to leverage deterministic state machines to replay the behaviour of other servers, which goes back to PeerReview [51]. There, servers exchange logs of received messages for auditing to eventually detect and expose faulty behaviour. This idea was taken up by block DAG approaches—but with the twist to leverage determinism to *not* send those messages that can be determined. This allows compressing messages to the extent of only indicating that a message has been sent as we do in Chapter 4.

Related Protocols. For both Chapters 3 and Chapter 4 there are closely related concrete protocols. Close to Stellar is Ripple [103] also relying on mutual trust, and the follow-up protocol called Cobalt that allows for a federated setting [71]. Close to our block DAG framework are Hashgraph [10], Blockmania [30], Aleph [41], and Flare [96]. Underlying all of these systems is the same idea: first, build a common block DAG, and then locally interpret the blocks and graph structure as communication for some protocol: Hashgraph encodes a consensus protocol in a block DAG structure, Blockmania [30] encodes a simplified version of PBFT [25], Aleph [41] employs atomic broadcast and consensus, and Flare [96] builds on federated byzantine agreement from Stellar combined with block DAGs to implement a federated voting protocol. Naturally, the correctness arguments of these systems focus on their system, *e.g.*, the correctness proof in Coq of byzantine consensus in Hashgraph [29]. In our work, we aim for a different level of generality: we establish structure underlying protocols which employ block DAGs. To that end, and opposed to previous approaches, we treat the protocol interpreted on the block DAG completely as a black-box, *i.e.*, our framework is parametric in this protocol. While our work focuses on correctness, two recent works show that DAG-based approaches for concrete protocols are efficient and even optimal: DAG-Rider [57] implements the asynchronous byzantine atomic broadcast abstraction and is shown to be optimal with respect to resilience, amortized communication complexity, and time. Different to our work, DAG-Rider relies on randomness, which is an extension in our setting. Also Narwhal and Tusk [31] for BFT consensus reports impressive—also empirically evaluated—performance gains. Moreover, as argued in [31], our approach enjoys two further benefits for implementations: load balancing, as we do not rely on a single leader, and equal message size.

On Stellar. Lokhava *et al.* [66] describe the whole Stellar eco-system, not only the consensus protocol—including implementation, empirical evaluation, and deployment and even provide some formal verification. Losa *et al.* [68] prove safety and liveness of Stellar under partial synchronicity in Isabelle/HOL and Ivy. One key point in Stellar is the idea to build quorums based on trust. This is also the key point in the following works [69, 38]. It is orthogonal to

Chapter 3, where we take the quorum as given and focus on the protocol. Losa *et al.* [69] propose a generalisation of Stellar's quorums that does not prescribe constructing them from slices, yet allows different participants to disagree on what constitutes a quorum. They then propose a protocol solving consensus over intact sets in this setting that provides better liveness guarantees than the protocol in [75], but is impractical. Florian *et al.* [38] reason about the FBQS underlying the Stellar consensus protocol and give formal definitions for safety and liveness guarantees based on notions of minimal quorums, minimal blocking sets, and minimal splitting sets. The authors give some algorithms and a tool to compute the quorums.

Threshold Logical Clocks. A recently proposed work related to our block DAG framework in Chapter 4 is the threshold logical clock abstraction [39], which allows a higher-level protocol to operate on an asynchronous network as if it were a synchronous network. They do so by defining an abstraction of the communication at the level of groups. Similar to our framework, also threshold clocks rely on causal relations between messages by including a threshold number of messages for the next time step. In our setting, this would roughly correspond to including a threshold number of predecessor blocks for every block. In contrast, our framework, by only providing the abstraction of a reliable point-to-point link to $\mathcal{P}$, pushes reasoning about messages to $\mathcal{P}$.

9.2 Blockchain Programs

The work in Part II relates to the following major fields: superoptimisation, compiler optimisations based on SMT solvers, and analysis of smart contracts.

Superoptimisation. Our work relies heavily on the advances made to push enumeration and search into a SAT or SMT solver. Joshi *et al.* [56] leverage a SAT solver to encode superoptimisation. Gulwani *et al.* [50] introduce templates to leverage a solver to synthesise a function implementing a specification relating desired input and output. Most importantly, Jangda *et al.* [55] introduce *unbounded superoptimisation* giving an encoding to shift the search for an optimal program to the SMT solver. This encoding is the basis for our work in Chapter 6. To our knowledge, our approach is the first application of superoptimisation to smart contracts, but superoptimisation has been used in other domains. Most notably, the tool Souper [97] is a superoptimiser for LLVM [65]. Similar to our rewrite and simplification rules in Chapter 7 and Chapter 8, Souper caches common optimisation patterns. Mukherjee *et al.* [82] extend Souper by heuristics to prune the search space to reduce calls to the SMT solver to check the equivalence between a candidate program and the original program. In our approach, we circumvent this by pushing the enumeration into the SMT solver. Similar to our approach in Chapter 8, Bansal *et al.* [12] use superoptimisation to automatically generate a peephole optimiser for x86 binaries. However, they do not generalize optimisations into rules but instead keep them in an optimisation database in order to reapply them. Moreover it uses an enumeration based superoptimiser, which is more exhaustive, but limits instruction sequences to length 3. We picked our window size in ppltr to be 5 and similar to Phothilimthana *et al.* [91] also use a sliding window. There are several works on superoptimisation where ideas could be explored in our context: Sharma *et al.* [104] find optimisations that hold under certain conditions *i.e.* in certain contexts, such as for some fixed input. They synthesise non-trivial and useful conditions for x86 from test cases. The tool TOAST [18] superoptimises machine code using *Answer Set Programming* [20] instead of SAT or SMT solvers. Phothilimthana *et al.* [91] combine three search heuristics for finding a cheaper program (enumerative, SAT-solver based, stochastic) and view superoptimisation as a graph search problem.

Optimisations through SMT. In a more general setting, we next look at compiler optimisations through SMT solvers. Alive [67] is a framework to specify peephole optimisation rules for LLVM in the Alive domain specific language (DSL), to then verify their correctness with an SMT solver using the theory of bit-vectors—and extensions *e.g.* floating points [78]. Alive also exploits context information about the input such as `isPowerOfTwo()` or `cannotOverflow()`. Finally, Alive can generate C++ code for the peephole optimisation rules to use within LLVM.

Similar to Alive, in Chapter 6–7, we rely on an SMT solver to verify correctness of an optimisation in our approach—but we also rely on the SMT solver to find them in the first place. However, we do not exploit context information as Alive does, which would make for interesting future work. As the EVM does not operate on floating point numbers, neither do our tools. Especially for ppltr in Chapter 8, the pipeline from Alive is alluring: the Alive DSL may be a guidance for a specification of peephole optimisation rules—rather than the *ad hoc* specification in ppltr—and code generation for the peephole optimiser of *e.g.*, the Solidity compiler solc would ease adoption. Also OptGen [21] automatically generates local optimisation rules with unary and binary integer operations such as $\neg$, &, + for 8 bit values by enumerating (isomorphic) terms with up to 2 operators on the left and right hand side of a rule and checking equivalence with an SMT solver. OptGen generates "symbolic constants" c_1, $c_2, \ldots$ and eval_plus, to find a rule such as $c_1 + c_2 \rightarrow \mathsf{eval\_plus}(c_1, c_2)$. OptGen also generates rules like $x \mathbin{\&} 0 \rightarrow 0$ with enumerating x which can also be a term, not only a constant, and OptGen can also suggest conditional rules such as $c_1 \mathbin{\&} c_2 = 0 \implies (x \mid c_1) \mathbin{\&} c_2 \rightarrow x \mathbin{\&} c_2$. The enumeration approach is similar to basic superoptimisation in Chapter 6 and templates [50]. OptGen only operates on 8 bit values and does not seem to lift this restriction, which we do by translation validation in Chapter 6. Similar to Alive, OptGen uses context information by expressing conditional rules, which seems a promising area of further work. In our rules in Chapter 8, we currently only have constants c_i, and not as OptGen, variables for terms x. It might be interesting to overcome this in ppltr, but it may require to go towards higher-order rules.

Smart Contract Analysis. In recent years, several tools for analysis of smart contracts were developed. Oyente [70] uses control flow analysis in order to detect security defects such as reentrancy bugs. The tool Gastap [6] provides an upper bound on gas consumption of a smart contract by combining static analysis tools. More recently, tools are looking at optimising smart contracts.

Chen *et al.* [26] identified 7 expensive patterns on Solidity contracts with respect to *(i)* useless code (dead code, opaque predicates), and *(ii)* loops (*e.g.* expensive operations in loops). Their tool Gasper rewrites these expensive patterns. By manual inspection from nearly 300k snippets with window size 1-5, Chen *et al.* [27] identified 24 anti patterns, such as `OP POP` optimises to the `POP` instruction. Their tool GasReducer applies anti-patterns to EVM bytecode. Our tool syrup in Chapter 7 subsumes 21 anti-patterns concerning stack layout and commutativity. These enumerated anti-patterns show how difficult it is to capture all the interleaving concerning stack layout. We avoid this, by leveraging the SMT solver. We also capture the anti-pattern `OP ISZERO ISZERO` to `OP` with `OP` one of `LT`, `GT`, `SLT`, `SGT`, `EQ` as part of our simplification rules in syrup. Two anti-patterns in [27] we cannot support in our approach are the collapsing multiple `JUMPDEST` to one `JUMPDEST` and `OP STOP` to `STOP` for `OP` not a jump instruction.

The system Gasol [2] also incorporates an automatic optimisation for storage operations that consists of replacing accesses to the storage (`SSTORE` and `SLOAD`) by equivalent accesses to memory locations (`MSTORE` and `MLOAD`), when a static analysis identifies that it is sound and efficient doing such transformations. Brandstaetter *et al.* [19] analyse the applicability of "optimisation strategies" from software engineering on 3k Solidity smart contracts. Their optimisation strategies include ideas like loop unrolling, parallel computation, reordering tests, or exploiting algebraic identities. Finally, recent work analysed the alignment between gas cost and actual execution costs. Yang *et al.* [114] experimentally prove that the gas model for some EVM instructions is not correctly aligned with respect to the observed computational costs in real experiments. Perez *et al.* [90] use this misalignment in gas to show that this can lead to gas-related attacks. However, our work is parametric in the gas model used, and new adjustments in the gas model of Ethereum are integrated by just updating the cost for the corresponding modified instructions in our implementation.

Chapter 10

Conclusion

To reiterate the research hypothesis [H]: *by applying formal reasoning to blockchain technologies we can reduce execution costs while guaranteeing correctness.* In my thesis I provide two case studies as evidence towards [H]: for *blockchain protocols* in Part I and for *blockchain programs* in Part II.

10.1 Summary

In Part I I provide evidence towards the sub-hypothesis [H_a]: *by applying formal reasoning to communication protocols we can reduce the number of exchanged messages while guaranteeing correctness.* This was achieved by compressing messages. The basis for Chapter 3 is the Stellar consensus protocol [75]. We first define an abstract—but simpler—version of the Stellar consensus protocol. In the abstract protocol we use federated voting [75], which is known to be a reliable byzantine broadcast [42], as a black-box. We then prove that the properties of (weak) byzantine consensus hold. However, the abstract protocol relies on sending infinitely many messages. To improve this, we propose a more realistic concrete consensus protocol compressing the infinitely many messages to a finite number of messages. We then show that the concrete protocol refines the abstract protocol and thus the properties of (weak) byzantine consensus hold. In Chapter 4 we compress messages by two means: first, by not sending messages which can be inferred due to determinism of the protocol, and second by batching the execution of multiple parallel instances of a protocol. We give a generic formalization of a block DAG and its properties and show that a block DAG is an implementation of a reliable point-to-point channel, which can be used to implement any deterministic BFT protocol $\mathcal{P}$ efficiently. Hereby, messages emitted by $\mathcal{P}$, which are the results of the deterministic execution of $\mathcal{P}$, can be omitted. At the same time, multiple parallel instances of $\mathcal{P}$ using the same block DAG are executed essentially 'for free'. Our main result is that using the block DAG framework for a deterministic BFT protocol $\mathcal{P}$ maintains its interfaces, and safety and liveness properties.

In Part II I give evidence towards my sub-hypothesis [H_b]: *by applying formal reasoning to smart contracts we can reduce the monetary fees of their execution while guaranteeing correctness.* We reduce monetary fees by optimising gas consumption of EVM bytecode in basic blocks, *i.e.*, EVM bytecode within a node in the control flow graph of a smart contract. We start by modelling the EVM state and superoptimisation for EVM bytecode as an SMT *satisfiability* problem, based on the encoding of unbounded superoptimisation [55] in Chapter 6, to automatically find optimised bytecode. We then looked at superoptimisation for EVM bytecode as a synthesis and an SMT *optimisation* problem in Chapter 7. We improve the performance of our first approach by using symbolic execution to generate a stack functional specification to solve the SMT optimisation problem efficiently as a synthesis problem, and not encoding the semantics of the bit-vector operations of the EVM in the SMT problem. This allows us to express the problem using only existential quantification. Orthogonally in Chapter 8, we generalize the optimisations found in Chapter 6 to optimisation rules to populate the peephole optimiser of a smart contract compiler. We implemented three prototypes: a superoptimiser for EVM bytecode ebso, a synthesizer of super-optimised smart contracts syrup, and a technique for populating an EVM bytecode peephole optimiser ppltr. The prototypes are available on www.github.com/mariaschett[1] under the Apache-2.0 license. We evaluated our work on large-scale, real-world data sets from

[1]Side remark: the prototypes have been forked 12 times and together have 60+ stars as of July 23, 2021.

the Ethereum blockchain. In our first evaluation in Chapter 6 we found that relying on the heavily optimised search heuristics of a modern SMT solver is a feasible approach—albeit still having performance challenges, *e.g.*, timing out on 92.12 % of the blocks. Tackling the performance challenges in Chapter 7 in our next evaluation we found a suitable trade-off between expressiveness and performance. In our final evaluation in Chapter 8 we automatically generated 993 peephole optimisation rules from the 250 most called contracts and applied them to the 1000 most called contracts. Applying the rules allowed us to discard more than 145k superfluous instructions, saving more than 43 000 g and 4.5 % storage space.

