Laboratory for Automated Reasoning and Analysis

Interchain Foundation

Master Thesis

Verifying Distributed Systems with Stainless

Author:
Stevan Ognjanovic

Academic supervisor:
Prof. Viktor Kuncak
Company supervisor:
Dr. Zarko Milosevic

August 28, 2020

Acknowledgements

I want to take this opportunity to thank the following people who have made my thesis possible.

Firstly, I would like to express my sincere gratitude to Professor Viktor Kuncak, my supervisor at EPFL. He has provided valuable insights into the world of formal verification and gave me the necessary freedom to pursue this topic in a way that was exciting for me and will hopefully be exciting for others.

I would also like to thank Dr. Zarko Milosevic, my advisor at Interchain Foundation, for his emphasis on building real-world systems, which made this effort immensely more challenging, interesting, and relevant.

Crucial people in implementing this work were Romain Ruetschi and Dr. Jad Hamza. Their understanding of the internals of Stainless was instrumental for my understanding of formal verification in Stainless.

For the TLA^+ side of things, Professor Josef Widder and Dr. Igor Konnov were kind enough to share their knowledge of both TLA^+ and specifications on which this work is based.

Besides those that provided academic guidance, I would like to thank Informal Systems for including me in their meetings, allowing me to gain valuable insights for building a successful implementation.

Other than people who helped directly during this effort, I would like to thank my brother and Felix de Souza. They are ”responsible” for why I put such a strong emphasis on software development, which spilled into the report.

I want to thank my parents for being there whenever I needed it through all ups and downs, and especially through downs which were constructs of my imagination.

Finally, I would like to thank my family and friends, you were an indispensable part of this journey.

Could not have done this without all of you!

Abstract

Verifying distributed systems is an immensely complicated task requiring substantial effort by the designers of those systems. Even if verifying and specifying the system was straightforward, any subsequent implementation introduces its own set of challenges such as performance issues, bugs, and alignment with specifications. While there is a multitude of tools designed to help with formal verification of specifications, there is limited support for verifying concrete implementations of such specifications.

This thesis explores formal verification of distributed systems at the level of implementation. Specifically, we use Stainless as a tool for formally verifying Scala programs. A substantial part of the thesis explains how we translate a specification written in TLA^+ to an implementation in a formally verifiable subset of Scala.

Firstly, we explain how programs can be verified using Stainless and the main approaches for doing that. Continuing from there, we cover the basic notions of a Core Verification protocol in the context of Tendermint blockchain Light Client we use as a case study. Finally, we describe this protocol (which we later verify and implement), and discuss its formal specification in TLA^+.

After the initial background, we show how to build executable models in Stainless based on the formal specifications. Along the way we encounter challenges related to library restrictions, non-verified code and limitations of Stainless for specifying global invariants of the protocol. We also propose solutions where they are possible, and also discuss issues surrounding formal verification of code designed to be extended.

The result of this work is a verified Scala implementation of the Core Verification protocol along with a full implementation of the Light Client. This implementation conforms to the intended behavior of the Light Client, which we assert through conformance tests. We demonstrate that it is possible to use Stainless to verify concrete implementations that can be deployed and used in the same manner as the conventionally developed software.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**
1.1 Roadmap . . . 1
1.2 Contributions . . . 2

2 Background **3**
2.1 Verifying Scala Programs . . . 3
2.1.1 Functions . . . 4
2.1.2 Inductive Proofs . . . 5
2.1.3 Classes and Objects . . . 6
2.2 TLA^+ as a Standard for Protocol Verification . . . 8
2.3 Tendermint Software . . . 8
2.3.1 Tendermint Core as Consensus Engine . . . 8
2.3.2 Tendermint Application Blockchain Interface (ABCI) for Application Agnostic State Replication . . . 9
2.4 Tendermint Light Node and Light Client . . . 10
2.4.1 Core Verification . . . 11
2.4.2 Light Client and Light Node . . . 12

3 Blockchain Model **14**
3.1 TLA^+ specification . . . 14
3.2 State Machine Formulation . . . 16
3.3 Enforcing Global Invariants Through Local Type Invariants . . . 18
3.4 Processing Events . . . 21
3.4.1 Append Block . . . 21
3.4.2 Time Progress . . . 21
3.4.3 Failures . . . 21

4 Core Verification **23**
4.1 TLA^+ Specification . . . 23
4.2 Light Store Split and State Simplification . . . 24
4.3 State Machine Formulation . . . 25
4.4 Proving Termination . . . 28
4.5 Liskov Substitution Principle as a Solution for Non-Verified Code and Performance . . . 29
4.5.1 Implementations with External Code . . . 30
4.5.2 Flexibility for Open World Design . . . 31
4.6 Abstracting for Concurrency and Limitations/Thread-Local Design . . . 34

5 Using the Verified Algorithm **35**
5.1 Communication with Blockchain . . . 35
5.2 Core Verification State . . . 36
5.3 Cryptography and Hashing . . . 36
5.4 Contracts as Hindrance on Performance . . . 37
5.5 Testing . . . 38
5.6 Light Node as Implementation of a Light Client . . . 39

6 TLA^+ to Stainless Experience **40**
6.1 Implementation . . . 40
6.1.1 Verification Statistics . . . 40
6.1.2 Set as List . . . 41
6.2 TLA^+ Guided Modeling . . . 42
6.3 Software Architecture . . . 43
6.4 Verification-Driven Development . . . 43

7 Related Work **46**

8 Conclusion **48**
8.1 Influence on Rust Implementation . . . 48
8.2 Limitations . . . 49
8.2.1 Properties in TLA^+ . . . 49
8.2.2 Verified Libraries . . . 49
8.2.3 Combining the Core Verification and Blockchain Model . . . 49
8.2.4 External Code . . . 50

A Original TLA^+ Blockchain Specification **51**

B Current TLA^+ Blockchain Specification Optimized for Apalache **59**

C Light Client TLA^+ specification **62**

Bibliography **65**

Chapter 1

Introduction

Distributed software tends to be very complex, requiring a complicated development process where teams that are developing them go through iterative steps of specification and implementation. Tendermint blockchain is an example of one such system. Tendermint is a distributed system designed for state replication with several protocols based on it. One of these protocols is Core Verification as an integral part of Tendermint Light Client. Light Client is a part of a much broader effort to expand the Tendermint blockchain ecosystem and blockchains in general. One such effort is the Interblockchain Communication Protocol (IBC) designed to provide a TCP like communication protocol between blockchain, of which Light Client is an integral part.

Having this in mind, the effort presented in this thesis is complementary to the ones made by the team implementing the Rust version of the Light Client, and the one specifying and formally verifying the core parts of the Light Client using Temporal Logic of Actions (TLA^+). This work bridges the gap between efforts in Rust and TLA^+ by implementing a formally verified implementation of Light Client Core Verification and a complete implementation of the Light Client. We are extensively using the fact that Scala is a well-known JVM based programming language providing rich library support. Using Scala, we achieve seamless interoperability with the existing Scala code. This work is located at the intersection of software verification and software development.

1.1 Roadmap

This thesis explores building formally verified programs in the context of distributed systems, using Scala as a popular and widely used programming language. The goal is to show how we can build parts of the protocols in a formally verified way adapting the existing specifications to the subset of Scala supported by Stainless program verification tool.

In the following chapters, we show how to build executable models based on existing TLA^+ specifications [1], how to deal with unverified code, global invariants, termination, and limitations of Stainless [2] formal verification tool.

In Chapter 2, we introduce Stainless, and how the Light Client and Core Verification fit into the larger Tendermint [3] and IBC ecosystem. Chapter 3 presents a case study of how to build a Stainless model from a TLA^+ specification with some of the basic techniques used more extensively in Chapter 4.

In Chapter 4, we expand on the techniques from the previous chapter to deal with issues arising from building an executable model. We demonstrate how to build verified algorithms while still being able to prove useful properties and use them in a real world setting. As a continuation of Chapter 4, in Chapter 5 we cover the parts of the code which are not formally verified but are composed with verified code to build a full system. We explain how we deal with non verified code such as cryptography, hashing, communication, testing, and reasons why this implementation is suitable for extension.

An essential part of this work is reasoning about the TLA^+ specification and building an equivalent verified implementation utilizing the well-established approaches to program design and architecture (e.g., Liskov substitution principle). This is covered in Chapter 6 alongside software modeling, and the approach to software development when formal verification is a first-class concern.

In the end, Chapter 8 covers the expected impact on the mainline Rust implementation, limitations, as well as related work.

1.2 Contributions

As a summary, the main contributions of this thesis are:

1. Stainless verified implementation in Scala of Light Client Core Verification,
2. implementation of Light Client in Scala which conforms to the expected behavior of Light Clients,
3. Light Client implementation which can be run to verify data received from a real Tendermint blockchain,
4. a detailed analysis of translation of TLA^+ specifications to a Stainless verifiable implementation, and
5. experience report and recommendations for designing runnable and verifiable Stainless implementations.

Chapter 2

Background

Formal verification is a well-known discipline in computer science and is treated by numerous papers examining different areas of possible applicability. These approaches can be broadly split into two groups:

- hardware verification, and
- software verification.

Between the two mentioned, hardware verification has been significantly more successful in influencing industry than software verification. There are two main reasons for that, first of all, verification is usually built-in the tools used for modeling, and secondly fixing bugs on already shipped hardware is almost impossible, and can result in high financial costs for the company developing it.

On the other hand, software verification is almost exclusively an elective choice. The cost of educating engineers how to use the tools for software verification and the average complexity of the software being developed in most cases make formal verification prohibitively expensive for an average company. However, there are cases where formal software verification is preferable and even necessary; such examples include software certification, mission-critical software like air-traffic control, and other high-risk software. In these cases, the cost of bugs is significantly higher and justifies the overhead associated with formal verification.

Formal verification is used extensively in distributed systems design. In this area of computer science, formal verification is almost an industry standard. The complexity of implementation needed to express the high-level goal of the distributed systems is orders of magnitude higher than in sequential algorithms providing the same capabilities in the non-distributed setting. Examples of such problems are consensus [4], distributed hash tables [5], and key-value stores [6]. In these cases, non-distributed implementations are substantially less involved as a result of existing hardware and software primitives.

However, formal verification of software programs has a varying level of success when discussing executable code. Some approaches, like TLA^+ [7], are not designed to be executable. Others like Dafny [8] and the language of Coq [9] are verification-oriented languages and require transpilation and introduce additional overhead for programmers. In the following sections of this chapter, we will describe the problem space we are working in and the tools we are using for formal verification.

2.1 Verifying Scala Programs

As the basis of the implementation, we are using a verifiable subset of Scala, which can be proven correct using the Stainless verification tool. As a result, we can use the verified code in the same manner as the unverified code. We can assume that Scala programmers would be able to reasonably quickly adapt to the restrictions imposed by

Stainless and would be able to write formally verifiable code. Existing work showes that it is possible to use Stainless to model protocols for leader elections, distributed key-value stores, and replicated counter [10], however in isolation and with verification in mind. The following examples will show the main building blocks of Stainless for implementing verified programs.

2.1.1 Functions

Code such as the one shown in Listing 2.1 can be regarded as the basic building block of any verified Scala program as it allows us to define the contracts of functions. In the given code snippet we can notice three basic constructs for specifying the function's contract:

- require - function precondition,
- decreases - termination measure of recursive calls, and
- ensuring - function postcondition.

Listing 2.1: Basic Stainless function verification.

```
def factorialFaulty(n: Int): Int = {
    require(n >= 0)
    decreases(n)
    if (n == 0) 1 else n * factorialFaulty(n-1)
}.ensuring(res => res > 0)
```

However, the example does not pass verification since Stainless is trying to confirm that there will be no runtime errors for any input. In this case, the problem is that *Int* types are standard 32bit signed integers prone to overflows resulting in the post-condition of the *factorialFaulty* violated with a sufficiently large input number. In such cases, Stainless will be able to find violations of contracts and report them to users. The report will include the location where it occurs and a counter-example for which the violation happens. In this case, it will detect an overflow resulting in a negative result.

Each of the three primary constructs for specifying the contract of the function is translated into a Verification Condition (VC). Stainless then translates these VCs to formats solvable by Satisfiability modulo theories (SMT) solvers (Z3 [11] and CVC4 [12]) using DPLL(T) solving [13] with the unfolding of function definitions and candidate model validation [14].[1] Translated VCs are then checked for satisfiability by the previously mentioned SMT solvers, the results of these solvers are used to confirm the correctness of the implementation.

To make this function pass verification, we can use two approaches: restricting the input parameter or using *BigInts* (which do not have overflows), as shown in Listing 2.2. Furthermore, using *decreases* function, we instruct Stainless to build a VC, which allows us to prove the termination of the function.

To prove termination, measures are assigned to an input parameter of a function. Measures can be numeric or a combination of numeric values and can be automatically inferred by Stainless in most cases. Stainless analyses recursive calls of a function and tries to prove that a measure decreases between two recursive calls. Developers can arbitrarily compute the measure as long as it is non-negative. Stainless assumes

[1]The approach described is for a predecessor system of Stainless called Leon however, the approach stays the same.

that when the measure reaches value 0, the function terminates. For a detailed formal description, Hamza et al. [2] provide a detailed explanation how such VCs are built.

Listing 2.2: Options for correct factorial implementation.

```
def factorialInt(n: Int): Int = {
  require(n >= 0 && n < 13)
  decreases(n)
  if (n == 0) 1 else n * factorialInt(n-1)
}.ensuring(res => res > 0)

def factorialBigInt(n: BigInt): BigInt = {
  require(n >= 0)
  decreases(n)
  if (n == 0)
    BigInt(1)
  else
    n * factorialBigInt(n - 1)
}.ensuring(res => res > 0)
```

2.1.2 Inductive Proofs

A powerful feature of Stainless is writing inductive proofs over recursive data structures. Such proofs are essential when dealing with recursive data structures such as *Lists* and can be used to prove properties with almost zero overhead for the user of the tool. We would advise users to use automatic inductive proof capability explained in the rest of this section, and only if that does not work help by guiding the proofs. Using inductive proofs can help with proving lemmas used for more complex parts of the code. For example, we can formulate a lemma:

- *"Trying to remove a number from a list of positive numbers will decrease the sum of all the elements in the resulting list or the sum will stay the same if the value is not contained in the list."*

Writing that for Stainless would result in Listing 2.3 (with the definition of the *sum* function).

Listing 2.3: Removal lemma with no help.

```
def sum(list: List[BigInt]): BigInt = {
  require(list.forall(value => value > 0))
  list match {
    case Nil() => BigInt(0)
    case head :: tl => head + sum(tl)
  }
}

def removalLemma(elem: BigInt, list: List[BigInt]): Unit = {
  require(list.forall(value => value > 0))
}.ensuring(_ => sum(list - elem) <= sum(list))
```

If we were to try and run Stainless to prove the *removalLemma*, we would get two timeouts, one on the precondition of *sum(list - elem)* and another for the postcondition of *removalLemma*. We have to help Stainless verify this lemma; we first try to help

Stainless prove the postcondition by adding an *@induct* on the list parameter. After this addition in Listing 2.4 Stainless manages to prove the postcondition, while still timing out on the precondition of *sum(list - elem).* The reason for this outcome is that it will be able to transform the function in a way that will allow the generation of VCs such that it is possible to prove the postcondition. However, when applying the operation of element removal, Stainless cannot automatically deduce that all the elements of the resulting list will still be positive. As a result, Stainless will not be able to prove the precondition of the *sum* method without additional help.

Listing 2.4: Guiding remove lemma with *@induct.*

```scala
def removalLemma(
    elem: BigInt,
    @induct list: List[BigInt]): List[BigInt] = {
  require(list.forall(value => value > 0))
}.ensuring(_ => sum(list - elem) <= sum(list))
```

Now we are in a situation where we can prove the intended lemma if we can prove another one:

- *Trying to remove an element from a list of positive numbers, results in a list of positive numbers.*

Proving this lemma separately and introducing it to the context as in the Listing 2.5 allows us to prove the original lemma.

Listing 2.5: Helping Stainless with an intermediate lemma.

```scala
def helperLemma(elem: BigInt, @induct list: List[BigInt]): Unit = {
  require(list.forall(value => value > 0))
}.ensuring(_ => (list - elem).forall(value => value > 0))

def removalLemma(elem: BigInt, @induct list: List[BigInt]): Unit = {
  require(list.forall(value => value > 0))
  // brings the postcondition of the lemma to the context
  helperLemma(elem, list)
}.ensuring(_ => sum(list - elem) <= sum(list))
```

The process described here is the primary building block of the composition of proofs when working with Stainless, used whenever proofs do not succeed because of timeouts. Usually, timeouts mean that lemma is provable; however, Stainless cannot find a good enough proof strategy for it, and it needs some guidance, as shown in *removalLemma.*

2.1.3 Classes and Objects

Stainless supports Scala objects as top-level grouping constructs, abstract classes for interface specification, and case classes/objects. It is possible to specify contracts on methods of those types in the same manner as for functions in general. As an added feature, we can specify invariants over concrete implementations, allowing us to statically prove that invariants will be respected throughout all the possible executions which use those classes.

In Listing 2.6 is an example of object-oriented constructs which can be used to build verifiable programs. *Height* class is a basic example of the procedure for specifying invariants and methods in classes. On the other hand, *RoundUp*, class hierarchy shows how to structure our code so that we can implement different strategies for certain operations while still being able to reason about them. *SimpleRoundUp*, in itself, does not maintain the contract of the *roundUp* specified in the parent class.

One might think that this could be a problem when verifying. However, Stainless applies a strategy where it verifies that on the call sites that expect a *RoundUp*, the concrete implementations will behave in the manner defined by the abstract base class. Such an approach is valid as we statically prove that the behavior is maintained for all implementations of the abstraction. If we were to call the *roundUp* method directly on the concrete implementation, Stainless would use the contract of the particular method being called and base the verification on it instead of the contract of the abstract class.

