Toychain

Formally Verified Blockchain Consensus

George Pîrlea

MEng Computer Science

Advisors: Dr Earl Barr, Dr Ilya Sergey

Submission Date: 29th April 2019

Thesis advisors: Dr Earl Barr, Dr Ilya Sergey George Pîrlea

Toychain

Formally Verified Blockchain Consensus

Abstract

We present Toychain, the first proven-correct implementation of a Nakamoto-style blockchain consensus protocol. We build upon our previous work, the first formalisation of a blockchain-based distributed consensus protocol with a mechanised proof of consistency.

Our formalisation includes a reference implementation of the block forest data structure and provably-correct message handlers for the protocol logic. The formal model is parametric *wrt.* implementations of several security primitives, such as hash functions, a notion of a proof object, a Validator Acceptance Function, and a Fork Choice Rule.

We improve our original model by removing several overly-strong assumptions, notably the assumption that hashing is injective. Then, we instantiate the model with a SHA256-based proof-of-work scheme and extract our proven-correct OCaml implementation of Nakamoto consensus. Finally, we execute our implementation on a local area network to test its effectiveness.

All our results are formalised in the Coq interactive theorem prover.

Acknowledgments

I would like to thank both my supervisors, Earl Barr and Ilya Sergey, for their continued support. Without them, this work would not have been possible.

When I was stuck on a difficult proof or confused by Coq's peculiarities, Ilya always provided the insight to get me through. Earl, with his extraordinary ability to ask questions that stump and his attention to detail, has made my thinking clearer and this report much better than it would have been otherwise. I am grateful to both of them.

I also wish to thank Anton Trunov and Karl Palmskog for their great technical advice. Anton, sharing his expertise, helped me understand the bugs in Coq's extraction mechanism and implement work-arounds. Karl, too, has been tremendously helpful. In fact, his comments on the original Toychain triggered this work. Moreover, he graciously offered his time to maintain the OPAM package and build infrastructure. Without either Karl or Anton, Toychain would not be executable.

I also want to thank the NUS School of Computing, particularly for arranging my visit to Singapore, where a large part of this thesis has been written and thank researchers at both NUS and Zilliqa for their feedback.

Finally, I would like to thank all colleagues and staff at UCL for their support throughout my degree. I am very grateful for the 4 years I spent studying in London.

Contents

1 Introduction

> *I don't know exactly how this can be prevented from happening again, but I do know that it would be a mistake for the community to brush off this bug just because it ended up being mostly harmless this time.*
>
> Michael Marquardt, founder of BitcoinTalk.org

This chapter motivates the need for provably-correct consensus protocols, explains the benefits and limitations of formal verification, and presents our approach to developing a framework for reasoning about blockchain consensus.

1.1 Need for Trustworthy Consensus Protocols

Distributed systems are widely deployed in industry and a significant amount of economic activity depends on them functioning reliably. Nevertheless, outages do happen, and quite regularly. For example, the entire VISA payment network was unavailable in the UK and Ireland for 8 hours on June 1st, 2018 due to a "hardware failure" [VIS18].

Outages in a distributed system can be prevented, but prevention comes at a cost. If some data needs to be available for the system to function, the risk of failure can be mitigated by *replication*. If the data is stored on multiple servers, when one server fails, another can take its place. However, replication introduces a significant challenge: the different servers in the system must be in *agreement* about the data they hold.

The mechanism by which a number of servers come into agreement with each other is called a *consensus protocol*. The consensus problem is a difficult one and it comes in many flavours. Is the set of servers fixed, or does it change? Is it acceptable if distinct clients receive different responses to the same query? What if some of the servers act maliciously, rather than just crash? Do all the servers even know about each other?

Regardless of the answer to these questions, experience shows that consensus protocols are, generally speaking, difficult to understand and implement correctly. This is due to the nature of the problem. When implementing a consensus protocol, some mistakes lead to obvious, immediate failures — no agreement is reached even in the absence of crashes. Such mistakes are easily caught by testing. Other mistakes, on the

other hand, can be surprisingly subtle and lead to failure only in very rare scenarios — but with disastrous consequences.

It is this subtle kind of mistake which motivates the need for *formal methods*. As distributed systems become more critically important, the cost of catastrophic failure grows. To minimise risks, organisations are incentivised to seek a higher level of assurance about the correctness of their systems.

1.2 Failures in Permissionless Systems

High levels of assurance are particularly important for *permissionless* systems such as Bitcoin, in which anyone can join the protocol at any time and act in a potentially malicious manner. When a system that is under the control of a single organisation malfunctions in production, mitigating the error is often as simple as "turn it off and on again". No such possibility exists for decentralised systems like Bitcoin. And errors do happen in these systems as well.

1.2.1 Value Overflow Incident

On August 15, 2010, Bitcoin experienced what came to be called the "*value overflow incident*" — block number 74,638 contained a maliciously crafted transaction that created 184 billion bitcoin [Gar10] (the intended total supply is 21 million). This happened because the code that validated transactions checked for overflow in individual transaction outputs, but not in their sum. Thus, a transaction with 0.5 bitcoin as input and 92 + 92 billion bitcoin as output was treated as valid, since the total output of the transaction, 184 billion bitcoin, overflowed to a value less than the input. A new version of the Bitcoin software that changed the consensus rules to reject the value overflow was released within 5 hours [Nak10]. In a decentralised system, however, you cannot force users to upgrade their software. How, then, was the overflowing transaction reverted?

The new release introduced a *soft fork* — a protocol change that tightens the consensus rules, but keeps them compatible with older versions of the software. More precisely, all behaviours that are acceptable under the new rules are also acceptable under the old rules, but not vice versa. The new version of the Bitcoin software saw the transaction as invalid, and rejected the block that contained it. Meanwhile, older versions continued to view the transaction as valid. However, as miners — protocol participants who extend the blockchain — migrated to the updated rules, the chain that did not include the invalid block became the *heaviest chain**. At that point, nodes running the

*The chain with the most accumulated proof-of-work. Consensus rules dictate that nodes adopt the heaviest *valid* chain. Honest miners create blocks only on top of this chain.

old software migrated to that chain, and the invalid block was orphaned.

1.2.2 Accidental Hard Fork

Even in the absence of malicious actors, simple software updates can lead to consensus failures. Bitcoin v0.8.0, released in February 2013, changed its internal transaction database from BerkeleyDB to LevelDB. On March 11, 2013, the Bitcoin network created a block that had a larger number of transaction inputs than previously seen. This lead to an unintentional *hard fork* — versions v0.7.2 and prior rejected block number 225,430, whereas v0.8.0 accepted it [Gar13]. It turns out that the configuration of the database had implicitly become a network consensus rule: v0.7.2 and earlier could not process blocks for which transaction processing acquired more than 10,000 database locks; v0.8.0 did not have this limitation. At that point, the v0.7.2 consensus rules had majority hashpower, so the Bitcoin developers addressed the issue by releasing a new version, v0.8.1, that temporarily rejected blocks that would cause more than 10,000 locks to be taken [And13].

1.2.3 Inflation Bug

Sometimes, bugs lie undiscovered for years. Bitcoin was severely vulnerable for a period of 18 months from March 2017 to September 2018. The root cause, similarly to the value overflow incident, was a fault in the transaction processing logic. Versions 0.14.X of the Bitcoin software would *crash* when processing blocks that include a transaction that attempts to spend the same output twice. This made the Bitcoin network vulnerable to a Denial of Service attack. Much more dangerously, in Bitcoin versions 0.15.X up to 0.16.2, the same transaction would in some circumstances allow a *succesful double-spend*, *i.e.* the nodes would not only *not* crash — they would see the money-cloning transaction as valid! [Bit18; Son18; Cor18]

> "The whole community screwed up by not reviewing consensus changes thoroughly enough, more developers need to pay attention! It's your responsibility." — Wladimir J. van der Laan [Laa18]

It is stupendously easy to introduce severe bugs into critical consensus code — the history of such errors in Bitcoin and other similar systems proves as much. But we are not doomed to perpetual failure.

1.3 Promise of Formal Verification

1.3.1 Well-suited for Consensus Protocols

Consensus protocols are complicated when we consider *how* they work, but *what* they do is very simple — they ensure agreement between multiple parties. This combination of having a complex implementation, but a simple specification makes consensus protocols *ideal targets* for formal methods. On one hand, it means that bugs that will not be caught by testing are very likely to creep in, which motivates the need for more assurance than simple testing can provide. On the other hand, as a verifier, it also means that it is easy to convince yourself that you are indeed verifying the right thing, *i.e.* that your specification means what you intend.

1.3.2 Powerful but not Magical

It is crucially important to understand what verification is in order to understand what it guarantees, but also, equally importantly, what it does not. Fundamentally, *formal verification* means proving software correct with respect to a given rigorous specification using mathematical reasoning. In other words, you are proving that the software meets the specification. Even if you trust the proof, which you largely can trust if it is *mechanised* (machine-checked), this does not mean that your software is correct. Indeed, it may be that your specification is wrong [Ser18].

For this reason, formally verified software can and does have errors. For example, CompCert, the verified C compiler, had a bug in how it compiled C99-style for loops: `i = 10; for (int i = 0; i < 3; i++) {}; return i;` would return 3 rather than 10 [Bee17]. There was no error in CompCert's proofs — all were mechanised and correct. The error was in CompCert's specification of C scoping rules.

By contrast, the specification of a blockchain consensus protocol is comparatively simple. With less scope for specification error, we can be more certain that our formal proofs actually do imply that the software is correct. Risks still exist, especially at the interface between the verified and the unverified components of the software, but these can be minimised in other ways.

1.4 Towards Verified Blockchain Consensus

Given the importance of systems such as Bitcoin, we argue that there is a strong need for formally verified, provably-correct implementations. In this thesis, we make a significant step towards achieving that goal.

We build upon our previous work, the first formalisation of Nakamoto consensus mechanised in an interactive proof assistant [PS18], and extend it. In particular, we remove all unrealizable assumptions from the original Toychain development, instantiate the protocol with an example set of consensus rules, and extract a *proven-correct implementation* directly from Coq.

Our long term goal is to have Toychain evolve into a principled framework for reasoning about the correctness and security properties of blockchain consensus protocols. One immediate application of such a framework is that it would allow developers to prove their updates correct before they are deployed, and thus guarantee that no failures will be introduced.

Overview of Contents

Chapter 1 motivated the need for formally-verified protocols and gave an overview of our approach.

Chapter 2 describes previous work on formal verification of distributed systems and gives a high-level description of Toychain.

Chapter 3 explains our effort to remove onerous assumptions we made in the original Toychain development, as a prerequisite for code extraction.

Chapter 4 details the process of extracting executable code from Coq and, looking at the full Toychain system, precisely distinguishes which components are proven correct and which are part of the trusted computing base.

Chapter 5 concludes by reflecting on what we have achieved thus far, discussing limitations and identifying avenues for future work, with a focus on driving Toychain towards its goal of informing real-world protocol implementations.

2
Overview

> *Defect-finding mechanisms must be applied until human confidence reaches the level appropriate to the impact of an unfound bug.*
>
> Ron Jeffries, co-inventor of Extreme Programming

This chapter describes related work on formal verification of distributed systems and gives a high-level overview of Toychain [PS18], the first mechanised formalisation of a blockchain consensus protocol — which we extend in this work.

2.1 Formal Verification of Distributed Systems

Distributed systems are notoriously difficult to implement correctly. This is completely understandable, given their inherently non-deterministic nature — programmers find it difficult to reason about the combination of concurrency, network delays, and failure scenarios. Nonetheless, distributed systems are used in industry, in situations where downtime or data loss can be seriously costly.

The quest for improved reliability of distributed systems has triggered research into better programming abstractions, defect-finding mechanisms, and formal verification. This has lead to developments such as actor-based programming [BSd82], fault-injection [DJ94; DJa96], model-checking of distributed protocols [Lam02; Yan+09], and recently, formally verified implementations of practical distributed systems [Haw+15; Wil+15; LBC16; Woo+16; SWT17].

Historically speaking, the first advancements in this area were abstractions meant to make it easier for programmers to reason about and implement distributed systems. This improved the situation, but only slightly. As tools developed, however, it became possible to model-check protocols against formal specifications. This helped validate the *protocols*, but crucially did not apply to actual implementations of distributed systems. To overcome this shortcoming, two different approaches sprung up. The systems research community focused on fault-injection and, to a lesser degree, instrumenting implementations for model-checking. By contrast, the programming languages community,

spurred by recent advancements in theorem proving, pursued full formal verification of distributed system implementations, using tools such as Coq [BC04] and Dafny [Lei10].

Both the "systems" approach and the "verification" approach are valuable. One crucial advantage of the systems approach is that it can be retrofitted onto existing systems. Given an existing distributed system, it takes little effort to plug it into a fault-injection framework such as Jepsen [Kin13] and start discovering bugs immediately. This makes the systems approach to improving reliability very attractive for industrial actors — it is low-investment and high-payoff. By contrast, formal verification has very high initial costs and requires a completely new implementation. It cannot be retrofitted. Nevertheless, it still *is* appealing in some situations because it gives strong correctness guarantees that the systems approach cannot provide. Permissionless systems, which we study in this work, are one of the key areas for which verification is well-suited. Since the overall system is not under any one entity's control, and thus cannot be easily changed, it becomes much more important to ensure no faults exist *before* the system is deployed.

For the purpose of formal verification, a distributed system can be modelled as a collection of stateful processes that pass messages to each other over a network. The network, in turn, can be either asynchronous, partially synchronous, or synchronous. The choice dictates when messages get delivered. Verification consists of discovering and establishing *invariants* — statements that relate the processes' and the network's state and are preserved by system transitions — and proving that these invariants imply the desired overall system property, *e.g.* safety or liveness. However, the concrete state of the network is usually not sufficient to establish strong invariants. For example, proving many properties requires storing historical information, rather than just the most recent state. As such, for verification, the "concrete" state usually needs to be augmented with "logical" or "ghost" state — information that is necessary for proofs, but does not impact the protocol's execution. Discovering what "logical" state needs to be kept is an important part of the process of discovering system invariants.

Key to formal verification is that both the system model and the associated proofs can be expressed in a programming language, *e.g.* Coq, such that the correctness of the proofs can be *machine-checked*. This means that the proofs themselves do not need to be trusted, as long as the checker is trusted. Moreover, since the model is expressed in a programming language, it is possible to make it executable. In this case, the proofs establish properties of an actual implementation of the system. In Section 2.1.1, we review some important formally verified implementations of distributed systems.

2.1.1 Verified Implementations

Several recent works propose frameworks for reasoning about distributed protocols and have developed formally verified implementations of realistic distributed systems. All of them, however, implement traditional consensus protocols, rather than permissionless protocols like Nakamoto consensus. While many of these existing frameworks could be adapted to reason about blockchain consensus, permissionless protocols are significantly different from traditional protocols and warrant different reasoning patterns.