10.2 Critical Discussion

Validity of Protocols. An open challenge for the work on protocols is validation: how to validate that the specification in Chapter 3 corresponds to the Stellar protocol? How to validate a run of $\mathcal{P}$ in the block DAG framework in Chapter 4? There are different strategies to validation depending on the use case. For one, we could *empirically evaluate* and implement our specification and test it against a reference implementation[2]. Similarly, for the block DAG framework we could implement protocol $\mathcal{P}$ and the framework and empirically evaluate them. However, also this approach comes with several challenges such as defining the source of truth: the specification or the implementation? Additionally, our implementation of the specification may not correspond to the specification. Finally, it remains to be determined, what exactly we want to compare in the evaluation: given that Stellar is heavily optimised, it may be much faster than our implementation, so we definitely would require some abstraction over timing. Another way of validating our specifications is through *manual inspection*—preferably by the protocol designers. Drawbacks are that these are laborsome, but certainly flexible and able to capture intuition. Several works have addressed the gap between specification and implementation by *extracting a formalised implementation* of a protocol, such as Velisarios PBFT

[2] *e.g.*, github.com/stellar/stellar-core/tree/master/src/scp

in Coq extracting verified code [93] or Raft in Coq [113]. This approach is expensive and offers no guarantees concerning the performance of the extracted code. Still, this could be combined with empirical evaluations for very high assurance. Finally, in the last year another formal specifications of Stellar was developed independently [68]. If one would show that the two *specifications are equivalent*, they would strengthen each other, thereby making a good case for validity.

Validity of Programs. Similar to the question for protocols is the question for programs: how to validate that our found programs and our model actually correspond to the EVM specification. In Chapter 6 we validated every optimisation by comparing a run of the original and the optimised program with pseudo-random input on a reference implementation of the EVM (*cf.* Section 6.3). A downside to this approach is that we cannot consider every input. However, we are convinced that if an instruction would have modified the part of the EVM state which we did not model, this would have been found by this approach. Clearly, also the question remains, how to be certain that the implementation adheres to the specification of the EVM. Another possibility is to run the test cases of the smart contracts and run compliance tests. This would require non-trivial engineering work, as we are currently not re-building the optimisations in the smart contracts. We validated our encodings of the instructions by manual inspection. Fortunately the encodings of the instructions are relatively small, self-contained, and correspond well to the definitions in the EVM specification. Finally, as sketched in Section 6.2, one could formally proof correctness of the optimisations with a formalisation of the EVM in a proof assistant. This would also be suitable for integration in verified compilers with correctness guarantees: they come with proofs of correctness. Indeed, I have integrated part of the peephole optimisation rules from Chapter 8 in a verified compiler compiling to EVM bytecode.

10.3 Outlook

In this final part I outline several ways to build on the results in my thesis.

The idea of message compression in Part I could be *generalised*. Here we believe the idea could be transferred in both directions: several messages are compressed into one message, or, a message is decompressed into several messages. For the first direction, compressing messages, this could be similar to Chapter 3, where one message triggers several actions, such as PREPb aborting every below-and-incompatible ballot. Similarly, in Chapter 4 one edge in the block DAG has several meanings essentially enabling parallelism 'for free'. The other direction, decompressing messages, can either facilitate easier proofs as in Chapter 3, or can be used for simulation such as in Chapter 4. In Part I, we give *modular definitions with clearly defined interfaces*. Our approach in Chapter 3 a simpler, but unrealistic, protocol with proof of correctness refining a more realistic implementation can serve as a blueprint for decomposing other protocols. For the block DAG framework in Chapter 4 future work could try different modules for *e.g.*, gossip in Algorithm 6. Similarly, the work could be extended with different high-level protocols $\mathcal{P}$—most notably by moving from interpreting a deterministic to a non-deterministic protocol $\mathcal{P}$. Then some care needs to be applied around the security properties assumed from randomness. If randomness is at the discretion of a server, the server can share the result by writing it in its next block. For unbiased randomness, one could use the shared coin protocol from Kokoriskogias *et al.* [59], secure under BFT assumptions and in a synchronous network.

While a formal paper proof of correctness gives high assurance, higher assurance is provided by a *mechanised proof in a proof assistant*, which also enables extracting a provably correct implementation. Indeed, in recent years many authors used proof assistants to proof correctness of protocols: Rahli *et al.* gave a safety proof of PBFT in Coq [93]. Woos *et al.* show the correctness of Raft in Coq [113]. Crary gave a correctness proof in Coq of byzantine consensus in Hashgraph [29]. Alturki *et al.* gave a Coq proof of asynchronous safety in Algorand. Casper has been shown correct in Coq [89] and in Isabelle/HOL [86]. IronFleet uses Dafny for showing safety and liveness of crash-tolerant Multi-

Paxos [52]. Moreover, safety and liveness under partial synchronicity of Stellar have been shown in Isabelle/HOL and Ivy in [68]. So future work could be to mechanize our proofs. Especially, for the block DAG framework as a core network abstraction the high level of assurance of mechanised proofs is certainly desirable. Moreover, it would ease the checking of optimisations in future work. In both, Chapter 3 and Chapter 4, we do not consider that correct servers can *crash and recover*—which is relevant for real world applications. Especially the block DAG approach seems to be well suited: it allows servers that recover to re-synchronise the block DAG, and continue execution—assuming that the remaining servers stored all the information persistently. This has a caveat: unless there is a mechanism for the higher level protocol to signal that some information will never again be needed, the full block DAG has to be stored by all correct parties forever. This seems to be a limitation of both our abstraction of block DAG but also the traditional abstraction of reliable point-to-point channels and the protocols using them. The latter seem to not require protocols to ever signal that a message is not needed any more (to stop re-transmission attempts to crashed or byzantine servers). Fixing this issue, and proving that protocols can be embedded into a block DAG, that can be operated and interpreted using a bounded amount of memory to avoid exhaustion attacks, is a challenging and worthy future avenue for work. Another open question is changes of the servers maintaining the protocol, *i.e.*, *reconfiguration*. Some work has been done on different views on the system in Stellar in [44], and also in [69]. Supporting reconfiguration of servers in block DAG protocols seems to be an open issue, besides splitting protocol instances in pre-defined epochs.

In Part II our approach is tailored towards new, rapidly evolving languages and their compilers with clear cost models such as gas metering. Thus we believe it should readily *generalise for other bytecodes of other smart contract languages* such as Move [107] and Michelson [61]. Facebook's Move is a gas-metered and verification-friendly designed language. The machine model of Move is stack-based with typed locals. To adapt the presented approach, the

SMT encodings in Chapter 6 and Chapter 7 would need to be extended to incorporate types and locals. Michelson, the smart contract language for the Tezos blockchain, also comes with a detailed formal semantics. Like the EVM it is a stack-based language, but features high-level data types, like lists, sets, and maps. To use the presented approach these data types need to be handled in the SMT encoding and SMT solvers do support complex theories such as sets and lists. Moreover, type information could be used to prune the search space, resulting in a positive performance impact.

Further future work is to *extend the coverage of EVM bytecode*. With the new PETERSBURG VERSION 3E2C089 of the EVM yellow paper [112], new instructions are available, such as the addition of shift-operators to the EVM. A second major point is the extension to cover EVM bytecode related to the EVM's memory and storage. In Chapter 6 and Chapter 8 we do not optimize instructions related to the semantics of the EVM's memory. Conceptually this would be a straightforward extension similar to storage. However, as the number of universally quantified variables and size of blocks are already posing challenges for performance, we believe that performance improvements are more important first. We explored performance improvements via the encoding in Chapter 7. Another avenue would be to improve the solvers themselves. To facilitate efforts in this direction we contributed benchmarks generated by ebso to the SMT community[3].

Similarly in Chapter 7 we do not optimize instructions related to storage and the memory. Again, the same methodology we have formalized for the stack could be extended to optimize the memory and storage bytecode operations. Finally, future work is the *integration into a compiler*. Two ideas are to *(i)* discover optimizations *ad hoc* throughout compilation, and *(ii)* apply optimization rules/peephole optimizations. For finding *ad hoc* optimizations, our work in Chapter 7 seems most promising. A next step would be a careful investigation of performance trade-offs between compile time and optimizations—an

[3] *cf.* clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks-tmp/benchmarks-pending/-/commit/93ba6a5e76c5b850bde8b83ed16a91dc1e64db81.

avenue we have explored in [3]. To automatically integrate the rules generated by pplτr into a compiler a domain-specific language like the one used by gcc[4] or Alive [67] might prove useful.

The two ideas could even inform each other: in Chapter 7 we do not encode the semantics of bit-vector instructions, and instead employ hand-crafted simplification rules, which could be inspired by, or even automatically derived from, rules generated by pplτr in Chapter 8.

[4] `gcc.gnu.org/onlinedocs/gccint/The-Language.html`

Bibliography

[1] Alexander Aiken. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices*, 31(7):19–24, 1996. `doi:10.1145/381841.381847`.

[2] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 118–125, Cham, 2020. Springer International Publishing. `doi:10.1007/978-3-030-45237-7_7`.

[3] Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, Albert Rubio, and Maria A. Schett. Super-Optimization of Smart Contracts. *Under Submission*, 2021.

[4] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 513–520, Cham, 2018. Springer International Publishing. `doi:10.1007/978-3-030-01090-4_30`.

[5] Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A. Schett. Synthesis of Super-Optimized Smart Contracts Using Max-SMT. In *Computer Aided Verification*, volume 12224 of *Lecture Notes in Computer Science*, pages 177–200, Cham, 2020. Springer International Publishing. `doi:10.1007/978-3-030-53288-8_10`.

[6] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Running on Fumes: Preventing Out-of-Gas Vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis. In *Verification and Evaluation of Computer and Communication Systems*, volume 11847 of *Lecture Notes in Computer Science*, pages 63–78, Cham, 2019. Springer International Publishing. `doi:10.1007/978-3-030-35092-5_5`.

[7] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987. `doi:10.1007/BF01782772`.

[8] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77, New York, NY, USA, 2018. ACM. `doi:10.1145/3167084`.

[9] Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money Transfer Made Simple. *arXiv:2006.12276 [cs]*, 2020. `arXiv:2006.12276`.

[10] Leemon Baird. The Swirlds Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance. Technical Report SWIRLDS-TR-2016-01, Swirlds, Inc, 2016.

[11] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. SoK: Consensus in the Age of Blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 183–198, New York, NY, USA, 2019. ACM. `doi:10.1145/3318041.3355458`.

[12] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 394–403, New York, NY, USA, 2006. ACM. `doi:10.1145/1168857.1168906`.

[13] Clark Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 20–23, Berlin, Heidelberg, 2005. Springer. doi:10.1007/11513988_4.

[14] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.

[15] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-22110-1_14.

[16] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The Barcelogic SMT Solver. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 294–298, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-70545-1_27.

[17] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

[18] Martin Brain, Tom Crick, Marina De Vos, and John Fitch. TOAST: Applying Answer Set Programming to Superoptimisation. In *Logic Programming*, Lecture Notes in Computer Science, pages 270–284, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11799573_21.

[19] Tamara Brandstätter, Stefan Schulte, Jürgen Cito, and Michael Borkowski. Characterizing Efficiency Optimizations in Solidity Smart Contracts. In *Proceedings of the 2020 IEEE International Confer-*

ence on Blockchain (Blockchain), pages 281–290. IEEE, 2020. `doi:10.1109/Blockchain50366.2020.00042`.

[20] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011. `doi:10.1145/2043174.2043195`.

[21] Sebastian Buchwald. Optgen: A Generator for Local Optimizations. In *Compiler Construction*, Lecture Notes in Computer Science, pages 171–189, Berlin, Heidelberg, 2015. Springer. `doi:10.1007/978-3-662-46663-6_9`.

[22] Vitalik Buterin. Ethereum Whitepaper: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper, 2013.

[23] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, Berlin Heidelberg, second edition, 2011.

[24] Christian Cachin and Marko Vukolic. Blockchain Consensus Protocols in the Wild. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.DISC.2017.1`.

[25] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186, USA, 1999. USENIX Association.

[26] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Underoptimized smart contracts devour your money. In *Proceedings of the*

2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 442–446. IEEE, 2017. doi:10.1109/SANER.2017.7884650.

[27] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards Saving Money in Using Smart Contracts. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 81–84, New York, NY, USA, 2018. ACM. doi:10.1145/3183399.3183420.

[28] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 93–107, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-36742-7_7.

[29] Karl Crary. Verifying the Hashgraph Consensus Algorithm. *arXiv:2102.01167 [cs]*, 2021. arXiv:2102.01167.