Listing 2.6: Stainless for object-oriented constructs.

```
object Example {
    case class Height(value: BigInt) {
        // invariant of Height ADT and can be arbitrarily complex
        require(value > 0)

        def +(diff: BigInt): Height = {
            // standard precondition on the function
            require(diff >= 0)
            // invariant automatically proved
            Height(value + diff)
        }
    }

    abstract class RoundUp {
        def roundUp(value: BigInt): BigInt = {
            // precondition for all implementations of roundUp when
            // called using virtual dispatch
            require(value > 0)
            ??? : BigInt
        }.ensuring(res => res % 10 == 0 && res >= value)
        // postcondition for all results of function implementation
    }

    case object SimpleRoundUp extends RoundUp {
        override def roundUp(value: BigInt): BigInt = {
            value + (10 - (value % 10))
        }
    }

    def differenceInTypeBehaviour() {
        val concrete = SimpleRoundUp
        // no problem
        concrete.roundUp(-1)
        val interface: RoundUp = SimpleRoundUp
        // applies the contract of abstract method
        interface.roundUp(-1)
    }
}
```

2.2 TLA^+ as a Standard for Protocol Verification

Temporal Logic of Actions (TLA^+) has been an industry standard for a long time for formalization of algorithms and protocols in sequential problem space and, more importantly, in concurrent and distributed one. Introduced by Leslie Lamport, it provides support for specifying safety and liveness properties [1] of protocols.

In industry, TLA^+ has been used very successfully for formally verifying protocols such as Paxos [4], Pastry [5], 2-phase commit to name a few. Using TLA^+ for these protocols was instrumental in finding protocol errors and confirming their correctness. After the successful application of TLA^+ to these problems, such specifications are almost a necessary prerequisite for any new protocol which guarantees specific properties and wants to give confidence to the implementation.

However, TLA^+ specifications tend to be very abstract compared to concrete implementation and do not necessarily prevent bugs during the implementation phase. For example, a consensus problem can be specified in just several lines. [2]

As a result, formally verifying the protocol does not guarantee that the implementation will be bug-free or that the invariants, as specified in the formally verified protocol, are correctly implemented. A recent example of such issues is Jepsen's report regarding the Redis-Raft early build implementation [15] of a well-specified protocol proved with TLA^+. In the report the Jepsen team found several sever bugs, one of those being a total data loss.

2.3 Tendermint Software

In this section, we will cover what is Tendermint, what it aims to solve, and some of the necessary prerequisites for understanding later sections. The descriptions are based on the publicly available documentation[3] for Tendermint and represent a digest of the most important concepts without going into details about each one.

In a nutshell, Tendermint is a software for safely and consistently replicating applications across multiple processes. To be able to do that, Tendermint is split into two main components:

- Tendermint Core in 2.3.1, and
- Tendermint Application Blockchain Interface (ABCI) in 2.3.2.

2.3.1 Tendermint Core as Consensus Engine

Tendermint Core is the consensus engine of the Tendermint software stack and performs Byzantine Fault Tolerant State Machine Replication for an arbitrary deterministic state machine. We can decompose the previous sentence into three parts:

- Byzantine Fault Tolerant - the guarantees it provides,
- State Machine Replication - the functionality it provides, and
- for an arbitrary deterministic state machine - requirements imposed on the application running on top, implementing Tendermint ABCI.

[2]`https://github.com/tlaplus/Examples/blob/master/specifications/Paxos/Paxos.tla`
[3]`https://docs.tendermint.com/master/`

In the Byzantine [16] Fault Tolerant consensus systems, we assume that process can exhibit arbitrary faults, including abrupt termination, hacking, and malicious behaviors. The network model in which we operate assumes partial synchrony [17]. Under these preconditions, it is necessary for more than 2/3 of all process to be correct to be able to implement state machine replication.

In the Tendermint case, processes have an assigned voting power proportional to their monetary investment into the blockchain. Monetary investment into the blockchain, otherwise known as stake, is used to monetary incentivize correct behaviour of processes. This approach for implementing process accountability is called Proof of Stake. To strengthen the partial synchrony assumption of the network model, Tendermint assumes an unbonding period during which a malicious process can be held accountable.

Processes that participate in consensus protocol are called validators. As previously mentioned, every validator participating in consensus has an associated voting power representing its stake in the blockchain.

For the process to append a block to the blockchain, a consensus has to be reached among a set of validators. This set is assigned for each blockchain's height and can change arbitrarily between two adjacent heights of the blockchain. In Tendermint, this means that validators who agreed on a value need to hold more than 2/3 of total voting power in the current validator set.

When talking about safe and consistent replication what we mean is that:

- safety - is maintained as long as less than 1/3 of total voting power is held by faulty validators, and
- consistently - all correct processes will see the same ordering of decided values in their logs.

Values, in the case of Tendermint Core, are a sequence of transactions grouped in a structure called a block. Therefore, Tendermint Core reaches a consensus on the execution of transactions in terms of a chain of blocks. As mentioned previously, every correct process will see the same ordering of decided values in their logs, now using Tendermint specific terminology; each correct process will see the same order of decided blocks which constitute a blockchain.

For the following sections, it is essential to mention a Full Node with which a Light Client communicates. A Full Node is a node that maintains an up-to-date consensus state.

2.3.2 Tendermint Application Blockchain Interface (ABCI) for Application Agnostic State Replication

ABCI allows possible users to write their applications in any language they chose, giving them the ability to use the consensus engine almost seamlessly. Consequently, users are no longer restricted as they are with similar blockchain technologies such as Etherum, Bitcoin, and others, requiring custom languages and other similar approaches.

However, there are certain restrictions on what can and can not be implemented on top of Tendermint Core. The main limitation is the determinism of transactions submitted to Tendermint Core. In situations where transactions are non-deterministic, state replication using the ordering of transactions makes no sense. Even though we have a total order on the execution, the results of applying those transactions over different processes can vary, breaking the state replication.

2.4 Tendermint Light Node and Light Client

There are several essential concepts necessary for understanding the problem space, in square brackets are other concepts explained in this list:

- *BlockHeader* - one block of a blockchain containing the information related to one consensus height, [*Header*, *LastCommit*],
- *Header* - contains metadata about the block and consensus, [*ValidatorSetHash*, *NextValidatorSetHash*],
- *ValidatorSetHash* - hash of the validators which should sign the current block, analogous for *NextValidatorSetHash*,
- *LastCommit* - consensus information for the previous Block, represented as a *Commit* data structure,
- *SignedHeader* - pair of (*Header*, *Commit*), where *Commit* is taken from the *Last-Commit* field of the next Block,
- *LightBlock* - tuple of (*Header*, *Commit*, *ValidatorSet*, *NextValidatorSet*) used by Light Clients to perform verification [*ValidatorSet*, *NextValidatorSet*],
- *ValidatorSet* - set of validators used to sign the Header of the *LightBlock*, analogous for *NextValidatorSet*,
- Core Verification - logic for verifying *LightBlocks*,
- Fork Detection - process of detecting divergence in chain information from different Full Nodes, otherwise known as forks,
- Evidence Submission - process of informing Full Nodes of existence of forks,
- Light Client - represents the logic for verifying *LightBlocks*, combining Core Verification, Fork Detection and Evidence Submission, and
- Light Node - wrapper over a Light Client providing users with a useful API.

Blockchain Light Clients, like the one for Tendermint, are a well-known concept in the world of blockchains since the Bitcoin white-paper [18] known as Simplified Payment Verification. Given that they require significantly less computational power and bandwidth to verify the consensus process of Full Nodes, they are an indispensable part of blockchain ecosystems and are widely used for state synchronization. Light Clients need to be adapted for different blockchains depending on the security model, and in this chapter, we will cover the specifics of the Tendermint Light Client and Core Verification.

Light Clients are a crucial part of building Interblockchain Communication Protocol (IBC). The general goal of IBC is to allow different blockchains to communicate with each other to decentralize authority and allow greater flexibility for users of blockchains. To enable this, Light Clients are a central part that will allow synchronization of state between blockchains. In Figure 2.1, is an example of one such blockchain network.

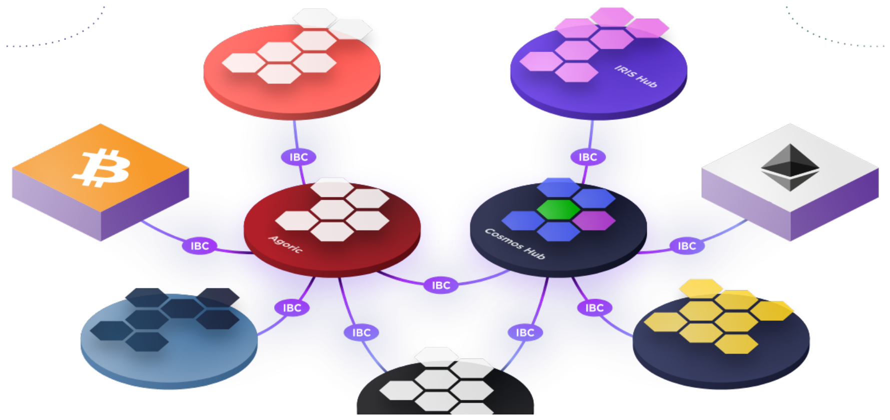

FIGURE 2.1: Internet of Blockchains

2.4.1 Core Verification

Light Client Core Verification is one part of the Simplified Payment Verification specific to the failure model in which a blockchain is supposed to operate (Tendermint in this case). The failure model of Tendermint on which Light Client depends, assumes the two following properties:

1. the sum of voting power of correct validators is more than 2/3 of the total voting power of validators at a certain blockchain height, and
2. there is a trusting period during which the property (1) holds such that, trusting period is less than the unbonding period from 2.3.1

Core Verification deals with verifying that the data we have received from a Full Node is valid concerning the failure model. Having that in mind, it is the Core Verification that depends the most on the blockchain with which it communicates.

Core Verification is a module that checks the validity of blocks, it is a process that communicates with one Full Node of the blockchain, as shown in Figure 2.2. Considering that it is communicating with a single Full Node, it can be defined as a sequential process, as is the case in the official English language specification [4] and the formal one in TLA^+ [19]. A more detailed discussion about the properties of the algorithm, invariants, and guarantees it relies upon are covered in Chapter 4.

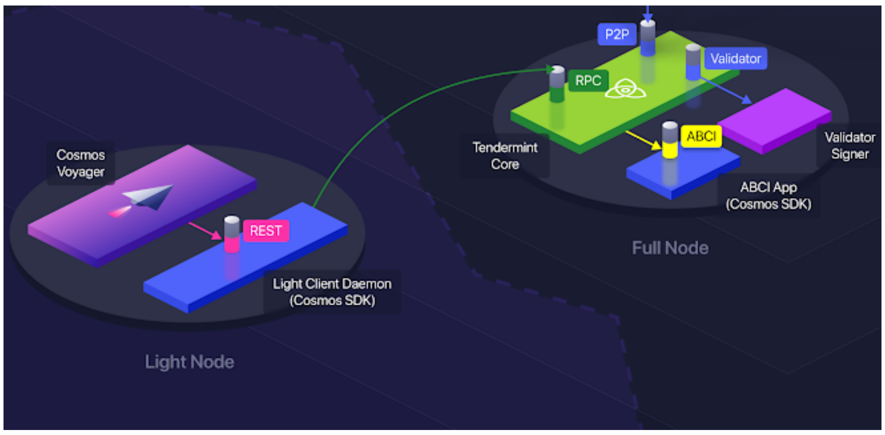

FIGURE 2.2: Tendermint Stack

[4]https://github.com/informalsystems/tendermint-rs/blob/0657d2746730c45f462a4d6f607f64e4eb8c7f66/docs/spec/lightclient/verification/verification.md

2.4.2 Light Client and Light Node

As mentioned previously, Light Node is the wrapper around the logic provided by a Light Client broken down into Core Verification, Fork Detection, and Evidence Submission. The decomposition is designed to give more confidence to the users interested in *LightBlocks* of a blockchain. Primary source of this increased confidence is the Fork Detection module. This module uses several instances of Core Verification, each one querying different Full Node of the blockchain, verifying the *LightBlocks* it receives using the Core Verification logic and comparing them between different Full Nodes.

With Fork Detection, we reduce the chance that the Light Client communicates with a faulty Full Node. As a result of no detected forks, *LightBlocks* can be considered correct for future use. We can summarize Core Verification and Fork Detection as:

- Core Verification - confirms that a certain *LightBlock* obtained from some Full Node forms a chain with respect to a given predecessor *LightBlock* at a lower height, and
- Fork Detection - confirms that a set of *LightBlocks* for the same height are identical, such that: (1) each block in this set is received from a different Full Node, and (2) each block passes Core Verification given the same predecessor *LightBlock*, and (3) the "identity" function applies to the *Headers* of the *LightBlocks*.

In situations when the Fork Detection manages to detect a conflict, Light Client informs Full Nodes of such cases. The process of sending the verification traces which we believe to be from a fork is called Evidence Submission. With the supplied evidence, Full Nodes can go through the process of Fork Accountability, during which Full Nodes try to penalize misbehaving nodes. Full Nodes are generally dependant on others, giving them the information on the occurrence of forks, so Light Clients are instrumental in this sense.

Listing 2.7: Core Verification algorithm taken from the English language specification.

```
func VerifyToTarget(
    primary PeerID,
    lightStore LightStore,
    targetHeight Height)
        (LightStore, Result) {
    nextHeight := targetHeight

    for lightStore.LatestVerified.height < targetHeight {
        // Get next LightBlock for verification
        current, found := lightStore.Get(nextHeight)
        if !found {
            current = FetchLightBlock(primary, nextHeight)
            lightStore.Update(current, StateUnverified)
        }

        // Verify
        verdict = ValidAndVerified(lightStore.LatestVerified, current)

        // Decide whether/how to continue
        if verdict == OK {
            lightStore.Update(current, StateVerified)
        }
        else if verdict == CANNOT_VERIFY {
            // do nothing the light block current passed validation,
            // but the validator set is too different to verify it.
            // We keep the state of current at StateUnverified.
            // For a later iteration, Schedule might decide to try
            // verification of that light block again.
        }
        else {
            // verdict is some error code
            lightStore.Update(current, StateFailed)
            // possibly remove all LightBlocks from primary
            return (lightStore, ResultFailure)
        }
        nextHeight = Schedule(lightStore, nextHeight, targetHeight)
    }
    return (lightStore, ResultSuccess)
}
```

Chapter 3

Blockchain Model

Formal specification of the Light Client protocol is split into two parts:

- simplified blockchain model as modeled from the perspective of one Full Node, and
- Core Verification mentioned in Section 2.4.1 and explained in more detail in Section 4.

In this chapter, we will explain the blockchain TLA^+ specification in more detail, explain the process of modeling an equivalent Stainless program from it, and cover in more detail how to deal with events. The model in this section follows the initial specification from Appendix A. [1] While taking into account the insights from the latest one in Appendix B [2] which optimizes for speed of verification using Apalache [20], and does not include events such as *BlockHeader* append and faults. However, as the Stainless model is based on the previous version, we are also modeling those.

3.1 TLA^+ specification

TLA^+ specifications of protocols define state machines in terms of possible events and invariants on those states. However, those states, transitions, and events are not expressed in terms of clear data types but in terms of transition formulas over the variables and constants of the modules. The definition of blockchain follows the same standard. Consequently, it is much easier to expose the global state and prove invariants in a specification language that allows global control of the state, as is done in TLA^+.

In Listing 3.1, we see some of the basic building blocks of TLA^+ blockchain specification. There are constants like *ULTIMATE_HEIGHT*, variables like *height*, and types such as *Height* and *BlockHeaders*. As a continuation of the the basic type constructs there are transition function like *AdvanceChain*, invariants *NeverStuck*, and helper functions *TwoThirds* shown in Listing 3.2. Using these basic constructs the users are able to build specifications like the ones in the Appendix.

Listing 3.1: Constants, variables, and types in TLA^+ blockchain specification. (comments are removed)

```
--------------- MODULE Blockchain ---------------
EXTENDS Integers, Sequences

Min(a, b) == IF a < b THEN a ELSE b
```

[1] taken from https://github.com/informalsystems/verification/blob/c7b834ef06103ba83fb07ec6825ed0fa2920a821/spec/light-client/Blockchain.tla

[2] taken from https://github.com/informalsystems/tendermint-rs/blob/45ee6b620abd2f4de3a45389bd94ed68847d4a14/docs/spec/lightclient/verification/Blockchain_A_1.tla

```
CONSTANT
  AllNodes,
  ULTIMATE_HEIGHT,
  MAX_POWER

Heights == 0..ULTIMATE_HEIGHT

Powers == 1..MAX_POWER

Commits == SUBSET AllNodes

BlockHeaders == [
  height: Heights,
  lastCommit: Commits,
  VP: UNION {[Nodes -> Powers]: Nodes \in SUBSET AllNodes \ {{}}},
  NextVP: UNION {[Nodes -> Powers]: Nodes \in SUBSET AllNodes \ {{}}}
]

VARIABLES
    tooManyFaults,
    height,
    minTrustedHeight,
    Faulty
```

Listing 3.2: Invariants, helper functions, and transition functions

```
TwoThirds(pVotingPower, pNodes) ==
    LET TP == PowerOfSet(pVotingPower, DOMAIN pVotingPower)
        SP == PowerOfSet(pVotingPower, pNodes)
    IN
    3 * SP > 2 * TP

AdvanceChain ==
  /\ height < ULTIMATE_HEIGHT /\ ~tooManyFaults
  /\ AppendBlock
  /\ UNCHANGED <<minTrustedHeight, tooManyFaults, Faulty>>

NeverStuck ==
  \/ tooManyFaults
  \/ height = ULTIMATE_HEIGHT
  \/ minTrustedHeight > height \* the trusting period has expired
  \/ ENABLED AdvanceChain
```

To check the TLA^+ specifications, it is necessary to bound the model space; *CONSTANTS* are used as a solution for this. As the approach to verifying TLA^+ specification is based on model checking in which the full state space is explored, bounding the model with constants is unavoidable for tractable verification. Based on the initial state constraints, tools will try to build the starting state of possible models and try to find all possible reachable states based on the possible transitions on those states. This process continues until the final state is reached, model is bounded with a termination height. It is possible to improve the performance of the verification by using symbolic model checking with Apalache [20], and in that case, it is possible to check more complex space with less memory and time overhead

For this specific model, there are three main reasons for transition along with corresponding transition functions and event names in the Stainless model:

- advance time - *AdvanceTime* as *TimeStep*,
- append block - *AppendChain* as *AppendBlock*, and
- node failure - *OneMoreFault* as *Fault*.