In the original Toychain paper, we distilled the protocol semantics and demonstrated a typical proof pattern for blockchain consensus by formally verifying a basic safety property [PS18]. In this work, we build on the Toychain model and extract a verified implementation of Nakamoto consensus. Our long-term goal is to transform Toychain into a principled framework specifically for reasoning about the correctness and security properties of blockchain consensus protocols and their implementations. In other words, we want to *tame* the full generality of existing frameworks to make it easier to reason about blockchain protocols, which are totally unlike the traditional protocols existing frameworks were designed to work best with. In particular, counter-intuitively, reasoning about concurrency and fault tolerance (the main challenges current frameworks seek to address) is practically not required for Nakamoto consensus: protocol messages have no dependencies on each other, so the order in which they are delivered and processed is irrelevant, and fault tolerance follows trivially since all nodes store every block in perpetuity and make it available to anyone upon request via the gossip mechanism.

We describe the existing frameworks now, and overview our approach in Section 2.2.

IronFleet. IronFleet proposes a methodology to verify distributed systems based on state-machine refinement and Hoare logic. State-machine refinement is used to reason about concurrency at the protocol level. Crucially, it allows the verifier to "contain" the complexity of reasoning about concurrency at a high-level, such that Hoare logic can be used to verify the low-level message-handler implementations as if they operated sequentially. IronFleet is used to prove both safety and liveness for an implementation of MultiPaxos. IronFleet is implemented and verified in Dafny, which relies on the Z3 SMT solver [Haw+15].

Verdi. Verdi is a Coq framework for verifying distributed systems, based on verified system transformers. Users of Verdi prove their implementation correct under an idealised fault model, then translate, via the system transformer, this implementation to an implementation that is correct under a different, more realistic, fault model. In other words, Verdi decouples verifying application-level guarantees from the complexity

of verifying fault-tolerance mechanisms [Wil+15]. Using Verdi, Woos *et al.* implement the Raft consensus protocol and prove it satisfies linearizability [Woo+16].

PSync. PSync is a domain-specific language embedded in Scala, used for automated verification of fault-tolerant distributed systems. It is based on the Heard-Of (HO) model and, under partial synchrony assumptions, is capable of verifying both safety and liveness properties. PSync has been used to verify several protocol implementations, including Paxos and Two-Phase Commit [DHZ16].

Chapar. Chapar presents a methodology for modular verification of causal consistency (as opposed to strong consistency) and implements a verified replicated key-value store server and client. The Chapar framework is written in Coq and the verified implementations are extracted to OCaml directly from Coq [LBC16].

Ivy. Ivy is an interactive tool for discovering invariants in distributed systems, using bounded verification of infinite-state systems. The user proposes a candidate invariant and Ivy either validates that the invariant is inductive or provides a counterexample. This continues until an inductive invariant is found. Ivy has been used to verify the safety of several protocols, but no implementation has been extracted from the Ivy models [Pad+16].

Disel. Disel (Distributed Separation Logic) is a framework for *compositional* verification of distributed systems and their clients, based on a distributed Hoare Type Theory. It allows a human verifier to write protocol specifications, show that particular implementations follow the protocol, and then reason about the composition of protocols abstractly, without reference to their respective implementations. Disel is implemented in the Coq proof assistant and has been used to verify an implementation of Two-Phase Commit [SWT17]. The network model in Toychain is inspired by Disel and our OCaml networking code is based on Disel's.

Velisarios. Velisarios is a Coq framework for verifying Byzantine-fault tolerant distributed systems, based on the logic-of-events model. It has been used implement the PBFT consensus protocol and verify a crucial safety property — agreement. The PBFT implementation is extracted from Coq to OCaml [Rah+18].

Pretend synchrony. Pretend synchrony is an approach to verifying distributed systems written in Go, using Hoare-style verification conditions and SMT, based on the observation that, in certain cases, asynchronous protocols can soundly be reasoned

about as if they were synchronous. It has been used to verify the correctness of Go implementations of Two-Phase Commit, Raft leader election, Single-decree Paxos and Multi-Paxos [Gle+19].

2.2 Toychain

Toychain is the first formalisation of a blockchain-based consensus protocol, with a proof of eventual consistency mechanised in the Coq proof assistant [PS18]. In the following, we provide a high-level overview of Toychain, aimed in particular at an audience new to protocol verification. For a full description, please refer to the original paper.

2.2.1 Nakamoto Consensus

Broadly speaking, the idea of blockchain consensus is straightforward. A set of *stateful nodes* communicate with each other over a *network* in an asynchronous message-passing style. The goal of the nodes is to maintain and extend a *ledger* or *blockchain* of transactions and ensure agreement over its contents.

During normal operation, a node can either (a) announce a *transaction*, which typically represents a state update in the system (we intentionally leave out details of what goes into a transaction, as this is application-specific), or (b) create and broadcast a *block*, which builds on top of a previous block (identified by its hash value) and contains the encoding of a list of transactions to be appended to the blockchain. Crucially, each block has a notion of a *proof object*, which allows nodes in the network to independently verify that the block to which the proof object is attached was created in accordance to the rules of the protocol. Upon receiving a block, a node validates the sequence of transactions contained within it, checks the validity of the attached proof object, and then adds the block to its local state. This process continues as more messages are emitted and delivered.

Since nodes only accept blocks with valid proof objects, the network can control how often new (valid) blocks are created by adjusting how easy or difficult it is to create a proof object. The discipline by which blocks are created is dictated by a pair of locally-computable functions, *genProof* and the Validator Acceptance Function (*VAF*), that determine how often proof objects get created and, respectively, which proof objects are valid. Having a block b and its associated proof object pf, it is fast to check whether $VAF\ b\ pf$ is true or false. What is difficult, however, is producing a pf object that is valid with respect to a given block b. For example, in a proof-of-work scheme *genProof* hashes the given block b with a random nonce. If this passes the difficulty threshold set by *VAF*, *genProof* succeeds; otherwise it fails. Assuming the hash function used is

pre-image resistant, finding a suitable nonce is computationally hard. In other words, minting new blocks is probabilistically rare.

As we have seen, nodes create blocks and broadcast them to the network. However, this setup by itself does not deliver global agreement. Since the network has message delays, situations can arise in which two different nodes create two different, valid blocks that extend the same chain! This is called a *fork* — there are two conflicting blockchains in the network. Even if the network then has a period of synchrony where all block messages are delivered to all nodes, *i.e.* all nodes have knowledge of all blocks, the conflict will still exist. But the entire purpose of the protocol was to have all the nodes agree on *one* chain. Somehow, every node must decide which chain to adopt and they must all decide the same way. To do this, the nodes employ a third function, the Fork Choice Rule (*FCR*), which imposes a *strict total order* on all possible blockchains. In other words, given any two chains (including chains in the fork relation), *FCR* determines which of the chains the node should adopt as the "heavier" chain. Since after the period of synchrony, the nodes have full knowledge of all blocks and thus of all chains, they can use *FCR* to pick the "heaviest" chain they have, which by the strictness assumption is uniquely determined. Because they have the same blocks and the same *FCR*, all nodes will get the same result — and thus be in agreement.

Fork Choice Rule Strictness. As you can see in the informal argument above, the strictness assumption is crucial to guarantee agreement. It ensures that *FCR* can uniquely decide which chain is heavier in *any* situation. This assumption, which is a core assumption we make in Toychain, greatly simplifies our proofs, but does not match real-world implementations, which have situations in which the *FCR* cannot unambiguously decide between two chains — in such situations, consensus is not guaranteed. For example, in Bitcoin, two chains of the same length will typically* compare as equal. This means that global agreement is guaranteed only *if* the *FCR* imposes an unambiguous winner amongst all the chains in the network. If network delays are small and the time between consecutive block mintings is large (as they are for Bitcoin), one of the forks will eventually become larger and "win", so a non-strict *FCR* is not a problem in practice. Nonetheless, it complicates reasoning. The strictness assumption lets us ignore such complications.

2.2.2 Toychain by Example

The three functions *genProof*, *VAF*, and *FCR* are sufficient to ensure eventual agreement. Having explained what these functions do, we now give a step-by-step example of

*The exception is chains of the same length that cross difficulty-change boundaries.

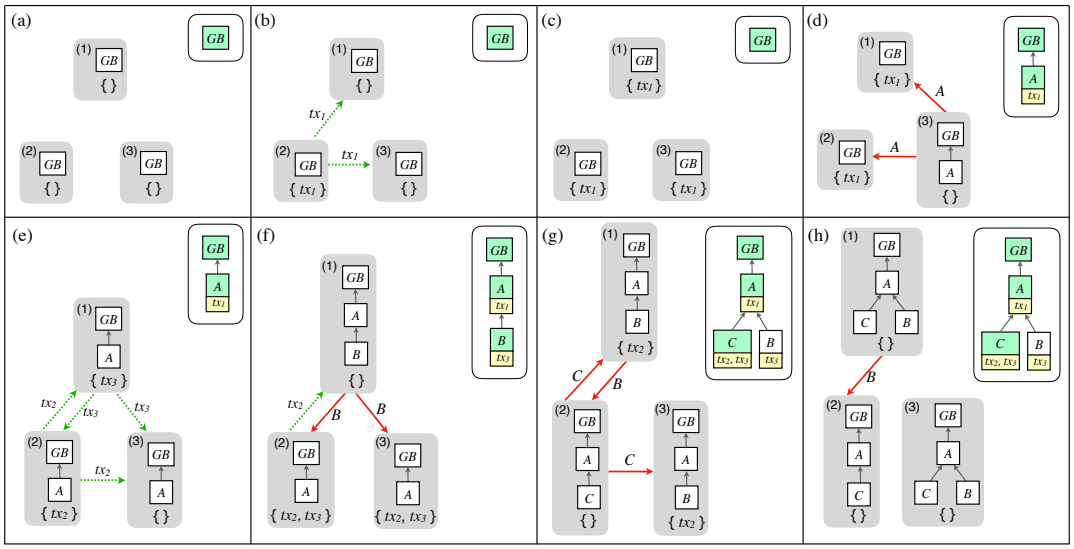

Figure 2.1: Stages of interaction in a blockchain network with 3 nodes. The (logical) "global" block forest is shown in the top-right corner for each stage.

the workings of a Nakmoto-style blockchain consensus protocol, as modelled in Toychain. Throughout this, we refer to the stages shown in Figure 2.1.

The goal of the protocol is to ensure that nodes agree on *one* ordering of transactions. For efficiency reasons, the protocol does not order transactions directly, but rather groups transactions into blocks and then orders the blocks. The nodes agree, using the *FCR*, on a blockchain, which implicitly imposes a single ordering of transactions.

The protocol execution starts with all nodes in agreement, as shown in stage (a) in Figure 2.1. They agree on the initial state of the system (*i.e.* everyone has the same genesis block *GB*) and have the same definitions for *genProof*, *VAF*, and *FCR*. Each stage in the figure also shows, in the top-right corner, a view of the *logical* "globally shared" tree of blocks, which all nodes will agree upon once all messages containing blocks (red arrows) are delivered. At any point, nodes can issue transactions. In stage (b), node (2) creates and broadcasts transaction tx_1 (dotted green arrow). Later, in stage (c), the messages containing this transaction are delivered and the transaction is included in the nodes' respective transaction pool, which is now $\{tx_1\}$. Nodes can also create ("mint") blocks. When a node mints and broadcasts a block, as in stage (d), it embeds the transactions in its transaction pool into the block. This can be seen in the top-right corner, where block A includes tx_1.

Fork Choice Rule is Additive. One assumption we make, which is implicit in how Figure 2.1 is drawn, is that adding a block to a chain always adds some strictly positive "weight", *i.e.* mining always improves your chain. For example, in stage (d), we assume that the chain [GB, A] is strictly "heavier" than [GB]. All real-world fork choice rules that we know of have this property, but it might be the case that it is not needed for protocol correctness.

Returning to the protocol execution example, since the system is distributed, multiple transactions can be issued concurrently. This is what happens in stage (e), where node (2) creates tx_2 and node (1) creates tx_3. Because of network delays, a node can mint a block without full knowledge of all transactions. This is what happens in stage (f), where node (1) creates block B before it receives tx_2. As such, block B only includes tx_3. Moreover, network delays also mean that blocks can be minted without full knowledge of other blocks. Stage (g) shows this happening, as node (2) mints block C before it receives block B. At this stage, a fork has been introduced in the network, as can be seen in the top-right "global" view. The same transaction, tx_3, has been included in two "competing" blockchains: [GB, A, B] and [GB, A, C]. In stage (g), the fork only exists in the "global" view — nodes are not aware of it. As block messages get delivered, however, nodes become aware of the fork, as seen in stage (h), and use the *FCR* on their local state to decide which of the competing chains to adopt. Since the fork choice rule imposes a *strict total order* on blockchains, it *uniquely determines* which of the two chains to adopt (for the sake of this example, we arbitrarily decide that *FCR* chooses [GB, A, C], shown highlighted in green in the "global" view). This means that in stage (h), nodes (1) and (3) agree to adopt chain [GB, A, C]. After stage (h), once all block messages are delivered, *i.e.* when the system is in a quiescent state with no block messages in transit, all the nodes will agree.

2.2.3 Modelling in Coq

Section 2.2.1 described the components that make up a Nakamoto consensus protocol and Section 2.2.2 showed how an execution of the protocol evolves stage by stage. This section explains how we represent all of this in the Coq proof assistant, to make it possible to write machine-checked proofs about the system's behaviour. Rather than reiterate the more formal presentation in the original Toychain paper, we present the model by directly listing Coq code and showing how it matches the description in the previous sections.

The core idea is that the state of the network (*i.e.* the individual stages in Figure 2.1) can be represented as a record data type, containing the state of each node and the list of "in flight" messages (*i.e.* the arrows in the figure). Moreover, transitions between

```
Record World :=
  mkW {
    localState : StateMap; (* Mapping from node ID to local state *)
    inFlightMsgs : seq Packet;
    consumedMsgs : seq Packet;
  }.
```

Listing 2.1: The network state consists of the local state of each node, the list of "in flight" messages, and a record of which messages have been delivered already.

```
Record State :=
  Node {
    id : nid;
    peers : peers_t;          (* Not shown in the figure *)
    blockTree : BlockTree;    (* Mapping from hash to block *)
    txPool : TxPool;
  }.
```

Listing 2.2: Each node's state contains the node's integer ID, a list of that node's peers, the local block forest data structure, and the node's transaction pool.

stages can be represented as an inductive data type with a separate constructor for each way the network state can evolve: either a network message is delivered or a node executes an internal action (issues a transaction or mints a block). For example, the transition between stage (a) and stage (b) involves node (2) issuing a transaction, tx_1. This is easier to show with code.

Listing 2.1 shows the full Coq definition of a `World`, which corresponds to one stage in Figure 2.1. The `localState` field corresponds to the rectangular blobs in the figure which show each node's state. Each of the blobs is represented by the data structure shown in Listing 2.2. The arrows in the figure correspond to `inFlightMsgs`. Interestingly, the "shared global state" in the top-right corner of each stage in the figure does not appear in the Coq definition. It will show up, however, when we start discussing the system invariant — the existence of the shared global state and its relation to each node's local state is a key part of the invariant.