[30] George Danezis and David Hrycyszyn. Blockmania: From Block DAGs to Consensus. *arXiv:1809.01620 [cs]*, 2018. arXiv:1809.01620.

[31] George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. *arXiv:2105.11827 [cs]*, 2021. arXiv:2105.11827.

[32] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[33] John R. Douceur. The Sybil Attack. In *Peer-to-Peer Systems*, Lecture

Notes in Computer Science, pages 251–260. Springer, Berlin, Heidelberg, March 2002. doi:10.1007/3-540-45748-8_24.

[34] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35:288–323, 1988.

[35] Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *Advances in Cryptology — CRYPTO' 92*, Lecture Notes in Computer Science, pages 139–147, Berlin, Heidelberg, 1993. Springer. doi:10.1007/3-540-48071-4_10.

[36] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for Concurrent Objects. In *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 252–266. Springer Berlin Heidelberg, 2009.

[37] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[38] Martin Florian, Sebastian Henningsen, Charmaine Ndolo, and Björn Scheuermann. The Sum of Its Parts: Analysis of Federated Byzantine Agreement Systems. *arXiv:2002.08101 [cs]*, May 2021. arXiv: 2002.08101.

[39] Bryan Ford. Threshold Logical Clocks for Asynchronous Distributed Coordination and Consensus. *arXiv:1907.07010 [cs]*, July 2019. arXiv: 1907.07010.

[40] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178, Rio Rico, AZ, USA, 1999. IEEE Comput. Soc. doi:10.1109/HOTOS.1999.798396.

[41] Adam Gagol, Damian Leśniak, Damian Straszak, and Michał Świetek. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, AFT '19, pages 214–228, New York, NY, USA, October 2019. ACM. `doi:10.1145/3318041.3355467`.

[42] Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine Quorum Systems. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.OPODIS.2018.17`.

[43] Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine Quorum Systems (Extended Version). *arXiv:1811.03642 [cs]*, November 2018. `arXiv:1811.03642`.

[44] Álvaro García-Pérez and Maria A. Schett. Deconstructing Stellar Consensus. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.OPODIS.2019.5`.

[45] Álvaro García-Pérez and Maria A. Schett. Deconstructing Stellar Consensus (Extended Version). *arXiv:1911.05145 [cs]*, December 2019. `arXiv:1911.05145`.

[46] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002. `doi:10.1145/564585.564601`.

[47] L M Goodman. Tezos: A Self-Amending Crypto-Ledger – Position Paper, 2014.

[48] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, October 2018. `doi:10.1145/3276486`.

[49] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The Consensus Number of a Cryptocurrency (Extended Version). *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*, pages 307–316, 2019. `arXiv:1906.05574`, `doi:10.1145/3293611.3331589`.

[50] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM. `doi:10.1145/1993498.1993506`.

[51] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 175–188, New York, NY, USA, 2007. ACM. `doi:10.1145/1294261.1294279`.

[52] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, New York, NY, USA, 2015. ACM. `doi:10.1145/2815400.2815428`.

[53] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KEVM: A complete seman-

tics of the ethereum virtual machine. In *Proc. 31st CSF*, pages 204–217. IEEE, 2018. doi:10.1109/CSF.2018.00022.

[54] Yoichi Hirai. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 520–535, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70278-0_33.

[55] Abhinav Jangda and Greta Yorsh. Unbounded Superoptimization. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*, pages 78–88, Vancouver, BC, Canada, 2017. ACM. doi:10.1145/3133850.3133856.

[56] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A Goal-directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 304–314, New York, NY, USA, 2002. ACM. doi:10.1145/512529.512566.

[57] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All You Need is DAG. *arXiv:2102.08325 [cs]*, February 2021. arXiv:2102.08325.

[58] Thomas King, Simon Butcher, and Lukasz Zalewski. Apocrita – High Performance Computing Cluster for Queen Mary University of London. *Zenodo*, 2017. doi:10.5281/zenodo.438045.

[59] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1751–1767, New York, NY, USA, October 2020. ACM. doi:10.1145/3372297.3423364.

[60] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin Heidelberg, second edition, 2016. doi:10.1007/978-3-662-50497-0.

[61] Nomadic Labs. Michelson: The Language of Smart Contracts in Tezos. Technical report, 2018.

[62] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.

[63] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. doi:10.1145/279227.279229.

[64] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4/3, 1982.

[65] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, March 2004. IEEE Computer Society.

[66] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 80–96, New York, NY, USA, October 2019. ACM. doi:10.1145/3341301.3359636.

[67] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Practical Verification of Peephole Optimizations with Alive.

Communications of the ACM, 61(2):84–91, January 2018. `doi:10.1145/3166064`.

[68] Giuliano Losa and Mike Dodds. On the Formal Verification of the Stellar Consensus Protocol. In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *OpenAccess Series in Informatics (OASIcs)*, pages 9:1–9:9, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/OASIcs.FMBC.2020.9`.

[69] Giuliano Losa, Eli Gafni, and David Mazières. Stellar Consensus by Instantiation. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.DISC.2019.27`.

[70] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM. `doi:10.1145/2976749.2978309`.

[71] Ethan MacBrough. Cobalt: BFT Governance in Open Networks. *arXiv:1802.07240 [cs]*, February 2018. `arXiv:1802.07240`.

[72] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998. `doi:10.1007/s004460050050`.

[73] Petros Maniatis and Mary Baker. Secure History Preservation Through Timeline Entanglement. In *Proceedings of the 11th USENIX Security Symposium*, pages 297–312, Berkeley, CA, USA, 2002. USENIX Association.

[74] Henry Massalin. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. doi:10.1145/36206.36194.

[75] David Mazières. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus. Technical report, Stellar Development Foundation, 2015.

[76] David Mazières, Giuliano Losa, and Eli Gafni. Simplified SCP, 2019.

[77] Libra Association Members. Libra White Paper — Blockchain, Association, Reserve. Technical report, Libra Association, 2020.

[78] David Menendez, Santosh Nagarakatte, and Aarti Gupta. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis*, Lecture Notes in Computer Science, pages 317–337, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-53413-7_16.

[79] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[80] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, Lecture Notes in Computer Science, pages 369–378, Berlin, Heidelberg, 1988. Springer. doi:10.1007/3-540-48184-2_32.

[81] Bernhard Mueller. Smashing Ethereum Smart Contracts for Fun and Real Profit. In *HITBSecConf2018*, Amsterdam, 2018.

[82] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. Dataflow-based pruning for speeding up superoptimization. *Proceedings*

of the ACM on Programming Languages, 4(OOPSLA):1–24, November 2020. doi:10.1145/3428245.

[83] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proc. 19th ICFP*, pages 175–188. ACM, 2014. doi:10.1145/2628136.2628143.

[84] Julian Nagele and Maria A. Schett. Blockchain Superoptimizer. In *Preproceedings of the 29th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2019)*, 2019. arXiv:2005.05912.

[85] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Technical report, 2008.

[86] Ryuya Nakamura, Takayuki Jimba, and Dominik Harz. Refinement and Verification of CBC Casper. In *2019 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 26–38, June 2019. doi:10.1109/CVCBT.2019.00008.

[87] Arvind Narayanan and Jeremy Clark. Bitcoin's Academic Pedigree. *Queue*, 15(4):20, 2017.

[88] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17. ACM, 1988.

[89] Karl Palmskog, Milos Gligoric, Lucas Peña, Brandon Moore, and Grigore Roşu. Verification of Casper in the Coq Proof Assistant. Technical report, Runtime Verification, Inc., November 2018.

[90] Daniel Perez and Benjamin Livshits. Broken Metre: Attacking Resource Metering in EVM. *arXiv:1909.07220 [cs]*, 2019. arXiv:1909.07220.

[91] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling Up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 297–310, New York, NY, USA, 2016. ACM. doi:10.1145/2872362.2872387.

[92] Justin Pombrio, Shriram Krishnamurthi, and Kathi Fisler. Teaching Programming Languages by Experimental and Adversarial Thinking. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SNAPL.2017.13.

[93] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Programming Languages and Systems*, volume 10801 of *Lecture Notes in Computer Science*, pages 619–650, Cham, 2018. Springer. doi:10.1007/978-3-319-89884-1_22.

[94] Rattle. Crytic/rattle. Crytic, April 2020.

[95] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

[96] Sean Rowan and Naïri Usher. The Flare Consensus Protocol: Fair, Fast Federated Byzantine Agreement Consensus. Technical report, 2019.

[97] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer. *arXiv:1711.04422 [cs]*, November 2017. arXiv:1711.04422.

[98] Maria A. Schett and George Danezis. Embedding a Deterministic BFT Protocol in a Block DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 177–186, New York, NY, USA, July 2021. ACM. `doi:10.1145/3465084.3467930`.

[99] Maria A. Schett and George Danezis. Embedding a Deterministic BFT Protocol in a Block DAG. *arXiv:2102.09594 [cs]*, February 2021. `arXiv:2102.09594`.

[100] Maria A. Schett and Julian Nagele. Populating the Peephole Optimizer of a Smart Contract Compiler. In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *OpenAccess Series in Informatics (OASIcs)*, pages 3:1–3:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[101] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, March 2013. `doi:10.1145/2490301.2451150`.

[102] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990. `doi:10.1145/98163.98167`.

[103] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple Protocol Consensus Algorithm. Technical report, 2014.

[104] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Conditionally Correct Superoptimization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 147–162, New York, NY, USA, 2015. ACM. `doi:10.1145/2814270.2814278`.

[105] Derek Sorensen. Establishing Standards for Consensus on Blockchains. In *Blockchain – ICBC 2019*, Lecture Notes in Computer Science, pages

18–33, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-23404-1_2.

[106] Venkatesh Srinivasan and Thomas Reps. Synthesis of machine code from semantics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 596–607, Portland, OR, USA, June 2015. ACM. doi:10.1145/2737924.2737960.

[107] Libra BFT Team. Move: A Language With Programmable Resources. Technical report, Novi, 2020.

[108] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi:10.1112/plms/s2-42.1.230.

[109] Marko Vukolić, The Distributed Computing Column, and by P. Fatourou. The Origin of Quorum Systems. *Bulletin of EATCS*, 2(101), September 2013.

[110] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. SoK: Diving into DAG-based Blockchain Systems. *arXiv:2012.06128 [cs]*, December 2020. arXiv:2012.06128.

[111] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Technical Report Byzantium Version e94ebda, Ethereum, 2018.

[112] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Technical Report Petersburg Version 3e2c089, Ethereum, 2020.

[113] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM*

SIGPLAN Conference on Certified Programs and Proofs, pages 154–165, New York, NY, USA, 2016. ACM. doi:10.1145/2854065.2854081.

[114] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2019, Stockholm, Sweden, June 17-19, 2019*, pages 310–319. IEEE, 2019. doi:10.1109/EuroSPW.2019.00041.

Appendix A

Appendix: Chapter 3

A.1 Ad Section 3.2

Example A.1.1 *Consider the FBQS containing four servers s_1 to s_4, where every set of three or more servers is a quorum, and every set of two or more servers is v-blocking for any $v \in \mathsf{Srvrs}$. Consider an execution of ASCP where the server v_3 is faulty. The FBQS has the intact set $I = \{v_1, v_2, v_4\}$. We assume that a set of alphabetical values, which we write in boldface. In the execution, servers s_1 and s_2 propose value $\boldsymbol{c}$, and server s_4 propose value $\boldsymbol{a}$. The faulty server s_3 sends a batch containing the messages $\mathtt{VOTE}(\langle 0, \perp\rangle, \mathit{false})$ and $\mathtt{VOTE}(\langle 1, \boldsymbol{a}\rangle, \mathit{false})$ to every correct server, thus helping them to prepare ballot $\langle 1, \boldsymbol{b}\rangle$. Since $\langle 1, \boldsymbol{b}\rangle$ exceeds s_4's candidate ballot $\langle 1, \boldsymbol{a}\rangle$, server s_4 will try to commit both $\langle 1, \boldsymbol{a}\rangle$ and $\langle 1, \boldsymbol{b}\rangle$. However, neither of s_1 or s_2 will try to commit any ballot since $\langle 1, \boldsymbol{b}\rangle$ is smaller than their candidate ballot $\langle 1, \boldsymbol{c}\rangle$, and therefore no quorum exists that tries to commit a ballot. Consequently, the timeout at round 1 of every correct server will expire, and since all of them managed to prepare $\langle 1, \boldsymbol{b}\rangle$, they all will try to prepare the increased ballot $\langle 2, \boldsymbol{b}\rangle$, and will ultimately commit that ballot and decide value $\boldsymbol{b}$. Notice that value $\boldsymbol{b}$ was not proposed by any correct server, but nevertheless all of them agree on the same decision. To the servers in I, server s_3 being faulty is indistinguishable from the situation where server s_3 is correct but slow, and it proposes $\boldsymbol{b}$. Therefore the servers in I cannot detect whether the decided value was proposed by some server in I or not.*

Figure A.1 depicts the trace of the execution of ASCP described above. In *each cell, we separate by a dashed line the events (above the line) that are triggered atomically, if any, from the batches of messages (below the line) that are sent by the server, if any. By BNS, the sending of every batch happens atomically with the events above the dashed line. At each cell, a server has received every batch in the rows above it. (For convenience, above the dashed line, we depict 'batched' events* vote-batch *and* deliver-batch*, which are defined in Section 3.3. Under the dashed line, we save the 'batched' send and receive primitives, and we depict one batch of messages per line.)*