Furthermore, there are three primary states which can be modeled based on the combination of constants and variables in the TLA^+ model, the names of the states are as implemented in the Stainless model:

- Running - Tendermint failure model is maintained and the final height has not been reached as,
- Faulty - there are more faults in the system than Tendermint failure model can handle and we have ultimate height has not been reached, and
- Finished - ultimate height has been reached.

Other parts of the specification are the invariants of the system, such as *NeverStuck*. However, properties like that one are not easily translatable to Stainless and very hard to prove. In the case of TLA^+, these invariants are checked on constructed states, while in the case of Stainless, the SMT solvers need to find models that can prove this property holds, which can be very hard to do. In certain situations, such properties are more natural to model depending on the guarantees they provide. In TLA^+, they can be split into two main classes: (1) safety, and (2) liveliness properties. When using Stainless, we analyze every property separately and find the approach best suited for modeling it.

3.2 State Machine Formulation

Translating the models from TLA^+ to formally verifiable Scala code requires a more verbose translation as model checking specifications with TLA^+ is more powerful and more general than what is immediately expressible in Stainless, as global checks maintain invariants over the states built by the tool.

Different qualitative states implicitly defined in the specification are explicitly defined in the Stainless code as is visible in the Listing 3.3. In the example, global invariants are maintained as invariants over the fields of the case classes. Along with the states of the machine, the second crucial aspect of it are the transitions. We are defining those in terms of events corresponding to transition functions in the TLA^+ specification.

In Table 3.1, we see the pairings of possible events with a given current state. The main takeaways from the table are that there is a termination state when a blockchain reaches predetermined target height, in which all events are ignored.

The defined steps are one part of the definition of the model transitions. In the Listing 3.4, we can see events defined in terms of the data they provide when such an event happens. However, the validity of those events is not enforced on this level as that is strictly dependant on the current blockchain state. For example, an event to append a block is not relevant in the *Faulty* state, as are all events in the *Finished* state. In the TLA^+ specification, valid events are restricted based on the constants of the model, as such approach is not reasonable in Stainless we are explicitly filtering events which are not valid by the current state of the blockchain.

Listing 3.3: States as defined in the blockchain model.

```
sealed abstract class BlockchainState {
    // State transition method.
    def step(systemStep: SystemStep): BlockchainState
}

// State of the blockchain where the chain maintains
// the blockchain failure modeland did not reach the final height.
case class Running(
    allNodes: Set[Address],
    faulty: Set[Address],
    maxVotingPower: VotingPower,
    blockchain: Blockchain)
        extends BlockchainState {
    require(
        runningStateInvariant(
            allNodes,
            faulty,
            maxVotingPower,
            blockchain))
    ...
}

// Blockchain is violating the failure model.
case class Faulty(
      allNodes: Set[Address],
      faulty: Set[Address],
      maxVotingPower: VotingPower,
      blockchain: Blockchain)
        extends BlockchainState {
    require(
        faultyStateInvariant(
            allNodes,
            faulty,
            maxVotingPower,
            blockchain))
    ...
  }

// The final height is reached.
case class Finished(
    allNodes: Set[Address],
    faulty: Set[Address],
    blockchain: Blockchain)
        extends BlockchainState {
    require(finishedStateInvariant(allNodes, faulty, blockchain))
    ...
}
```

Listing 3.4: Events as defined in the Stainless model.

State/Event	Advance Time	Append Block	Node Failure
Running	+	+	+
Faulty	+	-	+
Finished	-	-	-

TABLE 3.1: Valid State/Event pairs

```
// Abstract definition of a possible system step.
sealed abstract class SystemStep

// Models time progress, where duration is always
// a positive time delta (enforced as invariant on duration).
case class TimeStep(timeDelta: Duration) extends SystemStep

// Event representing a node fault in the system.
case class Fault(node: Address) extends SystemStep

// Simplified AppendBlock message where all validators have
// voting power of 1. Minimal data included needed to append a block.
case class AppendBlock(
    lastCommit: Commit,
    nextValidatorSet: ValidatorSet)
        extends SystemStep {
    require(
        nextValidatorSet.values.forall(_.votingPower.value == 1) &&
        lastCommit.forBlock.nonEmpty)
}
```

3.3 Enforcing Global Invariants Through Local Type Invariants

Enforcing invariants at the level of TLA^+ can sometimes be a straightforward process, for example:

1. define initial state, and
2. define transition functions.

With this, the user is done. Other than the initial invariant on the starting state, there might be an arbitrary number of invariants held by reachable states. For example, examining the specification the only thing we know about the chain of blocks is that it is a sequence of *BlockHeaders*, after we continue our analysis of initial state and *AppendBlock* function in Listing 3.5 used in the *AppendChain* we can infer the following simple invariant:

- *For every pair of adjacent block headers in a sequence(chain), the BlockHeader's height field with a higher index is greater by one than the height field of the BlockHeader with a lower index.*

Furthermore, in combination with the initial state where the starting *BlockHeader* is of height 1, we can prove the following invariant:

- *For every BlockHeader such that chain[i] == block, than block.height == i. (where indexing of the chain starts from 1).*

As a result of that invariant, we can be sure that when we request a *BlockHeader* of a certain height we can index the chain with the given value and get a *BlockHeader* of that height. However, these two mentioned invariants are not explicitly stated in TLA^+, but hold as a result of state exploration. Unfortunately, Stainless can not explore the whole context of execution and deal with global information of the model.

Listing 3.5: *AppendBlock* transition function.

```
AppendBlock ==
  LET last == blockchain[Len(blockchain)] IN
  \E lastCommit \in SUBSET (VS(last)) \ {{}},
     NextV \in SUBSET AllNodes \ {{}}:
     \E NextVP \in [NextV -> Powers]:
    LET new == [ height |-> height + 1, lastCommit |-> lastCommit,
                 VP |-> last.NextVP, NextVP |-> NextVP ] IN
    /\ TwoThirds(last.VP, lastCommit)
    /\ IsCorrectPower(Faulty, NextVP)
    /\ blockchain' = Append(blockchain, new)
    /\ height' = height + 1
```

To solve this problem in Stainless, this information is encoded at the type level. For example, there are two possible options: (1) as an invariant on a chain implemented as a *List*, or (2) a custom data type. In this situation, as the invariant of the chain can be defined recursively, an interesting approach would be to solve this problem with a recursive custom data structure, as shown in code snippet 3.6.

Listing 3.6: Chain model.

```
// Type representing the chain (parts are ommited).
sealed abstract class Chain

// Fixes the root sort to have header of height 1.
case class Genesis(blockHeader: BlockHeader) extends Chain {
    require(blockHeader.header.height == Height(1))
}

case class ChainLink(
    blockHeader: BlockHeader,
    tail: Chain)
        extends Chain {
    require(
        // height needs to be increasing in steps of 1
        blockHeader.header.height == tail.height + 1 &&
        // time needs to be increasing
        blockHeader.header.time > tail.head.header.time &&
        // Tendermint assumption about adjacent headers.
        blockHeader.validatorSet == tail.head.nextValidatorSet)
}
```

From the specification, we can see that the inferred invariants mentioned in TLA^+ are explicitly stated in the require clause of *ChainLink* along with some other assumptions. With this, Stainless can automatically build chains with the stated property holding on the whole chain as any model built for this type has to maintain the invariants of the *Chain* subtypes. As a result, we can say that Stainless maintains the same implicit higher-level invariant about the chain blocks and heights. We can also verify such an assumption by trying to verify the method in Figure 3.7, proving the correctness of this method; we can see that the *height* of the *BlockHeader* returned corresponds to the perceived index.

On the other hand, some invariants are somewhat more explicit in TLA^+ but still hold by transitions into that state rather than by any hard check. For example, transition into a *Faulty* state from *Running* can happen only after an additional *Fault* event has been processed. In this sense, the Stainless specification is the same as the TLA^+. However, the fault condition in TLA^+ will always hold in a faulty state due to transition history. At the same time, *Faulty* in Stainless can not count on the transitions to implicitly infer that the invariant will hold. Specifically, the invariant of the *Faulty* state:

- *For all BlockHeaders created in the trusting period, there is at least one BlockHeader for which voting power held by correct validators is less or equal than* 2/3 *of the total power of the validators.*

In this situation, the invariant on the state is modeled as a condition on the case class constructor parameters, as shown in Listing 3.8.

LISTING 3.7: Example of a method which depends on the type invariant.

```
def getHeader(height: Height, chain: Chain): BlockHeader = {
    require(height <= chain.height)
    chain match {
        case Genesis(blockHeader) => blockHeader
        case ChainLink(blockHeader, tail) =>
            // the postocndition can only be proved if
            // chain.height == res.header.height
            if (height == chain.height) blockHeader
            else    getHeaderInternal(height, tail)
    }
}.ensuring(res => res.header.height == height)
```

LISTING 3.8: Example of an invariant on an implementation of an abstract class.

```
// Blockchain is violating the failure model.
case class Faulty(
      allNodes: Set[Address],
      faulty: Set[Address],
      maxVotingPower: VotingPower,
      blockchain: Blockchain)
        extends BlockchainState {
    require(
        faultyStateInvariant(
            allNodes,
            faulty,
            maxVotingPower,
            blockchain))
    ...
}
```

3.4 Processing Events

Event processing is the final part of the Stainless state machine blockchain model. As previously mentioned, we are modeling three transition functions in TLA^+, each having a 1-1 mapping to the Stainless specification. Depending on the event, processing might require different proofs. The maintainability of certain invariants is not immediately provable by Stainless, an example being: *Detecting a fault event does not recover the blockchain and does not transition the system into a Running state.* Such invariants needed additional guidance in a similar manner as shown in Section 2.1.2. Since the *Finish* state does not have any meaningful transitions, it is not covered in this section.

3.4.1 Append Block

The event which models appending of a new block is only possible in the *Running* state. In the case of the *Finished* state, append does not make any sense, while for Faulty, we say that blockchains in that state are frozen until they leave that state by time progressing, as explained in Section 3.4.2. As a result, *AppendBlock* is an event by which the blockchain will either stay *Running* or transition into the Finished state. This property is not only intuitively correct, but Stainless manages to prove it.

3.4.2 Time Progress

As mentioned previously, this event has no semantic value in the *Finished* state. Time progress, as modeled here, represents the point in time for which all blocks with creation time after it maintains the invariant of the Tendermint blockchain failure model. As such, time progress can influence the model in the following ways:

1. in the *Running* state after time progresses, depending on the time delta, the height of the highest block after which all blocks maintain the invariant can increase or stay the same, and

2. in the *Faulty* state, the trusting window can move past blocks violating the failure model resulting in the state transitioning to *Running*.

From the previous description, we can see that time progress in case of *Running* state does not result in a state transition, while for *Failure*, we can either stay in the same state or recover and move to *Running*. Stainless proves the validity of this assumption.

3.4.3 Failures

Node failures are the only events that can bring the blockchain state into a situation where the Tendermint failure assumption does not hold. From the perspective of the concrete implementation of Full Nodes in Go [3] detection of failed nodes is unfeasible, and is not explicitly implemented because of the nature of the distributed systems. In this sense, failures, as modeled here, are purely a verification construct. Considering the way failure assumption of the blockchain is defined, we can describe the expected behavior after this event:

1. a blockchain already in a *Faulty* state can not recover if another fault is detected, and

2. *Running* chain may move into *Failure* state or stay *Running* depending on the failure condition of the blockchain.

[3] https://github.com/tendermint/tendermint

As with the previous transitions, these are also explicitly proved by Stainless.

Chapter 4

Core Verification

Core Verification is the primary protocol we are verifying. Its goal is to confirm that *LightBlocks* received from a Full Node, as described in Chapter 2, are valid under the assumptions based on the blockchain we are communicating with, in this case Tendermint. Compared to the last section, the model described here is very different in terms of the end goal. While the blockchain model is strictly a modeling artifact and not designed for execution, while changes to make it such are not impossible. This model can be used as an executable and formally verifiable implementation of Core Verification. In this section, we will cover how the TLA^+ specification in Appendix C [1] formalizes this protocol, differences in it compared to the Stainless model, modeling termination of Core Verification, and external code. We are not verifying network communication and possible storage solutions as they are outside the scope, and we assume them to be correct.

4.1 TLA^+ Specification

The original TLA^+ specification of Core Verification is modeled as a sequential process abstracting away the IO calls for *LightBlocks*, assuming that such calls will not block indefinitely. This approach is the most straightforward solution when using TLA^+, with the whole state of the world implicitly global with direct access to it. In this sense, the definition of Core Verification is contained in one transition function, in Listing 4.1, representing one iteration of the verification loop from the English spec. For tracking the validity of the algorithm, a *LightStore* is used to store the *LightBlocks* retrieved from the blockchain instance in three states:

- unverified - *LightBlock* for which it is currently not possible to confirm that it originated from the same blockchain as the current highest verified *LightBlock*,
- verified - initial *LightBlock* (trusted in the spec) and all *LightBlocks* for which we are able to prove that originated from the same blockchain as the initial *LightBlock*, and
- failed - *LightBlock* which violates the blockchain assumptions about a correct chain.

Unverified *LightBlocks* drive the loop of the Core Verification algorithm. The protocol is trying to retrieve more and more intermediate ones until we can make a decision. The specification uses the *LightStore* for two purposes: (1) cache of fetched light blocks, and (2) state over which invariants of the Core Verification are to be proven.

[1] taken from https://github.com/informalsystems/tendermint-rs/blob/45ee6b620abd2f4de3a45389bd94ed68847d4a14/docs/spec/lightclient/verification/Lightclient_A_1.tla

4.2 Light Store Split and State Simplification

Compared to the original formal specification, the Stainless model is simplified while maintaining the power to express the same algorithm and valid states of the *LightStore*. The goal of this simplification is to make the verification process simpler for Stainless. We can adapt it so that we can prove useful properties like termination, improve performance, and make reasoning about the implementation more natural. Having that in mind, we analyzed the specification and reached the following conclusions:

1. *LightStore* can be split in two, trace of verified *LightBlocks* (*VerificationTrace*) and a stack of unverified ones *FetchedStack*,
2. *Verification* is modeled as a prepend only list for which we can peek into the top element, which always has at least one element,
3. prepend to a list of verified *LightBlocks* succeeds only if the height of the current top element is lower that the height of the one we are trying to append,
4. unverified *LightBlocks* are stored on a stack which allows us to peek, pop and push,
5. height of the top element of verified list is always lower than the height of the top element of the stack storing unverified headers,
6. *CanScheduleTo* given heights $h1$ and $h2$ where $h1 + 1 < h2$ returns a height res such that $h1 < res < h2$, and
7. there can only be one failed *LightBlock*, as the situation when we decide that a *LightBlock* should be failed is the situation when we can exit the loop and return it as the reason for failure.

Abstracting the *CanScheduleTo* with *HeightCalculator*. We can tune the worst-case performance of the number of checks done between initial *LightBlock* and target height in terms of n, where n is the difference in height between the starting height and the target height:

- returning the height at the midpoint between the two heights *LightBlocks* (bisection) will result in the worst case complexity $O(n * log(n))$, and
- returning the height which is greater by one than the lower of the two heights, the worst case complexity is $O(n)$.

However, as IO is also an imporant contributor of time complexity, the algorithm achieves worst-case performance in this sense when the algorithm needs to retrieve all *LightBlocks* between the initial *LightBlock* and target height to check the target *LightBlock* successfully.

In Figure 4.1, we see an example of the state of the components guiding the verification process. The arrows represent the ordering of heights of *LightBlocks*, such that directions of the arrows point to increasing heights. The differences of those heights can vary depending on the *HeightCalculator* used. Currently, we provide two versions of the calculator, one which returns height greater by one than the base height and the second one, which returns the midpoint between the two.

In the *VerificationTrace* part of the figure, the arrows also have an additional meaning. They signify a trust link established during Core Verification based on the failure model of the blockchain.

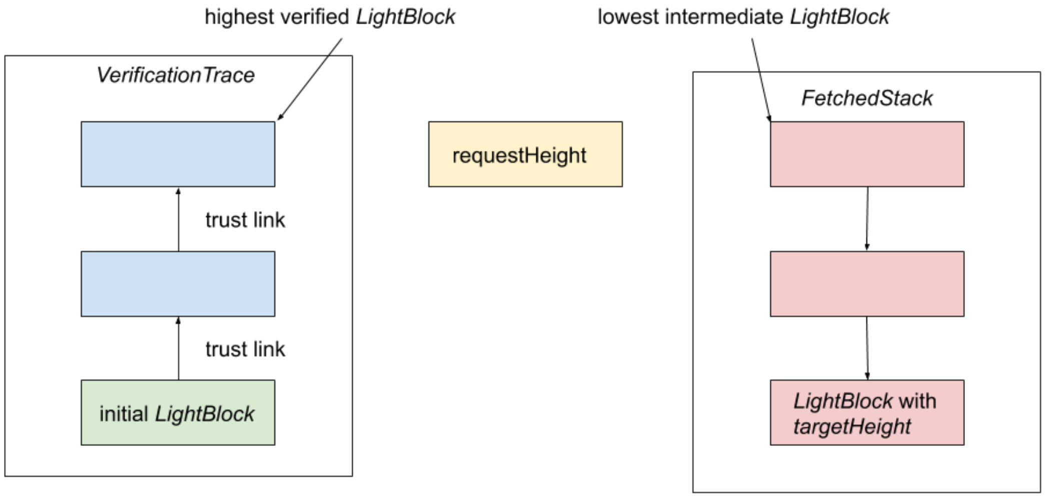
highest verified LightBlock
VerificationTrace
trust link
trust link
initial LightBlock
requestHeight
lowest intermediate LightBlock
FetchedStack
LightBlock with targetHeight

FIGURE 4.1: Usual state of *VerificationTrace* and *FetchedStack* during execution.

4.3 State Machine Formulation

The state machine of Core Verification is influenced by the IO requests for new *LightBlocks*. As such we formulate two states:

- *Finished* - a verdict has been reached, and
- *WaitingForHeader* - verification is waiting for an intermediate header which is not present locally.

