Towards Mechanised Consensus in Isabelle

Elliot Jones
Department of Computer Science, University of Exeter, UK

Diego Marmsoler
Department of Computer Science, University of Exeter, UK

Abstract

A blockchain acts as a universal ledger for digital transactions between two parties that require no moderation from a third party. Such transactions are cheaper, quicker, and more secure with high traceability and transparency, with the decentralised structure of a blockchain network allowing for greater scalability and availability. For these reasons, blockchain is at the forefront of emerging technologies, with a wide variety of industries investing billions into the technology. A blockchains consensus protocol is what allows a blockchain network to be decentralised but can be subject to malicious behaviour and faults in its design and implementation that can lead to catastrophic effects like the DAO hack that resulted in a loss of $60 million. From this it is clear to see that the verifications of these protocols are paramount to ensure the safe use of blockchain. In this research, we formally verify the Proof-of-Work consensus protocol, used by Bitcoin, in Isabelle/HOL by modelling the blockchain as the longest branch in a binary tree and proving that the common prefix property holds with the assumption that the network is in majority honest. In this paper, we discuss the validity of our approach, key functions and lemmas we used to complete the proof, advantages and drawbacks of the model, related work and how this research can be taken further.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Formal Methods, Blockchain, Isabelle/HOL, Consensus, Verification, Theorem Provers

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.4

Supplementary Material *Software (Theory Files)*: `https://doi.org/10.5281/zenodo.10479776` [16]

Funding *Diego Marmsoler*: This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/X027619/1].

Acknowledgements We would like to thank the FMBC 2024 reviewers for the careful reading and constructive suggestions on the paper and the formalisation.

1 Introduction

Blockchain was first introduced to the world in 2008, when the illusive Satoshi Nakamoto published his paper on Bitcoin [27]. Using cryptographic hash functions and consensus protocols, blockchain allows parties to carry out digital transactions 'peer-to-peer', meaning no third-party is required to mediate and requires no trust between parties. This allows transactions to be faster, cheaper, and more secure with high traceability and transparency. Furthermore, the decentralised structure of blockchain allows for greater scalability and availability. Whilst its original purpose was for cryptocurrencies, blockchain has since been identified as a means to transform a variety of industries such as finance, healthcare and energy [25]. It is for these reasons that blockchain has become one of the most promising emerging technologies, with worldwide spending expected to grow from $6.6 billion in 2021 to an estimated $19 billion by 2024 [34].

However, blockchain presents its own unique challenges. The consensus protocol of a blockchain is what allows it to be decentralised and Byzantine Fault Tolerant (BFT), but these protocols can act as a vector of attack for malicious users. For example, a 51% attack is

when an attacker controls most of the computing power of a blockchain network, giving the attacker the opportunity to alter transactions on the blockchain and use the same money for multiple transactions in a double spending attack. An example of this was when Ethereum Classic suffered 51% attacks in 2019 and 2020, losing approximately $1.1 million [26] and $5.6 million [11] in double spending.

Moreover, the design and implementation of a blockchain and its consensus protocol led to challenges themselves, with the infamous DAO hack stemming from a smart contract flaw and resulting in a loss of $60 million [8]. Furthermore, A faulty consensus protocol would mean nodes would not be able to agree on the correct chain, allowing malicious actors to potentially change history and double spend without most of the network being malicious. Consequently, the need to verify blockchain consensus protocols is paramount to ensure the security of users. The birth of blockchain has caused a surge of investment into software verification. No organisation showcases this better than CertiK, founded in 2018 and now the world's leading company in Web3 security auditing. With a valuation of $2 billion and annual revenue of $40 million [13], Certik uses cutting-edge formal verification techniques to audit blockchains and smart contracts. The most popular consensus protocol in the world is Proof-of-Work (PoW) due to its usage by Bitcoin, the largest cryptocurrency in the world with a market capitalisation of over $700 billion [9]. PoW assumes that the majority of the network's computing power is acting honestly, making it susceptible to a 51% attack. Despite this, it is still important to verify that the protocol works as expected in the presence of a majority honest network. In doing so, we ensure that consensus will continue to hold as the blockchain network progresses through time and block stability will remain. A verified consensus protocol will promote confidence and encourage participation in the network by minimising faults and malicious behaviour.

In this paper, we formalise a general PoW protocol in Isabelle/HOL [29] and verify consensus by proving the common prefix property [12]. To this end, the paper provides the following main contributions:

1. We describe a general model for blockchain and its formalisation in Isabelle/HOL. A blockchain is modeled as a tree and accompanied by a function to check its validity (Subsubsection 3.2.4). The model provides abstract characterizations of minings (Subsection 3.3) and honest minings (Subsection 3.4) as special types thereof. The protocol is modeled as an inductively defined set of valid event traces, where each event is either a honest or dishonest mining (definition `traces` in Listing 11).
2. We formalise and discuss the common prefix property in Isabelle/HOL (Listing 13). This is an important safety property for consensus protocols which asserts that, once confirmed, blocks cannot be modified anymore.
3. We verify the property from our model. Thereby we discover important assumptions which are required for consensus, such as `b1` and `b2` in Listing 11 or the preconditions for `dishonest_induct` in Listing 11.

In Section 2, we highlight key blockchain properties that are essential to understand our verification. In Section 3, we explain our model, the functions and datatypes we have used, and the mining locales. In Section 4, we describe the blockchain locale and the proof of our consensus property. We conclude the paper with a discussion (Section 5) in which we highlight the simplifications and assumptions we have made for our model.

2 Background

As described in [27], a blockchain is a sequence of blocks, where each block contains a collection of transactions between parties in a network. A blockchain is initialised by a genesis block, a block that contains no transactions but has a unique hash value. To add a new block to the chain, a network participant must first assemble enough transactions into a block, then find the Merkle root hash of these transactions and combine it with the previous block's hash and a nonce to produce its own hash that meet the network's hash requirements. Once the participant finds a nonce that produces a suitable hash, it can link its block to the chain.