In the first row of Figure A.1, the correct servers s_1, s_2, and s_4 try to prepare the ballots that they propose (lines 5–7 of Algorithm 3 and lines 3–6 of Algorithm 2), which results in each of the s_1, s_2 and s_4 sending a $\mathtt{VOTE}(b, \mathit{false})$ *message for each $b \lnapprox \langle 1, x\rangle$, where x is respectively $\boldsymbol{c}$, $\boldsymbol{c}$, and $\boldsymbol{a}$. The faulty server s_3 sends a* $\mathtt{VOTE}(b, \mathit{false})$ *message for each $b \lnapprox \langle 1, \boldsymbol{b}\rangle$. Notice the use of the sequence comprehension notation to denote sequences of events triggered in a cell, as well as sequences of messages in a batch. Server s_1 triggers* $\mathsf{propose}(\boldsymbol{c})$ *followed by the batched event* $\mathsf{vote\text{-}batch}([b,\ b \lnapprox \langle 1, \boldsymbol{c}\rangle], \mathit{false})$*, which stands for*

$$[\mathsf{ballots}[\langle 0, \bot\rangle].\mathsf{vote}(\langle 0, \bot\rangle, \mathit{false}),\ \mathsf{ballots}[\langle 1, \boldsymbol{a}\rangle].\mathsf{vote}(\langle 1, \boldsymbol{a}\rangle, \mathit{false}),$$
$$\mathsf{ballots}[\langle 1, \boldsymbol{b}\rangle].\mathsf{vote}(\langle 1, \boldsymbol{b}\rangle, \mathit{false})],$$

and it sends a batch with the sequence of messages $[\mathtt{VOTE}(b, \mathit{false}),\ b \lnapprox \langle 1, \boldsymbol{c}\rangle]$*, which stands for*

$$[\mathtt{VOTE}(\langle 0, \bot\rangle, \mathit{false}),\ \mathtt{VOTE}(\langle 1, \boldsymbol{a}\rangle, \mathit{false}),\ \mathtt{VOTE}(\langle 1, \boldsymbol{b}\rangle, \mathit{false})].$$

In the second row of Figure A.1, servers s_1, s_2, and s_4 start the timer with delay $F(1)$, since there exist ballot $\langle 1, \boldsymbol{a}\rangle$ and open interval $[\langle 0, \bot\rangle, \langle 1, \boldsymbol{a}\rangle)$ such that the quorum $\{s_1, s_2, s_4\}$ receives from itself a message $\mathtt{VOTE}(\langle 0, \bot\rangle, \mathit{false})$*, and $[\langle 0, \bot\rangle, \langle 1, \boldsymbol{a}\rangle)$ is the singleton containing the null ballot $\langle 0, \bot\rangle$ (lines 15–17 of Algorithm 3). This means that all correct servers receive from themselves vote messages that support preparing ballots with rounds bigger or equal than* 1.

In addition to this, servers s_1 *and* s_2 *send the batch* $[\texttt{READY}(b, \mathit{false}),\ b \lnapprox \langle 1, \boldsymbol{b}\rangle]$*, since they receive a message* $\texttt{VOTE}(b, \mathit{false})$ *for each* $b \lnapprox \langle 1, \boldsymbol{b}\rangle$ *from the quorum* $\{s_1, s_2, s_3\}$*, to which they belong (lines 7–9 of Algorithm 2). And similarly, server* s_4 *sends a* $\texttt{READY}(\langle 0, \perp\rangle, \mathit{false})$*, since it receives the message* $\texttt{VOTE}(\langle 0, \perp\rangle, \mathit{false})$ *from all servers, which constitute a quorum to which* s_4 *belongs. Notice that server* s_4 *cannot send* $\texttt{READY}(\langle 1, \boldsymbol{a}\rangle, \mathit{false})$ *because no quorum to which* s_4 *belongs exists that sends* $\texttt{VOTE}(\langle 1, \boldsymbol{a}\rangle, \mathit{false})$*.*

In the third row of of Figure A.1, servers s_1*,* s_2*, and* s_4 *deliver false for ballot* $\langle 1, \boldsymbol{a}\rangle$*, since they receive the message* $\texttt{READY}(\langle 0, \perp\rangle, \mathit{false})$ *from the quorum* $\{s_1, s_2, s_4\}$ *to which they all belong (lines 13–15 of Algorithm 2), which results in each of those servers preparing ballot* $\langle 1, \boldsymbol{a}\rangle$ *and triggering lines 8–12 of Algorithm 3. Since the prepared ballot* $\langle 1, \boldsymbol{a}\rangle$ *reaches* s_4*'s candidate ballot, then* s_4 *triggers the batched event* $\textsf{vote-batch}([\langle 1, \boldsymbol{a}\rangle], \mathit{true})$ *and prepares a batch with the message* $\texttt{VOTE}(\langle 1, \boldsymbol{a}\rangle, \mathit{true})$ *that it will send later (lines 8–12 of Algorithm 3 and lines 3–6 of Algorithm 2). In addition to this, server* s_4 *also prepares a batch with the message* $\texttt{READY}(\langle 1, \boldsymbol{a}\rangle, \mathit{false})$ *that it will also send later, since it receives* $\texttt{READY}(\langle 1, \boldsymbol{a}\rangle, \mathit{false})$ *from the* s_4*-blocking set* $\{s_1, s_2\}$ *(lines 10–12 of Algorithm 2). Recall that the rule in lines 10–12 of Algorithm 2 allows a server to send a ready message with some Boolean even if the server previously voted a different Boolean for the same ballot. Finally, server* s_4 *sends the two batches prepared before atomically.*

In the fourth row of Figure A.1, servers s_1*,* s_2 *and* s_4 *deliver false for ballot* $\langle 1, \boldsymbol{b}\rangle$*, since they receive a message* $\texttt{READY}(b, \mathit{false})$ *for each* $b \lnapprox \langle 1, \boldsymbol{b}\rangle$ *from the quorum* $\{s_1, s_2, s_4\}$ *to which they all belong (lines 13–15 of Algorithm 2), which results in each of those servers preparing ballot* $\langle 1, \boldsymbol{b}\rangle$ *and triggering lines 8–12 of Algorithm 3. Since the prepared ballot* $\langle 1, \boldsymbol{b}\rangle$ *exceeds* s_4*'s candidate ballot, then* s_4 *updates its candidate ballot to* $\langle 1, \boldsymbol{b}\rangle$ *and triggers* $\textsf{vote-batch}([\langle 1, \boldsymbol{b}\rangle], \mathit{true})$*, which results in* s_4 *sending* $\texttt{VOTE}(\langle 1, \boldsymbol{b}\rangle, \mathit{true})$ *(lines 8–12 of Algorithm 3 and lines 3–6 of Algorithm 2).*

At this point no server can decide any value, because there exists not any *ballot such that a quorum of servers votes true for it, and the timeouts of all correct servers will expire after $F(1)$ time.*

In the sixth row of Figure A.1, servers s_1, s_2 and s_4 trigger timeout*, and since they all prepared ballot $\langle 1, \boldsymbol{b}\rangle$, they update their candidate ballot to $\langle 2, \boldsymbol{b}\rangle$ and trigger the batched event* vote-batch.$([b,\ b \lesssim\!\!\!\!/\ \langle 2, \boldsymbol{b}\rangle], \mathit{false})$ *(lines 18–20 of Algorithm 3). Servers s_1, s_2 and s_4 send the batch* $[\texttt{VOTE}(\langle 2, \boldsymbol{b}\rangle, \mathit{false}),\ \langle 1, \boldsymbol{c}\rangle \leq b \lesssim\!\!\!\!/\ \langle 2, \boldsymbol{b}\rangle]$*, which contains infinitely many messages that are sent at once by BNS.*

In the seventh row of Figure A.1, servers s_1, s_2, and s_4 start the timer with delay $F(2)$, since there exist ballot $\langle 2, \boldsymbol{b}\rangle$ and open interval $[\langle 1, \boldsymbol{b}\rangle, \langle 2, \boldsymbol{b}\rangle)$ such that the quorum $\{s_1, s_2, s_4\}$ receives from itself the infinitely many messages $\texttt{VOTE}(b, \mathit{false})$ *with $b \in [\langle 1, \boldsymbol{b}\rangle, \langle 2, \boldsymbol{b}\rangle)$ (lines 15–17 of Algorithm 3), which are received at once by BNS. This means that all correct servers receive from themselves vote messages that support preparing ballots with rounds bigger or equal than 2. Then, servers s_1, s_2, and s_4 send the batch* $[\texttt{READY}(b, \mathit{false}),\ \langle 1, \boldsymbol{c}\rangle \leq b \lesssim\!\!\!\!/\ \langle 2, \boldsymbol{b}\rangle]$*, since they receive a message* $\texttt{VOTE}(b, \mathit{false})$ *for each b such that $\langle 1, \boldsymbol{c}\rangle \leq b \lesssim\!\!\!\!/\ \langle 2, \boldsymbol{b}\rangle$ from the quorum $\{s_1, s_2, s_3\}$ to which they belong (lines 7–9 of Algorithm 2). The batch contains infinitely many messages, which are sent at once by BNS.*

In the eight row of Figure A.1, servers s_1, s_2, and s_4 trigger deliver-batch$(b,\ \langle 1, \boldsymbol{c}\rangle \leq b \lesssim\!\!\!\!/\ \langle 2, \boldsymbol{b}\rangle], \mathit{false})$*, which stands for a vote false for each b below and incompatible than $\langle 2, \boldsymbol{b}\rangle$ for which the server didn't vote any Boolean yet, since they receive a message* $\texttt{READY}(b, \mathit{false})$ *for each of such b's from the quorum $\{s_1, s_2, s_4\}$ to which they all belong (lines 13–15 of Algorithm 2). Since the prepared ballot $\langle 2, \boldsymbol{b}\rangle$ reaches the candidate ballot of all correct servers, they trigger the event* vote-batch$([\langle 2, \boldsymbol{b}\rangle], \mathit{true})$ *and send a* $\texttt{VOTE}(\langle 2, \boldsymbol{b}\rangle, \mathit{true})$ *(lines 8–12 of Algorithm 3 and lines 3–6 of Algorithm 2).*

In the ninth row of Figure A.1, servers s_1, s_2 and s_4 send the batch $[\texttt{READY}(\langle 2, \boldsymbol{b}\rangle, \mathit{true})]$*, since they all received* $\texttt{VOTE}(\langle 2, \boldsymbol{b}\rangle, \mathit{true})$ *from the quorum $\{s_1, s_2, s_4\}$ to which all belong (lines 7–9 of Algorithm 2).*

Finally, in the tenth row of Figure A.1, servers s_1*,* s_2 *and* s_4 *trigger* $\mathsf{deliver\text{-}batch}([\langle 2, \boldsymbol{b}\rangle], \mathit{true})$*, since they all received* $\mathtt{READY}(\langle 2, \boldsymbol{b}\rangle, \mathit{true})$ *from the quorum* $\{s_1, s_2, s_4\}$ *to which all belong (lines 13–15 of Algorithm 2), and they all decide value* $\boldsymbol{b}$ *and end the execution.*