There is one slight issue with the definition in Listing 2.1. An object of type `World` can represent one of the stages in Figure 2.1, but it can also represent junk, *i.e.* worlds that do not really make sense. For example, a `World` could have a node with an incorrectly constructed `BlockTree`, that has a mapping from some hash h to some block b, but with $\mathsf{hash}(b) \neq h$. We want to eliminate the possibility of such worlds when we reason about our protocol. To do this, we define a coherence predicate `Coh`, which lets us restrict our reasoning only to `World` objects that are correctly constructed.

This is enough to represent a static network, but several questions remain. How

```
Inductive system_step (w w' : World) : Prop :=
| Idle of Coh w & w = w'

| Deliver (p : Packet) (st : State) of
     Coh w &
     p \in inFlightMsgs w &
     find (dst p) (localState w) = Some st &
     let: (st', ms) := procMsg st (src p) (msg p) in
     w' = mkW (upd (dst p) st' (localState w))
              (seq.rem p (inFlightMsgs w) ++ ms)
              (rcons (consumedMsgs w) p)

| Intern (proc : nid) (t : InternalTransition) (st : State) of
     Coh w &
     find proc (localState w) = Some st &
     let: (st', ms) := (procInt st t) in
     w' = mkW (upd proc st' (localState w))
              (ms ++ (inFlightMsgs w))
              (consumedMsgs w).
```

Listing 2.3: A world `w` can evolve into `w'` in three different ways: (1) do nothing, (2) deliver a packet to a particular node, or (3) have a particular node execute an internal transition.

exactly can the network evolve, *i.e.* what is our network model? And how do we represent it in Coq?

Network Model. We model an asynchronous network, where packets might be rearranged or arbitrarily delayed, but make several simplifying assumptions about how the network behaves. In particular, we assume packets are never dropped or corrupted. At face value, this seems outrageous, but the assumption makes sense in our context. In a real-world deployment, nodes in Nakamoto consensus do not communicate directly to each other, but indirectly via a gossip mechanism. The gossip mechanism's job is to guarantee that blocks and transactions are delivered to all nodes, regardless of network conditions. In other words, the gossip mechanism allows nodes to pretend messages are not dropped or corrupted — we discuss this further in Section 2.2.5. We do not directly model packet duplication, but the protocol logic is defined in a way that tolerates message duplication, which is equivalent (packets wrap messages). Finally, we do not model Byzantine adversaries, *i.e.* malicious participants.

To represent the way the network can evolve (*i.e.* our network model) in Coq, we define an inductive data type, with a constructor for each way in which the network can change.

Listing 2.3 shows the different ways a world `w` can evolve into a different world `w'`. Each of these corresponds to one constructor of the `system_step` inductive data type.

```
1 Lemma property_step w w' :
2   property w → system_step w w' → property w'.
```

Listing 2.4: For an arbitrary `property` of worlds, this is the statement we need to prove to show that the property is an invariant. The proof proceeds by case analysis on the constructors of `system_step`.

We only define transitions for coherent worlds (lines 2, 5 and 14). Firstly, a world can do nothing and remain the same. This is the `Idle` constructor on line 2. Secondly, a world can deliver one of its "in flight" messages, `p` (line 6) to a node with state `st`. When this happens, the state `st`, which must be the actual state of the destination node (line 7), is updated as dictated by the message handler `procMsg` (line 8). The message handler returns the node's new state `st'` and a list of messages the node sends `ms`. The new world, `w'`, is obtained from the original world `w` by updating the node's state (line 9), removing `p` and adding `ms` to the "in flight" messages (line 10), and marking message `p` as consumed (line 11). This is the `Deliver` constructor on lines 4–11. Finally, a world can process an internal transition, *i.e.* either issue a transaction or create a block. The transitions are handled by the `procInt` handler (line 16). Note that the "in flight" messages have no inherent order: on line 8, `ms` is appended, whereas on line 18 it is prepended. The `Coh w` predicate, which is a precondition for all three kinds of system step, requires that the original world `w` is coherent, *e.g.* the node IDs are consistent, all nodes have the genesis block GB, each node's local state is valid *etc.*

With this infrastructure in place, we can begin stating properties of the Toychain system and proving them by induction. Listing 2.4 shows the general induction scheme: if a `property` holds on an arbitrary world `w`, `w` evolves into `w'` (as defined by `system_step`), and the `property` holds on `w'`, then `property` is an invariant. In other words, proving the statement on line 2 of Listing 2.4 establishes `property` as an invariant of the system. After that, it suffices to establish that `property` holds of the initial state of the system in order to prove that it holds at any point in the protocol's execution.

It should now be clear how we model Toychain in Coq and how we prove properties of the system. In essence, we represent stages of the protocol execution as values of the `World` data type and transitions between stages as different constructors of the `system_step` data type. Then, we establish invariants by induction over `system_step`, *i.e.* by considering every possible way in which the system can evolve. This is conceptually simple, but induction proofs over the state of an entire distributed system are, in practice, very messy. Finally, we show that the invariants we establish imply desired systems properties, *e.g.* quiescent consistency ("when all block messages are delivered, everyone agrees").

One question about the Coq model remains unaddressed. Where do *genProof*, *VAF*,

```
1  Definition procInt (st : State) (tr : InternalTransition) :=
2     let: (Node n prs bt pool) := st in
3     match tr with
4     | TxT tx ⇒ pair st (emitBroadcast n prs (TxMsg tx))
5
6     (* Assumption: nodes broadcast to themselves as well! ⇒ simplifies logic *)
7     | MintT ⇒
8       let: bc := (btChain bt) in
9       let: attempt := genProof n bc in
10      match attempt with
11      | Some(pf) ⇒
12         if VAF pf bc then
13           let: allowedTxs := [seq t <- pool | txValid t bc] in
14           let: prevBlock := (last GenesisBlock bc) in
15           let: block := mkB (hashB prevBlock) allowedTxs pf in
16           if tx_valid_block (btChain bt) block then
17             let: newBt := (btExtend bt block) in
18             let: newPool := [seq t <- pool | txValid t (btChain newBt)] in
19             let: ownHashes := (keys_of newBt) ++ [seq hashT t | t <- newPool] in
20             pair (Node n prs newBt newPool) (emitBroadcast n prs (BlockMsg block))
21           else
22             pair st emitZero
23         else
24           pair st emitZero
25      | _ ⇒ pair st emitZero
26      end
27    end.
```

Listing 2.5: Handler for node internal transitions: either (1) issuing a transaction or (2) *attempting* to mint a block.

and *FCR* show up? They have appeared nowhere thus far. The answer, unsurprisingly, is that they show up in the definition of the protocol, *i.e.* in `procMsg` and `procInt`. Until now, we have only shown the definition of the network semantics. The model of the protocol is given by the executable functions `procMsg` and `procInt`, which internally make use of *genProof*, *VAF*, and *FCR*. For example, lines 9 and respectively 12 in Listing 2.5 show the process of attempting to create a valid proof object to attach to a newly-created block (line 15), as well as the change to the node's local state and corresponding block broadcast (line 20). This uses *genProof* and *VAF*. Similarly, `btChain`, a function which returns the "heaviest" chain in the local `BlockTree`, internally uses *FCR*. The `MintT` transition attempts to create a new block building on top of this "heaviest" chain, `bc` (lines 8–9). If a proof object is produced, it can be attached to a new block which extends the last block of `bc` (lines 14–15).

The `procMsg` message handler is defined in a similar fashion.

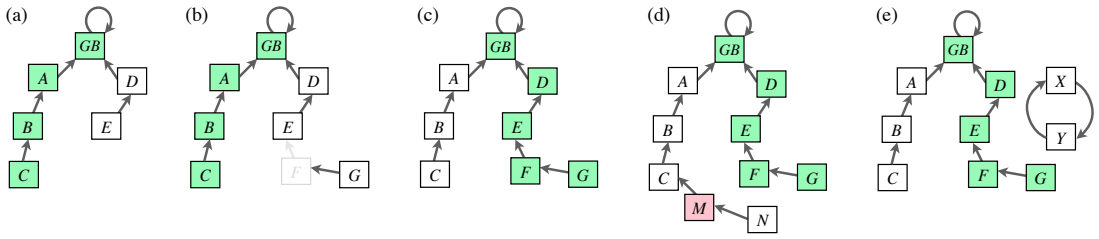

Figure 2.2: Different states of a valid block forest and its heaviest chain (green). States (d) and (e) are abnormal and can only come to exist in the presence of malicious participants. The red block in state (d) contains invalid transactions. The abnormal states *are* valid, *i.e.* consistent instances of the type.

2.2.4 Block Forests

The core data structure in the Toychain protocol is the "block forest" that nodes use to store the blocks they have heard of. Figure 2.2 shows a valid (*i.e.* consistent) block forest in different states, growing as blocks are added to it. The gray arrows depict the hash-links between blocks, *i.e.* each block's `prev` field. The heaviest chain in the forest (according to *FCR*) is shown in green. Stage (a) shows a valid block forest with largest chain [GB, A, B, C]. In stage (b), the forest is extended by adding a new block, G, which points to another block F (shown in light gray) which has not yet been added to the block forest. This can happen due to out-of-order message delivery. In this case, G is called an "orphan" block.

Forests vs Trees. To align with prevailing terminology, the block forest data structure is called `BlockTree` in the Toychain source code. Real-world node implementations such as Bitcoin Core only add blocks at the tips of their chains, *i.e.* their forests are always trees. For example, in a situation like stage (b), Bitcoin Core would reject block G — it would first wait to receive block F. This seems harmless, but has important implications for the correctness of the protocol. Accepting orphan blocks, as Toychain does, guarantees that message-delivery order does not matter. Rejecting them, as real-world implementations do, protects against spam attacks (if F does not exist, the node has accepted junk), but is correct only if G can be retrieved later. Generally speaking, G can indeed be retrieved via gossip, but it is nonetheless important to explicitly point out the assumption. For full generality, the node's local data structure is a forest, not a tree.

Coming back to Figure 2.2, stage (c) shows that when block F is finally delivered, the heaviest chain in the local block forest (now a tree) is [GB, D, E, F, G]. Stage (d) shows a block M (red) that contains an invalid transaction. The block forest is still valid (*i.e.* correctly defined), but contains an invalid block, *i.e.* a block that will never be part of

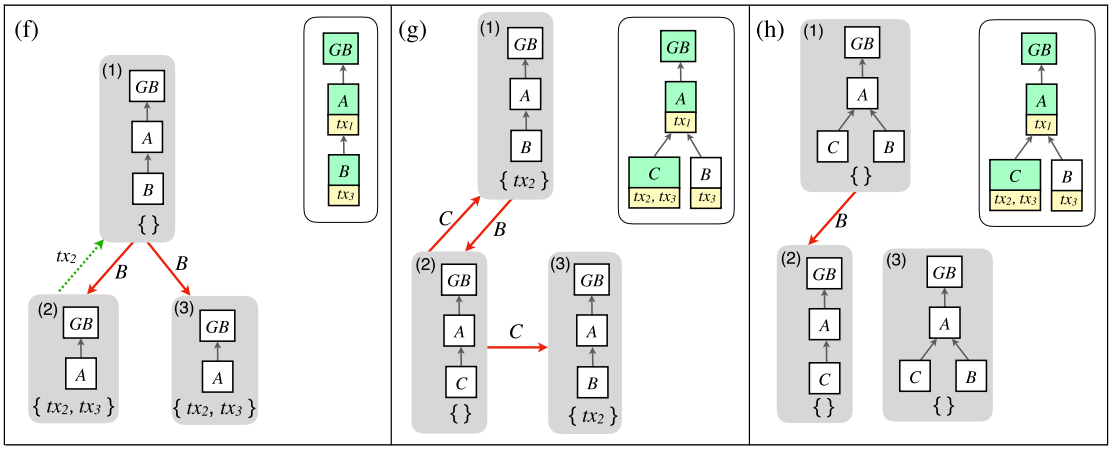

Figure 2.3: The (logical) "global" block forest is shown in the top-right corner for each stage. For any node, when you add the "in flight" block messages to that node's state, you obtain the "global" block forest — *the law of block conservation*.

the heaviest chain. Only malicious nodes can create invalid blocks. Similarly, stage (e) shows a valid block forest, with two blocks X and Y that form a cycle. Neither of them will ever be considered as part of a candidate chain.

KEY PROPERTIES. The operation that adds a block to a block forest has two key properties: it is *idempotent* and *commutative*. Block-addition being idempotent means that adding the same block b multiple times to a block forest is equivalent to adding it once. This allows for graceful handling of message duplication — duplicated blocks are simply ignored. Similarly, the fact that block-addition is commutative means that message delivery order is unimportant. As long as two nodes receive the same blocks, they will have the same block forest, no matter the order in which they received the blocks. These two properties are crucial in the formulation of our system invariant in Section 2.2.5.

For a full description of the block forest data structure and the operations that can be performed on it, please refer to the original Toychain paper.

2.2.5 System Invariant

What is the strongest invariant property we can prove for the Toychain model? One way to discover invariants is to look at what is "conserved" throughout the execution of the protocol. Figure 2.3, which shows a possible execution of the Toychain protocol, can give us some ideas. For example, if we assume that all nodes in the network are directly connected to each other, *i.e.* the network graph is a clique, the following statement is invariant: there exists a logical ("ghost") global block forest that is the union of all local

block forests *and*, for each node, if you add the "in flight" block messages to that node's state, you obtain the global block forest. This property, which we call *the law of block conservation*, is an invariant.

Gossip Protocol. The law of block conservation relies on two assumptions: (1) the network graph is a clique and (2) block messages are not dropped. Clearly, neither of these two assumptions is realistic. Nonetheless, the property is still very useful, since the Toychain protocol has a mechanism — the gossip protocol — to ensure that block messages are propagated as if the network was a clique. The gossip protocol guarantees that all blocks are available to all nodes even if block messages are dropped, as long as the network graph is connected. We have not formally verified that the gossip mechanism in Toychain provides this guarantee, but in Chapter 4 we empirically validate the claim. As such, morally[†], if the gossip mechanism is correct, then the law of block conservation holds in an altered form: for each node, if you add the blocks "available" to that node through the gossip protocol to the node's state, you obtain the global block forest.

In fact, there is a stronger invariant than the law of block conservation. Figure 9 in the original Toychain paper states it formally [PS18], but here we offer an informal description.

Clique Invariant. At any point in the execution of the protocol, there exists a logical global block forest $\widehat{bf}$, a chain c, and a "canonical" node n which holds this chain, such that:

(1) the law of block conservation holds with respect to $\widehat{bf}$

(2) $\widehat{bf}$ is a tree and its heaviest chain is c

(3) any node has chain c or a less heavy chain

(4) the canonical node n has chain c

One direct implication of the clique invariant (and indeed, of the law of block conservation) is *quiescent consistency*: when all block messages are delivered, every node agrees on what chain to adopt.