However, through a process called forking, a blockchain can split into two or more chains. This happens when two or more blocks are linked to the same block. A blockchain network's consensus protocol allows participants to decide which chain is the 'correct' chain so that they can continue to add blocks to that chain. This mitigates the need for a central authority to decide which chain is the correct chain. A violation of consensus will halt the progress of the blockchain as participants cannot decide where to add a block.

The PoW consensus protocol works on the rule that the longest chain is the correct chain. It does this by assuming that the majority of the network's computing power is honest, meaning that it should be able to solve block hashes quicker than dishonest computing power and keep the correct chain the longest chain. If there are two or more longest chains of the same length, participants will randomly select a chain to work on until one of the chains has a block added and prevails as the longest chain.

Garay et al. identifies the two fundamental properties of PoW consensus as the *common prefix* and *chain quality* properties that ensure consensus if they hold [12]. The common prefix property states that honest network participants agree on the longest chain up to k blocks where k is a variable set by the network. The chain quality property states that the number of blocks produced from dishonest participants will not be very large. As part of our model, we make the assumption of majority honesty, which mitigates the need for the chain quality property. Because of this, the objective of our verification is to prove the common prefix property which subsequently proves consensus.

3 Model

3.1 Binary Tree

As mentioned previously, we will model the blockchain as the longest branch in a binary tree. The rationale behind this is that the tree structure offers an effective way to model the stages a blockchain has as it progresses through time. In a binary tree, each node represents a block of the blockchain, with the root node acting as the genesis block as all branches stem from the root node the same way all blockchains stem from the genesis block. Additionally, the different branches of the binary tree model represent the different competing chains at different stages of the network. It is possible to contain data of an arbitrary type within the tree nodes in Isabelle but it is not necessary for our verification as we are only considering the general construction of a blockchain so there is no need to consider the transactions inside each block.

For each node in a binary tree, there can be zero, one or two child nodes. If there are no child nodes, then we are at the end of a branch and it is either the longest chain or was competing to be the longest chain at an earlier point in time, known as a stale block. If there is a single child node, then this means a block was added to the chain at this point. If

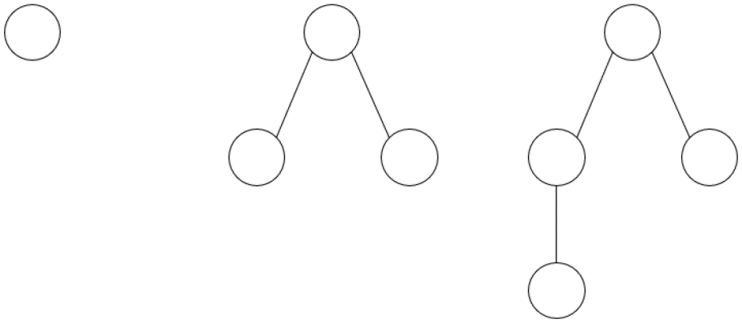

Figure 1 Binary Tree Progression.

```
datatype 'a tree =
    Tip | Node "'a tree" 'a "'a tree"

definition exampleTree :: "nat tree" where
  "exampleTree = Node
                  (Node
                    (Node
                      (Node Tip 5 Tip)
                      2
                      Tip)
                    1
                    (Node Tip 3 Tip))
                  0
                  (Node Tip 4 Tip)"
```

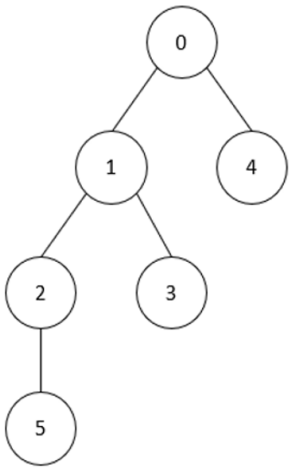

Figure 2 Binary Tree in Isabelle.

there are two child nodes, this is when two blocks are solved simultaneously, so there are now competing chains. Figure 1 showcases each case of binary tree progression. We start with the root node (genesis block), then two nodes (blocks) are added to the tree simultaneously. We now have two valid chains. Eventually, a block is added to the left chain, so we have a single longest chain. Figure 2 shows how we construct a binary tree in Isabelle/HOL with a visual representation of the example tree.

As part of our model, we have made the assumption that all participants in the network are synchronised, meaning they all share the same view of the blockchain. In reality, there is no global view of the blockchain, and each participant has their own copy of the blockchain which may not always be the same for every participant. In our model we only consider a single binary tree for each network so we must assume that participants views are synchronised.

An alternative method for modelling a PoW blockchain network is using a list. Here each entry acts as a block in the longest chain, with the ith entry being the ith block in the chain. However, a list would only keep track of the longest chain and so would not give us a full picture of the network. As a result, we cannot consider a scenario in which a chain that is not the longest chain eventually becoming the longest chain as we would not be keeping a list for it. In reality, this situation would arise from a 51% attack on the network.

3.2 Isabelle Functions

In this section, we will detail the key functions and datatypes that are used to verify the consensus property. We will explain what each function does, how it is encoded into Isabelle, give an example, and showcase any relevant proofs that were later used in the verification. For the functions that require a binary tree as an input, we will use the example binary tree from Figure 2 to show the effect the function has.

3.2.1 Nodes

The *nodes* function counts the number of nodes in the inputted binary tree. In Listing 1 we can see its usage to count six nodes in the example tree. There were no relevant proofs associated with this function.

Listing 1 nodes Function in Isabelle.

```
primrec nodes :: "'a tree ⇒ nat" where
  "nodes Tip = 0"
| "nodes (Node l e r) = Suc (nodes l + nodes r)"

value "nodes exampleTree"
"6" :: "nat"
```

3.2.2 Height

The *height* function calculates the height of the inputted binary tree. In Listing 2 we can see its definition in Isabelle and its usage to calculate that the example tree has a height of four. The height function satisfies some monotonicity properties which can be found in the appendix A.

Listing 2 height Function in Isabelle.

```
primrec height :: "'a tree ⇒ nat" where
  "height Tip = 0"
| "height (Node l e r) = Suc (max (height l) (height r))"

value "height exampleTree"
"4" :: "nat"
```

3.2.3 Longest

The *longest* function returns a set of lists of node data where each list represents a longest chain. In Listing 3 we can see its definition as well as its usage to calculate the longest chain of the example tree. In the example, we have used natural numbers as the node data. There were no relevant proofs associated with this function.