	Server s_1	Server s_2	Server s_3	Server s_4
1	propose($\mathbf{c}$) vote-batch($[b,\ b \lnapprox \langle 1, \mathbf{c}\rangle]$, false) $[\texttt{VOTE}(b, \mathit{false}),\ b \lnapprox \langle 1, \mathbf{c}\rangle]$	propose($\mathbf{c}$) vote-batch($[b,\ b \lnapprox \langle 1, \mathbf{c}\rangle]$, false) $[\texttt{VOTE}(b, \mathit{false}),\ b \lnapprox \langle 1, \mathbf{c}\rangle]$	$[\texttt{VOTE}(b, \mathit{false}),\ b \lnapprox \langle 1, \mathbf{b}\rangle]$	propose($\mathbf{a}$) vote-batch($[b,\ b \lnapprox \langle 1, \mathbf{a}\rangle]$, false) $[\texttt{VOTE}(\langle 0, \bot\rangle, \mathit{false})]$
2	start-timer($F(1)$) $[\texttt{READY}(b, \mathit{false}),\ b \lnapprox \langle 1, \mathbf{b}\rangle]$	start-timer($F(1)$) $[\texttt{READY}(b, \mathit{false}),\ b \lnapprox \langle 1, \mathbf{b}\rangle]$		start-timer($F(1)$) $[\texttt{READY}(\langle 0, \bot\rangle, \mathit{false})]$
3	deliver-batch($[b,\ b \lnapprox \langle 1, \mathbf{a}\rangle]$, false)	deliver-batch($[b,\ b \lnapprox \langle 1, \mathbf{a}\rangle]$, false)		deliver-batch($[b,\ b \lnapprox \langle 1, \mathbf{a}\rangle]$, false) vote-batch($[\langle 1, \mathbf{a}\rangle]$, true) $[\texttt{VOTE}(\langle 1, \mathbf{a}\rangle, \mathit{true})]$ $[\texttt{READY}(\langle 1, \mathbf{a}\rangle, \mathit{false})]$
4	deliver-batch($[\langle 1, \mathbf{a}\rangle]$, false)	deliver-batch($[\langle 1, \mathbf{a}\rangle]$, false)		deliver-batch($[\langle 1, \mathbf{a}\rangle]$, false) vote-batch($[\langle 1, \mathbf{b}\rangle]$, true) $[\texttt{VOTE}(\langle 1, \mathbf{b}\rangle, \mathit{true})]$
	⋮	⋮	⋮	⋮
6	timeout vote-batch($[b \lnapprox \langle 2, \mathbf{b}\rangle]$, false) $[\texttt{VOTE}(b, \mathit{false}),\ \langle 1, \mathbf{c}\rangle \leq b \lnapprox \langle 2, \mathbf{b}\rangle]$	timeout vote-batch($[b \lnapprox \langle 2, \mathbf{b}\rangle]$, false) $[\texttt{VOTE}(b, \mathit{false}),\ \langle 1, \mathbf{c}\rangle \leq b \lnapprox \langle 2, \mathbf{b}\rangle]$		timeout vote-batch($[b \lnapprox \langle 2, \mathbf{b}\rangle]$, false) $[\texttt{VOTE}(b, \mathit{false}),\ \langle 1, \mathbf{c}\rangle \leq b \lnapprox \langle 2, \mathbf{b}\rangle]$
7	start-timer($F(2)$) $[\texttt{READY}(b, \mathit{false}),\ \langle 1, \mathbf{c}\rangle \leq b \lnapprox \langle 2, \mathbf{b}\rangle]$	start-timer($F(2)$) $[\texttt{READY}(b, \mathit{false}),\ \langle 1, \mathbf{c}\rangle \leq b \lnapprox \langle 2, \mathbf{b}\rangle]$		start-timer($F(2)$) $[\texttt{READY}(b, \mathit{false}),\ \langle 1, \mathbf{c}\rangle \leq b \lnapprox \langle 2, \mathbf{b}\rangle]$
8	deliver-batch($[b,\ \langle 1, \mathbf{c}\rangle \leq b \lnapprox \langle 2, \mathbf{b}\rangle]$, false) $[\texttt{VOTE}(\langle 2, \mathbf{b}\rangle, \mathit{true})]$	deliver-batch($[b,\ \langle 1, \mathbf{c}\rangle \leq b \lnapprox \langle 2, \mathbf{b}\rangle]$, false) $[\texttt{VOTE}(\langle 2, \mathbf{b}\rangle, \mathit{true})]$		deliver-batch($[b,\ \langle 1, \mathbf{c}\rangle \leq b \lnapprox \langle 2, \mathbf{b}\rangle]$, false) $[\texttt{VOTE}(\langle 2, \mathbf{b}\rangle, \mathit{true})]$
9	$[\texttt{READY}(\langle 2, \mathbf{b}\rangle, \mathit{true})]$	$[\texttt{READY}(\langle 2, \mathbf{b}\rangle, \mathit{true})]$		$[\texttt{READY}(\langle 2, \mathbf{b}\rangle, \mathit{true})]$
10	deliver-batch($[\langle 2, \mathbf{b}\rangle]$, true) decide($\mathbf{b}$)	deliver-batch($[\langle 2, \mathbf{b}\rangle]$, true) decide($\mathbf{b}$)		deliver-batch($[\langle 2, \mathbf{b}\rangle]$, true) decide($\mathbf{b}$)

Figure A.1: Execution of ASCP.

	Server s_1	Server s_2	Server s_3	Server s_4
1	propose($\mathbf{c}$) brs.prepare($\langle 1, \mathbf{c}\rangle$) $\mathtt{VOTE}(\textsc{prep}\ \langle 1, \mathbf{c}\rangle)$	propose($\mathbf{c}$) brs.prepare($\langle 1, \mathbf{c}\rangle$) $\mathtt{VOTE}(\textsc{prep}\ \langle 1, \mathbf{c}\rangle)$	$\mathtt{VOTE}(\textsc{prep}\ \langle 1, \mathbf{b}\rangle)$	propose($\mathbf{a}$) brs.prepare($\langle 1, \mathbf{a}\rangle$) $\mathtt{VOTE}(\textsc{prep}\ \langle 1, \mathbf{a}\rangle)$
2	start-timer($F(1)$) $\mathtt{READY}(\textsc{prep}\ \langle 1, \mathbf{b}\rangle)$	start-timer($F(1)$) $\mathtt{READY}(\textsc{prep}\ \langle 1, \mathbf{b}\rangle)$		start-timer($F(1)$) $\mathtt{READY}(\textsc{prep}\ \langle 1, \mathbf{a}\rangle)$
3	brs.prepared($\langle 1, \mathbf{a}\rangle$)	brs.prepared($\langle 1, \mathbf{a}\rangle$)		brs.prepared($\langle 1, \mathbf{a}\rangle$) brs.commit($\langle 1, \mathbf{a}\rangle$) $\mathtt{VOTE}(\textsc{cmt}\ \langle 1, \mathbf{a}\rangle)$ $\mathtt{READY}(\textsc{prep}\ \langle 1, \mathbf{b}\rangle)$
4	brs.prepared($\langle 1, \mathbf{b}\rangle$)	brs.prepared($\langle 1, \mathbf{b}\rangle$)		brs.prepared($\langle 1, \mathbf{b}\rangle$) brs.commit($\langle 1, \mathbf{b}\rangle$) $\mathtt{VOTE}(\textsc{cmt}\ \langle 1, \mathbf{b}\rangle)$
⋮	⋮	⋮	⋮	
6	timeout brs.prepare($\langle 2, \mathbf{b}\rangle$) $\mathtt{VOTE}(\textsc{prep}\ \langle 2, \mathbf{b}\rangle)$	timeout brs.prepare($\langle 2, \mathbf{b}\rangle$) $\mathtt{VOTE}(\textsc{prep}\ \langle 2, \mathbf{b}\rangle)$		timeout brs.prepare($\langle 2, \mathbf{b}\rangle$) $\mathtt{VOTE}(\textsc{prep}\ \langle 2, \mathbf{b}\rangle)$
6	start-timer($F(2)$) $\mathtt{READY}(\textsc{prep}\ \langle 2, \mathbf{b}\rangle)$	start-timer($F(2)$) $\mathtt{READY}(\textsc{prep}\ \langle 2, \mathbf{b}\rangle)$		start-timer($F(2)$) $\mathtt{READY}(\textsc{prep}\ \langle 2, \mathbf{b}\rangle)$
7	brs.prepared($\langle 2, \mathbf{b}\rangle$) brs.commit($\langle 2, \mathbf{b}\rangle$) $\mathtt{VOTE}(\textsc{cmt}\ \langle 2, \mathbf{b}\rangle)$	brs.prepared($\langle 2, \mathbf{b}\rangle$) brs.commit($\langle 2, \mathbf{b}\rangle$) $\mathtt{VOTE}(\textsc{cmt}\ \langle 2, \mathbf{b}\rangle)$		brs.prepared($\langle 2, \mathbf{b}\rangle$) brs.commit($\langle 2, \mathbf{b}\rangle$) $\mathtt{VOTE}(\textsc{cmt}\ \langle 2, \mathbf{b}\rangle)$
8	$\mathtt{READY}(\textsc{cmt}\ \langle 2, \mathbf{b}\rangle)$	$\mathtt{READY}(\textsc{cmt}\ \langle 2, \mathbf{b}\rangle)$		$\mathtt{READY}(\textsc{cmt}\ \langle 2, \mathbf{b}\rangle)$
9	brs.committed($\langle 2, \mathbf{b}\rangle$) decide($\mathbf{b}$)	brs.committed($\langle 2, \mathbf{b}\rangle$) decide($\mathbf{b}$)		brs.committed($\langle 2, \mathbf{b}\rangle$) decide($\mathbf{b}$)

Figure A.2: Execution of CSCP.

A.2 Ad Section 3.3

Example A.2.1 *Recall Example A.1.1. Compare the execution of ASCP in Figure A.1 with infinitely many events and messages with the finite execution of CSCP in Figure A.2. The servers propose the same values as in Example A.1.1. In particular, in the first row, the faulty server s_3 sends* $\mathtt{VOTE}(\textsc{prep}\ \langle 1, \boldsymbol{b}\rangle)$ *to every correct server. As in ASCP every correct server starts a timer in the second row. As in ASCP server s_4 has prepared $\langle 1, \boldsymbol{a}\rangle$ and sends* $\mathtt{READY}(\textsc{prep}\ \langle 1, \boldsymbol{a}\rangle)$ *after receiving* $\mathtt{VOTE}(\textsc{prep}\ b_u)$ *from a quorum for $b_u \in \{\langle 1, \boldsymbol{b}\rangle, \langle 1, \boldsymbol{c}\rangle\}$ where $b' \in \{\langle 0, \perp\rangle\}$ and $b' \lessapprox b_u$ (lines 10–12 of Algorithm 4). In the third row, the servers s_1, s_2 and s_4 prepare the maximum ballot $\langle 1, \boldsymbol{a}\rangle$, as they received* $\mathtt{READY}(\textsc{prep}\ b_u)$ *from a quorum for $b_u \in \{\langle 1, \boldsymbol{a}\rangle, \langle 1, \boldsymbol{b}\rangle\}$ where $b' \in \{\langle 0, \perp\rangle\}$ and $b' \lessapprox b_u$ (lines 18–12 of Algorithm 4). Now server s_4 reaches its candidate value $\langle 1, \boldsymbol{a}\rangle$ and therefore votes for it. But at the same time, s_4 receives* $\mathtt{READY}(\textsc{prep}(\langle 1, \boldsymbol{b}\rangle))$ *from the s_4-blocking set $\{s_1, s_2\}$ and sends* $\mathtt{READY}(\textsc{prep}\ \langle 1, \boldsymbol{b}\rangle)$ *(lines 14–16 of Algorithm 4). In the fourth, server s_4 only votes one commit statement* $\textsc{cmt}\ \langle 1, \boldsymbol{b}\rangle$*, as opposed to voting* *true for the two ballots $\langle 1, \boldsymbol{a} \rangle$ and $\langle 1, \boldsymbol{b} \rangle$ in the fourth row of Figure A.1. Similar to Example A.1.1, the correct servers decide value $\boldsymbol{b}$, which was not proposed by any correct server. As in Figure A.2, at this point no server can decide any value, because there is no ballot with a quorum of servers for it, and the timeouts of all correct servers will expire after $F(1)$ time. Then, in the sixth row of Figure A.2, servers s_1, s_2 and s_4 trigger* timeout*, and since they all prepared ballot $\langle 1, \boldsymbol{b} \rangle$, they update their candidate ballot to $\langle 2, \boldsymbol{b} \rangle$. Now s_1, s_2 and s_4 have all the same candidate ballot and analogues to row six to nine in of Figure A.1 can execute CSCP to decide value $\boldsymbol{b}$ and end the execution.*

For illustration, the executions in Figure A.2 and A.1 entail concrete and abstract traces τ and ρ respectively, which consist of the events on the left of each cell when traversing the tables in left-to-right, top-down fashion, and where the network events on the right of each cell are intermixed in such a way that the assumptions on atomic and batched semantics are met. It is routine to check that $H(\tau|_{\{1,2,4\}}) = H(\rho|_{\{1,2,4\}})$ and that $\rho|_{\{1,2,4\}} = \sigma(\tau|_{\{1,2,4\}})$.

Because the proof of Lemma A.2.6 from Chapter 3 is not intrinsically difficult, but verbose, I give it only in the appendix. The next lemmas help to establish Lemma A.2.6. The first lemma shows that round, prepared, and candidate coincide in executions of ASCP and CSCP.

Lemma A.2.1 *Let $\mathcal{F}$ be an FBQS with some intact set I, s be a server with $s \in I$, and τ be a trace entailed by an execution of CSCP. If $\sigma(\tau)$ is a trace entailed by an execution of ASCP, then s.*round*, s.*prepared*, and s.*candidate *coincide in both executions.*

Proof A.2.1 *We prove the statement by induction on τ. For the base case, it suffices to observe, that* candidate*,* prepared*, and* round *coincide when initialised in line 3 and 4 of Algorithm 5 and line 3 and 4 of Algorithm 3. For the step case $\tau = \tau' \cdot e$ we consider only the interesting cases, where* candidate*,* prepared*, or* round *are modified in line 6, line 11, line 16, line 19, and line 20 of Algorithm 5. For the other events in the concrete trace τ, the fields are not modified and the statement holds.*

Case $e = s.\mathsf{propose}(x)$**:** *By definition* $\sigma(\tau)$ *contains* $s.\mathsf{propose}(x)$*, and by line 6 of Algorithm 5 and by line 6 of Algorithm 3,* candidate *coincides.*

Case $e = \mathsf{prepared}(b)$**:** *By definition* $\sigma(\tau)$ *contains* $s.\mathsf{deliver\text{-}batch}([b', b' \lessapprox b], \mathit{false})$*. By induction hypothesis* prepared *coincide, and therefore* prepared $< b$*. Then by line 9 of Algorithm 5 and by line 9 of Algorithm 3,* prepared *coincides. Again, by induction hypothesis* candidate *coincides, and therefore* candidate $\leqslant$ prepared *coincides. If* candidate $\leqslant$ prepared *holds then by line 11 of Algorithm 5 and by line 11 of Algorithm 3,* candidate *coincides.*

Case $e = \mathsf{start\text{-}timer}(n)$**:** *By line 15 of Algorithm 5 trace* τ' *contains* $s.\mathsf{receive}(\mathtt{M}_u(\textsc{stmt}_u\ b_u), u)$ *from* u *with* $\textsc{stmt}_u \in \{\textsc{cmt}, \textsc{prep}\}$ *for a quorum* $U \in \mathcal{Q}$ *such that* $s \in U$ *and for each* $u \in U$ *exists* $\mathtt{M}_u \in \{\mathtt{VOTE}, \mathtt{READY}\}$ *and* $b_u \in \mathsf{Ballot}$ *such that* round $< b_u.n$*.*

Sub-case $\mathtt{M}_u(\textsc{prep}\ b_u)$**.** *By definition* $\sigma(\tau')$ *contains a batch with* $\mathtt{M}_u(b'_u, \mathit{false})$ *for every* $b'_u \lessapprox b_u$ *and every* $\mathtt{M}_u(\textsc{prep}\ b_u)$*.*

Sub-case $\mathtt{M}_u(\textsc{cmt}\ b_u)$**.** *By definition* $\sigma(\tau')$ *contains a batch with* $\mathtt{M}_u(b_u, \mathit{true})$ *for every* $\mathtt{M}_u(\textsc{cmt}\ b_u)$*.*

By induction hypothesis, round *and therefore* round $< b_u.n$ *coincides. By line 16 of Algorithm 5 and by line 16 of Algorithm 3,* round *coincides.*

Case $e = \mathsf{timeout}$**:** *By definition* $\sigma(\tau)$ *contains* $s.\mathsf{timeout}$*, and by induction hypothesis* candidate*,* prepared*, and* round *coincide. Then by line 19 and 20 of Algorithm 5 and line 19 and 20 of Algorithm 3,* candidate*,* prepared*, and* round *coincide.*

The next lemmas relate the prepared ballots between ASCP and CSCP. First, we establish an invariant on the prepared ballot in CSCP.