Comparing this to the TLA^+ formulation, we notice a more decomposed approach than the one in *VerifyToTargetLoop* from Listing 4.1. In the sense that in *WaitingForHeader*, after receiving the requested header, we try to prepend as many *LightBlocks* as possible from the stack of unverified blocks to the *VerificationTrace* before making a decision, or requesting an intermediate *LightBlock*. On the other hand, TLA^+ specification only recognizes one iteration of a loop that encompasses all options.

Another significant difference in writing an executable Core Verification model is the checks which are used to show that a *LightBlock* is from the same chain as the previous ones. In the TLA^+ case, it is straightforward to solve this problem as one can use global variables and constants to write these checks directly in terms of the blockchain module instance. When implementing this in Stainless, the check is expressed in terms of data validity using hashes and cryptography. Consequently, we are checking hashes and signatures of *LightBlocks* instead of explicitly checking if it originated from a blockchain. As such, the properties proved in the TLA^+ are based on end state of the verification.

The state of the Core Verification model is split into two parts, as described previously, a trace of verified *LightBlocks* and stack of the ones which were fetched but could not yet be successfully verified. We can say that verification is the process of improving the trace such that it reaches the target height. To achieve this, the process of Core Verification is trying to establish trust, in terms of the blockchain model, between *LightBlocks*, such that at the end we can say that we can trust that a *LightBlock* with target height originates from the same blockchain as the one which we used as the starting point.

Between a previously verified and the *LightBlock* currently verified, we recognize two situations:

1. adjacent (two consecutive headers) - the next validator set of the verified one needs to be the same as the validator set of one being verified, or
2. non-adjacent - in this case we are using the failure model of Tendermint.

LISTING 4.1: Example of an invariant on an implementation of an abstract class.

```
VerifyToTargetLoop ==
   /\ latestVerified.header.height < TARGET_HEIGHT
   /\ \E current \in BC!LightBlocks:
      /\ IF nextHeight \in DOMAIN fetchedLightBlocks
         THEN
            /\ current = fetchedLightBlocks[nextHeight]
            /\ UNCHANGED fetchedLightBlocks
         ELSE
            /\ FetchLightBlockInto(current, nextHeight)
            /\ fetchedLightBlocks' = LightStoreUpdateBlocks(fetchedLightBlocks, current)
      /\ nprobes' = nprobes + 1
      /\ LET verdict == ValidAndVerified(latestVerified, current) IN
         CASE verdict = "OK" ->
            /\ lightBlockStatus' = LightStoreUpdateStates(
                                     lightBlockStatus,
                                     nextHeight,
                                     "StateVerified")
            /\ latestVerified' = current
            /\ state' =
                  IF latestVerified'.header.height < TARGET_HEIGHT
                  THEN "working"
                  ELSE "finishedSuccess"
            /\ \E newHeight \in HEIGHTS:
               /\ CanScheduleTo(newHeight, current, nextHeight, TARGET_HEIGHT)
               /\ nextHeight' = newHeight

         [] verdict = "CANNOT_VERIFY" ->
            /\ lightBlockStatus' = LightStoreUpdateStates(
                                     lightBlockStatus,
                                     nextHeight,
                                     "StateUnverified")
            /\ \E newHeight \in HEIGHTS:
               /\ CanScheduleTo(newHeight, latestVerified, nextHeight, TARGET_HEIGHT)
               /\ nextHeight' = newHeight
            /\ UNCHANGED <<latestVerified, state>>

         [] OTHER ->
            /\ lightBlockStatus' = LightStoreUpdateStates(
                                     lightBlockStatus,
                                     nextHeight,
                                     "StateFailed")
            /\ state' = "finishedFailure"
            /\ UNCHANGED <<latestVerified, nextHeight>>
```

For the non-adjacent case, we rely heavily on the failure model of Tendermint. As the validator sets can change from one *LightBlock* to the next, we need to find a way to establish trust between the two non-adjacent blocks. For this check, we are using the properties of the Tendermint failure model. In the rest of the explanation we assume that the checks related to time, specifically execution inside the trusting period, have been performed.

To prove that the higher *LightBlock* is from the same blockchain as the one we currently believe to be from a valid chain, we are using the set of validators that committed the higher block and the next validators of the *LightBlock* we believe to be valid. The validators at the intersection of the two sets need to hold more than 1/3 of the voting power in the next validator set of the predecessor *LightBlock*. If we pass this check, we can say that the set of validators that committed the *LightBlock* we are testing has at least one correct validator. As this validator is also present in the commit of the *LightBlock* we are testing, we can say that all correct validators decided in line with the one we believe to be correct. By the Tendermint model, we can prove that all validators which committed this *LightBlock* are correct.

As a way of proving that only blocks that can be proven to originate from the same chain are inserted into the list of checked *LightBlocks*, the data structure rejects any attempts to insert a *LightBlock* which can not pass this check. This property is proved by Stainless to always hold in the implementation we have in the model.

Suppose we look at the invariants from the TLA^+. In that case, we notice that some of them, such as *StoredHeadersAreVerifiedOrNotTrustedInv*, are defined using direct checks based on the correctness of the peer (Full Node) we are communicating with and the state of the said peer. As a result, the checks that it provides are more robust than the ones we have in our model.

Listing 4.2: An example of an invariant which bases its checks on the global state.

```
StoredHeadersAreVerifiedOrNotTrustedInv ==
  state = "finishedSuccess"
    =>
    \A lh, rh \in DOMAIN fetchedLightBlocks:
      \/ lh >= rh
      \/ \E mh \in DOMAIN fetchedLightBlocks:
        lh < mh /\ mh < rh
      \/ "OK" = ValidAndVerified(fetchedLightBlocks[lh], fetchedLightBlocks[rh])
      \/ ~BC!InTrustingPeriod(fetchedLightBlocks[lh].header)
```

For example, specifying properties in terms of the node's correctness is impossible to achieve, as one can not assume anything about the correctness property of the peer in the code. Also, checking the *LightBlocks* with the state of the blockchain we are supposedly communicating with is unreasonable. To solve this, as previously mentioned, we assume that we can check hashes and signatures. These checks are predicated on the blockchain model in which we assume that faulty (Byzantine) peer can not forge signatures and hashes, which are used to detect malicious *LightBlocks* and Full Nodes which provided them.

One can say that compared to the TLA^+ specification, the one we have now described is less powerful. However, the core algorithm is of the same power but adapted to the problem space in which the model should be formally verified. We can also argue that with a trace of verified *LightBlocks* rejecting the ones which it can not prove to originate from the same blockchain, this model is less error-prone than the one with a *LightStore* when talking in terms of implementation. The main downside of this implementation is the impossibility to check the existence of accepted *LightBlocks* on the main chain.

4.4 Proving Termination

An essential property of every algorithm is its termination and the condition under which it terminates. In this case, the algorithm's termination requires that the target height is lower than the current height of the blockchain on the Full Node used to retrieve *LightBlocks*. Under this precondition and the guarantees provided by the correct blockchain Full Node, we can assume that all the intermediate headers we need to verify the target height exist. A Full Node may be malicious and violate this assumption. If this happens, Stainless is not able to model such violations which result in runtime errors. Users of the implementation may choose to catch these runtime errors.

Since we covered the preconditions needed for the algorithm to terminate, the second part is to prove it. In the TLA^+ case, termination can be defined in terms of temporal logic, alternatively, as is the case in the TLA^+ spec, as a deadlock which we inspect manually. Unlike the previously mentioned approach, in Stainless, it is almost impossible to prove termination using such global checks. Luckily as the Core Verification is defined as a sequential process we can use the measure *decreases* construct as presented in Section 2.1.1 to prove termination.

Now the question is: "What is the appropriate measure for proving termination?". First of all, errors, like network timeouts, are ignored as they are considered either a bug in the external code covered in Section 4.5 or as blockchain assumption violations. The measure can then be formulated as the distance to deciding if the *LightBlocks* originated from the same blockchain as the initial *LightBlock*.

Looking at how that check is defined, we are now searching for situations in which a decision has to be made, in the non-adjacent case, we can always try to get an intermediate one. However, in the adjacent case, we have to decide as there is no other way to make a decision. Therefore, we will use the difference in heights between two *LightBlocks* as a way to measure termination.

However, blindly taking the difference between the highest one in the *VerificationTrace* and the target height is not sufficient. For example, we have to prove that comparing with intermediate headers will also terminate, and then there is the question of how to show improvement of the state. To do that we are taking a two-part measure as a measure of state improvement in terms of distance to reaching the target height:

1. difference between the target and trusted height (goal measure), and
2. difference between the current intermediate height and highest confirmed *LightBlock* (decision measure)

The reason why a second part is necessary are the situations in which we are continually requesting intermediate *LightBlocks*. When this is happening, the first part of the measure is fixed, but we are still making progress as we are getting closer to the situation where we will retrieve the adjacent *LightBlock* of the highest checked one. When we retrieve that *LightBlock*, a decision can be made to either terminate or insert a new verified *LightBlock* to the *VerificationTrace* (decreasing the first part of the measure).

We are using this measure to quantify the improvement between two requests for a *LightBlock*. Assuming that the request will eventually finish and the measure decreases between two requests, we prove that the algorithm will eventually terminate. More informally, between two requests, we are closer to making a decision on the confirmation that the *LightBlocks* received originated from the same blockchain given an initial *LightBlock* of that chain. Using additional intermediate lemmas, we prove the validity of termination measure.

In Listing 4.3, we can see how this measure looks at the high level without going into the internals of Core Verification and just looking at the part which deals with the state transitions of Core Verification. In it, we see the precondition which makes the Core Verification possible, usage of measure computed in *LightClientLemmas.terminationMeasure*, call to main in-client logic for Core Verification implemented inside *processHeader* and the decision based on that outcome.

LISTING 4.3: Main Core Verification recursive function.

```scala
@scala.annotation.tailrec
private def verify(waitingForHeader: WaitingForHeader): Finished = {
    // assumption about state of blockchain
    require(waitingForHeader.fetchedStack.targetLimit <=
    lightBlockProvider.currentHeight)
    // measure
    decreases(LightClientLemmas.terminationMeasure(waitingForHeader))

    // single point of network IO
    val intermediateLightBlock =
    lightBlockProvider.lightBlock(waitingForHeader.requestHeight)

    // main verification logic
    val nextState = processHeader(
        waitingForHeader,
        intermediateLightBlock)

    // decision based on the outcome of verification
    nextState match {
      case state: WaitingForHeader => verify(state)
      case state: Finished => state
    }
}
```

Colors in Figures 4.2 and 4.3 denote different semantic types of *LightBlocks* and *Heights*: (1) red, fetched but not verified, (2) yellow, the height which we are waiting for, (3) blue, *LightBlocks/Height* verified during Core Verification, and (4) green, for the starting *LightBlock/Height*

4.5 Liskov Substitution Principle as a Solution for Non-Verified Code and Performance

An overarching problem with formal verification of programs is the restrictiveness of verified code concerning the possible industry implementation. Core Verification needs to communicate over the network with Full Nodes, process HTTP requests, and responses, and possibly store data somewhere. All of the mentioned situations are a necessary part of any industrial application of almost any problem, that is, interaction with non-verified code is unavoidable. Another approach that goes hand in hand with industrial applications is the usage of existing libraries for pervasive functionalities. In this sense, using existing libraries in which users believe in and external non-verified code in formal verification can be considered as similar approaches for solving the reusability problem and trust in "alien" implementations.

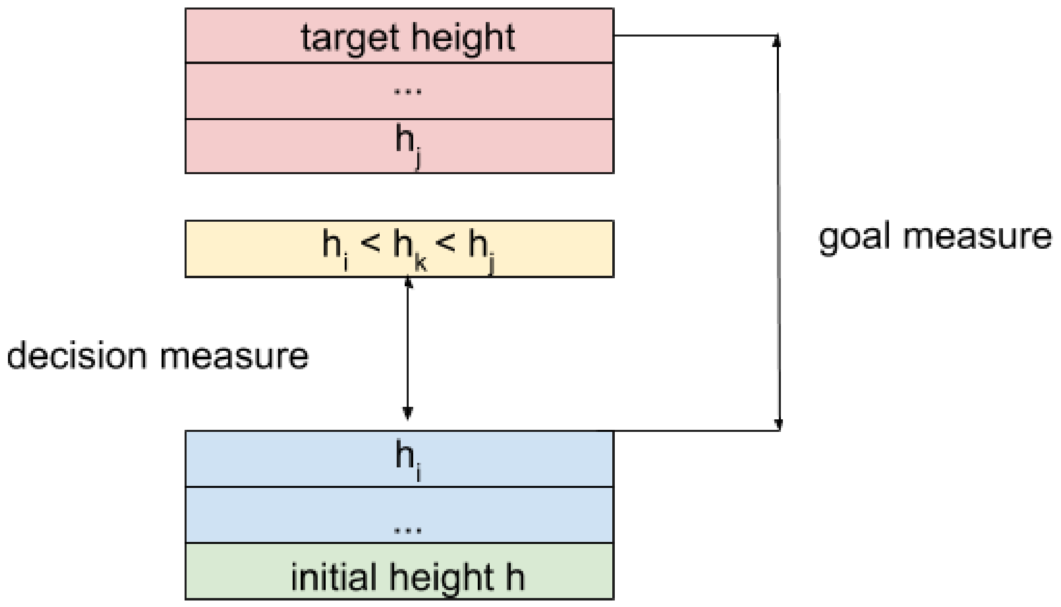

FIGURE 4.2: Simplification of state during Core Verification for measures

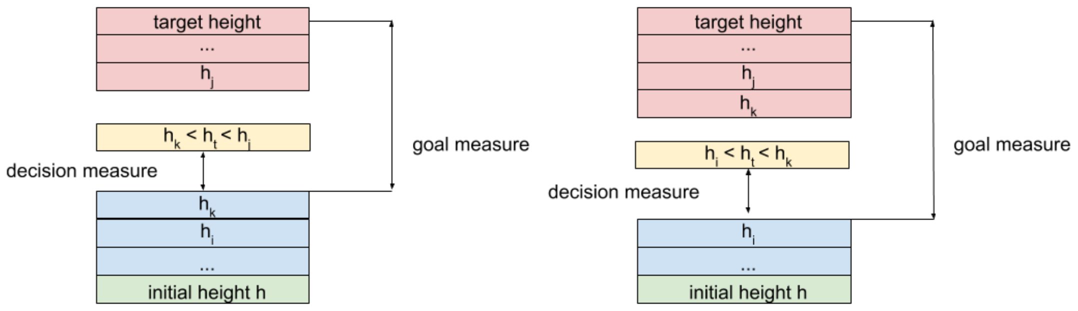

FIGURE 4.3: Two basic changes in measure differences

4.5.1 Implementations with External Code

The usual approach when dealing with external code in Stainless is to mark code either as an *@extern* or *@ignore*. Code marked as external can have pre, and postconditions, checked by Stainless and postconditions assumed to hold as shown in Listing 4.4 [2]. Also, *@extern* can be used to mark specific fields or parameters as opaque, so Stainless will not extract them. On the other hand, ignored code is invisible to Stainless, and proofs do not use it. Using *@ignore* annotation is not explored here as this functionality is used only during testing.

LISTING 4.4: Working with existing Scala code with *@extern*

```
import stainless.lang._
import stainless.annotation._
import scala.annotation.meta.field
```

[2] *@pure* is a custom Stainless annotation telling us that the function is pure in the mathematical sense.

```scala
import scala.collection.concurrent.TrieMap

case class TrieMapWrapper[K, V](@extern theMap: TrieMap[K, V]) {
  @extern @pure
  def contains(k: K): Boolean = {
    theMap contains k
  }

  @extern
  def insert(k: K, v: V): Unit = {
    theMap.update(k, v)
  } ensuring {
    this.contains(k) &&
    this.apply(k) == v
  }

  @extern @pure
  def apply(k: K): V = {
    require(contains(k))
    theMap(k)
  }
}

object TrieMapWrapper {
  @extern @pure
  def empty[K, V]: TrieMapWrapper[K, V] = {
    TrieMapWrapper(TrieMap.empty[K, V])
  } ensuring { res =>
    forall((k: K) => !res.contains(k))
  }
}
```

From the first two paragraphs, we can see that Stainless enables us to move the boundary between verified and unverified code to wherever we consider reasonable. However, this way of marking a boundary is more suited for whole functions and classes. On the other hand, this might not be the approach we want, and in general, it seems to be heavy-handed. With this approach, we are saying we are going to fix the concrete implementation and mark it as external, so we do not verify anything about it. Even though this might be a viable solution, it imposes severe restrictions on the code's usability. We are forcing the users of verified code to use specific, unverified solutions we implemented, preventing them from adapting it to their needs, as is the standard when writing reusable code.

4.5.2 Flexibility for Open World Design

To deal with situations where the implementations are not known upfront, and we need to provide the flexibility developers are accustomed to, we need to provide different solutions than those with *@extern*. The approach we are taking is defining abstract classes, which should provide the definitions of an interface and the expected behavior of the interface methods. This approach gives us the benefit of loosely coupling the algorithm's building blocks and a better understanding of necessary parts for a complete implementation.

This way of defining supertypes and subtypes is called behavioral subtyping [21], Liskov Substitution Principle(L in SOLID [3]) [22] or design by contract [23] [4]. Designing systems with this approach in mind is the industry standard in the object-oriented community. We think that such an approach is very appealing, works quite well with other approaches, which are the current understanding of software design, and allows us to provide elegant solutions for non-verified code.

Listing 4.5: Abstract TrieMap on which other formally verified code depends.

```
import stainless.lang._
import stainless.annotation._

abstract class TrieMap[K, V] {
  @pure
  def contains(k: K): Boolean = {
    ??? : Boolean
  }

  def insert(k: K, v: V): Unit = {
    ??? : Unit
  } ensuring {
    this.contains(k) &&
    this.apply(k) == v
  }

  @pure
  def apply(k: K): V = {
    require(contains(k))
    ??? : V
  }
}
```

However, this approach is not a silver bullet, incorrect implementation of abstract classes can result in the algorithm breaking in unknown ways. For example, we expect that all *LightBlock* requests for a particular height will result in responses with a *LightBlock* of the same height and always from the same Full Node. Suppose there is a bug in the implementation of the checks that are supposed to be enforced by the postcondition of the function. In that case, we will lose any guarantees related to termination, as described in Section 4.4. Such issues are an overarching problem in software design, so we do not consider this approach as detrimental or more error-prone compared to what is the current state of the art in software development.