Understanding the proof that the clique invariant is inductive is not necessary to understand the rest of this thesis, but we nonetheless present an outline. There are two key ideas. First key idea: $\widehat{bf}$ is always a tree and any local block forest is a subset of $\widehat{bf}$ (this is implied by the law of block conservation). The `Deliver` system step preserves

[†]We have not found a good abstraction for the gossip protocol, so have been unable to formally verify this claim.

this: delivering a message (1) does not change $\widehat{bf}$ and (2) follows the law of block conservation (for one node, a block message being delivered moves the block from "in flight" to the node's local block forest). Moreover, delivering a message does not change which chain is heaviest or which node holds the heaviest chain, *i.e.* c and n remain unchanged. The `Intern` system step is trickier to handle, as it may change $\widehat{bf}$, c, and n. Second key idea: because $\widehat{bf}$ is a tree, mining locally is the same as mining globally. An individual node mining is equivalent to the global block forest (*i.e.* the "network", abstractly) mining — a valid extension of the local chain is a valid extension of *some* chain in the global block forest, *i.e.* mining only happens at the tips of the global block forest (*i.e.* mining keeps $\widehat{bf}$ a tree). As such, if the newly-mined chain c' is heavier than the previously heaviest chain in the network, then c' becomes the new heaviest chain (and n potentially changes). Otherwise, c remains the heaviest chain and n remains unchanged. Since the node that mined the block also broadcast it, the law of block conservation is conserved as well. This concludes the proof. The "canonical" node's existence shows that there is always at least one node which holds the heaviest chain, *i.e.* the heaviest chain is not just a logical artefact, but a concrete value that specific nodes see. If you are interested in the details, check the Toychain paper or, better yet, the Coq source.

2.2.6 Dropped Assumptions

While the proof sketch in the previous section might seem straightforward, the devil is in the details. Indeed, one of the benefits of an interactive theorem prover is that it forces you to consider all the corner-cases that are easy to miss in a pen-and-paper proof. Moreover, it forces you to precisely identify your assumptions. In the previous sections, we have described two of Toychain's crucial assumptions, namely that the fork choice rule is strict and additive. In this section, we review some assumptions we made in the original Toychain and explain the reasoning behind them, in anticipation of Chapter 3, where we will remove these assumptions.

Genesis Block Hash. In the original Toychain, we make an assumption that is both unreasonable and, with the benefit of hindsight, pointless. If you recall Figure 2.2, there is a peculiarity we have not yet mentioned — the genesis block points back to itself! In other words, $\texttt{prev}\ GB = \texttt{hash}(GB)$. The idea behind this assumption is that it slightly simplifies the logic for computing the chain that ends at a particular block b. Since blockchains cannot have cycles, the recursive function `compute_chain` stops whenever it encounters a cycle. The GB-hash assumption then guarantees that the genesis block can only appear at the beginning of a chain. Of course, `compute_chain` can be defined

such that the GB-hash assumption is not needed. Moreover, the GB-hash assumption is completely unrealistic. If hash is a cryptographic hash function, it is actually impossible (or at least computationally infeasible) to create a genesis block.

Valid Chains Have No Cycles. Another assumption we make in the original paper is that *VAF* will not accept a block b as a valid extension of a chain c if $b \in c$. In other words, you cannot "reuse" blocks. This assumption, called *VAF_nocycle*, is perfectly reasonable, but as it turns out, stronger than necessary. As we have previously said, blockchains, irrespective of their validity, cannot have cycles — this follows from their definition as chains of blocks linked by hashes. With this, *VAF_nocycle* reduces to saying that *VAF* will accept the genesis block only at the beginning of a chain. We prove this reduction in Chapter 3.

Hash is Injective. The most interesting assumption in the original Toychain is the *hash_inj* assumption, which states that if two blocks have the same hash, then they are in fact the same block. If you are familiar with hash functions, you will notice that this assumption is *false*. Hash functions do have collisions! Collisions are probabilistically rare and computationally infeasible to produce, but they do exist. However, when verifying a blockchain consensus protocol (which makes extensive use of hashes), it is very convenient to pretend that hash collisions do not exist. For example, it allows you to represent a node's local block forest as a map from hashes to blocks and easily prove that additions to this data structure are commutative. As you recall, this is one of the key properties needed for the protocol's correctness. But the reality still stands — the assumption is false! Does this undermine the entire Toychain formalisation? Are all our proofs junk[‡]? Yes and no. Making this assumption does in fact undermine our claims of correctness, since you now have to trust us not to have misused it to derive falsehood. If we have, then all we have accomplished is a complicated, mechanically-checked proof of False. But our proofs are not junk. In fact, we have been careful to use the *hash_inj* assumption only in sound ways. One of the major accomplishments of the present work is to *prove* this claim by completely removing the *hash_inj* assumption from the Toychain development. Our explanation of this effort forms the bulk of Chapter 3.

[‡]Assuming falsehood lets you prove anything.

3
Toychain Made Realistic

> *I have learned throughout my life as a composer chiefly through my mistakes and pursuits of false assumptions, not by my exposure to founts of wisdom and knowledge.*
>
> Igor Stravinsky, Russian composer

This chapter explains our work to make the Toychain model realistic by removing unreasonable assumptions. If you have not read Section 2.2.6 of Chapter 2, please do.

3.1 Unrealistic Assumptions

The original Toychain development makes two assumptions that no implementation can possibly satisfy: *GB_hash* and *hash_inj*. These say that the genesis block points back to itself and, respectively, that the hash function is injective. Before we can extract a proven-correct implementation of Toychain, *i.e.* an implementation that we can prove satisfies all the model's assumptions, we need to remove these.

3.1.1 Genesis Block Hash

```
Axiom init_hash : prevBlockHash GenesisBlock = #GenesisBlock.
```

Listing 3.1: The *GB_hash* assumption in Coq. $\#GB$ is shorthand for hash(GB).

We made the *GB_hash* assumption very early in Toychain's development and without much thought, in order to simplify the definition of the `compute_chain` function. The assumption meant that `compute_chain` terminated only when it encountered a cycle — and we made use of this implication in several proofs. As such, removing the assumption, rather than just being a case of trivially redefining `compute_chain`, involved rewriting some important proofs. For example, the proof of `btExtend_chain_prefix` (whose statement is shown in Listing 3.2) heavily relies on the termination condition of `compute_chain`. Removing the assumption means changing the termination condition, which necessarily changes the proof. Other proofs had to be changed in a similar fashion.

```
Lemma btExtend_chain_prefix bt a b :
  valid bt → validH bt →
  ∃ p, p ++ (compute_chain bt b) = compute_chain (btExtend bt a) b .
```

Listing 3.2: Adding a block a to a block forest possibly "fills gaps", with some blocks no longer being orphans, *i.e.* existing chains (which did not extend all the way back to GB) can become longer.

```
Definition BlockTree := union_map Hash block.
```

Listing 3.3: Block forests are defined as maps from hashes to blocks. This is a valid representation only if hash collisions do not occur. If they do, we cannot represent the resulting forest.

Removing the *GB_hash* assumption also forced us to write some new proofs. Interestingly, in these proofs, we need to explicitly handle the corner-case where `prev` $GB =$ hash(GB), *i.e.* where *GB_hash* is true. For example, one of the properties of blockchains is that they do not have cycles, *i.e.* blocks only appear once. One implication of this is that blocks in a chain do not have self-cycles, *i.e.* `prev` $B \neq$ hash(B). The first block in a chain, however, can be an exception. We need to handle this case explicitly.

3.1.2 Hash Injectivity

Compared to *GB_hash*, the *hash_inj* assumption is not only meaningful, but also central to the entire Toychain formalisation. Indeed, to remove the hash injectivity assumption, we had to make changes in every single proof. Moreover, we had to rewrite almost all of the functions that operate on block forests, in order to make them resilient to hash-collisions. This was a significant undertaking.

In our Coq development, we represent block forests as maps from hashes to blocks, as shown in Listing 3.3. At first impression, it might seem like this is an issue, since the definition itself fails if a hash collision does indeed occur — we cannot represent the resulting block forest. However, this limitation is not significant. To see why, consider what happens when a hash collision occurs during the execution of the protocol. For one, the very definition of a blockchain breaks down. If a block's `prev` field is h and there are two different blocks with hash value h, which chain is being represented? It could be either. What can we do?

Remember, the *hash_inj* assumption is false. Removing it means we have to redesign the protocol such that it "works" even if hash collisions do happen. How can we do this? Well, the easiest way is to say that, when a hash collision does happen, everything breaks. Crucially, though, everything needs to break for all nodes and in the same way for each node — we still need agreement. This is the solution we adopt. When a hash collision does occur, all local block forests become invalid and all the nodes agree on the

```
1 Definition btExtend bt b :=
2   (* Only add "fresh" blocks *)
3   if #b \in dom bt then bt
4   else (#b \\→ b \+ bt).
```

Listing 3.4: If hash collisions cannot happen, it suffices to check whether the hash is already included in the forest.

```
1 Definition btExtend bt b :=
2   (* Only add "fresh" blocks *)
3   if #b \in dom bt then
4     if find (# b) bt == Some b then bt
5     (* Hash collision → undefined *)
6     else um_undef
7   else (#b \\→ b \+ bt).
```

Listing 3.5: When a hash collision happens, we invalidate the entire forest.

```
valid bt = valid (btExtend bt b)
```

Listing 3.6: Forward reasoning.

```
valid (btExtend bt b) → valid bt
```

Listing 3.7: Backward reasoning.

trivial chain consisting of only the genesis block. Importantly, everything is guaranteed to break only when block messages are delivered directly, rather than via the gossip mechanism, since the current (unverified) implementation of the gossip mechanism uses hashes as identifiers for its advertise-and-pull functionality. If a collision occurs, a node which has already seen that hash will not request it again and thus will never see the colliding block. It *is* possible to design a Nakamoto-style consensus protocol that continues to function after a hash collision, but this requires extra effort, and is mainly of theoretical, rather than practical interest. To our knowledge, no current real-world implementation of Nakamoto consensus tolerates hash collisions in blocks.

To see the difference between the old logic to extend block forests and the new, hash-collision aware logic, compare Listing 3.4 with Listing 3.5. They are different in two significant ways. Firstly, the old `btExtend` could assume that if $\mathsf{hash}(b)$ is in the map's domain, then b is included in the forest. Obviously, this assumption no longer holds if hash collisions can happen. This is why line 4 exists in Listing 3.5 — it checks that it is indeed b that is included, rather than some other block that hashes to $\mathsf{hash}(b)$. Secondly, and much more importantly, the new `btExtend` can return invalid block forests (line 6), whereas the old definition only returned valid ones. This is a *major* difference and requires changing all proofs.

With the old definition, a valid block forest could be extended in any way and it would remain valid. Since the initial block forest for any node is valid, the node's block forest would remain valid, unconditionally, throughout the execution of the protocol. With the new definition, however, this reasoning principle no longer holds. In fact, we have to switch from forward-reasoning to backward-reasoning. Compare Listings 3.6 and 3.7. This is truly unfortunate, since `valid bt` is a hypothesis in all of the proofs, *i.e.* the reasoning principle needs to change in all of the proofs.

```
Lemma btExtendV_fold_comm (bt : BlockTree) (xs ys : seq block) :
  valid (foldl btExtend (foldl btExtend bt xs) ys) =
  valid (foldl btExtend (foldl btExtend bt ys) xs).
```

Listing 3.8: We want to prove that, *wrt.* validity, adding lists of blocks is commutative. If hash collisions can happen, proving this is non-trivial.

```
1 valid (foldl btExtend (foldl btExtend bt xs) ys) =
2 valid (foldl btExtend (foldl btExtend bt ys) xs).
```

Listing 3.9: Induction hypothesis.

```
1 valid (foldl btExtend (foldl btExtend bt (rcons xs x)) ys) =
2 valid (foldl btExtend (foldl btExtend bt ys) (rcons xs x)).
```

Listing 3.10: Goal for the inductive step.

The change from the old definition and reasoning principle to the new is, in some cases, very painful. For example, consider the key property that the block-addition operation is commutative. A proof that used to be 10 lines long becomes 150 lines long! Moreover, proving that the addition of lists of blocks is commutative becomes not only painful, but actually theoretically interesting, *i.e.* difficult. To see why, let us look at a related property.

Consider the statement in Listing 3.8, which is the property we want to prove. It says that if you have two different lists of blocks, it does not matter in which order you add the lists to your forest — at the end, either both resulting forests will be valid or neither will be valid. Typically, statements such as this one are proven by induction. The inductive step gives you an induction hypothesis such as the one shown in Listing 3.9 and requires you to prove the statement in Listing 3.10. If you are being attentive, you will notice the issue. It is the issue we have been talking for the last three paragraphs. We can no longer do forward-reasoning for validity. Listing 3.9 does not imply Listing 3.10. Nonetheless, the property is true and we do want to prove it.

The problem is that validity is not an inductive property. To see why, consider the following. Imagine you are the computer and you are executing both lines 1 and 2 in Listing 3.9 simultaneously. The issue is that, at some point in the execution, line 1 can encounter a hash collision and become invalid, while line 2 keeps executing and adding more blocks before *it* becomes invalid. In other words, the two lines are guaranteed to have the same validity only at the end of the execution, not "in the middle". How do we get around this? We transform validity into an inductive *almost*-equivalent. This is

```
1  Definition no_collisions (bt : BlockTree) (xs : seq block) :=
2    valid bt ∧                                        (* You start with a valid bt *)
3    ∀ a, a \in xs →
4      (∀ b, b \in xs → # a = # b → a = b) ∧           (* No collisions within xs *)
5      (∀ b, b ∈ bt → # a = # b → a = b).              (* Or between xs and bt *)
6  (* The following statements are true: *)
7  valid (foldl btExtend bt xs) → no_collisions bt xs
8  validH bt → no_collisions bt xs → valid (foldl btExtend bt xs)
```

Listing 3.11: This property is slightly weaker than validity, but is inductive. The two notions are not equivalent because of the `validH bt` hypothesis on line 8, which says that if hash h maps to block b in `bt`, then $\mathsf{hash}(b) = h$. In other words, `validH bt` means that `bt` is correctly constructed.

the theoretically interesting part.

Listing 3.11 shows our inductive *almost*-equivalent of validity. Remember, a valid block forest becomes invalid only if a collision happens during block-addition.

If starting with some block forest `bt`, you make a list of additions `xs` and the result is valid, that means that no collisions were introduced. This is what line 7 says. More precisely, it says that the original forest was, via the backward-reasoning principle in Listing 3.7, valid (line 2), that there were no collisions between blocks in `xs` (line 4), and that there were no collisions between blocks in `xs` and the blocks in the original forest `bt` (line 5).

Similarly, the fact that no collisions were introduced when adding a list of blocks `xs` to some block forest `bt` implies that the resulting block forest is valid, *as long as* the original block forest `bt` was correctly constructed. This is what line 8 says. To understand why the extra condition is required, look at line 5. Now suppose `bt` is incorrectly constructed, such that it has a mapping from some hash h to some block b, but with $\mathsf{hash}(b) \neq h$. This is a valid block forest. Moreover, $b \notin$ `bt`, since the $\in$ operator checks for "correct" inclusion. In this situation, if `xs` contains some block that hashes to h, adding this block introduces a hash collision that neither line 4 nor line 5 rule out. It is a very subtle observation. Indeed, if we did not have mechanically-checked proofs, we would have missed this detail and would have come to the incorrect conclusion that `no_collisions` is equivalent to validity.