Lemma A.2.2 *Let* $\mathcal{F}$ *be an FBQS with some intact set* I*,* s *be a server with* $s \in I$*, and* τ *be a trace entailed by an execution of CSCP. Then for every ballot*

$b \in s.\mathsf{Bllts\text{-}dl\text{-}cmt}$ *(respectively,* $b \in s.\mathsf{Bllts\text{-}rd\text{-}cmt}$*) holds* $b \leqslant s.\mathsf{max\text{-}dl\text{-}prep}$ *(respectively,* $b \leqslant s.\mathsf{max\text{-}dl\text{-}prep}$*).*

Proof A.2.2 *Assume towards a contradiction, that there is a ballot* $b \in \mathsf{Bllts\text{-}dl\text{-}cmt}$ *(respectively,* $b \in s.\mathsf{Bllts\text{-}rd\text{-}cmt}$*) such that* $b > \mathsf{max\text{-}dl\text{-}prep}$ *(respectively,* $b > s.\mathsf{max\text{-}dl\text{-}prep}$*). This is only possible, if* s *sent* $\mathtt{READY}(\text{PREP } b')$ *and* $\mathtt{READY}(\text{CMT } b)$ *to itself where* $b' < b$ *(lines 19 and 20, and lines 32 and 33 of Algorithm 4), but then* s *sent contradicting messages, which contradicts that* $s \in I$*.*

The next lemma guarantees that for no ballot above the maximal delivered ballot in CSCP, in ASCP this ballot was delivered.

Lemma A.2.3 *Let* $\mathcal{F}$ *be an FBQS with some intact set* I*,* s *be a server with* $s \in I$*, and* τ *be a trace entailed by an execution of CSCP. If* $\sigma(\tau)$ *is a trace entailed by an execution of ASCP, for every* $b > s.\mathsf{max\text{-}dl\text{-}prep}$ *holds* $s.\mathsf{brs}[b].\mathsf{delivered}$ *is false.*

Proof A.2.3 *Assume towards a contradiction, that* $s.\mathsf{brs}[b].\mathsf{delivered}$ *is true. By lines 13–15 of Algorithm 2 this is only possible if* $\sigma(\tau)$ *contains an event* $s.\mathsf{send\text{-}batch}(ms, u)$ *with* $\mathtt{READY}(b, a) \in ms$ *for* $a \in \{true, false\}$ *from every* u *in a quorum* U*. Assume* $\mathtt{READY}(b, true) \in ms$*. Then by definition* $\sigma(\tau)$ *contains* $s.\mathsf{send}(\mathtt{READY}(\text{CMT } b), u)$ *and by lines 32 and 33 of Algorithm 4,* $b \in s.\mathsf{Bllts\text{-}dl\text{-}cmt}$*, but then* $b \leqslant s.\mathsf{max\text{-}dl\text{-}prep}$ *by Lemma A.2.2. As* $b > s.\mathsf{max\text{-}dl\text{-}prep}$*,* $\sigma(\tau)$ *contains an event* $s.\mathsf{send\text{-}batch}(ms, u)$ *with* $\mathtt{READY}(b, false) \in ms$ *and by lines 13–15 of Algorithm 2 this is only possible if* $\sigma(\tau)$ *contains an event* $s.\mathsf{send\text{-}batch}(ms, u)$ *where* $\mathtt{READY}(b, false) \in ms$ *from every server* u *in a quorum* U *where* $s \in U$*. Again, by definition of* σ *and BNS this entails that* τ *contains* $s.\mathsf{receive}(\mathtt{READY}(\text{PREP } b_u), u)$ *for* $b' \lnapprox b_u$ *for every* $b' \lnapprox b$*, but then, by lines 18 and 19 of Algorithm 4,* $s.\mathsf{max\text{-}dl\text{-}prep}$ *is assigned to* b *and this contradicts* $b > s.\mathsf{max\text{-}dl\text{-}prep}$.

The next lemma guarantees that for no ballot above the maximal readied ballot in CSCP, in ASCP this ballot is not ready.

Lemma A.2.4 *Let $\mathcal{F}$ be an FBQS with some intact set I, s be a server with $s \in I$, and τ be a trace entailed by an execution of CSCP. If $\sigma(\tau)$ is a trace entailed by an execution of ASCP, for every $b > s.\mathsf{max\text{-}rd\text{-}prep}$ holds $s.\mathsf{brs}[b].\mathsf{ready}$ is false.*

Proof A.2.4 *Assume towards a contradiction, that $s.\mathsf{brs}[b].\mathsf{ready}$ is true. By lines 7–9 and lines 10–12 of Algorithm 2 this is only possible if $\sigma(\tau)$ contains an event $s.\mathsf{send\text{-}batch}(ms, u)$ with $\mathtt{READY}(b, a) \in ms$ for $a \in \{true, false\}$ for every u in either a quorum U or a s-blocking set B. Assume $\mathtt{READY}(b, true) \in ms$. Then by definition $\sigma(\tau)$ contains $s.\mathsf{send}(\mathtt{READY}(\textsc{cmt}\ b), u)$ for every u and by lines 10 and 11, or lines 14 and 15 of Algorithm 4, $b \in s.\mathsf{Bllts\text{-}rd\text{-}cmt}$, but then $b \leqslant s.\mathsf{max\text{-}rd\text{-}prep}$ by Lemma A.2.2. As $b > s.\mathsf{max\text{-}rd\text{-}prep}$, $\sigma(\tau)$ contains an event $s.\mathsf{send\text{-}batch}(ms, u)$ with $\mathtt{READY}(b, false) \in ms$ for every u in either a quorum U or a s-blocking set B. Assume $\mathtt{READY}(b, true) \in ms$. Then again, by definition of σ and BNS this entails that τ contains $s.\mathsf{receive}(\mathtt{READY}(\textsc{prep}\ b_u), u)$ for $b' \lessapprox b_u$ for every $b' \lessapprox b$ for every u in either a quorum U or a s-blocking set B, but then, by lines 10 and 11, or lines 14 and 15 of Algorithm 4, $s.\mathsf{max\text{-}rd\text{-}prep}$ is assigned to b and this contradicts $b > s.\mathsf{max\text{-}rd\text{-}prep}$.*

The following lemma relates the committed ballots from CSCP to the delivered ballots in ASCP.

Lemma A.2.5 *Let $\mathcal{F}$ be an FBQS with some intact set I, s be a server with $s \in I$, and τ be a trace entailed by an execution of CSCP. If $\sigma(\tau)$ is a trace entailed by an execution of ASCP and $b \notin \mathsf{Bllts\text{-}dl\text{-}cmt}$ then $b.\mathsf{delivered}$ is false.*

Proof A.2.5 *Assumes towards a contradiction that $b.\mathsf{delivered}$ is true. By lines 13–15 of Algorithm 2 and BNS, this is only possible if $\sigma(\tau)$ contains an event $s.\mathsf{receive\text{-}batch}(ms, u)$ with $\mathtt{READY}(b, a) \in ms$ for $a \in \{true, false\}$ from a quorum U such that $s \in U$. If a is true, then by definition of σ, τ contains $s.\mathsf{receive}(\mathtt{READY}(\textsc{cmt}\ b), u)$ from a quorum U such that $s \in U$. By lines 32 and 33 in Algorithm 5, $b \in \mathsf{Bllts\text{-}dl\text{-}cmt}$ and this contradicts $b \notin \mathsf{Bllts\text{-}dl\text{-}cmt}$. If a is false, then $s.\mathsf{receive}(\mathtt{READY}(\textsc{prep}\ b_u), u)$ from a quorum U such that $s \in U$ and*

$b' \lesssim b_u$ for every $b' \lesssim b$. Then by lines 18 and 19 of Algorithm 4, max-dl-prep *is assigned to b and b.*delivered *is true contradicts Lemma A.2.3.*

Finally, we show the key lemma:

Lemma A.2.6 *Let $\mathcal{F}$ be an FBQS with some intact set I and τ be a trace entailed by an execution of CSCP. For every finite prefix τ' of the projected trace $\tau|_I$, the simulated $\rho' = \sigma(\tau')$ is the prefix of a trace entailed by an execution of ASCP.*

Proof A.2.6 *We proceed by induction on the length of τ'. The case $\tau' = [\,]$ is trivial since $\sigma([\,]) = [\,]$ is the prefix of any trace. We let $\tau' = \tau'_1 \cdot [e]$ and consider the following cases:*

Case $e = s.\mathsf{prepare}(b)$: *For any execution of the CSCP with trace τ'_1, the prefix τ'_1 contains either the event $s.\mathsf{propose}(b.x)$ by lines 5 and 7 of Algorithm 5, or the event $s.\mathsf{timeout}$ by lines 18 and 21 of Algorithm 5. The definition of σ entails that the simulated prefix $\rho'_1 = \sigma(\tau'_1)$ contains either $s.\mathsf{propose}(b.x)$ or $s.\mathsf{timeout}$. By the induction hypothesis, the simulated prefix ρ'_1 is entailed by an execution of ASCP. Let the sub-trace that simulates event e be $\rho'_e = s.\mathsf{vote\text{-}batch}([b', b' \lesssim b], \mathit{false})$. We show that $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP.*

Sub-case s proposes $b.x$. *By lines 5–7 of Algorithm 3, s triggers $s.b'.\mathsf{vote}(\mathit{false})$ for every $b' \lesssim \langle 1, b.x\rangle$ is in the execution of ASCP.*

Sub-case s triggers timeout *By line 21 of Algorithm 5 ballot b equals* candidate *and by Lemma A.2.1* candidate *coincides. By lines 18–21 of Algorithm 3, $s.b'.\mathsf{vote}(\mathit{false})$ for every $b' \lesssim b$ is in the execution of ASCP.*

As s triggered vote*(false) for every $b' \lesssim b$ in both cases. When batched, this results in the event* vote-batch*$([b', b' \lesssim b], \mathit{false})$, and $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP.*

Case s.commit(b). *By lines 8 and 12 of Algorithm 5, for any execution of CSCP with trace τ', the prefix τ'_1 contains the event s.*prepared(b). *The definition of σ entails that the simulated prefix $\rho'_1 = \sigma(\tau'_1)$ contains the event s.*deliver-batch$([b', b' \lesssim b], \mathit{false})$. *By the induction hypothesis, the simulated prefix ρ'_1 is entailed by an execution of ASCP. Let the sub-trace that simulates event e be $\rho'_e = s$.*vote-batch$([b'', \phi(\tau'_1) < b'' \leqslant b], \mathit{true})$. *We show that $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP. Fix a ballot b'' where $\phi(\tau'_1) < b'' \leqslant b$. By definition $\phi(\tau'_1)$ equals* prepared *and for every b'' holds* prepared $< b''$. *Since ρ'_1 contains the event s.*deliver-batch$([b_i, b' \lesssim b], \mathit{false})$, *$s$ triggered b'.*deliver(false) *for each $b' \lesssim b$, and* candidate *and* prepared *coincide by Lemma A.2.1, the guard at line 8 of Algorithm 3 holds after any of such executions of ASCP. We can reason in the same fashion for every b'' in $\phi(\tau'_1) < b'' \leqslant b$. By processing b'' in increasing order of ballots,* candidate *increases monotonically and triggers s.*vote(b'', true) *for every ballot b''. As s triggered* vote(b'', true) *for every $\phi(\tau'_1) < b'' \leqslant b$. When batched, this results in the event s.*vote-batch$([b'', \phi(\tau'_1) < b'' \leqslant b], \mathit{true})$, *and therefore $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP.*