The approach we recommend and consider very useful for confirming intuitions about the implementation is the following:

1. design core part of the algorithm in terms of abstractions with a certain behavior (can be arbitrarily complex) as in 4.5,

[3]acronym describing the five main principles of modern software design

[4]in formal theory these names are not precisely equivalent and the reader is encouraged to explore the differences, however in the industry it is clear what is the intended approach

2. prove the algorithm with abstractions and refine the necessary abstractions to provide the least amount of restrictions necessary for proving required properties,
3. implement verified instances of those abstraction (optional), and
4. implement instances of those abstractions for performance or for hiding non-verified code as in 4.6.

When following this approach, what usually happens is that the programmer will very quickly have an implementation ready for integration testing or ready for further verification. This approach is also what we used when extracting abstractions. Stainless will be able to prove the core parts' correctness, allowing developers to concentrate on drilling down the dependency graph. Managing to implement dependencies in a verified way, one would reach a situation in which unit testing is redundant, assuming the specification corresponds to the intended behavior.

LISTING 4.6: External implementation of the *TrieMap* abstraction.

```
import scala.collection.concurrent.{TrieMap => ScalaTrieMap}

// assumption is that this code is implemented
// outside of the codebase visible to Stainless
class TrieMapWrapper[K, V](theMap: ScalaTrieMap[K, V])
    extends TrieMap {
  def contains(k: K): Boolean = {
    theMap contains k
  }

  def insert(k: K, v: V): Unit = {
    theMap.update(k, v)
  }

  def apply(k: K): V = {
    require(contains(k))
    theMap(k)
  }
}
```

We recommend that users implement verifiable implementation first, if possible, for quicker integration testing and to confirm the sufficiency of abstractions. After that proceed to implement possibly non-verifiable implementations of the same abstractions with performance as the primary concern. From the verifiable implementation, users should be able to discover interesting edge cases, get a better understanding of what is needed, and possibly come up with a better solution. However, when external code is the only solution, a full non-verified solution adhering to the expected behavior and standard testing is the expected approach.

Another useful result of this approach to dealing with external code is its inherent suitability for Behavioral-driven development. The by-product of the approach is that we provide a built-in testing artifact for others to use instead of using specification tools like JBehave [5]. We can imagine Stainless being able to provide automatic test cases or other tools capable of doing that.

[5]https://jbehave.org/

4.6 Abstracting for Concurrency and Limitations/Thread-Local Design

Continuing from the previous section, we are dealing with concurrency in the same manner as with external code. Stainless can not reason about concurrent execution without custom code to deal with it, as is the case in the actor system [10]. In certain situations, we need to build code such that it provides support for concurrent execution without going through the same process as for the actor system. There are two ways of doing this, either provide thread-safe implementations that can be shared between threads or use thread-local instances and combine the results of different threads in specialized code designed for this. The question is also how to define abstractions that can be used to solve this.

The approach which is preferred, when possible, is to use thread-local instances. For example, in the Core Verification case, the only possible location where one can imagine concurrent access is the list of confirmed *LightBlocks* and the stack of previously fetched ones. In the implementation, we assume that the list of checked is strictly thread-local and that no concurrent thread can change it. Suppose we can build a *FetchedStack*, which can be shared between threads such that it maintains invariants imposed by the sequential execution. In that case, we could imagine a situation where other threads can help us by inserting *LightBlocks* as long as the view we are using is adhering to the expected behavior. Under these conditions, our verified implementation guarantees termination and correct execution concerning the pre and postconditions on method invocations and the profs we have.

The implementation allows for concurrent executions of Core Verification over the same Full Node to help each other by storing the headers they downloaded. Using those headers from the stack as a way of helping with performance. However, the preferred approach is to use a thread-local approach without help from the other concurrent Core Verification instances:

- implementation is easier to reason about,
- it is harder to violate the behaviour of abstractions,
- it is not expected to have multiple concurrent Core Verification instances over the same Full Node, and
- sharing of *LightBlocks* becomes essential only when thinking about performance related to network requests, and that can be solved at the IO level with much simpler semantics (this is the approach we are taking and implemented in external code).

Chapter 5

Using the Verified Algorithm

Continuing from Section 4 where we explained how the Core Verification is implemented and formally verified in this section we will cover the solutions for external code 5.1, 5.2, 5.3, how it was tested 5.5, and the implementation of Light Node 5.6 as a proof of usability of the verified components. Throughout these sections, we will also cover what are the libraries used and how they are abstracted away so that they do not interfere with verified code.

5.1 Communication with Blockchain

Communicating over a network with a Full Node requires a HTTP client, which is used to query the Full Node for *Headers*, *Commits*, and *ValidatorSet*, information out of which we construct *LightBlocks*. This functionality would require a significant effort to verify formally; another solution is to write an implementation from scratch and test it in a standard way. Both of those approaches are not recommended when dealing with problems encountered by people all around the world. Usually, recommended approach is to use the widely used libraries providing implementations for what we need without significant extra effort. We have decided to use an existing library for HTTP requests and implement this functionality on top of it treating all of that code as external, abstracted away by a *LightBlockProvider*. However, we do assume some invariants on the interface level:

- it always communicates with the same Full Node - from English spec, and
- if it successfully returns the *LightBlock* it is for the requested height.

While the first requirement is the assumption that the rest of the code does not depend explicitly, the second one is necessary for proving the Core Verification algorithm's correctness.

This abstraction also proved useful when testing Core Verification and in general Light Node as both of them directly depend on this abstraction. Abstracting providers away allowed us first to verify the core algorithm and swap the providers between the one communicating with Full Nodes and the one providing the *LightBlocks* locally. The second one was used for conformance tests offered by the tendermint-rs repository.

Background library used for RPC requests over HTTP is sttp [1] and circe [2] for decoding JSON responses, both of those being actively maintained and popular libraries in the Scala community. However, the choice of these libraries was arbitrary. It can be easily changed as the provider is not a core part of the algorithm, other than the previously mentioned expected properties. Users of the *LightBlockProvider* can also decide to decorate it with caching capability and can use provided *CachingLightBlockProvider*

[1] https://github.com/softwaremill/sttp

[2] https://github.com/circe/circe

which uses Caffeine [3] for caching. To test the Full Node provider's implementation, we used a docker [4] container supplied by Tendermint [5] to confirm that it is possible to use the provider in a real-world situation.

5.2 Core Verification State

As mentioned previously in the 4.2 the state of the Core Verification in the TLA^+ specification is mostly based on the *LightStore* concept, however, the implementation we propose is somewhat different and in this section, we will explain how to get the same behavior. Unlike the implementation with the *LightStore*, assuming that during Core Verification, the algorithm will access *LightBlocks* in it based on the given height, the access pattern we use is the one explained in the 4.2. This access pattern allows us to make a more restricted model that is easier to formally verify and have better abstractions while still being able to express the same algorithm as before. In this sense, violated invariant *StoredHeadersAreVerifiedOrNotTrustedInv* in the TLA^+ is now no longer relevant and the equivalent correct property is now achieved through correct by construction types.

It is possible to model both the *VerificationTrace* and *FetchedStack* as abstractions over the *LightStore* such that each insert into the trace is an update of a *LightBlock* to a verified state. At the same time, pushes into the fetched stack are also inserts into the *LightStore* as unverified blocks. As such, it is straightforward to implement both states on top of *LightStore*. However, such implementations are not verified as they do not provide significant improvement to the algorithms' understanding.

5.3 Cryptography and Hashing

A fundamental building block of blockchains is cryptography and hashing, from purely cryptography-based like Bitcoin, whose Proof of Work model of reaching consensus is based on cryptographic primitives. Others like Tendermint use the same primitives for security reasons related to their Byzantine Fault Tolerance. As a result, it is a necessary part of the implementation to provide support for such operations and checks. Unfortunately, formally verifying cryptographic primitives is out of the scope of this report, and doing that would require a significant dedicated effort. We approach this problem in the same manner as in the case of *LightStoreProvider* using the already existing libraries. An essential aspect of Tendermint is that all hashing and cryptographic primitives are defined in terms of the corresponding encoding of types defined using Amino encoding [6]. Luckily instead of writing everything from scratch, the subset we are interested in is the same as the ProtocolBuffer for which Tendermint provides the necessary protobuf files for which we have support in Scala.

The central cryptographic primitive used when dealing with signatures necessary to verify *LightBlock* commits is the Ed25519 [24] signature algorithm. For this, we are using the Google Tink library [7] as it provides the necessary primitive and is well maintained and has excellent documentation.

[3] `https://github.com/ben-manes/caffeine`

[4] `https://www.docker.com/`

[5] `https://hub.docker.com/r/tendermint/tendermint/`

[6] Tendemint specific encoding mostly compatible with ProtocolBuffer, which is currently being removed and in future releases will be switched to pure ProtocolBuffer.

[7] `https://github.com/google/tink`

Hashing is an integral part of the Tendermint used to provide Merkle proofs for the validity of data. With that in mind, we implemented a Merkle tree computation algorithm, which computes a balanced Merkle tree for an array of byte arrays provided as its input. This implementation is not verified as the gains of verifying the implementation are not very useful as we are using external libraries for computing hashes.

5.4 Contracts as Hindrance on Performance

Proving correctness of code using formal verification with Stainless will incur performance penalties compared to solutions that would use general Scala. Firstly, the data structures that can be used to prove properties about the code do not always have the best possible performance. Secondly, checking all pre and postconditions during execution will result in more time spent on ”useless” computation. These two problems can be solved to a degree by using different Stainless constructs and functions provided by the library.

The recommended approach to dealing with data structures that could be problematic concerning performance is to use of external code, as explained in Section 4.5. However, in the most general case, we can not count on users to correctly implement those abstractions. If we want to guard against this, we need to reconfirm those assumptions with *asserts* in the code, even though we have ”proved” the implementation to be correct. Another option is to implement delegating classes which check the contracts during calls.

From a correctness perspective, this is a risk as now we have a different situation. Stainless has formally proved that the code is correct but the users of it might get a false sense of security as runtime exceptions can still happen as a result of contract violations.

On the other hand, using *require* and *ensuring* and spraying *asserts* all around the code will result in more work for Stainless, more cluttered code, larger jars, more branching in code, and general performance degradation. To combat this, Stainless provides *StaticChecks* and ghost code, which go hand in hand.

The two mentioned constructs provided by Stainless allows us to mark code as only visible during verification. Such code is invisible for the Scala compiler and will not be present in the executable. Simply looking at this, one can say that performance issues are solved, ”Mark everything purely verification related as ghost and be done with it.” However, such an approach is problematic from the correctness perspective. Removing all those checks removes any defense from the code, making it completely reliant on correct usage.

As a result, marking checks as only visible during verification is not a straightforward task as it might look like on a first glance. However, there are situations where it is possible to mark them as ghost and remove them:

1. code not interacting with external code with assumption that inputs are previously checked,

2. asserts used to speed up and help verification,

3. type invariant checks inside the methods of the same type for which encapsulation is achieved, and

4. in general type invariant checks inside verified code not interacting with non verified code.

From the list of cases where removal of checks is safe, one can see that the main guiding principle when thinking about their removal is the interaction with non-verified code. If one can isolate parts of the code to use only statically verified code, all of the checks are redundant other than ones at entry points to that code. Unfortunately, that is hard to achieve in the most general case and can only be applied to private code in libraries.

Another thing to have in mind is the importance of how type invariants impact performance, summed up as ”Not all invariants are created equal.” Splitting them into two groups:

- invariants which can be specified inductively, and
- invariants which can not be specified inductively.

In situations in which invariants are specified inductively, they may have negligible performance overhead. When this is not possible, it is highly likely we need to check the whole data structure. An example of a data structure specified with a ”cheap” invariant is the *Chain* from 3.6.

5.5 Testing

Testing when combining formally verified and non-verified code is an important subject. In this case, we assume that the Core Verification algorithm is implemented correctly and that formal verification is sound. As a result, we have proof that the Core Verification will not result in a runtime error for any valid input. Furthermore, pre and postconditions could raise exceptions such that users of the implementation should be prevented from doing any serious harm to the systems.

As a result, we are mostly concentrated on integration testing or unit testing of external code used for cryptography and communication over the network. Tests are very well contained and quite clear with almost no mocking of data structures. One can look at how we test our code and realize that when using Stainless and design by contract. It is possible to omit any mocking at the level of Core Verification and let Stainless prove that we would get the correct output for all possible inputs. [8] Such an approach is beneficial as the manual work of writing mocks, and other code specific for testing can be removed and leave only the executable model as a testing artifact along with the output from Stainless.

Although it is possible to use Stainless to prove the correctness of the verified parts, we are still at the mercy of correct implementations of external code. To solve this, we are using standard approaches to testing. To that end, we either used specifications of algorithms[9], constructed test cases manually or used JSON files provided by tendermint-rs [10].

In the end, we used conformance tests from tendermint-rs to confirm that the complete implementation behaves in a manner we would expect. During those tests, we noticed no errors in the parts of the code that were formally verified. For example, the Core Verification worked fine, invariants over measures were correct, and formally verified code exhibited satisfactory behavior as one would expect from a formally verified implementation. The bugs discovered during testing were related to errors in logic related to checks of *LightBlock* hashes validity, the order of those checks, and possible misunderstanding of properties related to the unverified code.

[8] for situations where the implementations of dependencies are correct

[9] we used the official RFC examples for Merkle tree computation

[10] https://github.com/informalsystems/tendermint-rs

5.6 Light Node as Implementation of a Light Client

As previously mentioned, Core Verification is just a part of the final functionality needed for a full IBC implementation. The complete implementation is a Light Node capable of communicating with multiple Full Nodes of the blockchain and use the information provided by all of them to make a final decision on the validity of the *LightBlocks*. One might ask why Core Verification is not enough; the reason for that is a malicious Full Node. If we are communicating with only one Full Node of the blockchain, we have no other information, and we trust it without any additional checks. However, blindly trusting one Full Node is not advisable as we do not know if it is correct. Proving that we have communicated with correct nodes is done by contacting other Full Nodes of the blockchain and retrieving the *LightBlocks* from them and cross-checking with the previously received ones.

Cross-checking of those *LightBlocks* is otherwise called Fork Detection. It is implemented as a comparison of hashes of target headers and based on the result, we decide on the existence of forks. Unlike Core Verification, Fork Detection does not have a detailed formal specification in TLA^+ but an informal one in English. Fork Detection is not verified because the specification in itself does not have any specific properties which would prompt us to implement this functionality using Stainless. However, the *PeerList* of nodes used for keeping the relationships between the node we are communicating with does have a precise formal specification, which we implemented in Stainless and proved correct the intended behavior

From the description of Fork Detection in 2.4.2, it is possible to run multiple Core Verification instances in parallel and synchronize during the comparison of *LightBlocks* for detecting forks. This observation is currently ignored and is left as a future improvement of the solution.

The final part that is Evidence Submission is currently not implemented as at the time of the writing is part of an active discussion and is left for the future. However, we know that Evidence Submission should collect the evidence of forks and inform full nodes that Light Client has detected the existence of a fork. After that, the Full Nodes should go through the process of Fork Accountability (also area of active discussion) and penalize the misbehaving nodes.

Chapter 6

TLA^+ to Stainless Experience

As we have seen from the previous sections modeling Stainless verified models based on TLA^+ specifications is not always a straightforward task. The differences between the two come from the level of abstraction provided in TLA^+ and Stainless, and specifications in TLA^+ are written in a way which tends to be either non-feasible or impractical when confronted with verified Stainless code. One of the main differences between the two are properties requiring global access, which are not always possible to specify in Stainless. Properties like eventual guarantees of termination can be impossible to formulate in Stainless unless a termination measure can be specified.

6.1 Implementation

The implementation [1] is currently split in three subprojects, *light-client-core*, *light-client*, and *tendermint-general*. Each of these is semantically based on which functionality they provide, (1) *light-client-core* - verified implementation of Core Verification and blockchain model, (2) *light-client* - implementation of Light Node and external code, and (3) *tendermint-general* - code which in general can be used with Tendermint (ProtoBuff, Merkle tree and hashing). Lines of code in each of the subprojects are visible in the Table 6.1 without the code for testing.

Module	Lines of Code
light-client-core	1957
light-client	1000
tendermint-general	593

Table 6.1: Lines of code.

Other than the *light-client-core* module, which we assume to be correct by verifying with Stainless, the other two modules have corresponding tests using ScalaTest [2] framework. For confirming the correctness of the whole implementation, we have used the conformance tests from the tendermint-rs repository, managing to pass all of the Light Client conformance tests successfully. Using the docker image provided by the Tendermint repository, we were also able to write integration tests proving the usability of the implementation in a setting as close to the real-world as possible.

6.1.1 Verification Statistics

Verification is contained in the *light-client-core* in which both the blockchain model and Core Verification are contained. Our standard configuration for Stainless is shown in

[1] https://github.com/OStevan/Light-Client-Stainless

[2] https://www.scalatest.org/

Listing 6.1. We also provide a script for running verification if Stainless is installed locally, *run.sh*. For reference we are using Z3 version 4.7.1 as an SMT solver and Stainless version 0.7.1 with support for the sbt [3] plugin.

LISTING 6.1: Standard Stainless configuration.

```
vc-cache = false
timeout = 300
check-models = true
print-ids = false
batched = true
fail-invalid = false
infer-measures = true
check-measures = true
type-checker = true
```

In the Table 6.2, we see the timing results depending on the enable Stainless features when run using GitHub actions [4] and on Lenovo ThinkPad P51 with Intel Core i7-7820HQ, 16GB of RAM and Ubuntu 20.04.1 LTS. As one can see from the output of verification, using Stainless, for this implementation, does not introduce prohibitive time overheads, which would turn users away from using the tool for this specific implementation. Stainless does not provide the option to disable the insertion of automatic checks, so for the last row, we had to count the number of VCs manually; therefore, there are no timing results.

Features	Number of VC	GitHub	P51	P51 with cached VCs
all	578	153.843	116.030	21.375
no measures	537	158.878	106.717	14.32
no generated checks	505	/	/	/

TABLE 6.2: Number of VC and time spent on verification.