Listing 3 longest Function in Isabelle.

```
primrec longest:: "'a tree ⇒ ('a list) set" where
  "longest Tip = {[]}"
| "longest (Node l e r) =
  { e # p | p. p ∈
```

```
  (if height l > height r then longest l
    else if height r > height l then longest r
    else longest l ∪ longest r)}"

value "longest exampleTree"
"{[0, 1, 2, 5]}" :: "nat list set"
```

3.2.4 Check

The *check* function checks for input tree t that no other branch is within d nodes of one of the longest chains, up to depth n of the tree, returning true if it holds. We call d our difference value. In Listing 4 we can see its definition and its usage to show that the example tree does hold up to depth 1 with a difference value of 1 but does not hold at a depth of 2.

Listing 4 check Function in Isabelle.

```
fun check :: "nat ⇒ nat ⇒ 'a tree ⇒ bool" where
  "check 0 d t = True"
| "check (Suc n) d Tip = False"
| "check (Suc n) d (Node l e r) =
    (( (height l - height r > d) ∧ (check n d l) ) ∨
    ( (height r - height l > d) ∧ (check n d r) ))"

value "check 1 1 exampleTree"
"True" :: "bool"

value "check 2 1 exampleTree"
"False" :: "bool"
```

Furthermore, the first two properties we prove are the weakened statements of the depth and difference values, stating that if the check function is true for a given depth or difference value x then it will also be true for all values less than x. These statements are proven by structural induction over the tree parameter t as well as the depth parameter n. The common prefix lemma states that given a tree t that returns true on the check function for depth n and difference value d, all longest chains will have the same first n nodes. Listing 5 shows these properties in Isabelle, with the proof of the common prefix property in appendix A.

Listing 5 check Function Properties in Isabelle.

```
lemma check_weaken_distance:
  assumes "check n (Suc x) t"
    shows "check n x t"
using assms by (induction rule: check.induct, auto)

proposition check_weaken_depth:
  assumes "check (Suc x) d t"
    shows "check x d t"
using assms by (induction rule: check.induct, auto)

lemma common_prefix[rule_format]:
  "∀p p'. check n d t ∧ p ∈ longest t
  ∧ p' ∈ longest t ⟶ take n p = take n p'"
```

The common prefix lemma shows that the common prefix property holds for an arbitrary tree t that passes the check function. To prove consensus, we must now show that this property holds for the set of trees that can occur in our majority honest PoW network.

3.2.5 Event

We model our blockchain network as a sequence of events, where an event is the pair consisting of the *Honest* and *State* variable and describes an action carried out on a binary tree. The Honest field is of type Boolean and it is used to distinguish between honest and dishonest mining events, whilst the State field contains the state of the binary tree after the event has occurred. We then define a function *count* which can be used to count the number of honest or dishonest events in a list of events. We prove properties for adding events to a list of events in appendix A in Listing 16. The definitions can be seen in Listing 6.

Listing 6 Event Properties in Isabelle.

```
record 'a event =
  Honest :: bool
  State :: "'a tree"

definition count::"bool ⇒ ('a event) list ⇒ nat" where
  "count b = List.length ∘ filter (λx. Honest x = b)"
```

3.3 Mining

The mining locale is used to describe how we mine blocks and add them to the blockchain using add functions. It has two key properties:

1. Mining on top of an empty tree results in a tree with one node.
2. Mining on top of a non-empty tree adds the new block either to the left or the right branch.

From this, we prove the lemmas `mining_cases` and `height_add`. `mining_cases` is used to describe the cases of mining on either the left or right branch of a tree and follows easily from property 2 of the locale. `height_add` shows that the height of a tree either stays the same or increases by 1 after mining on it and is proved by induction on the binary tree variable. Listing 7 shows these properties in Isabelle, with the proof of `height_add` in appendix B.

Listing 7 Mining Locale in Isabelle.

```
locale mining =
    fixes add :: "'a tree ⇒ 'a tree"
    assumes m1: "∃e. add Tip = Node Tip e Tip"
        and m2: "⋀l e r. add (Node l e r) = Node (add l) e r
                    ∨ add (Node l e r) = Node l e (add r)"

  lemma mining_cases:
    fixes l e r
    obtains (l) "add (Node l e r) = Node (add l) e r"
          | (r) "add (Node l e r) = Node l e (add r)"
    using m2 by auto
```

```
12
13  lemma height_add:"height (add t) = height t
14                    ∨ height (add t) = Suc (height t)"
```

From these lemmas, we derive the lemma `check_add` which shows the three possible cases a tree can be after applying an add function to it, assuming the original tree passes the check function with difference value $d+1$ and depth n. The three cases are as follows:

1. We have mined on one of the longest chains. Here, the height of the new tree is higher than the original tree and can pass the check function with difference value $d+2$ and depth n.
2. We have mined on one of the second-longest chains. Here, the height of the new tree is equal to the original tree and can pass the check function with difference value d and depth n.
3. We have mined on one of the chains that is not one of the longest or second longest chains. Here, the height of the new tree is equal to the original tree and can pass the check function with difference value $d+1$ and depth n.

We prove this statement by structural induction over the tree variable. It can be seen in Listing 8 with its proof in appendix B.

Listing 8 *check_add* Lemma in Isabelle.

```
1 lemma check_add[rule_format]:
2 "check n (Suc d) t ⟶
3   height t < height (add t) ∧ check n (Suc (Suc d)) (add t)
4   ∨ height t = height (add t) ∧ check n (Suc d) (add t)
5   ∨ height t = height (add t) ∧ check n d (add t)"
```

Lastly, we create the corollary `check_add_cases` to distinguish between the cases of mining on a tree. This follows directly from `check_add` and can be seen in Listing 9.

Listing 9 *check_add_cases* Lemma in Isabelle.

```
1 corollary check_add_cases:
2     assumes "check n (Suc d) t"
3     obtains "check n (Suc (Suc d)) (add t)"
4           | "check n (Suc d) (add t)"
5           | "check n d (add t)"
6     using check_add[OF assms] by auto
```

3.4 Honest Mining

The honest locale is used to describe how honest participants mine blocks and add them to the blockchain. It has the same properties as the mining locale with the additional property that a mined block is always added to the longest chain of the network. Like in the mining locale, we prove similar `mining_cases`, `height_add` and `check_add` lemmas.