Case $e = s$.prepared(b). *By lines 18 and 20 of Algorithm 4, for any execution of CSCP with trace τ' there exists a maximum b such $b >$* max-dl-prep *and a quorum U that contains server s and for each $u \in U$ server s received* READY(PREP b_u) *where $b' \lesssim b_u$ for every $b' \lesssim b$. Therefore the prefix τ'_1 contains for every $u \in U$ the event s.*receive(READY(PREP b_u)$, u$). *The definition of σ entails that the simulated prefix $\rho'_1 = \sigma(\tau'_1)$ contains the event s.*receive-batch([READY$(b'_u, \mathit{false}), b'_u \lesssim b_u], u$) *for each s.*receive(READY(PREP b_u)$, u$) *that occurs in τ'_1. By the induction hypothesis, the simulated prefix ρ'_1 is entailed by an execution of ASCP. Let the sub-trace that simulates event e be $\rho'_e = s$.*deliver-batch$([b'', b'' \lesssim b \wedge \forall s.$deliver-batch$(bs) \in \sigma(\tau).\,(b', \mathit{false}) \notin bs], \mathit{false})$. *We show that $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP. Fix a* *ballot* b'' *where* $b'' \lesssim b$ *and there is no batch* $s.deliverBatch(bs, false)$ *with* $b'' \in bs$ *in* ρ'_1*. For each server* $u \in U$*, we know that* ρ'_1 *contains an event* $s.\mathsf{receive\text{-}batch}([\mathtt{READY}(b'_u, false), b'_u \lesssim b_u], u)$*. As for every* $b' \lesssim b$ *we know* $b' \lesssim b_u$*, we have* $b'' \lesssim b_u$*. Thus and by BNS, we know that* s *received* $\mathtt{READY}(b'', false)$ *from* u*. By Lemma A.2.3 and by* $b > \mathsf{max\text{-}dl\text{-}prep}$*, we know that* $b'.\mathsf{delivered}$ *is false. Therefore, by lines 13–15 of Algorithm 2, triggers* $s.b'.\mathsf{deliver}(b', false)$*. We can reason in the same fashion for every ballot* b' *and batch the delivers in the event* $s.\mathsf{deliver\text{-}batch}([b'',\ b'' \lesssim b \wedge \forall s.\mathsf{deliver\text{-}batch}(bs) \in \sigma(\tau).\ (b', false) \notin bs], false)$*, and therefore* $\rho' z_1 \cdot \rho'_e$ *is the prefix of a trace entailed by an execution of ASCP.*

Case $e = s.\mathsf{committed}(b)$**:** *By lines 32 and 34 of Algorithm 4, for any execution of CSCP with trace* τ'_1*, there exists a quorum* U *that contains server* s *which is such that* s *receives* $\mathtt{READY}(\text{CMT } b)$ *from every* u *in* U *and* $b \notin \mathsf{Bllts\text{-}dl\text{-}cmt}$*. The definition of* σ *entails that the simulated prefix* $\rho'_1 = \sigma(\tau'_1)$ *contains an event* $s.\mathsf{receive\text{-}batch}([\mathtt{READY}(b, true)], u)$ *for each* $s.\mathsf{receive}(\mathtt{READY}(\text{CMT } b), u)$ *that occurs in* τ'_1*. By the induction hypothesis, the simulated prefix* ρ'_1 *is entailed by an execution of ASCP. Let the sub-trace that simulates the event* e *be* $\rho'_e = s.\mathsf{deliver\text{-}batch}([b], true)$*. We show that* $\rho'_1 \cdot \rho'_e$ *is the prefix of trace entailed by an execution of ASCP. As* s *received* $\mathtt{READY}(b, true)$ *from a quorum* U *where* $s \in U$*. As* $b \notin \mathsf{delivered}$ *by Lemma A.2.5* $\mathsf{deliver}$ *is false, and by lines 7 and 9 of Algorithm 2 triggers* $s.\mathsf{deliver}(b, true)$*. When batched, this results in the event* $s.\mathsf{deliver\text{-}batch}([b], true)$*, and therefore* $\rho'_1 \cdot \rho'_e$ *is the prefix of a trace entailed by an execution of ASCP.*

Case $e = s.\mathsf{send}(\mathtt{VOTE}(\text{PREP } b), u)$**.** *By lines 4 and 7 of Algorithm 4, for any execution of CSCP with trace* τ'*, the prefix* τ'_1 *contains the event* $s.\mathsf{prepare}(b)$*. The definition of* σ *entails that the simulated prefix* $\rho'_1 = \sigma(\tau'_1)$ *contains the event* $s.\mathsf{vote\text{-}batch}([b', b' \lesssim b], false)$*. By the induction hypothesis, the simulated prefix* ρ'_1 *is a trace entailed by an* *execution of ASCP. Let the sub-trace that simulates event e be* $\rho'_e = s.\mathsf{send\text{-}batch}([\mathtt{VOTE}(b', \mathit{false}), b' \lnsim b \land \forall a \in \mathsf{Bool}. \forall s.\mathsf{send\text{-}batch}(ms, u) \in \sigma(\tau). \mathtt{M}(b', a) \notin ms], u)$. *We show that* $\rho'_1 \cdot \rho'_e$ *is the prefix of a trace entailed by an execution of ASCP. Fix a ballot* $b' \lnsim b$ *such that* ρ'_1 *does not contain the an event* $s.\mathsf{send\text{-}batch}(ms, u)$ *with* $\mathtt{VOTE}(b', \mathit{false}) \in ms$. *Then by lines 2 and 5 of Algorithm 2 we know that the Boolean* voted *is false. Hence, the condition in line 4 of the same figure is satisfied, and since* $s.\mathsf{vote\text{-}batch}([b', b' \lnsim b], \mathit{false})$, *s triggered* $b'.\mathsf{vote}(\mathit{false})$, *appending* $s.\mathsf{send\text{-}batch}(ms, u)$ *with* $\mathtt{VOTE}(b', \mathit{false}) \in ms$ *results in a trace entailed by an execution of ASCP by line 6 of the same figure. We can reason in the same fashion for every ballot* $b' \lnsim b$ *and conclude together with BNS that* $\rho'_1 \cdot \rho'_e$ *is the prefix of a trace entailed by an execution of ASCP.*

Case $e = s.\mathsf{receive}(\mathtt{VOTE}(\textsc{prep}\ b), u)$**.** *By assumption the network does not create or drop messages, hence s receives* $\mathtt{VOTE}(\textsc{prep}\ b)$ *only after u previously sent the same message and the prefix* τ'_1 *contains the event* $u.\mathsf{send}(\mathtt{VOTE}(\textsc{prep}\ b), s)$. *The definition of σ entails that the simulated prefix* $\rho'_1 = \sigma(\tau'_1)$ *contains an event with* $u.\mathsf{send\text{-}batch}([\mathtt{VOTE}(b', a), b' \lnsim b], s)$ *for* $a \in \mathsf{Bool}$. *By the induction hypothesis, the simulated prefix* ρ'_1 *is entailed by an execution of ASCP. Let the sub-trace that simulates event e be* $\rho'_e = s.\mathsf{receive\text{-}batch}([\mathtt{VOTE}(b', \mathit{false}), b' \lnsim b \land \forall a \in \mathsf{Bool}. \forall s.\mathsf{receive\text{-}batch}(ms, u) \in \sigma(\tau). \mathtt{VOTE}(b', a) \notin ms], u)$. *We show that* $\rho'_1 \cdot \rho'_e$ *is the prefix of a trace entailed by an execution ASCP. By the ascending-ballot-order convention, it is enough to show that each* $b' \lnsim b$, *s receives a batch with* $\mathtt{VOTE}(b', a)$ *for* $a \in \mathsf{Bool}$ *exactly once in* ρ'. *For a fixed* b', *an event with* $s.\mathsf{receive\text{-}batch}(ms, u)$ *with* $\mathtt{VOTE}(b', \mathit{false}) \in ms$ *is in* ρ'_e *only if* $s.b'.\mathsf{receive}(\mathtt{VOTE}(b', a), u)$ *is not in* ρ'_1. *On the other hand, u sent a batch event with* $u.b'.\mathsf{send}(\mathtt{VOTE}(b', a), s)$ *for each* $b' \lnsim b$. *Hence,* $\rho'_1 \cdot \rho'_e$ *is the prefix of a trace entailed by an execution of ASCP.*

Case $e = s.\mathsf{send}(\mathtt{READY}(\textsc{prep}\ b), u)$**.** *For any execution of the CSCP with trace* τ'_1, *the server s sends* $\mathtt{READY}(\textsc{prep}\ b)$ *either after hearing from* *a quorum in line 12 of Algorithm 4, or after hearing from a s-blocking set in line 16 of the same figure. We consider both cases:*

Sub-case s sends $\mathtt{READY}(\textsc{prep}\ b)$ after hearing from a quorum.

By lines 10–12 of Algorithm 4, exists a maximum ballot b such that $\textsf{max-rd-prep} < b$ *and there exists a quorum U such that $s \in U$ and for every server $u \in U$ the server s received* $\mathtt{VOTE}(\textsc{prep}\ b_u)$ *where $b' \lnapprox b_u$ for every $b' \lnapprox b$. The definition of σ entails that the simulated prefix $\rho'_1 = \sigma(\tau'_1)$ contains an event with* $s.\textsf{receive-batch}([\mathtt{VOTE}(b'_u, a), b'_u \lnapprox b_u \wedge \forall a \in \textsf{Bool}. \forall s.\textsf{receive-batch}(ms, u) \in \sigma(\tau). \mathtt{VOTE}(b'_u, a) \notin ms], u)$ *for each server $u \in U$ and each event* $s.\textsf{receive}(\mathtt{VOTE}(\textsc{prep}\ b_u), u)$*. By the induction hypothesis, the simulated prefix ρ'_1 is a trace entailed by an execution of ASCP. Let the sub-trace that simulates event e be* $\rho'_e = s.\textsf{send-batch}([\mathtt{READY}(b', \mathit{false}), b' \lnapprox b \wedge \forall s.\textsf{send-batch}(ms, u) \in \sigma(\tau). \mathtt{READY}(b', \mathit{false}) \notin ms], u)$*. We show that $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP. If $b.n = 1$ then by b maximal and $b \lnapprox b_u$, s received a batch with* $b'.\mathtt{VOTE}(b', \mathit{false})$ *for $b' \lnapprox b$ from every $u \in U$ such that $s \in U$. Then, by lines 7–9 in Algorithm 2, by BNS a batch with* $\mathtt{READY}(b_j, \mathit{false})$ *is in ρ'_1. If $b.n > 1$, and as s is correct, by lines 18–21 and lines 8–17 of Algorithm 5 s prepared the ballot $b^v_p = \langle b.n - 1, b^s.x\rangle$ in the previous round. By lines 18–20 of Algorithm 4, s sends* $\mathtt{READY}(\textsc{prep}\ b^s_p)$*. Hence by definition of σ, a batch with* $\mathtt{READY}(b_j, a)$ *is in ρ'_1 for every $b_j \lnapprox b^v_p$. It remains to show that a batch with* $s.\textsf{send-batch}([\mathtt{READY}(b_j, \mathit{false}), b^p_s < b_j < b], u)$ *is in $\rho'_1 \cdot \rho'_e$. By assumption, for each server u and $b'_u \lnapprox b_u$ the server s received* $\mathtt{VOTE}(b'_u, a)$*. It suffices to show that the server s receives* $\mathtt{VOTE}(b_j, \mathit{false})$ *from every $u \in U$ for every ballot b_j. Then, by lines 7–9 and BNS in Algorithm 2 a batch with* $\mathtt{READY}(b_j, \mathit{false})$ *is in ρ'_1. By Lemma A.2.4 and* $b' > b > \textsf{max-rd-prep}$, $\textsf{ready}$ *is false for b'.*

Sub-case s sends $\mathtt{READY}(\textsc{prep}\ b)$ after hearing from a s-blocking set.

By lines 14–16 of Algorithm 4 there exists a maximum ballot b such that $\mathsf{max\text{-}rd\text{-}prep} < b$ *and there exists a s-blocking set B such that for every $u \in B$ the server s received* $\mathtt{READY}(\textsc{prep}\ b_u)$ *where $b' \lessapprox b_u$ for every $b' \lessapprox b$. The definition of σ entails that the simulated prefix $\rho'_1 = \sigma(\tau'_1)$ contains an event* $s.\mathsf{receive\text{-}batch}([\mathtt{READY}(b'_u, \mathit{false}), b'_u \lessapprox b_u], u)$ *for each server $u \in B$ and each event* $s.\mathsf{receive}(\mathtt{READY}(\textsc{prep}\ b_u), u)$*. By the induction hypothesis, the simulated prefix ρ'_1 is entailed by an execution of ASCP. Let the sub-trace that simulates event e be* $\rho'_e = s.\mathsf{send\text{-}batch}([\mathtt{READY}(b', \mathit{false}), b' \lessapprox b \wedge \forall s.\mathsf{send\text{-}batch}(ms, u) \in \sigma(\tau).\, \mathtt{READY}(b', \mathit{false}) \notin ms], u)$*. Fix a ballot $b'' \in B$ such that $b'' \lessapprox b$ and for a batch with* $s.b''.\mathsf{send}(\mathtt{READY}(b'', \mathit{false}), u) \notin \sigma(\tau'_1)$*. By Lemma A.2.4 and* $b'' > b >$ $\mathsf{max\text{-}rd\text{-}prep}$, ready *is false for b''. We have to show that s received* $\mathtt{READY}(b'', \mathit{false})$ *from every u in the s-blocking set B. Then by lines 10–12 in Algorithm 2 s send* $\mathtt{READY}(b'', \mathit{false})$ *to u. As for every $b' \lessapprox b$ we know $b' \lessapprox b_u$, we have $b'' \lessapprox b_u$. Thus, we know that a batch with* $\mathtt{READY}(b'', \mathit{false})$ *is in ρ'_1.*

Both cases show that for the sub-trace ρ'_e that simulates event e, the trace $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP.