To prove the lemma in Listing 3.8, we add an extra hypothesis: `validH bt`. This hypothesis does support forward-reasoning, *i.e.* `valid bt` ∧ `validH bt` implies `validH (btExtend bt b)`. With this extra hypothesis, and the *almost*-equivalence between `no_collisions` and validity, we can prove the goal in the inductive case (that Listing 3.9 implies Listing 3.10) and thus prove the overall commutativity lemma. This is great, but not perfect. We claim that the lemma in Listing 3.8 is true even without the `validH bt` hypothesis — but we do not know how to prove it.

We conclude this section by showing how removing the hash injectivity assumption affects the clique invariant, which we recall below.

Original Clique Invariant. At any point in the execution of the protocol, there exists a logical global block forest $\widehat{bf}$, a chain c, and a "canonical" node n which holds this chain, such that:

(1) the law of block conservation holds with respect to $\widehat{bf}$

(2) $\widehat{bf}$ is a tree and its heaviest chain is c

(3) any node has chain c or a less heavy chain

(4) the canonical node n has chain c

Of course, it is now possible for the global block forest to become invalid. How does this affect the invariant? It is fairly simple, actually. If there is no hash collision, the updated invariant has the same clauses as the original. However, if a hash collision *has* occurred, then the invariant essentially degenerates into the law of block conservation. More concretely, the hash-collision resistant invariant is as follows.

Clique Invariant Under Hash Collisions. At any point in the execution of the protocol, there exists a logical global block forest $\widehat{bf}$, a chain c, and a "canonical" node n which holds this chain, such that:

(1) the law of block conservation holds with respect to $\widehat{bf}$

(2′) Either $\widehat{bf}$ is valid and

  (a) $\widehat{bf}$ is a tree and its heaviest chain is c

  (b) any node has chain c or a less heavy chain

  (c) the canonical node n has chain c

(2″) Or $\widehat{bf}$ is invalid

If there is no hash collision, in a quiescent state, the nodes will agree on the largest chain c. Otherwise, if there *is* a hash collision, when all block messages are delivered, everyone's local block forest will become invalid and the nodes will agree on the trivial chain consisting only of the genesis block.

```
Axiom VAF_nocycle :
  ∀ b bc,
    VAF b bc → b \notin bc.
```

Listing 3.12: The *VAF_nocycle* assumption is stronger than needed.

```
Axiom VAF_GB_first :
  ∀ bc,
    VAF GenesisBlock bc → bc = [::].
```

Listing 3.13: We want to reduce *VAF_nocycle* to this weaker version.

```
1 Lemma hash_chain_uniq_hash_nocycle b bc :
2   hash_chain (b :: bc) →              (* Correct hash-links *)
3   uniq (map hashB (b :: bc)) →        (* No hash-collisions *)
4   (∀ c, c \in bc → prevBlockHash c != hashB (last b bc)).
```

Listing 3.14: Correctly-constructed chains have no cycles, except perhaps from the first block.

3.2 Strong Assumptions

Finally, there are two assumptions that we have identified in Chapter 2 as being stronger than they need to be: FCR-strictness and *VAF_nocycle*. We remove the latter assumption, but leave removing the former for future work.

In essence, *VAF_nocycle* (shown in Listing 3.12) makes a claim that is too strong. Any reasonable *VAF* implementation does satisfy this claim, but that could be tricky to prove. Since in Chapter 4 we will, in fact, write a *VAF* implementation and will have to show that it satisfies the assumptions that the Toychain model makes about it, we might as well save ourselves some time and reduce *VAF_nocycle* to its weakest precondition* (shown in Listing 3.13) right now.

The idea is straightforward, but its proof is interesting. Blockchains, if correctly constructed, are hash-chains. They are a series of blocks linked by hashes, and each block has its own hash — there are no hash-collisions within the chain. It follows from this that there are no cycles within the chain, *i.e.* the lemma in Listing 3.14 is true. Looking at line 4 in the listing, you might wonder — is it not too weak? It only says that there are no cycles to the last block in the chain. What about cycles in the middle of the chain? Well, the lemma holds for any `bc`, so if you truncate `bc` appropriately, you can apply the lemma to say that there are no cycles anywhere in the chain, *i.e.* correctly-constructed blockchains have no cycles, except perhaps from the first block.

With this, the reduction we intended follows. For correctly constructed chains, `VAF_nocycle` is equivalent to the weaker version `VAF_GB_first`, since only the first block in a chain can have a cycle. With this, we are now ready to proceed to Chapter 4 and create an instance of Toychain that we *prove* satisfies all the model's assumptions.

*Listing 3.13 is a weakest precondition of *VAF_nocycle* only for correctly-constructed chains. This is fine for us, since Toychain only calls *VAF* on chains returned by `compute_chain`.

4

Toychain Made Practical

Talk is cheap. Show me the code.

Linus Torvalds

This chapter covers our effort to extract a proven-correct implementation of Toychain. We precisely describe the trusted computing base, highlighting which parts of the system need to be trusted, and explain the limitations of the current implementation.

4.1 From Proof to Program

One of the choices we made early on while designing Toychain was to actually *implement* the core parts of the protocol in Gallina, Coq's specification language. This way, our proofs both show the correctness of the protocol and give us a proven-correct implementation of the protocol. Nonetheless, transforming an executable specification into practical software that you can run on your computer is not straightforward.

Our expectation was that moving from the abstract model to a concrete instance of Toychain would, more or less, boil down to writing implementations for functions left unspecified in the model and proving that our implementations satisfy the model's assumptions. Indeed, at a high-level, this *is* what it boiled down to. But we ran into several difficulties. For example, we found that instantiating some of the unspecified protocol parameters, *e.g. FCR*, would break many of our existing proofs. The proofs expected functions to have opaque, not transparent, definitions. To work around this opacity issue and parameterise our existing proofs over definitions, we had to refactor Toychain to use Coq's confusing module system. We describe this in Section 4.2. Also, to our surprise, the main challenge in getting runnable code was not instantiating the functions we left unspecified in the formalisation, but instantiating the types. Since Coq is dependently-typed, whereas OCaml is not, this is harder than it sounds — most of the types we used in Coq do not exist in OCaml. In Section 4.3 we explain how we overcome this issue, instantiate Toychain with a specific implementation of proof-of-work Nakamoto consensus, and prove it satisfies the model's assumptions. This gets us Coq code, but we still have to transform it to OCaml and then into an executable.

Section 4.4 discusses the challenge of extracting OCaml code from the instantiated Toychain — including working around two separate bugs in the extraction mechanism! Finally, Section 4.5 shows how to take this verified consensus protocol logic and build an actual implementation out of it, with all the TCP/IP bits and the associated problems, including differences between how OCaml treats Linux system calls when it is interpreted, as compared to when it is compiled to native code.

Despite the difficulties, transforming Toychain into runnable software has been tremendously satisfying. The code works. We have computers on a LAN talking to each other, running a formally verified blockchain consensus protocol and reaching agreement. That is seriously cool.

Furthermore, having runnable code also allowed us to validate (empirically) the parts of the protocol which we have not yet verified — crucially, the gossip mechanism. This is particularly important since, in normal operation of the protocol over the Internet, blocks propagate through gossip rather than direct messages. Moreover, if a node goes offline and then comes back at a later time, it *has to* engage in gossip to catch up.

4.2 Coq Modules and Parametricity

Toychain is, in fact, not a single consensus protocol, but a family of consensus protocols parametric in terms of the implementation of several security primitives: a hash function, a fork choice rule, a validator acceptance function, and a notion of a proof object. For an introduction to these terms, read Section 2.2.1 of Chapter 2. To get a verified executable *instance* of Toychain, we have to provide a concrete implementation of these primitives and prove that they satisfy the assumptions our formal model makes about them.

4.2.1 Mechanised Proofs are Brittle

In the Toychain development, $hash_b$, FCR, and VAF are defined as `Parameter`s. When Coq encounters these symbols in a proof, it treats them as completely opaque — they have a type, but no definition. Then, when we instantiate these parameters by giving them concrete `Definition`s, they cease to be opaque to Coq. One implication of this is that Coq's tactics try to be helpful and unfold definitions whenever they can. This can be averted somewhat by declaring definitions to be `Global Opaque`, but this only affects Coq's simplification tactics; the behaviour of most tactics is unchanged [App18].

While this may seem harmless, it actually breaks several of Toychain's current proofs. As with most mechanised proofs, our proofs are very brittle in the face of changes in the program being reasoned about — and replacing an opaque definition with a transparent one is a big change. Because the structure of the proof is very tightly

linked with the structure of the program, as the program changes, mismatches break the proofs. The changes required in the proofs are almost always trivial, but time-consuming. Overcoming this inherent brittleness is an active area of research.

4.2.2 Modules Preserve Opaqueness

To overcome the issue of definition transparency without making modifications to the existing proofs, we have to use Coq's module system, which gives us a way to "parameterise proofs over structures" [Let17]. Coq's module system is similar to, for example, Java interfaces. You can define a `Module Type` to specify the module signature*, *i.e.* which types and functions it provides, and then state the proofs in terms of that module signature. Then, when you want to instantiate the proofs for a particular implementation, you define a `Module` which satisfies the `Module Type`. In this fashion, proofs work as if they were operating on `Parameter`s, but can be instantiated to particular `Definition`s when needed.

A Note about "Modules"

One point of confusion is that Coq overloads the word "module" to refer to two different concepts: the one we just described (which is used rarely), but also the typical software engineering practice of breaking down software into separate files that `Import` from each other. Coq "modules" in the second sense are not modules in the first sense. This differs, for example, from OCaml, where, for example, a file called `fintype.ml` automatically defines a module in the first sense, with the associated parametricity and definition-hiding.

Throughout the rest of this chapter, I will refer to definition-hiding modules simply as modules, whereas modules in the second sense (Coq code within a file) will be written in quotes: "modules".

4.2.3 Toychain Made Modular

At a high level, Toychain is made out of several components: type definitions, consensus parameters, a library to manipulate block forests, protocol logic, network semantics, and statements and proofs of network invariants.

These components form a hierarchy. The consensus parameters depend on the definitions, the block forest library depends on the definitions and consensus parameters, and so on. As such, the development lends itself well to modularity.

*When referring to modules, "type", "signature" and "interface" are used interchangeably.

```
1 Module Type ConsensusProtocol (T : Types) (P : ConsensusParams T)
2                               (F : Forest T P) (A : NetAddr).
3 Import T P F A.
4 (* Actual definitions and proofs *)
5 End ConsensusProtocol.
```

Listing 4.1: Define a module signature that includes actual definitions.

```
1 Module Protocol (T : Types) (P : ConsensusParams T)
2                 (F : Forest T P) (A : NetAddr)
3                 <: ConsensusProtocol T P F A.
4 Include (ConsensusProtocol T P F A).
5 End Protocol.
```

Listing 4.2: Instantiate a module by including (copy/pasting) the module signature.

Only the type definitions, consensus parameters, block forest library, and the protocol logic actually extract to OCaml code. The network semantics and invariant proofs are purely logical and do not appear in the extracted version of Toychain.

Turning "Modules" into Modules

Luckily, because Toychain was already split into files, most of the work to make the development modular consisted only of transforming Coq poor-man's "modules" into proper modules by defining a module type and an implementation that is the same as the module type. This approach, where we provide a full implementation in the module interface (line 4 in Listing 4.1) and then `Include`, *i.e.* copy/paste, that implementation in the module definition (line 4 in Listing 4.2) is an easy way to modularise the existing Toychain development, but is *bad engineering*. It defeats the entire purpose of modules, which is to hide unnecessary information — we expose everything. Again, we only do this to work around Coq's default definition transparency. We make one exception to this approach, for the block forests module.

As a technical point, in the hierarchy of Toychain components, only the type definitions are a first-order Coq module. The rest of the hierarchy is made out of *Coq functors*, *i.e.* higher-order modules that are defined in terms of other modules. For example, the `Protocol` module in Listing 4.2 is a module parametric in terms of some modules with signatures `Types`, `ConsensusParams T`, `Forest T P`, and `NetAddr` respectively.

Block Forests Module

For the `Forests` "module", the workhorse of our development, we chose to modularise it properly, *i.e.* have the module signature only export the symbols that the rest of the

Toychain development actually uses. This allows us to identify precisely which facts about block forests (which indirectly imply facts about the consensus parameters) are truly needed for protocol correctness and which are extraneous.

This is useful information in anticipation of future work. For example, in the future we might want to create a more efficient block forest implementation. Knowing exactly which facts are used in the proofs allows us to concentrate on proving only these facts.

As of now, the `Forests` module includes proofs of 140 different lemmas. Only 33 of these are necessary for protocol correctness and thus exposed in the module interface. The more interesting ones include:

`btChain_good`	`btChain` only returns chains that start with the genesis block (*i.e.* GB-founded chains)
`btExtend_comm`	Adding blocks to the block forest (`btExtend`) is commutative
`btExtend_preserve`	As long as no hash collisions occur, `btExtend` does not remove items from the block forest it operates on
`btExtend_mint`	Minting a block on top of `btChain` makes your new `btChain` larger than the old one
`btExtend_within`	If you have only a partial view of the global block forest, ($F_l \subseteq F_G$), and mint a block B on top of the local `btChain` in a way that does not give you a chain greater than the global `btChain`, adding B to F_G does not change the global chain

With the infrastructure to preserve definition opacity (*i.e.* modules) in place, our high-level proofs are now parametric over definitions. As such, we can instantiate Toychain with a specific implementation without fear of breaking proofs.

4.3 Instantiating Consensus Parameters

As previously mentioned, Toychain is a *family* of consensus protocols parametric on the choice of security primitives. To obtain an executable instance of Toychain, it suffices to instantiate the type definitions (*e.g.* transactions, hashes) and the consensus parameters (*genProof*, *VAF*, and *FCR*). Everything else necessary for the protocol is already fully specified in Gallina.

In the following, we instantiate Toychain with a SHA256-based proof-of-work scheme, similar to that of Bitcoin. Before we can write the function implementations, however,

```
Parameter Hash : ordType.
```

Listing 4.3: `Hash` can be any type with a strict comparison operation.

```
Parameter Hash : Set.

Axiom Hash_eqMixin : Equality.mixin_of Hash.
Canonical Hash_eqType := Eval hnf in EqType Hash Hash_eqMixin.

Axiom Hash_ordMixin : Ordered.mixin_of Hash_eqType.
Canonical Hash_ordType := Eval hnf in OrdType Hash Hash_ordMixin.
```

Listing 4.4: The `ordType` type class "broken apart". Hash can now be extracted to an OCaml type and the logical properties can be discarded.

we have to first determine how the types we use in Coq should translate to OCaml types.

4.3.1 Type Definitions

As mentioned before, instantiating the types we use in Toychain is *not* trivial. The root of the issue is that Coq has a much stronger type system than OCaml. Many of the types we initially used in the Toychain development to represent hashes, transactions or IP-port pairs simply *do not exist* in OCaml. For example, we made extensive use of the types `eqType` and `ordType`, *i.e.* the type of values with decidable boolean equality and strict comparison, respectively. In the Coq implementation, `eqType` and `ordType` are `Structure`s, similar to Haskell's type classes. Unfortunately, these do not translate directly into OCaml.