The cases for honest minings are adding to the left or right branch of a tree depending on which branch is longer. This follows from the new locale property. `height_add` shows that the height of the tree always increases by 1 when the mining is honest and is proved by induction on the binary tree variable. From these lemmas, we derive the `check_add` lemma which shows the only possible cases a tree can be after applying an honest add function to it,

assuming the original tree passes the check function with difference value d and depth n, is that the new tree passes the check function with difference value $d+1$ and depth n. This is the same as case 1 for the `check_add` lemma in the mining locale as we have mined on the longest chain. Listing 10 shows the locale in Isabelle, with the proofs of `check_add` and `height_add` found in Appendix C.

Listing 10 Honest Mining Locale in Isabelle.

```
1  locale honest = mining +
2    assumes h1:
3    "⋀l e r. height l ≥ height r ∧ add (Node l e r) = Node (add l) e r
4           ∨ height r ≥ height l ∧ add (Node l e r) = Node l e (add r)"
5
6  lemma mining_cases:
7    fixes l e r
8    obtains (l) "height l ≥ height r ∧ add (Node l e r) = Node (add l) e r"
9          | (r) "height r ≥ height l ∧ add (Node l e r) = Node l e (add r)"
10   using h1 by auto
11
12 lemma height_add: "height (add t) = Suc (height t)"
13
14 lemma check_add[rule_format]:
15   "check n d t ⟶ check n (Suc d) (add t)"
```

4 Verification

For the verification, we fix the state of the blockchain at a particular point in time, then define all possible future progressions of the tree structure and then show that all of them preserve the common prefix property.

To this end, we introduce a new locale `blockchain` which uses the mining locale for dishonest miners and the locale for honest miners. In addition, we introduce two new parameters to provide context to the model:

- A fixed tree $t0$ which represents the blockchain at a particular point in time.
- A natural number *depth*, which determines the length of the prefix which is considered stable, i.e., which should not change in the future.

In addition to these parameters, we assume two more properties:

1. The initial tree, $t0$ passes the check function for a threshold equal to the difference between its height and the depth value (property `b1` in Listing 11).
2. The height of the initial chain is greater than the depth (property `b2` in Listing 11).

We then define the set `traces`, which contains all possible event sequences which can occur in a network with the majority of miners being honest. The set is defined inductively and each sequence can be of one of the following cases:

1. The list with one event that is honest and applies an honest add function to $t0$ (Line 9 in Listing 11).
2. The list with one event that is dishonest and applies a dishonest add function to $t0$ (Line 10 in Listing 11).
3. A list of events that is already in traces, that is prepended with an honest event (Line 11 in Listing 11).

4. A list of events that is already in traces, with less dishonest events than the sum of the number of honest events and the threshold value, prepended with a dishonest event (Line 13 in Listing 11).

Listing 11 shows the blockchain locale and traces set in Isabelle.

Listing 11 Blockchain Locale in Isabelle.

```
1  locale blockchain =
2    honest hadd + mining dadd
3    for hadd::"'a tree ⇒ 'a tree" and dadd::"'a tree ⇒ 'a tree" +
4    fixes depth::nat and t0::"'a tree"
5    assumes b1: "check depth (Suc (height t0 - depth)) t0"
6        and b2: "height t0 > depth"
7
8  inductive_set traces :: "('a event list) set" where
9    honest_base: "[⦇Honest = True, State = hadd t0⦈] ∈ traces"
10 | dishonest_base: "[⦇Honest = False, State = dadd t0⦈] ∈ traces"
11 | honest_induct: "⟦t ∈ traces⟧
12   ⟹ ⦇Honest = True, State = hadd (State (hd t))⦈ # t ∈ traces"
13 | dishonest_induct:
14   "⟦t ∈ traces; count False t < count True t + (height t0 - depth)⟧
15     ⟹ ⦇Honest = False, State = dadd (State (hd t))⦈ # t ∈ traces"
```

Using traces, we create the lemmas `bounded_dishonest_mining` and `bounded_check`. `bounded_dishonest_mining` shows that given a list of events from traces, the sum of the number of honest events and the threshold value is greater than or equal to the number of dishonest events. This follows from assumption 2 of the blockchain locale. `bounded_check` states that given a list of events from traces then the most recent tree from that list passes the check function with a depth of depth and a difference value equal to the sum of number of honest events and the threshold value minus the number of dishonest events plus one. This statement is proven by induction of the tree variable for each case in traces. Listing 12 shows these lemmas in Isabelle with their proofs found in appendix D.

Listing 12 Blockchain Locale in Isabelle.

```
1  lemma bounded_dishonest_mining:
2    fixes t assumes "t ∈ traces"
3    shows "count True t + (height t0 - depth) ≥ count False t"
4
5  lemma bounded_check:
6    fixes t assumes "t ∈ traces"
7    shows "check depth
8          (Suc (count True t + (height t0 - depth) - count False t))
9          (State (hd t))"
```

We prove the consensus theorem within the blockchain locale. It states that given a list of events from traces and lists of the longest chains nodes, then the lists of these chains are the same up to index depth. This can be seen in Listing 13.

Listing 13 Consensus Theorem in Isabelle.

```
theorem consensus:
  fixes t assumes "t ∈ traces"
  and "p ∈ longest (State (hd t))"
  and "p' ∈ longest (State (hd t))"
shows "take depth p = take depth p'"
using assms(2,3)
common_prefix[of depth
  "Suc (count True t + (height t0 - depth) - count False t)"
  "(State (hd t))"]
bounded_check[OF assms(1)] by blast
```

The theorem is similar to the `common_prefix` lemma we proved earlier in Listing 5 with the key differences being that tree t is in the traces set and we use the *depth* value for parameter n in the check function. The `common_prefix` lemma was defined before we established any of our locales and so is just a lemma for the check function itself and does not consider anything related to consensus. The consensus theorem applies this lemma to the context of consensus and is essentially a proof of the common prefix property. As mentioned previously, we do not require the chain quality property under our assumption of majority honesty so the common prefix property is enough to show that consensus holds.