Case $e = s.\mathsf{receive}(\mathtt{READY}(\textsc{prep}\ b), u)$**.** *Analogue to case* $s.\mathsf{receive}(\mathtt{VOTE}(\textsc{prep}\ b), u)$.

Case $e = s.\mathsf{send}(\mathtt{VOTE}(\textsc{cmt}\ b), u)$**.** *By lines 23 and 25 of Algorithm 4, for any execution of CSCP with trace τ' the prefix τ'_1 contains the event* $s.\mathsf{commit}(b)$*. The definition of σ entails that the simulated prefix $\rho'_1 = \sigma(\tau'_1)$ contains the event* $s.\mathsf{vote\text{-}batch}([b', \phi(\rho'_1) < b' \leqslant b], \mathit{true})$*. By the induction hypothesis, the simulated prefix ρ'_1 is the prefix of a trace entailed by an execution of ASCP. Let the sub-trace that simulates event e be* $\rho'_e = s.\mathsf{send\text{-}batch}([\mathtt{VOTE}(b', \mathit{true}), \phi(\rho'_1) < b' \leqslant b], u)$*. We show that $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP. Fix a ballot b' such that ρ'_1 does not contain a* $s.\mathsf{send\text{-}batch}(ms, u)$ *with* $\mathtt{VOTE}(b', \mathit{true}) \in ms$*. By* *line 24 of Algorithm 4, we know that* $b' = \mathsf{max\text{-}vt\text{-}prep}$, *and by lines 4–7 of the same figure,* s *did not send* $\mathtt{VOTE}(\textsc{prep}\ b'')$ *for any* $b'' > \mathsf{max\text{-}vt\text{-}prep}$. *By definition of* σ, *a batch event with* $s.b'.\mathsf{send}(\mathtt{VOTE}(b', \mathit{false}), u) \notin \sigma(\tau)$ *for* $b' > b$. *As* $b \notin \mathsf{Bllts\text{-}vt\text{-}cmt}$, *again by definition of* σ, *a batch event with* $s.b'.\mathsf{send}(\mathtt{VOTE}(b', \mathit{true}), u) \notin \sigma(\tau)$. *Therefore we know that the Boolean* voted *is false. Hence, the condition in line 4 of the same figure is satisfied. Since* $s.\mathsf{vote\text{-}batch}([b', \phi(\rho_1') < b' \leqslant b], \mathit{true})$, s *triggered* $b'.\mathsf{vote}(\mathit{true})$, *appending an event* $s.\mathsf{send\text{-}batch}(ms, u)$ *with* $\mathtt{VOTE}(b, \mathit{true}) \in ms$ *results in the prefix of a trace entailed by an execution of ASCP by line 6. We can reason in the same fashion for every* b' *in* $\phi(\sigma(\tau)) < b' \leqslant b$, *and therefore and by BNS* $\rho_1' \cdot \rho_e'$ *is a trace entailed by an execution of ASCP.*

Case $e = s.\mathsf{receive}(\mathtt{VOTE}(\textsc{cmt}\ b), u)$. *By assumption the network does not create or drop messages, hence* s *receives* $\mathtt{VOTE}(\textsc{cmt}\ b)$ *only after* u *previously sent the same message and the prefix* τ_1' *contains the event* $u.\mathsf{send}(\mathtt{VOTE}(\textsc{cmt}\ b), s)$. *The definition of* σ *entails that the simulated prefix* $\rho_1' = \sigma(\tau_1')$ *contains an event with* $u.\mathsf{send\text{-}batch}([\mathtt{VOTE}(b, \mathit{false})], s)$. *By induction hypothesis* ρ_1' *is the prefix of a trace entailed an execution of ASCP. Let the sub-trace that simulates event* e *be* $\rho_e' = s.\mathsf{receive\text{-}batch}([\mathtt{VOTE}(b', \mathit{true}), b' \in \{b' \mid \phi(\rho_1') < b' \leqslant b\}], u)$. *We show that* $\rho_1' \cdot \rho_e'$ *is the prefix of a trace entailed by an execution of ASCP. As* u *sent* $\mathtt{VOTE}(\textsc{cmt}\ b)$ *to* s, *we know that* s *receives* $\mathtt{VOTE}(b', \mathit{true})$ *exactly once for every* $b' \in \{b' \mid \phi(\rho_1') < b' \leqslant b\}$ *and the batch is exactly once in* ρ'. *Hence,* $\rho_1' \cdot \rho_e'$ *is the prefix of a trace entailed by an execution of ASCP.*

Case $e = s.\mathsf{send}(\mathtt{READY}(\textsc{cmt}\ b), u)$. *For any execution of CSCP with trace* τ_1', *the server* s *sends* $\mathtt{READY}(\textsc{cmt}\ b)$ *either after hearing from a quorum in line 28 of Algorithm 4, or after hearing from a s-blocking set in line 31 of the same figure. We consider both cases:*

Sub-case s sends $\texttt{READY}(\text{CMT } b)$ after hearing from a quorum.

By lines 26–28 of Algorithm 4 there exists a quorum U such that $s \in U$ and for every server $u \in U$ the server s received $\texttt{VOTE}(\text{CMT } b)$ and $b \notin \mathsf{readied}$ and $b \geqslant \mathsf{max\text{-}rd\text{-}prep}$. The definition of σ entails that the simulated prefix $\rho'_1 = \sigma(\tau'_1)$ contains an event $s.\mathsf{receive\text{-}batch}([\texttt{VOTE}(b', a), \phi(\rho'_1) < b' \leqslant b], u)$ and for every $u \in U$ such that $s \in U$ for every event $s.\mathsf{receive}(\texttt{VOTE}(\text{CMT } b), u)$. By the induction hypothesis, the simulated prefix ρ'_1 is a trace entailed by an execution of ASCP. Let the sub-trace that simulates event e be $s.\mathsf{send\text{-}batch}([\texttt{READY}(b, \mathit{true})], u)$. We show that $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP. If s received $\texttt{VOTE}(b, \mathit{true})$ from a quorum U such that $s \in U$ and readied *in Algorithm 2 is false, then by lines 7–9 in Algorithm 2, a batch with $\texttt{READY}(b, \mathit{true})$ is in ρ'_1. Assume a $s.\mathsf{receive\text{-}batch}(ms, u)$ with $\texttt{VOTE}(b, \mathit{false}) \in ms$ is in ρ'_1. By definition of σ this is only possible, if s received $\texttt{VOTE}(\text{PREP } b_u)$ for some $b_u > b$. As s processed $s.\mathsf{receive}(\texttt{VOTE}(\text{CMT } b), u)$ and as s is correct, s cannot have processed $s.\mathsf{receive}(\texttt{READY}(\text{PREP } b_u), u)$. Hence s received $\texttt{VOTE}(b, \mathit{true})$ from u, and as s has not received $\texttt{VOTE}(b, \mathit{false})$,* readied *in Algorithm 2 is false.*

Sub-case s sends $\texttt{READY}(\text{CMT } b)$ after hearing from a s-blocking set.

By lines 29–31 of Algorithm 4 there exists a maximum ballot b and a s-blocking set B such that s received $\texttt{READY}(\text{CMT } b)$ from every server $u \in B$ and $b \notin \mathsf{readied}$ and $b \geqslant \mathsf{max\text{-}rd\text{-}prep}$. The definition of σ entails that the simulated prefix $\rho'_1 = \sigma(\tau'_1)$ contains the event $s.b.\mathsf{receive}(\texttt{READY}(b, a), u)$ for $a \in \{\mathit{true}, \mathit{false}\}$ for every $u \in B$. By the induction hypothesis, ρ'_1 is the prefix of a trace entailed by an execution of ASCP. Let the sub-trace that simulates e be $s.\mathsf{send\text{-}batch}([\texttt{READY}(b, \mathit{true})], u)$. We show that $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP. We have to *show that s received* $\texttt{READY}(b, \mathit{true})$ *from a s-blocking set B and* $\mathsf{readied}$ *in Algorithm 2 is false. Then by lines 10–12 in Algorithm 2,* $s.\mathsf{send\text{-}batch}(ms, u)$ *with* $\texttt{READY}(b, \mathit{true})$ *is in* ρ'_1*. Assume s received* $\texttt{READY}(b, \mathit{false})$ *from u. By definition of σ this is only possible, if s received* $\texttt{READY}(\text{PREP } b_u)$ *for some $b_u > b$. As s processed* $s.\mathsf{receive}(\texttt{READY}(\text{CMT } b), u)$ *and as s is correct, s cannot have processed* $s.\mathsf{receive}(\texttt{READY}(\text{PREP } b_u), u)$*. Hence s received* $\texttt{READY}(b, \mathit{true})$ *from u, and as s has not received* $\texttt{VOTE}(b, \mathit{false})$*,* $\mathsf{readied}$ *in Algorithm 2 is false.*

Both cases show that for the sub-trace ρ'_e that simulates event e, the trace $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP.

Case $e = s.\mathsf{receive}(\texttt{READY}(\text{CMT } b), u)$**.** *Analogue to case* $s.\mathsf{receive}(\texttt{VOTE}(\text{CMT } b), u)$.

Case $e = s.\mathsf{propose}(x)$**.** *Straightforward by definition of σ, since τ contains* $s.\mathsf{propose}(x)$ *iff the simulated $\rho = \sigma(\tau)$ contains* $s.\mathsf{propose}(x)$.

Case $e = s.\mathsf{decide}(x)$**.** *By lines 13–14 in Algorithm 5, for any execution of CSCP with trace τ' the server s decides value x only after s triggers* $\mathsf{committed}(b)$ *for a ballot b with $b.x = x$. The definition of σ entails that the simulated prefix $\rho'_1 = \sigma(\tau'_1)$ contains the event* $s.\mathsf{deliver\text{-}batch}([b], \mathit{true})$*. By induction hypothesis ρ'_1, the simulated prefix ρ'_1 is entailed by an execution of ASCP. Let the sub-trace that simulates event e be* $\rho'_e = [s.\mathsf{decide}(x)]$*. We show that $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP. As* $s.\mathsf{deliver\text{-}batch}([b], \mathit{true})$*, s triggered* $\mathsf{deliver}(\mathit{true})$ *for ballot b, by lines 13 and 14 of Algorithm 3,* $s.\mathsf{decide}(x)$ *is in the execution of ASCP and $\rho'_1 \cdot \rho'_e$ is the prefix of a trace entailed by an execution of ASCP.*

Case $e = s.\mathsf{start\text{-}timer}(n)$**:** *By lines 15–17 of Algorithm 5 for any execution of CSCP with trace τ'_1, there exists a quorum U which is such that s receives* $\texttt{M}(\text{STMT } b_u)$ *where* $\texttt{M} \in \{\texttt{VOTE}, \texttt{READY}\}$ *and* $\text{STMT} \in \{\text{CMT}, \text{PREP}\}$ *from* *every u in U and* round $< b_u.n$. *The definition of σ entails that the simulated prefix $\rho_1' = \sigma(\tau_1')$ contains the event s.*receive-batch$([\mathtt{M}(b', \mathit{false}), b' \lesssim b_u], u)$

*for every s.*receive$(\mathtt{M}(\text{PREP } b_u), u)$ *that occurs in τ_1', or s.*receive-batch$([\mathtt{M}(b_u, \mathit{true})], u)$ *for every s.*receive$(\mathtt{M}(\text{CMT } b_u), u)$ *that occurs in τ_1'. By induction hypothesis ρ_1' is the prefix of a trace entailed by an execution of ASCP. Let the sub-trace that simulates the event e be $\rho_e' = [s.\text{start-timer}(n)]$. We show that $\rho_1' \cdot \rho_e'$ is the prefix of trace entailed by an execution of ASCP. By Lemma A.2.1 coincides* round *and by assumption $n < b_u$.*round *holds. We have distinguish two cases:*

Sub-case s received $\mathtt{M}(\text{PREP } b_u)$ from u. *The definition of σ entails that the simulated prefix $\rho_1' = \sigma(\tau_1')$ contains an event with s.*receive-batch$([\mathtt{M}_u(b_u', \mathit{false}), b_u' \lesssim b_u], u)$ *and every $\mathtt{M}_u(\text{PREP } b_u)$.*

Sub-case $\mathtt{M}_u(\text{CMT } b_u)$. *The definition of σ entails that the simulated prefix $\rho_1' = \sigma(\tau_1')$ contains a s.*receive-batch$([\mathtt{M}_u(b_u, \mathit{true})], u)$ *for every $\mathtt{M}_u(\text{CMT } b_u)$.*

*Combining the cases leads to the conditions in line 15 in Algorithm 3 satisfied. Thus, by line 17 of the same figure, s.*start-timer(n) *is in the execution of ASCP and $\rho_1' \cdot \rho_e'$ is the prefix of a trace entailed by an execution of ASCP.*

Case $e = s$.timeout: *Straightforward by definition of σ, since τ contains s.*timeout *iff the simulated $\rho = \sigma(\tau)$ contains s.*timeout.