A less user-friendly part of working with Stainless results from working with sources without incremental compilation. As such every time verification is started, the pipeline goes through the process of compilation of sources. [5] Therefore when running the command *sbt 'set lightClientCore/stainlessEnabled := true; compile'*, which runs verification using sbt, all sources (including Stainless library) are compiled and verified, taking 147 seconds even with caching enabled. One possible solution for this problem is mentioned in Section 8.2.2.

6.1.2 Set as List

During the writing of the verified code, we used Stainless *List* data type extensively instead of *Sets*. There were several reasons for this, but in terms of verification the main ones were easier proof-writing with inductive proofs, and *Lists* proved more stable, with one comment in code being a prime example *//TODO if isCorrect is inlined this lemma can be proved, but calls to isCorrect at other locations fail verification.*

As such, we have extracted this functionality along with the proofs about *Lists*, which were general enough to be useful for others. This work resulted in a PR for Stainless with around 300 lines of proofs about *Lists*, and list backed set called *ListSet*.

[3]https://www.scala-sbt.org/

[4]https://github.com/features/actions

[5]One can use Stainless watch mode but it is not always what a user wants.

Expanding the space of verified implementations such as this will be crucial for the Stainless ecosystem. Such proofs can later be composed to build other proofs. Some might be specific to the implementation, but some may be useful for others, and we encourage such contributions.

6.2 TLA^+ Guided Modeling

Modeling programs based on an already defined TLA^+ specification poses different challenges than merely implementing a program from scratch. For example, the assumption is that TLA^+ formulation is also an artifact that can guide the modeling of software. One can argue that implementing software based on it without later formally verifying that code is much easier than also building the same software with formal verification in mind. For example, when building non verified implementation, one is concentrated on mapping the specification to code as close as possible. However, whether that code is verifiable is not taken into account.

In such situations, engineers do not have additional restrictions imposed by tools used for verification. Specifically, they are not restricted by the subset of the language they are using, and trust in the implementation is achieved through a combination of the non-executable specification and testing. This is contrary when verified code is a first-class concern. In such situations, the engineer has to be familiar with both TLA^+ and the tool for program verification. The main difference when comparing TLA^+ and Stainless is the encoding of correct behaviour in correct by construction types.

There is a qualitative difference in the way we are writing specifications in Stainless when compared to TLA^+. As a result of this, the protocol analysis, when making a verifiable executable model, requires a more detailed analysis of allowed states in TLA^+. Results of such an analysis are the implementations of *Chain*, *VerificationTrace*, and *FetchedStack*, where we need to introduce additional constraints on the data types for more straightforward proofs of the properties in which we are interested.

Therefore, building executable models will result in more analysis of the core data structures as they tend to be very important when Stainless is concerned. For example, without the approach described in Section 4.2, the code would become much more verbose, and we would need additional proofs without any contribution to the understanding of the algorithm but just as a pure exercise in writing proofs. In more extreme cases, not restricting the space at the type level results in performance degradation of proofs.

Consequently, when building an executable model in Stainless, the programmer needs to be able to have sufficient knowledge of TLA^+ to be ready to depart from it while still maintaining the same properties. Combining this with a detailed analysis of behavior, we reach an approach in which the state during execution is explicitly constrained by invariants instead of checking properties on the traces of execution. One can think of the two methods as complements of each other: while in TLA^+, properties are extrapolated from behavior, in Stainless behavior depends on the correct specifications of components.

For example, in TLA^+ Core Verification specification, we prove that when a successful decision is made, there exists a chain of *LightBlocks* from the starting *LightBlock* to the target one, which proves that the target block originated from the same chain as the starting one. While in Stainless, we have a trace in which it is impossible to insert *LightBlocks*, which are not on this proof chain. As a result, the only property we are interested in is that we have reached the target height, and we maintain that property as a property of the trace.

6.3 Software Architecture

There is also a question of software architecture. The standard approach to software development is to use object-oriented programming with other paradigms used in situations where useful. However, writing TLA^+ specifications depends on a significantly different approach than what are the current best practices and expected flexibility of libraries and programs provided.

As an example of this, we are taking the *VerificationTrace* abstraction. In our case, we implemented an in-memory verified solution implemented as a recursive type. However, because we wanted to show that we can still use the *LightStore* as the underlying storage solution, we have an implementation that stores *LightBlocks* in it, and even in this case, we have a level of extensibility which is not usually present in TLA^+. Requirements when building an executable model change from a closed world environment to an open one.

Therefore, design by contract is a clean way of writing easier to reason about proofs and a way of allowing the users of our code to adapt the implementation to their needs. We can imagine situations where users might want additional logging when accessing the trace, statistics of Core Verification, tracing capability, and possibly implementing the same abstractions in different JVM based languages.

In such situations implementing an open world system which also has core components and basic data types which are formally verified introduces a set of challenges which can no longer be considered in isolation as a standard TLA^+ to executable program mapping. Some of those challenges were discussed in Section 5.

6.4 Verification-Driven Development

Software architecture design is one of the leading hot topics in software development in this century. In the last 20 years, we heard of Test-driven development (TDD), Behaviour-driven development (BDD), Agile, Extreme programming, Scrum, and others. All designed to solve one ”simple” problem, how to write correct and maintainable code in a reasonable amount of time or, more informally, ”How not to destroy your product and your finances.” These approaches have brought forward interesting codifications of approaches to problems with which the computer science community has been fighting since its inception.

To bring more confusion to the mix, most of these tend to be somewhat vague, for example, what to test (testing vs. not testing private methods), and the acceptable metric for testing. Formal verification can be seen as another tool we can use to improve this process, as such a team at Informal Systems is proposing an approach called Verification-driven development. In it, the team proposes an approach where formally verified specifications of protocols should be the primary guiding document for developing software. However, this work takes an approach that goes from VDD in terms of guiding software development to verification working as a forcing function in architectural decisions. One might think that this approach would result in code which breaks the current best practices in software development, as shown in previous sections, this is completely contrary to the result we observed. Moreover, we noticed that verified code forces us to rely on best practices to make the verification tractable and meaningful.

This report’s approach described, using buzz words, as Extreme Verification-driven development where we strive for total code coverage in terms of formal verification and not just non-executable formal specifications. One might think, ”Why would another approach to development like this one bring anything new to the world of development.”

While writing this report, we were confronted with the same question, and at the same time, the solution was given:

- the observations made in the Stainless implementation are expected to influence the English language spec and Rust implementation,
- during development, we used the same tests as the Rust implementation which was developed in parallel, during the process we discovered bugs in that implementation which managed to be missed during testing,
- arguably the verified implementation should be easier to test with simpler abstractions, and
- we expect more changes to the TLA^+ as a result of English spec changes.

From the experience of writing this executable model, we have codified several principles of what should be the guiding principle for Verification-driven development:

1. Implementation in a well known general purpose language.
2. Verified implementation does not break the high level protocol properties.
3. Non-executable formal specifications are not the source of truth but a tool to explore possible implementations.
4. Code influences the non executable specifications.
5. 100% verified code.
6. Testing is not made obsolete by this approach.
7. Non-verified code is pushed to the edges of the system.

Each of the previously mentioned principles has a goal that it is trying to achieve. The overall goal is to make engineers trained in thinking more rigorously about the software they are implementing. The following list explains why is each principle important when developing with verification in mind:

1. Seamless interoperability with existing libraries and code. Flexibility of implementation needs to be of utmost importance.
2. It should be clear what the result is and what behavior the implementation should exhibit. Without it, formally verified code which does not conform to the expected behaviour is useless, also underspecified programs leave room for incorrect implementation.
3. Assumptions about the algorithms expressed in text specifications and formal ones that are not executable can be misleading if the implementations deviate from them. As such, the only source of truth should be the code, now with formally verified properties.
4. Artifacts like TLA^+ specifications should be as close as possible to the implementations. Because different decomposition might be reached in executable models, having multiple specifications that deviate between each-other pose a liability for developers and users.

5. Implementing 100% of code in a formally verified manner should help developers gain additional insights and expand the space of verified code. We hope that this will prompt them to implement pieces of software that are used frequently as libraries and publish them for others to use.

6. Verified code, even though deemed correct, can still contain hidden bugs. While formal verification proves correctness in terms of the properties specified in code, nothing prevents users from specifying properties erroneously.

7. In situations where external code is unavoidable, it should be abstracted away with clean abstractions and expected behaviour.

From the principles described above, a reasonable conclusion would be that formal verification would require more work. Testing might be considered sufficient; however, as Edsger W. Dijkstra correctly noted, ”Testing does not prove the absence of bugs.” As a result, testing can very easily miss cases for which the indeed behavior is violated. Formal verification can fill in the void, especially in instances where there are automatic tools like Stainless, which can strengthen an already existing language. However, verification can be a cumbersome task, and we do not expect all users to adopt it, and there is a question of how to export the properties the library provides. In situations where libraries do not have any formal guarantees, abstracting those libraries away and providing guarantees over those abstractions is currently the best practice in these situations, as we did for network IO and hashing.

On the other hand, comment from Donald Knuth, ”Beware of bugs in the above code; I have only proved it correct, not tried it.” When writing formally verified code, one still has to test to confirm that the implementation is correct. For example, in our case, parts of the code which depend on assumptions based on the Tendermint failure model are taken as correct, so errors in those parts can only be caught by testing.

One example of how Verification-driven development might look with sufficiently mature tools is the Dafny implementations of key-value stores and a replicated state machine [6]. In these implementations, the code follows all of the principles other than the first one, as Dafny is a language designed for verification, and not that well known with limited number of libraries. From our viewpoint, the implementation of Core Verification is also an example of code following these guidelines. However, it violates the 4th principle as large parts of the code have to be moved to the edges as they are not verified. Based on the two mentioned implementations, one can see that there is a trade-off between Dafny and Stainless. While Stainless provides support for verifying Scala-compatible programs and seamless interoperability with other libraries and Scala tools, Dafny does not have that capability and is transpiled to C# or other languages for it to be used.

Chapter 7

Related Work

In the area of formal verification of software, there is a significant push in building software with verification as a first-class concern. An important tool for formal verification both for software and hardware are TLA^+ specifications of protocols and systems, several of those proved to be an important tool for finding significant violations in cache-coherence protocol's memory module as in [1] and Pastry key-value protocol [5]. Violations that were found by the previously mentioned work prevented significant errors to propagate to the implementations of the said protocols and resulted in changes designed to help maintain the intended invariants. However, such specifications do not prevent significant errors from making it into the implementation. For example, the Raft consensus algorithm has a canonical and proved protocol with available TLA^+ spec, which did not prevent Redis-Raft implementation from violating the protocol found by using Jepsen tests.

TLA^+ has also been used extensively by companies to formalize their own solutions; Amazon uses is it for their web services [25], ARM for kernel verification [1], Microsoft for Azure Cosmos DB [2], and MongoDB for replication system [3].

Another approach to formally verifying code is directly implementing the executable version of the protocol and verifying that implementation. The main example of such an approach is the implementations in Dafny presented as part of the IronFleet project [6], in them the authors present an end-to-end approach to specifying, implementing, and running those protocols. The approach in those papers is an example of what we believe is a complete approach to building executable protocol specifications, where the work we present in this report is an initial guideline for doing the same with Stainless.

Cryptography is also an area in which formal verification can provide significant improvements in implementation trust. One effort in that direction is the Coq implementation of high-level cryptographic primitives in [26]. In it the authors take an approach in which they implement a general verified code which can be parameterized such that it specializes for a specific cryptographic primitive and is transpiled to C achieving very good performance.

Compared to work which we presented an interesting comparison can be made between their parametrization and the design by contract we used. Both of these approaches try to implement the core part of the algorithms in a general and verified way while keeping variable parts of usage parametric. Another example of a program implemented using Coq is the CompCert [27], a compiler for a large subset of C programming language. The goal of that project was to provide a compiler which is able to guarantee that safety properties proved in code will be maintained after compilation.

Other approaches work directly with existing general-purpose programming languages, without the need for transpilation. One example is ACL2, as a tool for formally

[1] https://git.kernel.org/pub/scm/linux/kernel/git/cmarinas/kernel-tla.git/about/
[2] https://github.com/Azure/azure-cosmos-tla
[3] https://github.com/visualzhou/mongo-repl-tla

verifying Common Lisp programs; it has been used for both hardware and software verification. In the software verification domain, it was notably used for verifying core parts of the Sun Java Virtual Machine and its bytecode verifier [28].

Chapter 8

Conclusion

With this project, we hope to have demonstrated how Stainless can model significant parts of existing TLA^+ specifications of distributed protocols into executable code which is formally verified. We have given an example of an implementation of Core Verification for the Tendermint blockchain along with a model of such blockchain, examined the process of developing executable models, shown that such a model can be used with data from the real world, discussed how such models can be used for finding useful insights which can be translated into better implementations in general and TLA^+ specifications.

Along with the executable model of the protocol, in this report, a significant focus is directed towards the insights and experience of using Stainless for building models which could be used in industry. Therefore we examine testing in combination with formal verification, object-oriented approaches to dealing with external code, limitations of verified code, and general guidelines to writing code with formal verification in mind when using Stainless. Furthermore, we have shown users can combine verified and non-verified code optimized for performance to achieve better results under the assumption that non-verified code adheres by principles of Design by Contract.

In the following sections, we will cover the expected influence of this implementation on the mainline Rust, limitations of the work presented here along with possible solutions and related work.

8.1 Influence on Rust Implementation

As usual in n-version programming, which this project might be seen as an example of, it is interesting to explore the results of combined work. However, the two implementation of Core Verification and its usage were influenced differently. While the Rust implementation was oriented towards mapping the English and the TLA^+ specification as close as possible to the code, the Stainless one was oriented towards translating the TLA^+ specification to an executable verifiable model. Since Stainless imposes limitations on the way one writes code we were forced to introduce stronger invariants, in cases where it is possible to model these properties, without restricting the expressiveness of the algorithm. As a result the implementation in Stainless is not only formally verified but provides stronger guarantees than Rust, TLA^+ and English specification while maintaining the same behaviour from the viewpoint of users.

We expect that these stronger guarantees will be also ported to the previously mentioned artifacts, there are several reasons for that: (1) testing should be easier as the allowed states are simpler, (2) from the discussions with the team at Informal Systems, authors of the Rust implementation, the current abstractions can provide more efficient evidence collection when forks are detected, (3) better performance can be achieved in terms of IO, and (4) there is a stronger emphasis on types which are correct by construction.

8.2 Limitations

From the previous sections, we can notice several limitations of the implementation we have presented. Some of those are the limitations imposed on us by Stainless and some are because of the scope limitations.

8.2.1 Properties in TLA^+

TLA^+ allows for much simpler specification of system properties than in Stainless. For example, a liveliness property like the following: ”Every correct process eventually decides on a value.” might be possible to prove with Stainless, while in TLA^+, such properties are specified using support for temporal logic [7]. To prove such properties one would have to construct ”fair” execution traces in Stainless and prove properties on it. This approach requires would require a significant effort to first formalize such ”fair” schedules and then using such schedules Stainless has to prove them which might require a detailed knowledge of Stainless internals to make the proofs tractable. Fortunately in the Core Verification model liveliness property of termination can be defined in terms of measure for which Stainless has good support.

Other properties which we found hard to prove and because of time constrains we did not manage to prove are properties like *NeverFaulty*. In cases such as this one, the idea would be to show that Stainless is able to find a counter-example such that the model actually transitions into the *Faulty* state. However, when trying to show this, we experienced constant timeouts in the most simple experimental cases which were supposed to bring us to the goal proof. We believe that the reason for that might be complexity of the model which Stainless tries to build, contracts of functions which make the work more time consuming than needed or possible bugs in Stainless. Detailed analysis of why this is the case is left for future work.

8.2.2 Verified Libraries

In large part, the restrictions imposed on us with respect to the boundaries of verified code are because of the unavailability of verified libraries that can be used. For example, compared with the full Scala ecosystem the data structures and libraries supported during verification are very limited. As an example, currently the only supported libraries are the ones from the Stainless repository which support basic data structures. It is impossible to share jars along with their formal specification so building verified ecosystem of libraries for Scala is time consuming. It is possible to share source dependencies and Stainless can use a cache of verified VCs to speed up the process. But recompiling all dependencies each time incurs significant overhead during the development cycle. Solutions to this problem are possible, one of the more reasonable is to use some intermediate representation of Scala as a way of sharing verified libraries [1].

8.2.3 Combining the Core Verification and Blockchain Model

One bigger downside of the implementation presented in this report is the separation of the Core Verification model from the Blockchain one. This is a result of the approach we made towards building a model that should be verified, as such global invariants like checking if the *LightBlock* is actually from the specified blockchain model are not possible. One approach to dealing with this might be the spec refinement as presented in [8], where the model is layered, from the high-level behavior to the protocol to

[1] One possible solution might be TASTy https://dotty.epfl.ch/docs/reference/metaprogramming/tasty-reflect.html

implementation. In this case, an implementation using actors like the one for replicated counter [10] might be the right approach towards proving this. Other approaches which could also be used are ghost variables supported by Stainless which we opted against in favour of keeping the code as clean as possible.

8.2.4 External Code

This can be regarded as a continuation of section 8.2.2 however since in our case significant parts of the code are composable and can be very easily swapped out with non-verified implementations. We consider this approach as the main liability of the implementation as bugs introduced in concrete implementations can easily break the expected behaviour. Currently, the best we can hope for is for possible users to follow the defined behavior closely and write tests to confirm that their implementation follows the interface. Unfortunately, this can be seen as a best-effort approach. In the ideal situation, any implementation of the interface would also be verified with Stainless statically confirming the behavior. A second-best approach would be tools that construct inputs based on the behavior specification to guide tests, either in the form of fuzzing or generating test cases which should try to exercise the code and confirm that behavior with reasonable certainty.