5 Discussion

Our verification of consensus in this model required us to make simplifications to how a blockchain network works. As mentioned in Section 3, we assume majority honesty and synchronisation of the network. Despite PoW assuming majority honesty, this is not always the case and so consensus can break down in the presence of a 51% attack. As for synchronisation, a blockchain network can become asynchronous because of dishonest behaviour or a fault within the network such as a participant having connectivity issues and not being able to update their private copy of the blockchain. As a result of this, consensus can still break down in a majority honest network.

Outside of these assumptions, we also made design choices for the model that lead to simplifications. The first of these is the use of a binary tree instead of an n-ary tree. With a binary tree, we can model forking at a node that can result in two chains. However, it is possible for there to be more than one fork at a node, resulting in three or more chains branching from a single node. A binary tree cannot be used to consider such an event, but in reality the likelihood of this event occurring is very unlikely as it would require three or more block hashes to be solved simultaneously and added to the chain at the same time. Using an n-ary tree in our verification would also make the process far more complicated as we would have to consider an arbitrary amount of branches for each node. It is for these reasons that we did not believe using a n-ary tree instead of binary tree was worth the effort required.

The main simplification of our model is that it is not probabilistic. Blocks are added to the blockchain by solving its hash which introduces a probabilistic element into the network as we cannot be absolutely sure how long such a brute force exercise will take. Because of this, majority honesty may not always hold as dishonest participants could get lucky in their hash searching whilst honest participants could get unlucky. Mining also plays a key role in the common prefix and chain quality properties described in [12], with their actual definitions being probabilistic in nature as a result. Specifically, these properties only have to hold with high probability but we show they always hold in our deterministic model.

6 Related Work

There is some work formalising traditional consensus protocols and some has even been mechanized in theorem provers. For example there exists a formalisation of Paxos [19] and Disk Paxos [15] in Isabelle, HotStuff [6] in Agda [3], and Velisarios [33] in Coq [35]. When it comes to blockchain, however, we usually do have less control over the actual network participants. Thus, verification of consensus in blockchain poses additional challenges compared to the verification of traditional BFT protocols.

There has been some early work on the verification of consensus in blockchain. In particular, blockchain in general [31, 22], the Bitcoin backbone protocol [12], general proof-of-stake [18], or a custom proof-of-work protocol known as Snow White [10]. While all these studies provide useful insides into blockchain consensus they are not mechanized and thus may contain mistakes.

More recently there has been some work on mechanizing blockchain consensus. For example there is a formalisation of Tendermint [4] using TLA+ and a formalisation of the Ethereum Beacon Chain [7] in Dafny. Moreover, there are formalisations of Casper [14], CBC Casper [28], Stellar [20], and general inter-blockchain Protocols [17] in Isabelle. Finally, there exists formalisations of Casper [30] (based on [14]), CKB [21, 5], Algorand [1], and Gasper [2] in Coq [35]. While all these works formalise various types of consensus protocols and some even verify certain properties for these protocols, none of them verify consistency in terms of common prefix which is the focus of our work.

There are three exceptions to this. First, there is the work of Pîrlea and Sergey [32] in which they formalise a general blockchain protocol in Coq. Similarly to our work, they model a blockchain as a tree (although implicitly as part of a forest). They then verify eventual ledger consistency, which is a property similar to the one we verify. The main difference to our work, however, is that they do not consider dishonest nodes in their analysis. Thus, by allowing for nodes with arbitrary behaviour, we complement their work.

In addition, Marmsoler formalised a type of general proof of work [23] in FACTum [24]. Similar to our work, the author considers honest as well as dishonest nodes in his model. Different to this work, however, a blockchain is modeled in terms of a list which does not allow to investigate forks. Thus, by modeling a blockchain as a tree instead of a list, we complement his work.

Finally, Thomsen and Spitters formalize a general, Nakamoto-Style Proof of Stake protocol in Coq [36]. They then verify a safety property similar to the common prefix property discussed in this paper. The main difference to our work lies in the type of considered consensus mechanism. While Thomsen and Spitters work is based on a general Proof of Stake consensus mechanism, our work focuses more on Proof of Work consensus.

7 Conclusion

In this paper, we have formally verified that consensus holds in a blockchain network that uses PoW and is assumed to be majority honest. This verification was carried out in Isabelle/HOL by modelling the blockchain as the longest chain in a binary tree. We have described the key functions and lemmas required to carry out this verification and outlined our assumptions of honesty and synchronisation for consensus to hold in our model. Lastly, we identify the key limitation of our model in that it is not probabilistic to account for probabilistic elements that occur within a PoW network such as mining. The successful verification of this paper serves as motivation for the implementation of the verified consensus protocol.

This paper could be expanded on in further research by aiming to address this limitation and developing a probabilistic verification of PoW using a different mathematical model such as state machines. Alternatively, a similar verification could be carried out on a different consensus protocol such as Proof of Stake (PoS) which is used by the Ethereum blockchain. However, PoS also has probabilistic elements that would need to be accounted for such as validator selection that would need to be considered or mitigated under a simplifying assumption.

References

1 Musab A Alturki, Jing Chen, Victor Luchangco, Brandon Moore, Karl Palmskog, Lucas Peña, and Grigore Roşu. Towards a verified model of the algorand consensus protocol in coq. In *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I 3*, pages 362–367. Springer, 2020. doi:10.1007/978-3-030-54994-7_27.

2 Musab A Alturki, Elaine Li, Daejun Park, Brandon Moore, Karl Palmskog, Lucas Pena, and Grigore Roşu. Verifying gasper with dynamic validator sets in coq. *Technical report*, 2020.

3 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, pages 73–78. Springer, 2009. doi:10.1007/978-3-642-03359-9_6.

4 Sean Braithwaite, Ethan Buchman, Igor V. Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper). In *FMBC@CAV*, 2020. doi:10.4230/OASIcs.FMBC.2020.10.