To allow extraction, we had to "break apart" the type classes such that they can still be used in Coq as before, but also extract nicely to OCaml. For example, the declaration in Listing 4.3 becomes the one in Listing 4.4, which says that `Hash` is a type in the `Set` kind[†] and *separately* says that `Hash` can be coerced into `eqType` and `ordType`. Later, when we instantiate `Hash` to a particular type (*i.e.* `string`), we will have to separately prove that `string` is `eqType` and `ordType`. This is laborious, but not difficult.

Thankfully, Coq treats the broken-up definition in Listing 4.4 as completely equivalent to the short `ordType` definition. This means that the changes in our development are contained at the type definition points — all proofs and function definitions remain otherwise unchanged.

Next, we describe how we instantiate the type for hashes. Instantiating transactions and proof objects follows in a similar fashion.

[†] `Set` is the type of types without logical content.

```
Inductive ascii : Set := Ascii (_ _ _ _ _ _ _ _ : bool).

Inductive string : Set :=
  | EmptyString : string
  | String : ascii → string → string.
```

Listing 4.5: Definition of ASCII characters and strings in the Coq standard library.

Instantiating Hashes

We instantiate hashes as hexadecimal `string`s. Passing strings from Coq to OCaml and vice-versa is relatively painless, whereas other data types (notably natural numbers) can cause serious headaches. Moreover, this allows us to use existing OCaml hash function implementations, *e.g.* the ones provided by the `cryptokit` library, rather than implement our own.

With this choice, there are two barriers we have to overcome. First, there is a mismatch between Coq's representation of strings (Listing 4.5) and OCaml's. We need to bridge the two representations. Second, as explained in the previous section, we must now prove that `string` is both `eqType` and `ordType`.

Firstly, to see why we need to bridge the Coq and OCaml representations of strings, consider the following. Coq represents strings as lists of ASCII characters. However, for compatibility reasons, the actual type of `string`s in Coq is not `list ascii`, but rather a type that is isomorphic (*i.e.* has the same constructors, but differently named) to `list ascii`. By contrast, OCaml has native strings. As such, in the extracted code, OCaml strings must be converted into lists of characters before passing them to Coq functions, and `string` outputs from Coq must be converted into OCaml native strings.

For the second issue, while it is "obvious" that strings accept decidable boolean equality and comparison operators, proving it takes some effort. Thankfully, the Coq standard library already provides definitions and proofs for boolean equality of ASCII characters and strings, and we can reuse these to prove that `string`s inhabit SSReflect's `eqType`. Sadly, though, we have no such luck when it comes to proving `string`s are `ordType`. Standard Coq has no predefined mechanism to compare ASCII characters or strings — we have to define a comparison function and prove it *irreflexive*, *transitive*, and *total* on our own. Given how ASCII characters are defined (Listing 4.5), this is quite awkward.

To ease some of the pain, we reuse Coq's and SSReflect's existing functions and proofs as much as possible. More concretely, Coq already defines a pair of functions, `N_of_ascii` and `ascii_of_N`, for converting characters to and from natural numbers. Moreover, the standard library does contain a proof that natural numbers are strictly

```
Definition WorkAmnt := N_ordType.

Definition work (b : block) : WorkAmnt :=
  count_binary_zeroes (hashB b).

Fixpoint total_work (bc : Blockchain) : N :=
  match bc with
  | b::bc' ⇒ (work b + total_work bc')%N
  | [::] ⇒ N_of_nat 0
  end.
```

Listing 4.6: Define the amount of "work" underlying blocks and blockchains.

ordered *and* a proof that `list ordType` is also `ordType`. We combine these to prove that `string` is `ordType`, by showing the existence of an embedding from `string` to `list ascii` to `list N`, which we prove is `ordType`.

This proof, that `string` is `ordType`, allows the Coq type system to treat hashes as if they had the original, generic `ordType` definition, but crucially also lets us work with hashes (with standard string operations) from both Coq code and OCaml code, *e.g.* when writing the implementation of *FCR* and, respectively, *genProof*.

We instantiate the types for transactions and proof objects in a similar way.

4.3.2 Proof of Work

With the types instantiated, we can proceed to define the interesting part of the protocol, namely the *consensus parameters*, and prove that our instances of *FCR* and *VAF* satisfy the assumptions that the Toychain model makes about them. We need to define the following components:

- Genesis block
- Hash functions for transaction and blocks
- Functions to validate transactions and manipulate transaction pools
- Fork choice rule
- Validator acceptance function

Not all of the components need to be defined in Coq. For example, we leave the hash functions as `Parameter`s and define them in OCaml as `cryptokit`'s implementation of SHA-256. This is fine since our model does not make any assumptions about the hash functions — as long as they have the right type, any functions suffice. By contrast, the

```
Definition VAF (b : Block) (bc : Blockchain) : bool :=
  (* GenesisBlock doesn't have work requirements *)
  if (b == GenesisBlock) then
    if (bc == [::]) then true else false
  (* All other blocks do *)
  else if (12 <? (work b))%N then true else false.

Lemma VAF_GB_first :
  ∀ bc, VAF GenesisBlock bc → bc = [::].
Proof. by rewrite/VAF eq_refl⇒bc; case: ifP⇒//=; move/andP; case⇒/eqP. Qed.
```

Listing 4.7: In a typical implementation, *VAF* would validate transactions and adjust its difficulty requirements based on the chain it takes as argument.

fork choice rule and validator acceptance functions have to be defined in Coq if we want to *prove* that they are consistent with the assumptions the Toychain model makes, *i.e.* that *VAF* only accepts the genesis block as the first block in a chain, and that *FCR* imposes a strict total order and is additive.

With hashes defined as hexadecimal strings, we can define our notion of "work" for a given block to be the number of leading binary zeroes in that block's hash — see Listing 4.6. We use this in *VAF* to decide whether a given block is valid, and use its cumulative version, `total_work`, in *FCR* to compare between two competing chains.

Using this notion of work, we are able to define the validator acceptance function, shown in Listing 4.7. From the definition, we immediately see that *VAF* will only accept the genesis block as the first block in a chain. We provide a proof of this fact to show that our instantiation of *VAF* satisfies Toychain's assumptions.

The fork choice rule, shown in Listing 4.8, is defined in a peculiar-looking way to make it easier to prove that it indeed imposes a strict total order on all possible blockchains. Nonetheless, its behaviour is entirely straightforward. Given two chains, it compares them based on, in the following order, their total amount of work, their lengths, and the order that we get from the fact that `Blockchain` is `ordType`. In other words, our *FCR* compares chains based on their `total_work`, and uses length and binary comparison as tie breakers. This *FCR* is additive, as well as transitive, total, and irreflexive, satisfying the model's assumptions.

Extracting Natural Numbers

Besides the odd choice to define *FCR* in the form of a `match` statement, rather than a more familiar series of `if`s, note also lines 13 and 14 in Listing 4.8. Two curious things happen there. Most notably, our implementation of *FCR* uses two different definitions of natural numbers!

```
1  Definition FCR bc bc' : bool :=
2    let w := total_work bc in
3    let w' := total_work bc' in
4    let l := (List.length bc) in
5    let l' := (List.length bc') in
6    let eW := w == w' in
7    let eL := l == l' in
8    let eO := bc == bc' in
9
10   match eW, eL, eO with
11   | true, true, true ⇒ false
12   | true, true, false ⇒ ords bc bc'
13   | true, _, _ ⇒ ~~ (Nat.leb l l')
14   | false, _, _ ⇒ ~~ (w <=? w')%N
15   end.
```

Listing 4.8: Real-world fork choices rules are additive, but not strict.

```
Extract Inductive nat ⇒ int [ "0" "succ" ]
  "(fun f0 fS n → if n=0 then f0 () else fS (n-1))".
```

Listing 4.9: While `nat` can be coerced to extract as OCaml's `int`, all operations defined in Coq operate on the Peano inductive structure (shown) and remain very slow.

Coq's usual `nat` type defines Peano natural numbers[‡], which are used throughout the standard library and, in this case, are the return type of `List.length`. Peano naturals have an elegant definition and are well-suited for inductive proofs, but are inefficient to represent and compute on. For example, the Coq process usually crashes when it tries to construct `nat`s larger than 70,000 and typical operations take linear or even quadratic time.

When extracting to OCaml, it is possible to override the definition such that `nat` extracts to OCaml's `int` — this is what we do for Toychain, as seen in Listing 4.9. However, this does not magically make operations on `nat`s fast. The boolean comparison we use in line 13 still runs in $O(n)$ time, which is infeasible even for relatively small numbers. Because of this performance limitation of `nat`, Coq also includes a binary definition of natural numbers, shown in Listing 4.10.

We use this latter, "binary", definition to represent the amount of work for blocks and blockchains, respectively. These can also be extracted to OCaml `int` and have a more reasonable performance profile: operations (*e.g.* the boolean comparison on line 14) take logarithmic time. This is still worse than the constant time we expect for operations on native `int`s, but is sufficient for most applications.

In the future, as we move towards making Toychain truly practical, all instances of

[‡]A natural number is either zero or the successor of a natural number.

```
Inductive positive : Set :=
  | xI : positive → positive (* 2*P + 1 is positive *)
  | xO : positive → positive (* 2*P is positive *)
  | xH : positive. (* 1 is a positive number *)

Inductive N : Set := (* A natural number is either 0 or positive *)
  | N0 : N
  | Npos : positive → N.
```

Listing 4.10: The Coq standard library includes a binary definition of natural numbers. Operations on them take logarithmic, rather than linear time.

```
13    | true, _, _ ⇒ ~~ (Nat.leb l l')
14    | false, _, _ ⇒ ~~ (w <=? w')%N
```

Listing 4.11: Coq naturals have no boolean "greater" operator defined in the standard library.

Coq's `nat` should be replaced with the binary version, `N`.

Designing Proof Libraries

The second issue with our definition of *FCR* is rather more trivial. Coq's standard library does not define a boolean "greater than" operator. As such, we are forced to use the convoluted, but equivalent "not less than or equal to", as seen in Listing 4.11.

The lack of a "greater than" operator might seem unusual, but is actually a reasonable design choice in a language designed for theorem proving, rather than general-purpose programming. Generally speaking, it is undesirable to have multiple theorems that are actually equivalent, *e.g.* transitivity of "less than or equal to" and transitivity of "greater than". Similarly, in a general-purpose programming language, it is considered a code smell to have multiple functions that do the same thing. Avoiding multiple equivalent definitions keeps the library easy to maintain.

4.4 Extracting OCaml Code

Coq comes with an *extraction mechanism* that takes Gallina terms and translates them to equivalent OCaml code. Having instantiated *VAF* and *FCR* and having proven that they satisfy the model's assumptions, we are now ready to extract OCaml code and supply the remaining protocol parameters, namely *genProof* and the hash functions, in order to get a full, executable implementation of Toychain instantiated with proof-of-work.

Note, however, that the extraction mechanism is not formally verified — we have to *trust* it to be correct. This trust might be misplaced. Indeed, to extract Toychain, we

```
(** val coq_SimplPred : 'a1 pred → 'a1 simpl_pred **)
let coq_SimplPred p = p
(** val predT : 'a1 simpl_pred **)
let predT = coq_SimplPred (fun _ → true)
```

Listing 4.12: Before the work-around is applied, OCaml infers `predT`'s type as `'_weak1 -> bool`, which is not included in the expected `'a1 simpl_pred` declared in the interface.

```
(** val predT : 'a1 simpl_pred **)
let predT _ = true
```

Listing 4.13: After in-lining, OCaml correctly infers `predT`'s type as `bool`, which is a subtype of `'a1 simpl_pred`. The extracted code now compiles.

had to work around two separate bugs in Coq's extraction mechanism. As far as we can tell, these work-arounds do not impact correctness, but this is a value judgement rather than a proof. Furthermore, the extraction produces OCaml code, which we compile to x86-64 assembly. As such, we also have to trust the OCaml compiler.

Listing 4.14 shows the entirety of the code that drives the extraction. Instructions are given to the extraction mechanism in a declarative style, and we only extract 3 high-level objects and their recursive dependencies (lines 44–47):

`procInt`	Handler for protocol internal transitions, *i.e.* issuing of transactions and mining attempts
`procMsg`	Handler for protocol messages, *i.e.* `TxMsg`, `BlockMsg`, `ConnectMsg`, `AddrMsg`, `InvMsg`, and `GetDataMsg`
`State`	Record data type describing a node's full local state

The instructions themselves are straightforward, but we run into two separate issues when extracting parts of Coq's SSReflect library, used in Toychain.

Firstly, the `ssrbool.v` file has a type mismatch issue when extracted, *i.e.* the type in the OCaml implementation of the `predT` function in `ssrbool.ml` does not match the type declared in the module interface `ssrbool.mli`. Essentially, the mismatch appears because the OCaml type inference algorithm infers the wrong type. We can see this in Listing 4.12, which shows the portion of code that fails to type-check. The expected type (generated by Coq's extraction mechanism) is provided as a comment above each definition. We work around this type inference error in line 15 of our extraction "recipe" (Listing 4.14) by in-lining the definition of `coq_simplPred` into the definition of `predT`, as seen in Listing 4.13 — this type-checks and the OCaml code extracted from Coq now compiles. Disel, which is also based on SSReflect, uses the same work-around [WP18].

```
Require Extraction.
From Toychain
Require Import Address Protocol Forests Parameters TypesImpl Impl.
Require Import ExtrOcamlBasic ExtrOcamlString ExtrOcamlZInt.

(* Instantiate modules *)
Module ForestImpl := Forests TypesImpl ProofOfWork.
Module ProtocolImpl := Protocol TypesImpl ProofOfWork ForestImpl Addr.

(* Avoid colliding with OCaml standard library names *)
Extraction Blacklist String List.

(* This solves an error where the implementation of ssrbool.ml
   doesn't match the interface *)
Extraction Inline ssrbool.SimplPred.

(* This works around what seems to be a bug in Coq's extraction
   mechanism. The normal extraction gives this code, but with "assert
   false" instead of "assert true". See Coq issue #7348. *)
Extract Constant fintype.Finite.base2 ⇒
"
  fun c →
    { Choice.Countable.base = c.base; Choice.Countable.mixin =
      (Obj.magic mixin_base (assert true (* Proj Args *)) c.mixin) }
".

(* ordinals are nat, and we want to extract nat to int *)
Extract Inductive nat ⇒ int [ "0" "succ" ]
    "(fun f0 fS n → if n=0 then f0 () else fS (n-1))".

Extract Constant ProofOfWork.hashT ⇒ "Core.hash_of_tx".
Extract Constant ProofOfWork.hashB ⇒ "Core.hash_of_block".
Extract Constant ProofOfWork.genProof ⇒
"
  fun bc tp ts →
  if List.length bc == 0 then None else
  let template = Core.get_block_template bc in
  let acc_txs = Core.get_acceptable_txs bc tp in
  let block = {template with txs = acc_txs} in
  if coq_VAF block bc then Some (acc_txs, (block.proof)) else None
".

Cd "Extraction/src/toychain".
Separate Extraction
  ProtocolImpl.procMsg
  ProtocolImpl.procInt
  ProtocolImpl.State.
Cd "../../..".
```

Listing 4.14: Full listing of Toychain's `Recipe.v` file, which contains the instructions for the extraction mechanism.

```
let get_block_template (bc : coq_Blockchain) =
  let prev = List.nth bc (List.length bc - 1) in
  let new_block = {
    prevBlockHash = (hash_of_block prev);
    txs = [];
    proof = Random.int 1073741823 (* 2^30 - 1 *)
  } in new_block
```

Listing 4.15: The OCaml definition of `get_block_template` for proof-of-work. It gives a new random nonce each time it is called.