Appendix A

Original TLA^+ Blockchain Specification

```
--------------------- MODULE Lightclient_A_1 ----------------------
(**
 * A state-machine specification of the lite client, following the English spec:
 *
 * https://github.com/informalsystems/tendermint-rs/blob/master/docs/spec/lightclient/verification.md
 *)

EXTENDS Integers, FiniteSets

\* the parameters of Light Client
CONSTANTS
  TRUSTED_HEIGHT,
    (* an index of the block header that the light client trusts by social consensus *)
  TARGET_HEIGHT,
    (* an index of the block header that the light client tries to verify *)
  TRUSTING_PERIOD,
    (* the period within which the validators are trusted *)
  IS_PRIMARY_CORRECT
    (* is primary correct? *)

VARIABLES       (* see TypeOK below for the variable types *)
  state,        (* the current state of the light client *)
  nextHeight,   (* the next height to explore by the light client *)
  nprobes       (* the lite client iteration, or the number of block tests *)

(* the light store *)
VARIABLES
  fetchedLightBlocks, (* a function from heights to LightBlocks *)
  lightBlockStatus,   (* a function from heights to block statuses *)
  latestVerified      (* the latest verified block *)

(* the variables of the lite client *)
lcvars == <<state, nextHeight, fetchedLightBlocks, lightBlockStatus, latestVerified>>

(****************** Blockchain instance ***********************************)

\* the parameters that are propagated into Blockchain
CONSTANTS
  AllNodes
    (* a set of all nodes that can act as validators (correct and faulty) *)

\* the state variables of Blockchain, see Blockchain.tla for the details
VARIABLES now, blockchain, Faulty

\* All the variables of Blockchain. For some reason, BC!vars does not work
bcvars == <<now, blockchain, Faulty>>
```

```
(* Create an instance of Blockchain.
   We could write EXTENDS Blockchain, but then all the constants and state variables
   would be hidden inside the Blockchain module.
 *)
ULTIMATE_HEIGHT == TARGET_HEIGHT + 1

BC == INSTANCE Blockchain_A_1 WITH
  now <- now, blockchain <- blockchain, Faulty <- Faulty

(************************* Lite client ************************************)

(* the heights on which the light client is working *)
HEIGHTS == TRUSTED_HEIGHT..TARGET_HEIGHT

(* the control states of the lite client *)
States == { "working", "finishedSuccess", "finishedFailure" }

(**
 Check the precondition of ValidAndVerified.

 [LCV-FUNC-VALID.1::TLA-PRE.1]
 *)
ValidAndVerifiedPre(trusted, untrusted) ==
  LET thdr == trusted.header
      uhdr == untrusted.header
  IN
  /\ BC!InTrustingPeriod(thdr)
  /\ thdr.height < uhdr.height
     \* the trusted block has been created earlier (no drift here)
  /\ thdr.time <= uhdr.time
  /\ untrusted.Commits \subseteq uhdr.VS
  /\ LET TP == Cardinality(uhdr.VS)
         SP == Cardinality(untrusted.Commits)
     IN
     3 * SP > 2 * TP
  /\ thdr.height + 1 = uhdr.height => thdr.NextVS = uhdr.VS
  (* As we do not have explicit hashes we ignore these three checks of the English spec:

     1. "trusted.Commit is a commit is for the header trusted.Header,
      i.e. it contains the correct hash of the header".
     2. untrusted.Validators = hash(untrusted.Header.Validators)
     3. untrusted.NextValidators = hash(untrusted.Header.NextValidators)
   *)

(**
  * Check that the commits in an untrusted block form 1/3 of the next validators
  * in a trusted header.
 *)
SignedByOneThirdOfTrusted(trusted, untrusted) ==
  LET TP == Cardinality(trusted.header.NextVS)
      SP == Cardinality(untrusted.Commits \intersect trusted.header.NextVS)
  IN
  3 * SP > TP

(**
 Check, whether an untrusted block is valid and verifiable w.r.t. a trusted header.

 [LCV-FUNC-VALID.1::TLA.1]
 *)
ValidAndVerified(trusted, untrusted) ==
    IF ~ValidAndVerifiedPre(trusted, untrusted)
    THEN "FAILED_VERIFICATION"
    ELSE IF ~BC!InTrustingPeriod(untrusted.header)
    (* We leave the following test for the documentation purposes.
       The implementation should do this test, as signature verification may be slow.
       In the TLA+ specification, ValidAndVerified happens in no time.
      *)
    THEN "FAILED_TRUSTING_PERIOD"
    ELSE IF untrusted.header.height = trusted.header.height + 1
            \/ SignedByOneThirdOfTrusted(trusted, untrusted)
         THEN "OK"
         ELSE "CANNOT_VERIFY"

(*
 Initial states of the light client.
 Initially, only the trusted light block is present.
 *)
LCInit ==
    /\ state = "working"
    /\ nextHeight = TARGET_HEIGHT
    /\ nprobes = 0  \* no tests have been done so far
    /\ LET trustedBlock == blockchain[TRUSTED_HEIGHT]
           trustedLightBlock == [header |-> trustedBlock, Commits |-> AllNodes]
       IN
        \* initially, fetchedLightBlocks is a function of one element, i.e., TRUSTED_HEIGHT
        /\ fetchedLightBlocks = [h \in {TRUSTED_HEIGHT} |-> trustedLightBlock]
        \* initially, lightBlockStatus is a function of one element, i.e., TRUSTED_HEIGHT
        /\ lightBlockStatus = [h \in {TRUSTED_HEIGHT} |-> "StateVerified"]
        \* the latest verified block the the trusted block
        /\ latestVerified = trustedLightBlock

\* block should contain a copy of the block from the reference chain, with a matching commit
CopyLightBlockFromChain(block, height) ==
    LET ref == blockchain[height]
        lastCommit ==
          IF height < ULTIMATE_HEIGHT
          THEN blockchain[height + 1].lastCommit
            \* for the ultimate block, which we never use, as ULTIMATE_HEIGHT = TARGET_HEIGHT + 1
          ELSE blockchain[height].VS
    IN
    block = [header |-> ref, Commits |-> lastCommit]

\* Either the primary is correct and the block comes from the reference chain,
\* or the block is produced by a faulty primary.
\*
\* [LCV-FUNC-FETCH.1::TLA.1]
FetchLightBlockInto(block, height) ==
    IF IS_PRIMARY_CORRECT
    THEN CopyLightBlockFromChain(block, height)
    ELSE BC!IsLightBlockAllowedByDigitalSignatures(height, block)

\* add a block into the light store
\*
\* [LCV-FUNC-UPDATE.1::TLA.1]
LightStoreUpdateBlocks(lightBlocks, block) ==
    LET ht == block.header.height IN
    [h \in DOMAIN lightBlocks \union {ht} |->
        IF h = ht THEN block ELSE lightBlocks[h]]

\* update the state of a light block
\*
\* [LCV-FUNC-UPDATE.1::TLA.1]
LightStoreUpdateStates(statuses, ht, blockState) ==
    [h \in DOMAIN statuses \union {ht} |->
        IF h = ht THEN blockState ELSE statuses[h]]
```

```
\* Check, whether newHeight is a possible next height for the light client.
\*
\* [LCV-FUNC-SCHEDULE.1::TLA.1]
CanScheduleTo(newHeight, pLatestVerified, pNextHeight, pTargetHeight) ==
   LET ht == pLatestVerified.header.height IN
   \/ /\ ht = pNextHeight
      /\ ht < pTargetHeight
      /\ pNextHeight < newHeight
      /\ newHeight <= pTargetHeight
   \/ /\ ht < pNextHeight
      /\ ht < pTargetHeight
      /\ ht < newHeight
      /\ newHeight < pNextHeight
   \/ /\ ht = pTargetHeight
      /\ newHeight = pTargetHeight

\* The loop of VerifyToTarget.
\*
\* [LCV-FUNC-MAIN.1::TLA-LOOP.1]
VerifyToTargetLoop ==
     \* the loop condition is true
   /\ latestVerified.header.height < TARGET_HEIGHT
     \* pick a light block, which will be constrained later
   /\ \E current \in BC!LightBlocks:
       \* Get next LightBlock for verification
       /\ IF nextHeight \in DOMAIN fetchedLightBlocks
          THEN \* copy the block from the light store
               /\ current = fetchedLightBlocks[nextHeight]
               /\ UNCHANGED fetchedLightBlocks
          ELSE \* retrieve a light block and save it in the light store
               /\ FetchLightBlockInto(current, nextHeight)
               /\ fetchedLightBlocks' = LightStoreUpdateBlocks(fetchedLightBlocks, current)
       \* Record that one more probe has been done (for complexity and model checking)
       /\ nprobes' = nprobes + 1
       \* Verify the current block
       /\ LET verdict == ValidAndVerified(latestVerified, current) IN
          \* Decide whether/how to continue
          CASE verdict = "OK" ->
            /\ lightBlockStatus' = LightStoreUpdateStates(lightBlockStatus, nextHeight, "StateVerified")
            /\ latestVerified' = current
            /\ state' =
                 IF latestVerified'.header.height < TARGET_HEIGHT
                 THEN "working"
                 ELSE "finishedSuccess"
            /\ \E newHeight \in HEIGHTS:
               /\ CanScheduleTo(newHeight, current, nextHeight, TARGET_HEIGHT)
               /\ nextHeight' = newHeight

          [] verdict = "CANNOT_VERIFY" ->
            (*
               do nothing: the light block current passed validation, but the validator
               set is too different to verify it. We keep the state of
               current at StateUnverified. For a later iteration, Schedule
               might decide to try verification of that light block again.
               *)
            /\ lightBlockStatus' = LightStoreUpdateStates(lightBlockStatus, nextHeight, "StateUnverified")
            /\ \E newHeight \in HEIGHTS:
               /\ CanScheduleTo(newHeight, latestVerified, nextHeight, TARGET_HEIGHT)
               /\ nextHeight' = newHeight
            /\ UNCHANGED <<latestVerified, state>>

          [] OTHER ->
          \* verdict is some error code
          /\ lightBlockStatus' = LightStoreUpdateStates(lightBlockStatus, nextHeight, "StateFailed")
          /\ state' = "finishedFailure"
          /\ UNCHANGED <<latestVerified, nextHeight>>

\* The terminating condition of VerifyToTarget.
\*
\* [LCV-FUNC-MAIN.1::TLA-LOOPCOND.1]
VerifyToTargetDone ==
  /\ latestVerified.header.height >= TARGET_HEIGHT
  /\ state' = "finishedSuccess"
  /\ UNCHANGED <<nextHeight, nprobes, fetchedLightBlocks, lightBlockStatus, latestVerified>>

(******************** Lite client + Blockchain *******************)
Init ==
  \* the blockchain is initialized immediately to the ULTIMATE_HEIGHT
  /\ BC!InitToHeight
  \* the light client starts
  /\ LCInit

(*
 The system step is very simple.
 The light client is either executing VerifyToTarget, or it has terminated.
 (In the latter case, a model checker reports a deadlock.)
 Simultaneously, the global clock may advance.
 *)
Next ==
  /\ state = "working"
  /\ VerifyToTargetLoop \/ VerifyToTargetDone
  /\ BC!AdvanceTime \* the global clock is advanced by zero or more time units

(************************ Types *****************************************)
TypeOK ==
  /\ state \in States
  /\ nextHeight \in HEIGHTS
  /\ latestVerified \in BC!LightBlocks
  /\ \E HS \in SUBSET HEIGHTS:
     /\ fetchedLightBlocks \in [HS -> BC!LightBlocks]
     /\ lightBlockStatus
        \in [HS -> {"StateVerified", "StateUnverified", "StateFailed"}]

(*********************** Properties ****************************************)

(* The properties to check *)
\* this invariant candidate is false
NeverFinish ==
  state = "working"

\* this invariant candidate is false
NeverFinishNegative ==
  state /= "finishedFailure"

\* This invariant holds true, when the primary is correct.
\* This invariant candidate is false when the primary is faulty.
NeverFinishNegativeWhenTrusted ==
  (*(minTrustedHeight <= TRUSTED_HEIGHT)*)
  BC!InTrustingPeriod(blockchain[TRUSTED_HEIGHT])
    => state /= "finishedFailure"

\* this invariant candidate is false
NeverFinishPositive ==
  state /= "finishedSuccess"
```

```
(**
  Correctness states that all the obtained headers are exactly like in the blockchain.

  It is always the case that every verified header in LightStore was generated by
  an instance of Tendermint consensus.

  [LCV−DIST−SAFE.1::CORRECTNESS−INV.1]
 *)
CorrectnessInv ==
    \A h \in DOMAIN fetchedLightBlocks:
        lightBlockStatus[h] = "StateVerified" =>
            fetchedLightBlocks[h].header = blockchain[h]

(**
 Check that the sequence of the headers in storedLightBlocks satisfies ValidAndVerified = "OK" pairwise
 This property is easily violated, whenever a header cannot be trusted anymore.
 *)
StoredHeadersAreVerifiedInv ==
    state = "finishedSuccess"
        =>
        \A lh, rh \in DOMAIN fetchedLightBlocks: \* for every pair of different stored headers
            \/ lh >= rh
              \* either there is a header between them
            \/ \E mh \in DOMAIN fetchedLightBlocks:
                lh < mh /\ mh < rh
              \* or we can verify the right one using the left one
            \/ "OK" = ValidAndVerified(fetchedLightBlocks[lh], fetchedLightBlocks[rh])

\* An improved version of StoredHeadersAreSound, assuming that a header may be not trusted.
\* This invariant candidate is also violated,
\* as there may be some unverified blocks left in the middle.
StoredHeadersAreVerifiedOrNotTrustedInv ==
    state = "finishedSuccess"
        =>
        \A lh, rh \in DOMAIN fetchedLightBlocks: \* for every pair of different stored headers
            \/ lh >= rh
              \* either there is a header between them
            \/ \E mh \in DOMAIN fetchedLightBlocks:
                lh < mh /\ mh < rh
              \* or we can verify the right one using the left one
            \/ "OK" = ValidAndVerified(fetchedLightBlocks[lh], fetchedLightBlocks[rh])
              \* or the left header is outside the trusting period, so no guarantees
            \/ ~BC!InTrustingPeriod(fetchedLightBlocks[lh].header)

(**
 * An improved version of StoredHeadersAreSoundOrNotTrusted,
 * checking the property only for the verified headers.
 * This invariant holds true.
 *)
ProofOfChainOfTrustInv ==
    state = "finishedSuccess"
        =>
        \A lh, rh \in DOMAIN fetchedLightBlocks:
                \* for every pair of stored headers that have been verified
            \/ lh >= rh
            \/ lightBlockStatus[lh] = "StateUnverified"
            \/ lightBlockStatus[rh] = "StateUnverified"
              \* either there is a header between them
            \/ \E mh \in DOMAIN fetchedLightBlocks:
                lh < mh /\ mh < rh /\ lightBlockStatus[mh] = "StateVerified"
              \* or the left header is outside the trusting period, so no guarantees
            \/ ~(BC!InTrustingPeriod(fetchedLightBlocks[lh].header))
              \* or we can verify the right one using the left one
        \/ "OK" = ValidAndVerified(fetchedLightBlocks[lh], fetchedLightBlocks[rh])

(**
 * When the light client terminates, there are no failed blocks. (Otherwise, someone lied to us.)
 *)
NoFailedBlocksOnSuccessInv ==
    state = "finishedSuccess" =>
        \A h \in DOMAIN fetchedLightBlocks:
            lightBlockStatus[h] /= "StateFailed"

\* This property states that whenever the light client finishes with a positive outcome,
\* the trusted header is still within the trusting period.
\* We expect this property to be violated. And Apalache shows us a counterexample.
PositiveBeforeTrustedHeaderExpires ==
    (state = "finishedSuccess") => BC!InTrustingPeriod(blockchain[TRUSTED_HEIGHT])

\* If the primary is correct and the initial trusted block has not expired,
\* then whenever the algorithm terminates, it reports "success"
CorrectPrimaryAndTimeliness ==
  (BC!InTrustingPeriod(blockchain[TRUSTED_HEIGHT])
    /\ state /= "working" /\ IS_PRIMARY_CORRECT) =>
      state = "finishedSuccess"

(**
  If the primary is correct and there is a trusted block that has not expired,
  then whenever the algorithm terminates, it reports "success".
  [LCV-DIST-LIVE.1::SUCCESS-CORR-PRIMARY-CHAIN-OF-TRUST.1]
 *)
SuccessOnCorrectPrimaryAndChainOfTrust ==
  (\E h \in DOMAIN fetchedLightBlocks:
        lightBlockStatus[h] = "StateVerified" /\ BC!InTrustingPeriod(blockchain[h])
    /\ state /= "working" /\ IS_PRIMARY_CORRECT) =>
      state = "finishedSuccess"

\* Lite Client Completeness: If header h was correctly generated by an instance
\* of Tendermint consensus (and its age is less than the trusting period),
\* then the lite client should eventually set trust(h) to true.
\*
\* Note that Completeness assumes that the lite client communicates with a correct full node.
\*
\* We decompose completeness into Termination (liveness) and Precision (safety).
\* Once again, Precision is an inverse version of the safety property in Completeness,
\* as A => B is logically equivalent to ~B => ~A.
PrecisionInv ==
    (state = "finishedFailure")
      => \/ ~BC!InTrustingPeriod(blockchain[TRUSTED_HEIGHT]) \* outside of the trusting period
         \/ \E h \in DOMAIN fetchedLightBlocks:
            LET lightBlock == fetchedLightBlocks[h] IN
                 \* the full node lied to the lite client about the block header
              \/ lightBlock.header /= blockchain[h]
                 \* the full node lied to the lite client about the commits
              \/ lightBlock.Commits /= lightBlock.header.VS

\* the old invariant that was found to be buggy by TLC
PrecisionBuggyInv ==
    (state = "finishedFailure")
      => \/ ~BC!InTrustingPeriod(blockchain[TRUSTED_HEIGHT]) \* outside of the trusting period
         \/ \E h \in DOMAIN fetchedLightBlocks:
            LET lightBlock == fetchedLightBlocks[h] IN
            \* the full node lied to the lite client about the block header
            lightBlock.header /= blockchain[h]

\* the worst complexity
```

```
Complexity ==
   LET N == TARGET_HEIGHT − TRUSTED_HEIGHT + 1 IN
   state /= "working" =>
      (2 ∗ nprobes <= N ∗ (N − 1))

(*
 We omit termination, as the algorithm deadlocks in the end.
 So termination can be demonstrated by finding a deadlock.
 Of course, one has to analyze the deadlocked state and see that
 the algorithm has indeed terminated there.
*)
==============================================================
```