5 Hao Bu and Meng Sun. Towards modeling and verification of the ckb block synchronization protocol in coq. In *Formal Methods and Software Engineering: 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1–3, 2021, Proceedings 22*, pages 287–296. Springer, 2020. doi:10.1007/978-3-030-63406-3_17.

6 Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. Towards formal verification of hotstuff-based byzantine fault tolerant consensus in agda. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, pages 616–635, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-06773-0_33.

7 Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. Formal verification of the ethereum 2.0 beacon chain. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–182. Springer, 2022. doi:10.1007/978-3-030-99524-9_9.

8 Coindesk. How the dao hack changed ethereum and crypto, 2023. [Accessed December 2023]. URL: https://www.coindesk.com/consensus-magazine/2023/05/09/coindesk-turns-10-how-the-dao-hack-changed-ethereum-and-crypto/.

9 CoinMarketCap. Bitcoin market capitalization, 2023. [Accessed December 2023]. URL: https://coinmarketcap.com/currencies/bitcoin/.

10 Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 23–41, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-32101-7_2.

11 Forkast. How ethereum classic's 51ethereum, 2020. [Accessed December 2023]. URL: https://forkast.news/video-audio/ethereum-classic-repeat-hacks-etc-labs-ceo-terry-culver-ben-sauter/.

12 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015. doi:10.1007/978-3-662-46803-6_10.

13 Growjo. Certik revenue and competitors, 2022. [Accessed December 2023]. URL: https://growjo.com/company/CertiK.

14 Yoichi Hirai. A repository for pos related formal methods. https://github.com/palmskog/pos, 2018.

15 Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, jun 2005. , Formal proof development. URL: https://isa-afp.org/entries/DiskPaxos.html.

16 Elliot Jones and Diego Marmsoler. Towards Mechanised Consensus in Isabelle. version 1.0., (visited on 02/05/2024). URL: https://doi.org/10.5281/zenodo.10479776.

17 Florian Kammüller and Uwe Nestmann. Inter-Blockchain Protocols with the Isabelle Infrastructure Framework. In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *Open Access Series in Informatics (OASIcs)*, pages 11:1–11:12, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.FMBC.2020.11.

18 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference*, pages 357–388. Springer, 2017. doi:10.1007/978-3-319-63688-7_12.

19 Philipp Küfner, Uwe Nestmann, and Christina Rickmann. Formal verification of distributed algorithms. In Jos C. M. Baeten, Tom Ball, and Frank S. de Boer, editors, *Theoretical Computer Science*, pages 209–224, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-33475-7_15.

20 Giuliano Losa and Mike Dodds. On the Formal Verification of the Stellar Consensus Protocol. In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *Open Access Series in Informatics (OASIcs)*, pages 9:1–9:9, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.FMBC.2020.9.

21 Xiaokun Luan and Meng Sun. Modeling and verification of ckb consensus protocol in coq. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 660–667. IEEE, 2021. doi:10.1109/QRS-C55045.2021.00100.

22 Bojan Marinković, Paola Glavan, Zoran Ognjanović, Dragan Doder, and Thomas Studer. Probabilistic consensus of the blockchain protocol. In Gabriele Kern-Isberner and Zoran Ognjanović, editors, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 469–480, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-29765-7_39.

23 Diego Marmsoler. Towards verified blockchain architectures: A case study on interactive architecture verification. In *Formal Techniques for Distributed Objects, Components, and Systems: 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings*, pages 204–223, Berlin, Heidelberg, 2019. Springer-Verlag. doi:10.1007/978-3-030-21759-4_12.

24 Diego Marmsoler and Habtom Kashay Gidey. Interactive verification of architectural design patterns in factum. *Form. Asp. Comput.*, 31(5):541–610, nov 2019. doi:10.1007/s00165-019-00488-x.

25 Ahmed Afif Monrat, Olov Schelén, and Karl Andersson. A survey of blockchain from the perspectives of applications, challenges, and opportunities. *IEEE Access*, 7:117134–117151, 2019. doi:10.1109/ACCESS.2019.2936094.

26 Neptune Mutual. Ethereum classic 51 [Accessed December 2023]. URL: https://neptunemutual.com/blog/ethereum-classic-51-attacks/.

27 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008.

28 R. Nakamura, T. Jimba, and D. Harz. Refinement and verification of cbc casper. In *2019 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 26–38, Los Alamitos, CA, USA, jun 2019. IEEE Computer Society. doi:10.1109/CVCBT.2019.00008.

29 Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002. doi:10.1007/3-540-45949-9.

30 Karl Palmskog, Milos Gligoric, Lucas Pena, Brandon Moore, and Grigore Roşu. Verification of casper in the coq proof assistant. Technical report, University of Illinois at Urbana-Champaign, 2018.

31 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 643–673, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-56614-6_22.

32 George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 78–90, 2018. doi:10.1145/3167086.

33 Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. Velisarios: Byzantine fault-tolerant protocols powered by coq. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 619–650, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-89884-1_22.

34 Statista. Blockchain - statistics and facts, 2023. [Accessed December 2023]. URL: https://www.statista.com/statistics/800426/worldwide-blockchain-solutions-spending/.

35 The Coq Development Team. The Coq reference manual – release 8.18.0. https://coq.inria.fr/doc/V8.18.0/refman, 2023.

36 Søren Eller Thomsen and Bas Spitters. Formalizing nakamoto-style proof of stake. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–15. IEEE, 2021. doi:10.1109/CSF51468.2021.00042.