A second, much more serious issue, has to do with how Coq extracts dependent records into OCaml. In some situations, projections (*i.e.* field accesses) into dependent records are extracted as `assert false` [Amo18]. Interestingly, this issue does not exist when performing extraction to Haskell, as the two extraction mechanisms (into OCaml and respectively, Haskell) have separate implementations [Gru18].

Luckily, we rely on a single such dependent-record access in our code, in the SSReflect `fintype` library's `base2` function. We work around this extraction fault in lines 20–25 of our extraction "recipe" (Listing 4.14) by replacing the `assert false` in the generated OCaml code with `assert true`. We claim this is legitimate, since a comment in the relevant portion of the extraction mechanism source code suggests the `assert false` is a "fake arg" that is supposed to be removed before the extracted code is output [Amo18]. Nonetheless, having to do this manual rewriting is worrying and makes it more difficult to trust that the resulting extracted OCaml code is faithful to the Coq original.

Finally, our extraction "recipe" (Listing 4.14) provides OCaml instantiations for the $hash_t$, $hash_b$, and *genProof* functions that we have left undefined in Coq. For the hash functions, these are wrappers around `cryptokit`'s implementation of SHA-256. The *genProof* function is implemented with the use of two helper functions, `get_block_template` (reproduced in Listing 4.15) and `get_acceptable_txs`.

With this done, we have a full instantiation of proof-of-work Nakamoto consensus. However, we still have *some* work to do before we can actually use this consensus code. We need to write OCaml code to handle TCP/IP connections, serialise/deserialise network packets, and pass them on to the appropriate protocol message handlers. We discuss this in the following section.

4.5 Formally-Verified Nakamoto Consensus

In Section 4.3, we instantiated the Toychain protocol family with consensus parameters for a Nakamoto-style proof-of-work protocol. Then, in Section 4.4, we extracted this Coq code into OCaml and provided definitions for the functions that were left unspecified.

```
let procMsg_wrapper () =
  (* Listen for incoming TCP/IP packets. *)
  let () = check_for_new_connections () in
  let fds = get_all_read_fds () in
  let (ready_fds, _, _) = retry_until_no_eintr
                          (fun () → Unix.select fds [] [] 0.0) in
  begin
    match get_pkt ready_fds with
    | None → (* nothing available *) None
    | Some pkt →
      begin
        (* For ConnectMsg and AddrMsg, update peer table in Net.the_cfg
           before processing the message. This ensures the appropriate sockets
           can be created when send_all is called later, in line 23. *)
        ( match pkt.msg with
          | ConnectMsg → add_peer_if_new pkt.src;
          | AddrMsg peers → List.map (fun pr → add_peer_if_new pr) peers; ();
          | _ → ();
        );
        (* Pass all messages to the message handler. *)
        let (st', pkts) = Pr.procMsg !st pkt.src pkt.msg 0 in
        st := st'; (* Update the local node state. *)
        send_all pkts; (* Send responses. *)
        Some (st, pkts)
      end
  end
```

Listing 4.16: The "wrapper" around procMsg has to dynamically (and transparently to the protocol logic) update the node's peer table and create the appropriate TCP/IP connections when it hears gossip — the protocol assumes it can send messages directly, with no notion of connections.

Now, we showcase how to take this consensus protocol code and transform it into a practical, executable implementation of Nakamoto consensus. Conceptually, all we need to do is define "wrappers" around the existing consensus code. These wrappers handle all interactions with the "real world" (*e.g.* TCP/IP connections, writing log files, *etc*), but rely on the message handlers extracted from Coq to carry out the protocol logic. As such, morally speaking, all our proofs also apply to the "wrapped" code *as long as* the "real world" does not contradict our model.

Other formally-verified implementations of distributed protocols, *e.g.* the Coq implementations of the protocols verified with Disel, rely on a similar "wrapper" mechanism [SWT17]. Indeed, we adapt and reuse a large part of the Disel networking code in our implementation of Toychain.

The core of the wrapper infrastructure is a table of peers that the node maintains TCP/IP connections with. As far as the protocol logic (*i.e.* message handling code) is concerned, these connections might as well not exist. Indeed, the protocol logic has no notion of a "connection" — it works by passing messages. It is the responsibility of the

```
let rec retry_until_no_eintr f =
  try f ()
  with Unix.Unix_error (EINTR, _, _) → retry_until_no_eintr f
```

Listing 4.17: We transform all Linux system calls into interrupt-resistant versions.

wrapper to make sure the messages are sent correctly.

Because of the gossip protocol, the peer table needs to be updated dynamically. There are several possible ways to implement this. For example, the `procMsg_wrapper` could inspect all outbound messages to see whether any of them need to be sent to a node who is *not* already in the peer table. If this is the case, the peer table needs to be updated. This is a valid way to keep the peer table up-to-date, but is not faithful to the underlying protocol. Instead, we choose to intercept `ConnectMsg` and `AddrMsg` protocol messages — emitted by the gossip mechanism — and dynamically add unknown peers to the peer table, as needed. We reproduce the full `procMsg_wrapper` in Listing 4.16.

Compared to the message-handling wrapper, handling internal transitions is much simpler. There is no need to listen for incoming network connections, since transitions are produced locally, *i.e.* when a transaction is issued or when the node attempts to mine a block. After every transition, the node's state is updated and the outbound messages, if any exist, are sent over the network.

Having defined the peer table infrastructure and the wrappers around `procMsg` and `procInt`, we have a complete, executable implementation of proof-of-work Nakamoto consensus.

4.5.1 Caution: TCP/IP

It is important to note that only the core, unwrapped protocol logic is formally verified. In a certain sense, there are no formal guarantees about the "wrapped" protocol. Bugs might exist in the overall system, and, in fact, they probably *do exist*. In particular, our networking code might fail to correctly process some packets.

While developing the wrapper infrastructure for Toychain, our process was to compile OCaml to bytecode and run it through the interpreter. This worked correctly: we had a cluster of nodes talking to each other and coming into agreement. However, as soon as we switched to compiling to machine code and executing it natively, our node processes would sometimes, seemingly randomly, crash.

After some investigation, we discovered that the crashes were caused by system interrupts — if a system call (*e.g.* `read` or `send` over a network socket) is interrupted in Linux, the call is not automatically restarted. As such, the call would fail, and an a `EINTR` exception would be thrown. The exception was never caught, and thus the

entire node process crashed. This did not happen when we ran the bytecode — somehow the runtime caught and handled the `EINTR` exception. We overcome this issue by automatically retrying system calls when they are interrupted, as seen in Listing 4.17.

It is entirely possible that similar problems might arise in a different environment, *e.g.* when running Toychain under Windows. As such, the "proven-correct" label must be taken with a grain of salt. Our formal proof does not eliminate the possibility of bugs. Nonetheless, it is very useful, because it restricts where in the code bugs can originate, *i.e.* faults can exist only in the unverified networking code, not in the verified consensus part. This claim is supported by a recent empirical study on the correctness of formally verified distributed systems. Looking at systems similar to ours, Fonseca *et al.* found bugs in the unverified portions of code, *e.g.* networking shims, but no faults in the actual protocol logic, which was formally verified [Fon+17].

4.5.2 Running Toychain

Toychain is open-source and is publically available online, on GitHub:

`https://github.com/certichain/toychain`

Building Toychain is known to work with Coq 8.9.0 and OCaml 4.06.1. We have tested Toychain on Ubuntu 16.04 and the Windows Subsystem for Linux (WSL) equivalent. All other dependencies can be installed using `opam`, as follows:

```
opam repo add coq-released https://coq.inria.fr/opam/released
opam install coq-mathcomp-ssreflect coq-fcsl-pcm
opam install cryptokit ipaddr
```

Then, to build Toychain, run `make` followed by `make node`. The first command compiles the Coq source files and the second extracts the OCaml code and builds the `node.native` executable file. Toychain nodes can be started on the command-line:

```
./node.native -me IP_ADDR PORT -cluster <CLUSTER>
```

where `<CLUSTER>` is a space-separated list of `IP_ADDR` `PORT` pairs.

The repository also contains a directory with shell scripts. The most important script, `run.sh`, starts a cluster of 3 Toychain nodes, running at `127.0.0.1`, ports 9000, 9001, and 9002 respectively. The nodes all know about each other and save their output in the respective `node-0{0,1,2}.log` file.

Additional nodes can be spawned "manually", *e.g.* :

```
./node.native -me 127.0.0.1 9003 -cluster 127.0.0.1 9001
```

The command above also shows that you can spawn a node with only partial knowledge of the cluster. As long as the node can connect to one entry-point into the network, the gossip mechanism will work and the node will eventually learn the full topology of the cluster.

Seeing the gossip mechanism in action has been one of the highlights of our work. The gossip mechanism is defined in Coq, in the `procMsg` message handler, but has no associated formal proof — it is entirely unverified. Nonetheless, it works in practice, as you can see in Figure 4.1. Nodes can join the network "late" and correctly come into agreement with their peers.

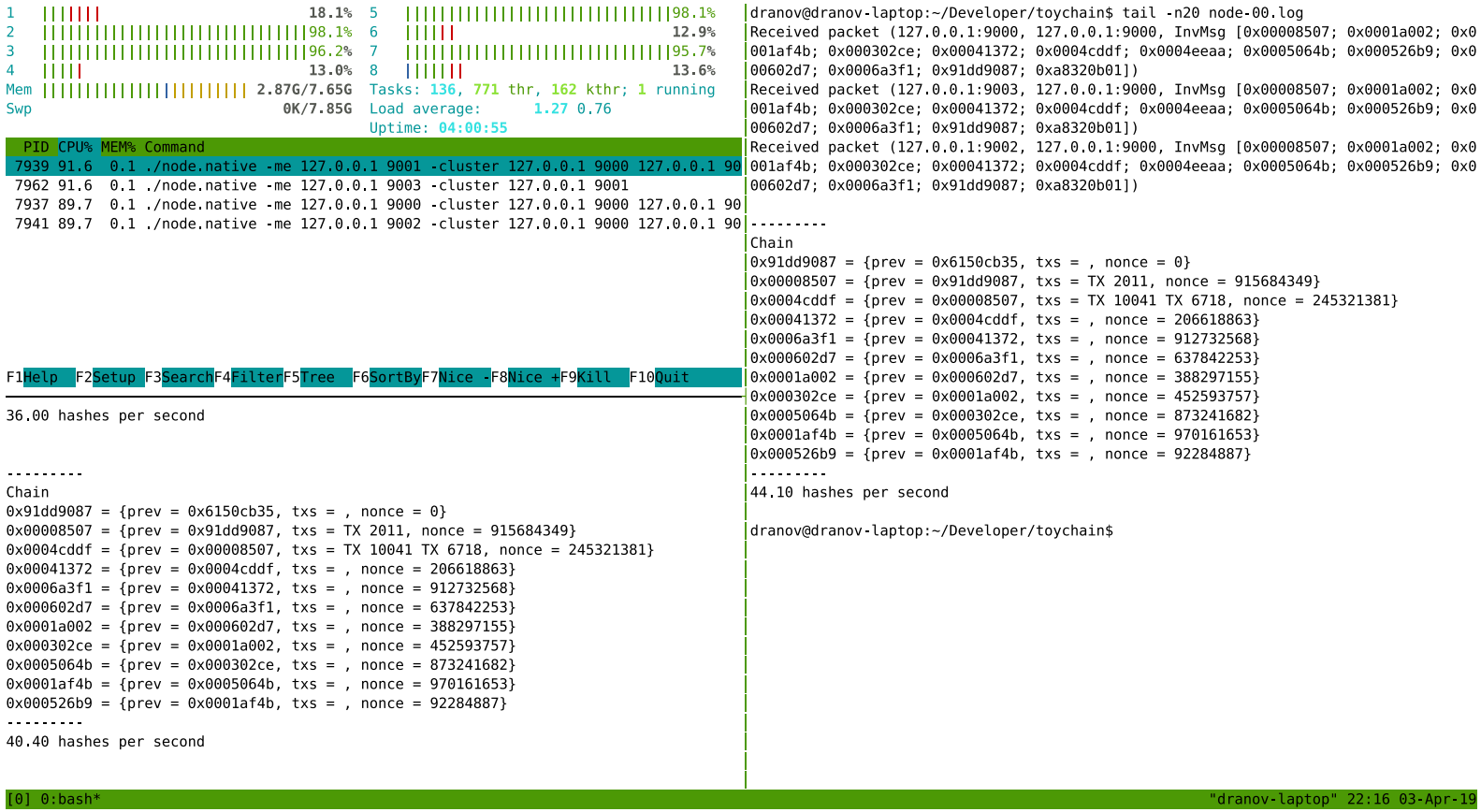

Figure 4.1: A Toychain cluster with 4 nodes in agreement. Three nodes were started using `run.sh` and the fourth (PID 7962), whose output is shown in the lower-left corner, joined "late", knowing only of node `127.0.0.1:9001`.

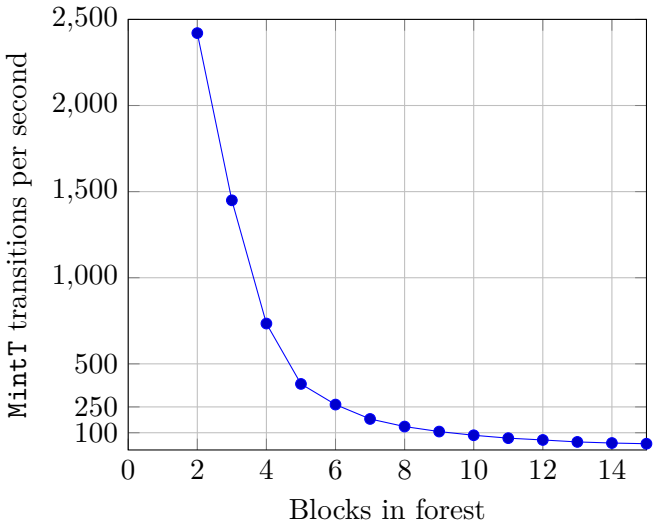

Figure 4.2: Executing a `MintT` transition, which should take $O(1)$ time, takes $O(n^4)$ time.

4.6 Limitations

This section describes the limitations of the Toychain implementation. We discuss the overall Toychain project in Chapter 5.

4.6.1 Performance

Toychain works. We have formally verified code that you can build and run on your PC. However, you would not be able to run Toychain as it exists today in a production environment. Currently, performance is *abysmal*. This is not entirely unexpected. Remember, the Toychain implementation evolved out of what was originally a *model*, albeit an executable one. Whenever we had a choice in how to implement a function, we chose the implementation that would be best suited for verification. Almost always, this is the slowest possible implementation.