Appendix B

Current TLA^+ Blockchain Specification Optimized for Apalache

```
----------------------------- MODULE Blockchain_A_1 -----------------------------
(*
  This is a high-level specification of Tendermint blockchain
  that is designed specifically for the light client.
  Validators have the voting power of one. If you like to model various
  voting powers, introduce multiple copies of the same validator
  (do not forget to give them unique names though).
 *)
EXTENDS Integers, FiniteSets

Min(a, b) == IF a < b THEN a ELSE b

CONSTANT
  AllNodes,
    (* a set of all nodes that can act as validators (correct and faulty) *)
  ULTIMATE_HEIGHT,
    (* a maximal height that can be ever reached (modelling artifact) *)
  TRUSTING_PERIOD
    (* the period within which the validators are trusted *)

Heights == 1..ULTIMATE_HEIGHT   (* possible heights *)

(* A commit is just a set of nodes who have committed the block *)
Commits == SUBSET AllNodes

(* The set of all block headers that can be on the blockchain.
   This is a simplified version of the Block data structure in the actual implementation. *)
BlockHeaders == [
  height: Heights,
    \* the block height
  time: Int,
    \* the block timestamp in some integer units
  lastCommit: Commits,
    \* the nodes who have voted on the previous block, the set itself instead of a hash
  (* in the implementation, only the hashes of V and NextV are stored in a block,
     as V and NextV are stored in the application state *)
  VS: SUBSET AllNodes,
    \* the validators of this bloc. We store the validators instead of the hash.
  NextVS: SUBSET AllNodes
    \* the validators of the next block. We store the next validators instead of the hash.
]

(* A signed header is just a header together with a set of commits *)
```

```
LightBlocks == [header: BlockHeaders, Commits: Commits]

VARIABLES
   now,
      (* the current global time in integer units *)
   blockchain,
   (* A sequence of BlockHeaders, which gives us a bird view of the blockchain. *)
   Faulty
   (* A set of faulty nodes, which can act as validators. We assume that the set
      of faulty processes is non−decreasing. If a process has recovered, it should
      connect using a different id. *)

(* all variables, to be used with UNCHANGED *)
vars == <<now, blockchain, Faulty>>

(* The set of all correct nodes in a state *)
Corr == AllNodes \ Faulty

(* APALACHE annotations *)
a <: b == a \* type annotation

NT == STRING
NodeSet(S) == S <: {NT}
EmptyNodeSet == NodeSet({})

BT == [height |−> Int, time |−> Int, lastCommit |−> {NT}, VS |−> {NT}, NextVS |−> {NT}]

LBT == [header |−> BT, Commits |−> {NT}]
(* end of APALACHE annotations *)

(***************************** BLOCKCHAIN ***********************************)

(* the header is still within the trusting period *)
InTrustingPeriod(header) ==
   now <= header.time + TRUSTING_PERIOD

(*
 Given a function pVotingPower \in D −> Powers for some D \subseteq AllNodes
 and pNodes \subseteq D, test whether the set pNodes \subseteq AllNodes has
 more than 2/3 of voting power among the nodes in D.
 *)
TwoThirds(pVS, pNodes) ==
   LET TP == Cardinality(pVS)
      SP == Cardinality(pVS \intersect pNodes)
   IN
   3 * SP > 2 * TP \* when thinking in real numbers, not integers: SP > 2.0 / 3.0 * TP

(*
 Given a set of FaultyNodes, test whether the voting power of the correct nodes in D
 is more than 2/3 of the voting power of the faulty nodes in D.
 *)
IsCorrectPower(pFaultyNodes, pVS) ==
   LET FN == pFaultyNodes \intersect pVS   \* faulty nodes in pNodes
      CN == pVS \ pFaultyNodes            \* correct nodes in pNodes
      CP == Cardinality(CN)              \* power of the correct nodes
      FP == Cardinality(FN)              \* power of the faulty nodes
   IN
   \* CP + FP = TP is the total voting power, so we write CP > 2.0 / 3 * TP as follows:
   CP > 2 * FP \* Note: when FP = 0, this implies CP > 0.

(* This is what we believe is the assumption about failures in Tendermint *)
FaultAssumption(pFaultyNodes, pNow, pBlockchain) ==
   \A h \in Heights:
```

```
   pBlockchain[h].time + TRUSTING_PERIOD > pNow =>
     IsCorrectPower(pFaultyNodes, pBlockchain[h].NextVS)

(* Can a block be produced by a correct peer, or an authenticated Byzantine peer *)
IsLightBlockAllowedByDigitalSignatures(ht, block) ==
    \/ block.header = blockchain[ht] \* signed by correct and faulty (maybe)
    \/ block.Commits \subseteq Faulty /\ block.header.height = ht \* signed only by faulty

(*
 Initialize the blockchain to the ultimate height right in the initial states.
 We pick the faulty validators statically, but that should not affect the light client.
 *)
InitToHeight ==
  /\ Faulty \in SUBSET AllNodes \* some nodes may fail
  \* pick the validator sets and last commits
  /\ \E vs, lastCommit \in [Heights -> SUBSET AllNodes]:
     \E timestamp \in [Heights -> Int]:
        \* now is at least as early as the timestamp in the last block
        /\ \E tm \in Int: now = tm /\ tm >= timestamp[ULTIMATE_HEIGHT]
        \* the genesis starts on day 1
        /\ timestamp[1] = 1
        /\ vs[1] = AllNodes
        /\ lastCommit[1] = EmptyNodeSet
        /\ \A h \in Heights \ {1}:
          /\ lastCommit[h] \subseteq vs[h - 1]   \* the non-validators cannot commit
          /\ TwoThirds(vs[h - 1], lastCommit[h]) \* the commit has >2/3 of validator votes
          /\ IsCorrectPower(Faulty, vs[h])       \* the correct validators have >2/3 of power
          /\ timestamp[h] > timestamp[h - 1]     \* the time grows monotonically
          /\ timestamp[h] < timestamp[h - 1] + TRUSTING_PERIOD    \* but not too fast
        \* form the block chain out of validator sets and commits (this makes apalache faster)
        /\ blockchain = [h \in Heights |->
            [height |-> h,
             time |-> timestamp[h],
             VS |-> vs[h],
             NextVS |-> IF h < ULTIMATE_HEIGHT THEN vs[h + 1] ELSE AllNodes,
             lastCommit |-> lastCommit[h]]
            ] \******


(* is the blockchain in the faulty zone where the Tendermint security model does not apply *)
InFaultyZone ==
  ~FaultAssumption(Faulty, now, blockchain)

(******************** BLOCKCHAIN ACTIONS ********************************)
(*
  Advance the clock by zero or more time units.
  *)
AdvanceTime ==
  \E tm \in Int: tm >= now /\ now' = tm
  /\ UNCHANGED <<blockchain, Faulty>>

(*
 One more process fails. As a result, the blockchain may move into the faulty zone.
 The light client is not using this action, as the faults are picked in the initial state.
 However, this action may be useful when reasoning about fork detection.
 *)
OneMoreFault ==
  /\ \E n \in AllNodes \ Faulty:
      /\ Faulty' = Faulty \cup {n}
      /\ Faulty' /= AllNodes \* at least process remains non-faulty
  /\ UNCHANGED <<now, blockchain>>
=================================================================
```

Appendix C

Light Client TLA^+ specification

```
------------------- MODULE Blockchain_A_1 -------------------------
(*
  This is a high-level specification of Tendermint blockchain
  that is designed specifically for the light client.
  Validators have the voting power of one. If you like to model various
  voting powers, introduce multiple copies of the same validator
  (do not forget to give them unique names though).
 *)
EXTENDS Integers, FiniteSets

Min(a, b) == IF a < b THEN a ELSE b

CONSTANT
  AllNodes,
    (* a set of all nodes that can act as validators (correct and faulty) *)
  ULTIMATE_HEIGHT,
    (* a maximal height that can be ever reached (modelling artifact) *)
  TRUSTING_PERIOD
    (* the period within which the validators are trusted *)

Heights == 1..ULTIMATE_HEIGHT   (* possible heights *)

(* A commit is just a set of nodes who have committed the block *)
Commits == SUBSET AllNodes

(* The set of all block headers that can be on the blockchain.
   This is a simplified version of the Block data structure in the actual implementation. *)
BlockHeaders == [
  height: Heights,
    \* the block height
  time: Int,
    \* the block timestamp in some integer units
  lastCommit: Commits,
    \* the nodes who have voted on the previous block, the set itself instead of a hash
  (* in the implementation, only the hashes of V and NextV are stored in a block,
     as V and NextV are stored in the application state *)
  VS: SUBSET AllNodes,
    \* the validators of this bloc. We store the validators instead of the hash.
  NextVS: SUBSET AllNodes
    \* the validators of the next block. We store the next validators instead of the hash.
]

(* A signed header is just a header together with a set of commits *)
LightBlocks == [header: BlockHeaders, Commits: Commits]

VARIABLES
    now,
        (* the current global time in integer units *)
    blockchain,
    (* A sequence of BlockHeaders, which gives us a bird view of the blockchain. *)
    Faulty
    (* A set of faulty nodes, which can act as validators. We assume that the set
       of faulty processes is non-decreasing. If a process has recovered, it should
       connect using a different id. *)

(* all variables, to be used with UNCHANGED *)
vars == <<now, blockchain, Faulty>>

(* The set of all correct nodes in a state *)
Corr == AllNodes \ Faulty

(* APALACHE annotations *)
a <: b == a \* type annotation

NT == STRING
NodeSet(S) == S <: {NT}
EmptyNodeSet == NodeSet({})

BT == [height |-> Int, time |-> Int, lastCommit |-> {NT}, VS |-> {NT}, NextVS |-> {NT}]

LBT == [header |-> BT, Commits |-> {NT}]
(* end of APALACHE annotations *)

(***************************** BLOCKCHAIN ***********************************)

(* the header is still within the trusting period *)
InTrustingPeriod(header) ==
    now <= header.time + TRUSTING_PERIOD

(*
 Given a function pVotingPower \in D -> Powers for some D \subseteq AllNodes
 and pNodes \subseteq D, test whether the set pNodes \subseteq AllNodes has
 more than 2/3 of voting power among the nodes in D.
 *)
TwoThirds(pVS, pNodes) ==
    LET TP == Cardinality(pVS)
        SP == Cardinality(pVS \intersect pNodes)
    IN
    3 * SP > 2 * TP \* when thinking in real numbers, not integers: SP > 2.0 / 3.0 * TP

(*
 Given a set of FaultyNodes, test whether the voting power of the correct nodes in D
 is more than 2/3 of the voting power of the faulty nodes in D.
 *)
IsCorrectPower(pFaultyNodes, pVS) ==
    LET FN == pFaultyNodes \intersect pVS   \* faulty nodes in pNodes
        CN == pVS \ pFaultyNodes            \* correct nodes in pNodes
        CP == Cardinality(CN)               \* power of the correct nodes
        FP == Cardinality(FN)               \* power of the faulty nodes
    IN
    \* CP + FP = TP is the total voting power, so we write CP > 2.0 / 3 * TP as follows:
    CP > 2 * FP \* Note: when FP = 0, this implies CP > 0.

(* This is what we believe is the assumption about failures in Tendermint *)
FaultAssumption(pFaultyNodes, pNow, pBlockchain) ==
    \A h \in Heights:
      pBlockchain[h].time + TRUSTING_PERIOD > pNow =>
        IsCorrectPower(pFaultyNodes, pBlockchain[h].NextVS)

(* Can a block be produced by a correct peer, or an authenticated Byzantine peer *)
IsLightBlockAllowedByDigitalSignatures(ht, block) ==
    \/ block.header = blockchain[ht] \* signed by correct and faulty (maybe)
```

```
   \/ block.Commits \subseteq Faulty /\ block.header.height = ht \* signed only by faulty

(*
 Initialize the blockchain to the ultimate height right in the initial states.
 We pick the faulty validators statically, but that should not affect the light client.
 *)
InitToHeight ==
  /\ Faulty \in SUBSET AllNodes \* some nodes may fail
  \* pick the validator sets and last commits
  /\ \E vs, lastCommit \in [Heights -> SUBSET AllNodes]:
     \E timestamp \in [Heights -> Int]:
        \* now is at least as early as the timestamp in the last block
        /\ \E tm \in Int: now = tm /\ tm >= timestamp[ULTIMATE_HEIGHT]
        \* the genesis starts on day 1
        /\ timestamp[1] = 1
        /\ vs[1] = AllNodes
        /\ lastCommit[1] = EmptyNodeSet
        /\ \A h \in Heights \ {1}:
          /\ lastCommit[h] \subseteq vs[h - 1]   \* the non-validators cannot commit
          /\ TwoThirds(vs[h - 1], lastCommit[h]) \* the commit has >2/3 of validator votes
          /\ IsCorrectPower(Faulty, vs[h])       \* the correct validators have >2/3 of power
          /\ timestamp[h] > timestamp[h - 1]     \* the time grows monotonically
          /\ timestamp[h] < timestamp[h - 1] + TRUSTING_PERIOD    \* but not too fast
        \* form the block chain out of validator sets and commits (this makes apalache faster)
        /\ blockchain = [h \in Heights |->
             [height |-> h,
              time |-> timestamp[h],
              VS |-> vs[h],
              NextVS |-> IF h < ULTIMATE_HEIGHT THEN vs[h + 1] ELSE AllNodes,
              lastCommit |-> lastCommit[h]]
             ] \******


(* is the blockchain in the faulty zone where the Tendermint security model does not apply *)
InFaultyZone ==
  ~FaultAssumption(Faulty, now, blockchain)

(******************** BLOCKCHAIN ACTIONS ******************************)
(*
  Advance the clock by zero or more time units.
  *)
AdvanceTime ==
  \E tm \in Int: tm >= now /\ now' = tm
  /\ UNCHANGED <<blockchain, Faulty>>

(*
 One more process fails. As a result, the blockchain may move into the faulty zone.
 The light client is not using this action, as the faults are picked in the initial state.
 However, this action may be useful when reasoning about fork detection.
 *)
OneMoreFault ==
  /\ \E n \in AllNodes \ Faulty:
      /\ Faulty' = Faulty \cup {n}
      /\ Faulty' /= AllNodes \* at least process remains non-faulty
  /\ UNCHANGED <<now, blockchain>>
=============================================================
```

Bibliography

[1] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications". In: *Correct Hardware Design and Verification Methods*. Ed. by Laurence Pierre and Thomas Kropf. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66. ISBN: 978-3-540-48153-9.

[2] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. "System FR: formalized foundations for the stainless verifier". eng. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30. ISSN: 2475-1421.

[3] Ethan Buchman, Jae Kwon, and Zarko Milosevic. *The latest gossip on BFT consensus*. 2018. arXiv: 1807.04938 [cs.DC].

[4] Leslie Lamport. "Byzantizing Paxos by Refinement". In: *Distributed Computing*. Ed. by David Peleg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 211–224. ISBN: 978-3-642-24100-0.

[5] Noran Azmy, Stephan Merz, and Christoph Weidenbach. "A machine-checked correctness proof for Pastry". eng. In: *Science of Computer Programming* 158.C (2018), pp. 64–80. ISSN: 0167-6423.

[6] Chris Hawblitzel et al. "IronFleet: Proving Practical Distributed Systems Correct". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, 1–17. ISBN: 9781450338349. DOI: 10.1145/2815400.2815428. URL: https://doi.org/10.1145/2815400.2815428.

[7] Leslie Lamport. "The temporal logic of actions". eng. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), pp. 872–923. ISSN: 01640925.

[8] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.

[9] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642058809.

[10] Romain Ruetschi and Jad Hamza. *Stainless Actors*. https://github.com/epfl-lara/stainless-actors. 2020.

[11] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[12] Clark Barrett et al. "CVC4". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177. ISBN: 978-3-642-22110-1.

[13] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)". In: *J. ACM* 53.6 (Nov. 2006), 937–977. ISSN: 0004-5411. DOI: 10.1145/1217856.1217859. URL: https://doi.org/10.1145/1217856.1217859.

[14] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. "Satisfiability Modulo Recursive Programs". In: *Static Analysis*. Ed. by Eran Yahav. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 298–315. ISBN: 978-3-642-23702-7.

[15] Kyle Kingsbury. *Redis-Raft 1b3fbf6*. 2020. URL: https://jepsen.io/analyses/redis-raft-1b3fbf6.

[16] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: https://doi.org/10.1145/357172.357176.

[17] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the Presence of Partial Synchrony". In: *J. ACM* 35.2 (Apr. 1988), 288–323. ISSN: 0004-5411. DOI: 10.1145/42282.42283. URL: https://doi.org/10.1145/42282.42283.

[18] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: *Cryptography Mailing list at https://metzdowd.com* (Mar. 2009), p. 5.

[19] Sean Braithwaite et al. *A Tendermint Light Client*. 2020.

[20] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. "TLA+ model checking made symbolic". eng. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30. ISSN: 2475-1421.

[21] Barbara H. Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (Nov. 1994), 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383. URL: https://doi.org/10.1145/197320.197383.

[22] Barbara Liskov. "Keynote Address - Data Abstraction and Hierarchy". In: *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum)*. OOPSLA '87. Orlando, Florida, USA: Association for Computing Machinery, 1987, 17–34. ISBN: 0897912667. DOI: 10.1145/62138.62141. URL: https://doi.org/10.1145/62138.62141.

[23] B Meyer. "APPLYING DESIGN BY CONTRACT". English. In: *Computer* 25.10 (1992), pp. 40–51. ISSN: 0018-9162.

[24] Daniel J. Bernstein et al. "High-Speed High-Security Signatures". In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 124–142. ISBN: 978-3-642-23951-9.

[25] Chris Newcombe et al. "How Amazon Web Services Uses Formal Methods". In: *Commun. ACM* 58.4 (Mar. 2015), 66–73. ISSN: 0001-0782. DOI: 10.1145/2699417. URL: https://doi.org/10.1145/2699417.

[26] A. Erbsen et al. "Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1202–1219.

[27] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7 (July 2009), 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[28] J Strother Moore and Hanbing Liu. “Formal specification and verification of a jvm and its bytecode verifier”. In: 2006.