A Functions

Listing 14 Monotonic Properties of height Function in Isabelle.

```
proposition height_mono_l:
"height r ≤ height l ⟹ height l < height l' ⟹
 height (Node l e r) < height (Node l' e r)"
by (induction l; simp)

proposition height_mono_r:
"height l ≤ height r ⟹ height r < height r' ⟹
 height (Node l e r) < height (Node l e r')"
by (induction r; simp)
```

Listing 15 Proof of the common prefix lemma in Isabelle.

```
lemma common_prefix[rule_format]:
  "∀p p'. check n d t ∧ p∈longest t ∧ p'∈longest t ⟶
  take n p = take n p'"
proof (induction rule: check.induct)
  case (1 d t)
  then show ?case by simp
next
  case (2 n d)
  then show ?case by simp
next
  case (3 n d l e r)
```

```
  show ?case
  proof (rule+, (erule conjE)+)
    fix p p'
    assume a1: "check (Suc n) d (Node l e r)"
       and a2: "p ∈ longest (Node l e r)"
       and a3: "p' ∈ longest (Node l e r)"
    from a1 consider (1) "d < height l - height r ∧ check n d l"
    | (2) "d < height r - height l ∧ check n d r" by auto
    then show "take (Suc n) p = take (Suc n) p'"
    proof cases
      case 1
      then have "height r < height l" by auto
      then have "tl p ∈ longest l" and "tl p' ∈ longest l"
      using a2 a3 by auto
      then have "take n (tl p) = take n (tl p')" using 1 3 by blast
      moreover have "take (Suc n) p = hd p # take n (tl p)"
      using a2 by auto
      moreover have "take (Suc n) p' = hd p' # take n (tl p')"
      using a3 by auto
      moreover have "hd p = hd p'" using a2 a3 by auto
      ultimately show ?thesis by simp
    next
      case 2 (*symmetric*)
    qed
  qed
qed
```

Listing 16 Proofs for Adding Events to Lists of Events in both Honest and Dishonest Cases.

```
lemma count_true_base[simp]:
  "count True [] = 0" unfolding count_def by simp
lemma count_false_base[simp]:
  "count False [] = 0" unfolding count_def by simp

lemma count_honest_true_ind[simp]:
  assumes "Honest e"
    shows "count True (e#es) = Suc (count True es)"
    unfolding count_def using assms by simp
lemma count_honest_false_ind[simp]:
  assumes "Honest e"
    shows "count False (e#es) = count False es"
    unfolding count_def using assms by simp

lemma count_dhonest_false_ind[simp]:
  assumes "¬ Honest e"
  shows "count False (e#es) = Suc (count False es)"
  unfolding count_def using assms by simp
lemma count_dhonest_true_ind[simp]:
  assumes "¬ Honest e"
  shows "count True (e#es) = count True es"
  unfolding count_def using assms by simp
```

B Mining Locale

Listing 17 Mining *height_add* Proof in Isabelle.

```
lemma height_add:
"height (add t) = height t ∨ height (add t) = Suc (height t)"
  proof (induction t)
    case Tip
    then show ?case using m1 by auto
  next
    case (Node l e r)
    show ?case
    proof (cases rule: mining_cases)
      case l
      moreover from this have
      "height (add (Node l e r)) = height (Node (add l) e r)" by simp
      ultimately show ?thesis using Node(1) by auto
    next
      case r
      moreover from this have
      "height (add (Node l e r)) = height (Node l e (add r))" by simp
      ultimately show ?thesis using Node(2) by auto
    qed
  qed
```

Listing 18 Mining *check_add* Proof in Isabelle.

```
lemma check_add[rule_format]:
"check n (Suc d) t ⟶
height t < height (add t) ∧ check n (Suc (Suc d)) (add t)
∨ height t = height (add t) ∧ check n (Suc d) (add t)
∨ height t = height (add t) ∧ check n d (add t)"
  proof (induction rule: check.induct)
    case (1 d t)
    then show ?case using height_add by auto
  next
    case (2 n d)
    then show ?case by simp
  next
    case (3 n d l e r)
    show ?case
    proof
      assume "check (Suc n) (Suc d) (Node l e r)"
      then consider (l) "Suc d < height l - height r ∧ check n (Suc d) l"
      | (r) "Suc d < height r - height l ∧ check n (Suc d) r" by auto
      then show "height (Node l e r) < height (add (Node l e r))
      ∧ check (Suc n) (Suc (Suc d)) (add (Node l e r))
      ∨ height (Node l e r) = height (add (Node l e r))
```

```
      ∧ check (Suc n) (Suc d) (add (Node l e r))
      ∨ height (Node l e r) = height (add (Node l e r))
      ∧ check (Suc n) d (add (Node l e r))"
      proof (cases)
        case l
        then consider
        "height l < height (add l) ∧ check n (Suc (Suc d)) (add l)"
        | "height l = height (add l) ∧ check n (Suc d) (add l)"
        | "height l = height (add l) ∧ check n d (add l)" using 3 by auto
        then show ?thesis
        proof cases
          case 1
          then show ?thesis
          proof (cases rule: mining_cases)
            case l2: l
            moreover from 1 l have
            "check (Suc n) (Suc (Suc d)) (Node (add l) e r)" by auto
            moreover from 1 l l2 have
            "height (Node l e r) < height (add (Node l e r))" by auto
            ultimately show ?thesis by simp
          next
            case r
            consider "height (add r) = height r"
            | "height (add r) = Suc (height r)" using height_add by auto
            then show ?thesis
            proof cases
              case l1: 1
              then have "height (Node l e r) = height (add (Node l e r))"
              using r by simp
              moreover have "check (Suc n) (Suc d) (add (Node l e r))"
              using l l1 r by auto
              ultimately show ?thesis by simp
            next
              case x: 2
              moreover have "height l > Suc (height r)" using l by auto
              ultimately have
              "height (Node l e r) = height (add (Node l e r))"
              using r by simp
              moreover have "check (Suc n) d (add (Node l e r))"
              proof -
                have "d < height l - height (add r)" using x l by auto
                moreover have "check n d l"
                using l check_weaken_distance by simp
                ultimately show ?thesis using r by simp
              qed
              ultimately show ?thesis by simp
            qed
          qed
```

```
      next
        case 2
        then show ?thesis
        proof (cases rule: mining_cases)
          case l2: l
          moreover have "check (Suc n) (Suc d) (Node (add l) e r)"
          using 2 l by auto
          moreover have
          "height (Node l e r) = height (add (Node l e r))"
          using 2 l2 by auto
          ultimately show ?thesis by simp
        next
          case r
          consider "height (add r) = height r"
          | "height (add r) = Suc (height r)" using height_add by auto
          then show ?thesis
          proof cases
            case l1: 1
            then have "height (Node l e r) = height (add (Node l e r))"
            using r by simp
            moreover have "check (Suc n) (Suc d) (add (Node l e r))"
            using l l1 r by auto
            ultimately show ?thesis by simp
          next
            case x: 2 (*symmetric to 1*)
          qed
        qed
      next
        case 3 (*symmetric to 2*)
      qed
    next
      case r (*symmetric to 1*)
    qed
  qed
```