Right now, the main bottleneck is the performance of the block forest library. In a typical implementation, you would expect the main `btChain` operation to take amortised $O(1)$ time. In other words, determining what the current chain is should be very fast, since you need this information for most operations in the protocol. Our implementation of `btChain` runs in $O(n^4)$ time! This makes it straightforward to prove that `btChain` does indeed return the heaviest chain — it compares all possible chains — but is obviously impractical.

Sadly, since `btChain` is such a core operation in the protocol, a slow implementation makes the entire protocol slow to the point of being unusable. Figure 4.2 shows that

attempting to mine a block, which should be a constant time operation (*i.e.* the graph should be a straight horizontal line), becomes unfeasible with as few as 10 blocks in the local state. Therefore, before Toychain can become truly practical, rather than simply executable, it is necessary to replace the current block forest implementation with an efficient one.

When executing a `MintT` transition, most of the time is spent executing `btChain`, but *some* of the slowness is due to having to hash blocks. Since in our definition, blocks include a list of transactions, longer blocks take a longer time to hash. This is not too much of an issue for Toychain, but would be unacceptable in a real protocol. Normally, hashes would only be computed over a fixed-size block header which contains the Merkle root of the block's transactions.

More performance issues are likely to start to appear in forests with around 10,000 blocks, because of our reliance on Coq Peano natural numbers. We discussed this in Section 4.3.2.

We leave improving Toychain's performance for future work.

4.6.2 Storage and Networking

Apart from bad performance, the current Toychain implementation is impractical in two other ways.

Lack of persistent storage. Toychain currently operates entirely in-memory, *i.e.* it does not save its local state to disk. This means that restarting the Toychain executable requires a full resynchronisation with the network. A practical implementation of Nakamoto consensus would save its block forest to disk after each newly-added block and resume execution from that state when restarted. This is conceptually straightforward, but has been a major pain point in real implementations, as the local chain state database would often become corrupted when the node process crashed. In the future, having a verified implementation that guarantees no corruption would be a significant win.

No UPNP support. As of now, the unverified networking layer in Toychain is not well-suited for real-world use on the Internet. The networking code is written with the assumption that connection establishment is bi-directional, *i.e.* if party A can establish a connection to party B, then B can establish a connection to A. Generally speaking, this is not true on the Internet, since many users reside behind a router that performs network address translation (NAT). Such users are not directly reachable from the open Internet — they need to set up port-forwarding in their router interface before they

can be reached. Real-world implementations of Nakamoto consensus function without this bi-directionality assumption *and* support the Universal Plug and Play (UPNP) standard to automatically set-up port forwarding when needed. Implementing at least one of these improvements is necessary before Toychain can be realistically used on the open Internet.

4.6.3 Trusted Computing Base

As mentioned in Section 4.5.1, only the core protocol logic is formally verified. The "wrapped" protocol, *i.e.* the actual implementation, is not. This means that the non-verified networking part of Toychain is *trusted*, *i.e.* we have to assume it is correct. If that assumption turns out to be false, we have no guarantees about the overall system.

Several other components in our system have to be *trusted* as well. For instance, we assume that Coq is sound and the Coq type-checker is correct. Arguably, this is a safe assumption, but our development relies on some of the less well-tested parts of Coq, *e.g.* functors, which might be risky. Furthermore, we assume that the non-trivial extraction from Coq to OCaml is correct — despite running into bugs in the extraction mechanism! We also have to trust the OCaml compiler and associated build tools, the `cryptokit` library, the operating system kernel, and the CPU we are executing the code on. All of these could have errors.

Notwithstanding the long list of trusted components, formal verification is still useful, because it restricts where in the system faults can originate from. A verified implementation *is* more trustworthy than a non-verified one [Fon+17]. Moreover, there is ongoing research, both in industry and academia, into how to reduce the TCB for real systems even further.

5
Conclusion

This is the best thing since sliced bread.
Wonder Bread*

This chapter offers some concluding remarks, discusses the major limitations of the current work, and proposes avenues for future work.

5.1 Toychain

In this work, we took the formal model of blockchain-based consensus we defined in [PS18], simplified it by removing onerous assumptions, and extracted the first proven-correct implementation of Nakamoto consensus. This is a major milestone towards a future where real-world implementations of blockchain consensus protocols are formally verified. We are not there yet, but we have made significant progress.

We hope this work will motivate the developers of existing permissionless systems to consider formal verification as a way to solve some of their current problems. Firstly, of course, verification can provide tight correctness guarantees. As the Bitcoin inflation bug discovered in September 2018 shows, serious flaws exist even in mature implementations and can remain undetected for years. Formal verification gives us a way to stamp them out. Moreover, verification provides a second, equally important benefit that is often forgotten. Making changes in permissionless protocols is *risky*. Bitcoin's history shows this very clearly. Verification gives us a way to eliminate the risk. Changes can be formally verified to be functionally correct and soft-forks before they are accepted. This would not remove the politics in the change-approval process, but might make fundamental protocol changes, *e.g.* to improve fungibility, easier to accept.

5.2 Limitations

While the future of formally verified implementations of blockchain consensus protocols is promising, Toychain as it currently stands is limited in several significant ways.

*Wonder Bread was the first mass-produced brand of pre-wrapped, pre-sliced bread.

Unverified Gossip Protocol. Our quiescent consistency proof is written in a completely unrealistic network model, where packets are never dropped or corrupted. We argue that this is legitimate, since the gossip mechanism is supposed to provide the guarantee that all protocol-relevant messages are delivered to all nodes, regardless of network conditions. However, we have not actually proven that the gossip mechanism in Toychain is correct. For this reason, our proofs are not fully convincing.

Fork Choice Rule Assumptions. We make the overly-strong assumption that the fork choice rule imposes a strict total order on all possible blockchains. Real-world *FCR*s do not satisfy this assumption. Moreover, we assume that the *FCR* is additive. This seems reasonable, and all protocols we know of have additive *FCR*s, but this might not be needed for correctness.

Abysmal Performance. Our extracted implementation of Nakamoto consensus works, but is practically unusable because of bad performance. This is not an issue with the verification tools. Rather, our formally verified implementation of block forests is simply inefficient. Replacing the current block forest library with a reasonable implementation should give good performance, at least with forests up to a few thousand blocks. With larger forests, the representation of Peano natural numbers will become the main performance bottleneck.

Security Against Byzantine Adversaries. As of now, we have not proven any security properties of Toychain. In particular, we have not proven that the protocol is resilient against Byzantine participants. Indeed, proving such properties requires introducing probabilistic reasoning, which severely complicates things [GS19].

5.3 Future work

In the long term, we intend to develop Toychain into a principled framework for reasoning about the correctness and security properties of blockchain consensus protocols and their implementations.

We plan to enhance our formal model to support reasoning about probabilistic security properties, find an appropriate abstraction for the gossip protocol and prove the protocol's correctness *wrt.* the abstraction, and remove the remaining overly-strong assumptions. On the implementation side, we will have to replace the block forest library with a verified, efficient implementation and make the networking code better-suited to operating on the open Internet.

More practically, we foresee that in the next few years it will become feasible to develop an inter-operable, formally verified implementation of an existing permissionless Nakamoto consensus system, *e.g.* Bitcoin.

Bibliography

[Amo18] Arthur Azevedo de Amorim. *Extraction with dependent records produces 'assert false'*. Apr. 2018. URL: https://github.com/coq/coq/issues/7348 (visited on 03/18/2019).

[And13] Gavin Andresen. *Bitcoin Improvement Proposal 50*. Mar. 2013. URL: https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki (visited on 02/04/2019).

[App18] Andrew Appel. *Unification blows up, even with "simple apply" and "Opaque"*. Mar. 2018. URL: https://github.com/coq/coq/issues/6998 (visited on 03/02/2019).

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Ed. by Wilfried Brauer, Grzegorz Rozenberg, and Arto Salomaa. Texts in Theoretical Computer Science An EATCS Series. Springer, 2004. URL: http://link.springer.com/10.1007/978-3-662-07964-5 (visited on 12/20/2017).

[Bee17] Marcel Beemster. *C99 for-loop defined variable gets incorrect scope*. Dec. 2017. URL: https://github.com/AbsInt/CompCert/issues/211 (visited on 02/28/2019).

[Bit18] Bitcoin. *CVE-2018-17144 Full Disclosure*. Sept. 2018. URL: https://bitcoincore.org/en/2018/09/20/notice/ (visited on 02/23/2019).

[BSd82] Roy J. Byrd, Stephen E. Smith, and S. Peter deJong. "An Actor-based Programming System". In: *Proceedings of the SIGOA Conference on Office Information Systems*. ACM, 1982, pp. 67–78. URL: http://doi.acm.org/10.1145/800210.806479.

[Cor18] Matt Corallo. *Fix crash bug with duplicate inputs within a transaction*. Sept. 2018. URL: https://github.com/bitcoin/bitcoin/pull/14247 (visited on 02/23/2019).

[DHZ16] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. "PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. ACM, 2016. URL: http://doi.acm.org/10.1145/2837614.2837650 (visited on 04/16/2019).

[DJ94] Scott Dawson and Farnam Jahanian. “Deterministic Fault Injection of Distributed Systems”. In: *in Lecture*. Springer-Verlag, 1994, pp. 178–196. DOI: 10.1.1.53.4114.

[DJa96] S. Dawson, F. Jahanian, and T. Mitton and. “Testing of fault-tolerant and real-time distributed systems via protocol fault injection”. In: *Proceedings of Annual Symposium on Fault Tolerant Computing*. June 1996, pp. 404–414.

[Fon+17] Pedro Fonseca et al. “An Empirical Study on the Correctness of Formally Verified Distributed Systems”. In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM Press, 2017, pp. 328–343. URL: http://dl.acm.org/citation.cfm?doid=3064176.3064183 (visited on 03/01/2019).

[Gar10] Jeff Garzik. *Strange block 74638*. Aug. 2010. URL: https://bitcointalk.org/index.php?topic=822.0 (visited on 02/04/2019).

[Gar13] Jeff Garzik. *Block #225430 chain fork dataset available*. Mar. 2013. URL: https://bitcointalk.org/index.php?topic=153170.0 (visited on 02/06/2019).

[Gle+19] Klaus V. Gleissenthall et al. “Pretend synchrony: synchronous verification of asynchronous distributed programs”. In: *Proceedings of the ACM on Programming Languages* POPL (Jan. 2019). URL: http://dl.acm.org/citation.cfm?doid=3302515.3290372 (visited on 04/16/2019).

[Gru18] Samuel Gruetter. *Running compiled fib example with haskell extraction works*. May 2018. URL: https://github.com/mit-plv/bedrock2/commit/1ae95056f91a562228b5825ce798b3d4aa1e433e (visited on 04/03/2019).

[GS19] Kiran Gopinathan and Ilya Sergey. “Towards Mechanising Probabilistic Properties of a Blockchain”. In: *CoqPL* (2019). URL: https://ilyasergey.net/papers/probchain-coqpl19.pdf.

[Haw+15] Chris Hawblitzel et al. “IronFleet: proving practical distributed systems correct”. In: *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP ’15*. ACM Press, 2015. URL: http://dl.acm.org/citation.cfm?doid=2815400.2815428 (visited on 04/08/2019).

[Kin13] Kyle Kingsbury. *Jepsen: A framework for distributed systems verification, with fault injection*. 2013. URL: https://github.com/jepsen-io/jepsen (visited on 04/10/2019).

[Laa18] Wladimir van der Laan. *It was wrong that the buggy code was merged. Yes, we screwed up but the "we" that screwed up is very wide.* Tweet. Sept. 2018. URL: https://twitter.com/orionwl/status/1043789573984333825 (visited on 02/23/2019).

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 978-0-321-14306-8.

[LBC16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. "Chapar: Certified Causally Consistent Distributed Key-value Stores". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. ACM, 2016, pp. 357–370. URL: http://doi.acm.org/10.1145/2837614.2837622 (visited on 04/10/2019).

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". en. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Springer, 2010, pp. 348–370. URL: http://link.springer.com/10.1007/978-3-642-17511-4_20 (visited on 04/16/2019).

[Let17] Pierre Letouzey. *A Tutorial on Using Modules*. Oct. 2017. URL: https://github.com/coq/coq/wiki/ModuleSystemTutorial (visited on 03/02/2019).

[Nak10] Satoshi Nakamoto. *Fix for block 74638 overflow output transaction*. Aug. 2010. URL: https://github.com/bitcoin/bitcoin/commit/d4c6b90ca3f9b47adb1b2724a0c3514f80635c84 (visited on 02/04/2019).

[Pad+16] Oded Padon et al. "Ivy: safety verification by interactive generalization". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*. ACM Press, 2016, pp. 614–630. URL: http://dl.acm.org/citation.cfm?doid=2908080.2908118 (visited on 04/16/2019).

[PS18] George Pîrlea and Ilya Sergey. "Mechanising Blockchain Consensus". In: *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2018. DOI: 10.1145/3167086. URL: http://pirlea.net/papers/toychain-cpp18.pdf.

[Rah+18] Vincent Rahli et al. "Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq". In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Springer International Publishing, 2018, pp. 619–650. URL: http://link.springer.com/10.1007/978-3-319-89884-1_22 (visited on 04/16/2019).

[Ser18] Ilya Sergey. *What We Talk about When We Talk about Formally Verified Systems*. National University of Singapore, Nov. 2018. URL: https://ilyasergey.net/slides/Sergey-BCSW18.pdf (visited on 03/01/2019).

[Son18] Jimmy Song. *Bitcoin Core Bug CVE-2018–17144: An Analysis*. Sept. 2018. URL: https://hackernoon.com/bitcoin-core-bug-cve-2018-17144-an-analysis-f80d9d373362 (visited on 02/23/2019).

[SWT17] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and Proving with Distributed Protocols". In: *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Dec. 2017). URL: http://doi.acm.org/10.1145/3158116 (visited on 02/03/2018).

[VIS18] VISA Europe. *Visa service disruption*. 2018. URL: https://www.visaeurope.com/newsroom/news/visa-service-disruption?linkId=52484014 (visited on 02/02/2019).

[Wil+15] James R. Wilcox et al. "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. ACM, 2015, pp. 357–368. URL: http://doi.acm.org/10.1145/2737924.2737958 (visited on 04/09/2019).

[Woo+16] Doug Woos et al. "Planning for change in a formal verification of the raft consensus protocol". In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs - CPP 2016*. ACM Press, 2016, pp. 154–165. URL: http://dl.acm.org/citation.cfm?doid=2854065.2854081 (visited on 04/03/2019).

[WP18] James R. Wilcox and Karl Palmskog. *DiSeLExtraction.v*. Sept. 2018. URL: https://github.com/DistributedComponents/disel/blob/master/Core/DiSeLExtraction.v (visited on 04/03/2019).

[Yan+09] Junfeng Yang et al. "MODIST: Transparent Model Checking of Unmodified Distributed Systems". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'09. USENIX Association, 2009, pp. 213–228. URL: http://dl.acm.org/citation.cfm?id=1558977.1558992 (visited on 04/10/2019).