C Honest Mining Locale

Listing 19 Honest Mining *height_add* Proof in Isabelle.

```
lemma height_add: "height (add t) = Suc (height t)"
proof (induction t)
  case Tip
  then show ?case using m1 by auto
next
  case (Node l e r)
  show ?case
  proof (cases rule: mining_cases)
```

```
    case l
    moreover from this have
    "height (add (Node l e r)) = height (Node (add l) e r)" by simp
    ultimately show ?thesis using Node(1) by simp
  next
    case r
    moreover from this have
    "height (add (Node l e r)) = height (Node l e (add r))" by simp
    ultimately show ?thesis using Node(2) by simp
  qed
qed
```

Listing 20 Honest Mining *check_add* Proof in Isabelle.

```
lemma height_add: "height (add t) = Suc (height t)"
proof (induction t)
  case Tip
  then show ?case using m1 by auto
next
  case (Node l e r)
  show ?case
  proof (cases rule: mining_cases)
    case l
    moreover from this have
    "height (add (Node l e r)) = height (Node (add l) e r)" by simp
    ultimately show ?thesis using Node(1) by simp
  next
    case r
    moreover from this have
    "height (add (Node l e r)) = height (Node l e (add r))" by simp
    ultimately show ?thesis using Node(2) by simp
  qed
qed
```

D Blockchain Locale

Listing 21 *bounded_dishonest_mining* Proof in Isabelle.

```
lemma bounded_dishonest_mining:
  fixes t
  assumes "t ∈ traces"
  shows "count True t + (height t0 - depth) ≥ count False t"
  using assms b2 by (induction rule:traces.induct; simp)
```

Listing 22 *bounded_check* Proof in Isabelle.

```
lemma bounded_check:
  fixes t
  assumes "t ∈ traces"
```

```
  shows "check depth
  (Suc (count True t + (height t0 - depth) - count False t))
  (State (hd t))"
  using assms
proof (induction rule:traces.induct)
  case honest_base
  define t where "t = [⦇Honest = True, State = hadd t0⦈]"
  then have "Suc (count True t + (height t0 - depth) - count False t)
  = Suc (Suc (height t0 - depth))" by simp
  moreover from b1 have "check depth
  (Suc (Suc (height t0 - depth))) (hadd t0)"
  using honest.check_add by (simp add: honest_axioms)
  ultimately show ?case unfolding t_def by simp
next
  case dishonest_base
  define t where "t = [⦇Honest = False, State = dadd t0⦈]"
  then have *: "Suc (count True t + (height t0 - depth) - count False t)
  = (height t0 - depth)" using b2 by simp
  show ?case
  proof (cases rule: check_add_cases[OF b1])
    case 1
    then have "check depth (Suc (height t0 - depth)) (dadd t0)"
    using * unfolding t_def using check_weaken_distance by simp
    then show ?thesis using * unfolding t_def
    using check_weaken_distance by simp
  next
    case 2
    then show ?thesis using * unfolding t_def
    using check_weaken_distance by auto
  next
    case 3
    then show ?thesis using * unfolding t_def by simp
  qed
next
  case (honest_induct t)
  define t' where "t' = ⦇Honest = True, State = hadd (State (hd t))⦈ # t"
  moreover have "Suc (count True t + (height t0 - depth)) > count False t"
  using honest_induct bounded_dishonest_mining[OF honest_induct(1)]
  by simp
  ultimately have "count True t' + (height t0 - depth) - count False t'
  = Suc (count True t + (height t0 - depth) - count False t)" by simp
  moreover from honest_induct have "check depth
  (Suc (Suc (count True t + (height t0 - depth) - count False t)))
  (hadd (State (hd t)))" using honest.check_add[OF honest_axioms]
  by (simp add: honest_axioms)
  ultimately show ?case unfolding t'_def
  using check_weaken_distance by simp
next
```

```
  case (dishonest_induct t)
  define t' where "t' = ⦇Honest = False, State = dadd (State (hd t))⦈ # t"
  then have *: "count True t' + (height t0 - depth) - count False t'
  = count True t + (height t0 - depth) - Suc (count False t)" by simp
  have "check depth
  (Suc (count True t' + (height t0 - depth) - count False t'))
  (State (hd t'))"
  proof (cases rule: check_add_cases[OF dishonest_induct(3)])
    case 1
    then have "check depth
    (Suc ((count True t + (height t0 - depth) - (count False t))))
    (dadd (State (hd t)))" using check_weaken_distance by simp
    then have "check depth
    (((count True t + (height t0 - depth) - (count False t))))
    (dadd (State (hd t)))" using check_weaken_distance by simp
    moreover have "count True t + (height t0 - depth) > (count False t)"
    using dishonest_induct(2) by simp
    ultimately have "check depth
    (Suc (count True t + (height t0 - depth) - Suc (count False t)))
    (dadd (State (hd t)))" using Suc_diff_Suc by simp
    then show ?thesis using * unfolding t'_def by simp
  next
    case 2
    then have "check depth
    (((count True t + (height t0 - depth) - (count False t))))
    (dadd (State (hd t)))" using check_weaken_distance by simp
    moreover have "count True t + (height t0 - depth) > (count False t)"
    using dishonest_induct(2) by simp
    ultimately have "check depth
    (Suc (count True t + (height t0 - depth) - Suc (count False t)))
    (dadd (State (hd t)))" using Suc_diff_Suc by simp
    then show ?thesis using * unfolding t'_def by simp
  next
    case 3
    moreover have "count True t + (height t0 - depth) > (count False t)"
    using dishonest_induct(2) by simp
    ultimately have "check depth
    (Suc (count True t + (height t0 - depth) - Suc (count False t)))
    (dadd (State (hd t)))" using Suc_diff_Suc by simp
    then show ?thesis using * unfolding t'_def by simp
  qed
  then show ?case unfolding t'_def by simp
qed
```